I Know That Face

By: Jed Menard, Clayton Howeth,
and Patrick Nichols

For: FullContact

Date: June 24, 2016

Introduction

Who Is FullContact?

FullContact is a company striving to keep users fully connected to the people who matter most in their lives. They achieve this by merging and managing all of their users' contacts in their application program interface on a consumer and business level. Their API allows users to easily query by email, phone number, twitter account, and more. Results include publicly-available social profiles, photos, basic demographics, job titles, and hundreds of other public data points.

Project Description

The project, named "I Know That Face," is primarily a facial recognition and association program. At a high level, the code had to be able to accept a picture with a face in it and respond with the name of the person in the picture. At a more technical level, the software had to be able to recognize a person's face in a given picture and process it into "eigenfaces," a set of pictures that combined form the real face. These would be used for comparison to other faces. A database would then be used to assign a tag to the given face, and integrate the tagged photo into FullContact's contact merging software.

Requirements

Functional Requirements

To achieve these goals, the software had to attain a handful of functional requirements. First, the code needed to be able to analyze a picture and extract faces from it. Before any comparisons could be made, several of these faces needed to be used as a training set from which an average face and a "facespace" would be calculated. It then needed to be able to calculate the distance between an individual face, not an entire picture, and the average face. Once this "difference face" had been attained, it needed to be compared to those of the other faces in the database, using principal component analysis, to find similarities. Finally, the software had to associate all similar faces together in the appropriate person's contact information.

Non-Functional Requirements

In attaining these functional requirements, the software also needed to meet some non-functional requirements. Namely, the code had to be fairly quick and able to be integrated into FullContact's other software. In order to process the large number of incoming photos, the software needed to be able to identify faces in less than a second. After the face had been recognized, it then needed to be able to compare it to other faces in a matter of seconds. Some peripheral calculations, such as finding the mean

face, calculating the eigenfaces, or generating the facespace, could take a considerable amount of time. Fortunately, these calculations only need to be made once and can be stored for later use. This cuts down on computational time by a massive amount, but increases bootstrap time marginally.

System Architecture

The project is designed to work as a black box for finding people in a picture as seen in Figure 1. All the user needs to know is that an image can be sent to an HTTP endpoint, “WhoAmI,” and the name of the most similar person would be returned. There are a handful of other endpoints that provide other functionality or act as a proof of concept, but these are not needed by the user. Behind the scenes, the WhoAmI endpoint calls a number of functional processes to compute similarities. These behind-the-scenes processes are colored blue for the purpose of report explanation.

First of all, WhoAmI uses the FaceDetector to locate a face in a picture. The coordinates for a bounding box around the face are returned, and passed to one of the functions in the ImgManipulation set. This function crops out the face from the picture and resizes it to a standard size for computations. A number of other image manipulation functions are called to break the face down to a gray-scaled version and find a difference face, then these new images are stored in the database.

Finally, the functionality of the CompareToPeople endpoint is called to iterate through the people and faces in the database to find the most similar match. Once this is done, the name of the most similar match is returned to the user through the request that was sent to the endpoint.

