MCECS Bot Jeeves
Face Recognition Package

Christopher Dean

ECE 589 Intelligent Robotics
Winter 2016
Portland State University

Table of Contents

- Introduction/Overview
  - FR Package At-A-Glance
- Face Detection and Recognition Algorithms
  - Haar Cascade Classifier
  - Correlation
  - Eigenfaces
  - Linear Subspace Method
  - Fisherfaces
  - Local Binary Pattern Histograms
- ROS Package
  - Face Recognition Implementation with OpenCV
  - Face Recognition Node
  - Performance
- Acknowledgements

Introduction/Overview

This write-up documents a software project in the form of a ROS package designed for the MCECS Bot, nicknamed Jeeves. It utilizes computer vision classes and functions from OpenCV 2.4 to perform face detection, recognition, and other actions.

Much of this document is dedicated to describing the various algorithms for face detection and recognition that I studied, as well as a description of the Face Recognition source code.

FR Package At-A-Glance

The central program in the ROS package is `face_recognition.cpp` which is launched on Jeeves' startup and starts the face detection and recognition node. Its contents are detailed later on in this document.

The node loads two pre-trained learning models: `haarcascade_frontalface_alt.xml` and `model.yaml`. The former is provided with OpenCV and is used to detect generic human faces in an image. When a face is seen by Jeeves, the node crops it out of the video frame and passes it to the Face Recognizer to predict who it is from the list of people the model is trained to, or if the face belongs to a stranger.

Currently Jeeves' FR model is trained to recognize four individuals. For more information about this model, including how to update it or add new people, see the "ROS Package" section.

The interface to the FR node is illustrated in the diagram below. It subscribes to images published by either the Freenect Node (controls a Microsoft Kinect) or a third-party node operating a webcam. It then publishes the coordinates of the face it sees to two topics (pitch and yaw). There is a Neck Control node that subscribes to these topics and uses the information to turn Jeeves' head toward the individual.