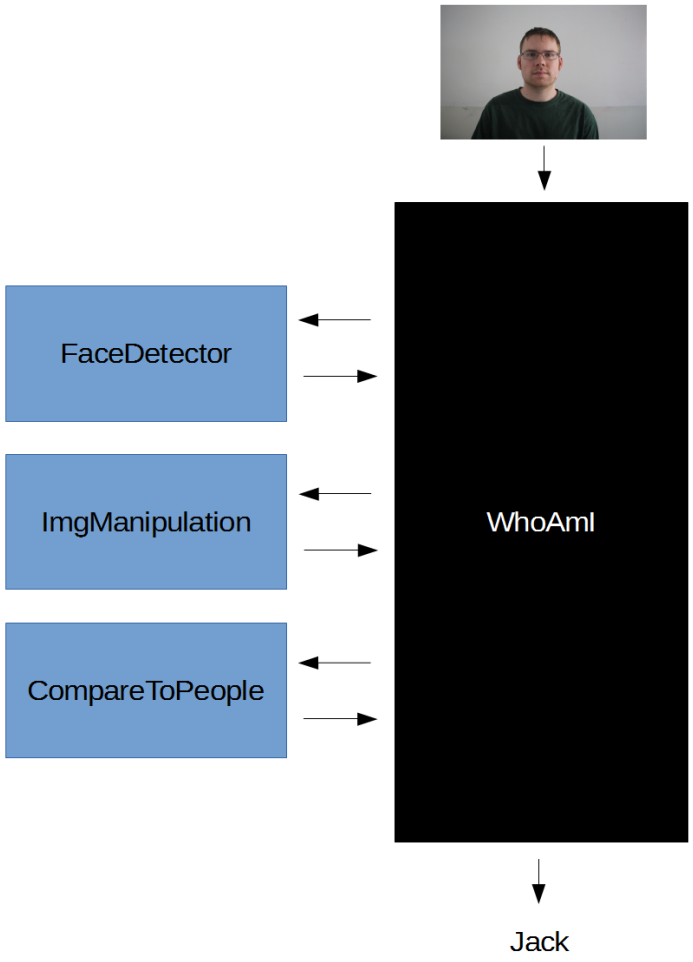

Figure 1

The functional code of the software is shown in the Figure 2 below. The HTTP endpoints have very little functionality written into them, and instead rely on calling functions that are written elsewhere.

The backbone of the design is integrated in the Database class. It keeps track of gray-scaled faces, original faces, people, and the Facespace. It can be used to instantiate a database on a new machine, and can load, update, and store any information that needs to be saved. When an endpoint needs to do any comparison or storing, the database is called.

The Person class is the main data structure used to keep track of relevant information. It contains the name of a person, an ID number in the database, and a list of the IDs of the face images that correspond to that person in the database. It can also initialize the mean face for that person's faces, and have it stored in the database.

DetectObject and ImgManipulation are where the vast majority of the computational logic is written. DetectObject contains the code to find a face, and ImgManipulation contains the code for everything needed to compare faces. It can pre-process an image to extract a face, prepare it for comparison, and perform the actual comparison using principal component analysis.

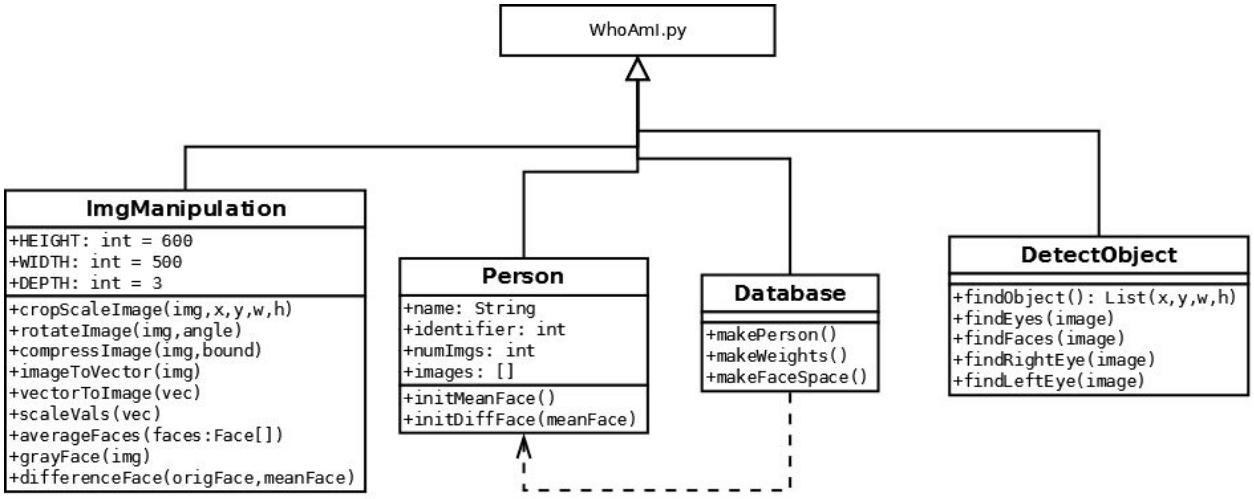

Figure 2

Tech Design

The vast majority of the complexity of the code comes from the ImgManipulation functions. The important functions in the collection are cropScaleImage, grayFace, averageImgArr, differenceFace, makeProjections, and compare.

DetectObject uses an open source library utilizing Haar Cascades to find faces. Once a face is found from DetectObject, the first function to be run is cropScaleImage. This function takes the bounding box of the face that was found and the original image, and crops the face out (Figure #3). It then scales it to a specific size for computations. Since large pictures get incredibly computationally expensive, all faces are scaled to a size of 500x600. This is set as a global constant and can be changed easily. Once the image has been cropped and scaled, it is returned.

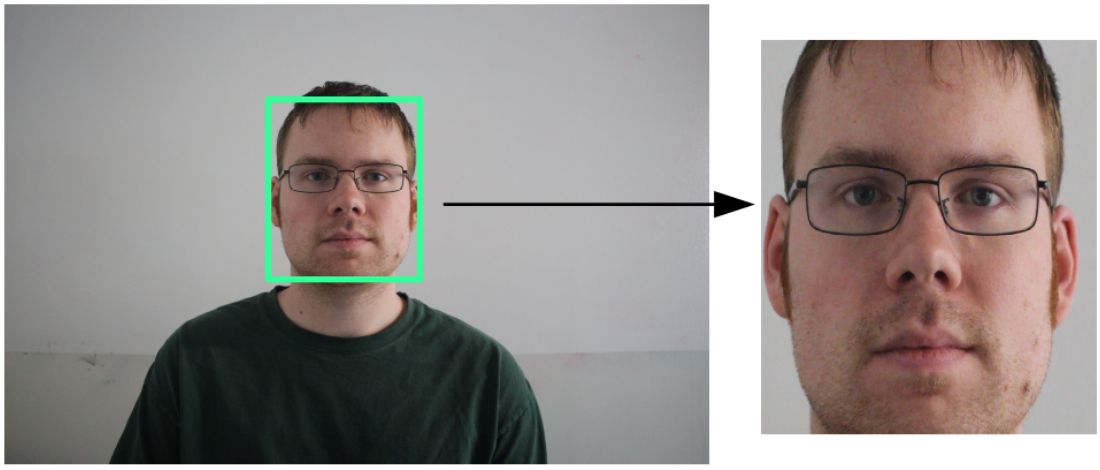

Figure 3

After a face has been extracted, the corresponding gray face is calculated (Figure #4). This is a fairly easy process, but takes a large number of computations. This is inherent in matrix calculations as every entry has to be iterated over. For every face, the code must run through 300,000 entries to find the average value in each pixel and a further 300,000 to store those in a new face.

Figure 4

When initializing a database on a new machine for the first time, averageImgArr must be called next. A large training set needs to be processed into gray faces, followed by combining them to find the “mean face,” or the averaged pixel values of all the faces. The individual mean face is used in calculations. If a person were to have ten saved pictures, the program would have to make ten times the number of calculations. Using the individual mean face in calculations reduces the computational process significantly.

Averaging a list of faces is the longest process, since it must iterate through every value in every face in the training set, average them, and store them in a new face. However, this only needs to be run once per database instantiation so this does not affect the computational speed of comparisons. The individual mean face of the person shown in Figure 4 is printed below in Figure 5.

Figure 5

After each individual mean face is calculated, the net mean face is calculated. This mean face, shown in Figure 6 below, is the average face of the entire dataset. The net mean face is used to calculate a difference face for each person. This face is a representation of how far the face is from the net mean face and is what is ultimately used to compare faces to one another. While this takes a reasonable amount of time to calculate, it does not need to be saved. This is only used when calculating the projection onto the FaceSpace, and is ultimately never used again.

Figure 6

After calculating the difference faces, the facespace is calculated. The facespace represents the Singular Value Decomposition of the set of faces it was trained on. Projections onto this facespace allow for the accounting of certain unique features of a person, increasing accuracy. all the images are turned into vectors by simply taking each row of the image and lining them into a single row. The vectors of this matrix undergo singular value decomposition, and they form the vector space that was previously referred to as the facespace. These rows can then be turned back into images, creating what are referred to as “eigenfaces”. These eigenfaces are the visual representation of each person’s uniqueness. A few of the truly haunting eigenfaces are displayed below in Figure 7.

Figure 7

In order to compare faces, a new face is projected onto the face space composed of the orthogonal components of the set of faces the detector was trained on. These projections are then compared to each other using their dot product and magnitudes.

Given two vectors, a and b, $\frac{a \cdot b}{|a||b|} = cos(\theta)$ where θ is the angle between the two vectors. This yields a value between 0 and 1. If the value is close to 1, it means the two vectors are very close together in the face space. If the value is close to 0, it means the two vectors are very far apart in the face space. If the projections are similar enough, it is assumed that the original pictures are of the same face and identifying information about the known face is returned. The projections are just the “weights” of each person’s mean face in the dataset. These weights are stored in the database to be compared against new faces in future comparisons.

Design Decisions

Python vs. Java

Several important design decisions were made regarding this project. Likely the most important one was deciding on which programming language to use. Most of FullContact’s API is built on Java, but the computer vision library being used, OpenCV, was very difficult to use in Java but easy to use in Python. Java unit testing is covered in the Mines curriculum, but Python unit testing is not. It came down to a question of which was easier, unit testing in Python, or the OpenCV native bindings in Java. A functional Python unit test was built in twenty minutes, and at that time the Java native binding procedure was no clearer, so a decision was made to use Python.

Storing Data

Another important design decision was made regarding the method used to store data. A choice had to be made between storing data on disk or using SQL. Using SQL would let the project grow better but would also be harder to implement. Storing all the data on disk would be easier to implement, but would be slower if the project ever had to grow. Storing the data on disk was chosen because the project was more of a proof of concept than something that was going to go into production.

Running Locally vs. Running on a Server

The final important design decision was whether to host the project on a local machine or to put it on a server. A local machine was familiar and easy, but it required somebody to have the entire project on their computer to run it. A server was unfamiliar, but it would allow a remote user to access the project. The decision was made to proceed with a server. The unreliability of local machines coupled with the clients desire to have the project on a server led to this decision.

Results

Stretch Goals

There were several stretch goals associated with this project that did not get completed. The first was the lack of code that would align faces. This code would have been valuable as it would have greatly increased that accuracy of the project. The project can currently correctly identify a face 10% of the time according to the tests that were run. While this is about 5 times better than random chance, it is still much less than would be needed for the project to go into production.

Another stretch goal that did not get accomplished was the ability to compare faces in real time. The idea was to implement a real-time video feed, most likely used by a computer's camera, that could box faces on screen with a name tag attached to the box. Though this would be a very valuable and useful feature, the six week deadline restricted the project deliverables.

Future Development

Along with the above stretch goals for the project are a few developments that could be applied in the future to continue the facial recognition process. Once such application is to apply the software to the entire photo database of FullContact. This would allow people to merge their contact photos, as well as recognize a person based on a photo passed into the FullContact API.

The next goal in the future of this project is the analysis of poor or sideways photos. The analysis of these photos would allow the dataset to restrict bad photos being passed in. For example, if someone had a silhouette photo of a person on a beach, the application could recognize that there is no face in the picture, therefore refraining from adding poor pictures to our control set.

A feature that was less than a week away of implementation, due to the six week restriction, is the ability to add a new face passed in if it did not match any of the known faces. A simple "Would you like to name this face?" function could quickly add this to the project.

Unforeseen Changes

Throughout development, several things changed in the project. The most significant was the shift from running the project on a local machine to running it on a remote server. This took almost a week of development time, but the client was very pleased with the results. The most valuable technical lesson learned in the development of this project was the ability to implement and run code on a server.

Summary of Data

Accuracy data for our project was obtained using a five-fold validation technique. This is where the software is trained on four-fifths of the pictures we had taken and the last fifth is tested against the training set. This is repeated five times, each time using a different fifth for testing. Once the data was collected, a full statistical analysis was performed.

The average likelihood of a correct match was only just above 10%. This is likely due to the nature of facial recognition and the fact that our training sets had very little variation. A number of confidence intervals, representing the range in which the true correct match percentage can be found, were also calculated and listed below.

Furthermore, the similarity rates for both correct and incorrect matches were analyzed individually. For each, the mean, standard deviation, variance, and three different percentiles were calculated. On average, the correct matches were only about 5% more similar than the incorrect matches.The incorrect matches were more widely varied than the correct ones, but only marginally.

Data

Accuracy data:
Correct: 44
Incorrect: 395
Total: 439
Most likely probability of correct match: .1002
Confidence intervals for probability of correct match:
80% confident: (.0853, .1224)
90% confident: (.0803, .1274)
95% confident: (.0754, .1322)
99% confident: (.0665, .1412)
Mean similarity:
Correct matches: .935169
Incorrect matches: .884915
Standard deviation/variance:
Correct matches: STD: .098810, Variance: .009763
Incorrect matches: STD: .100906, Variance: .010182
Percentiles:
Correct matches:
25th percentile: .924735
50th percentile (median): .958829
75th percentile: .983548
Incorrect matches:
25th percentile: .842611
50th percentile (median): .908124
75th percentile: .961679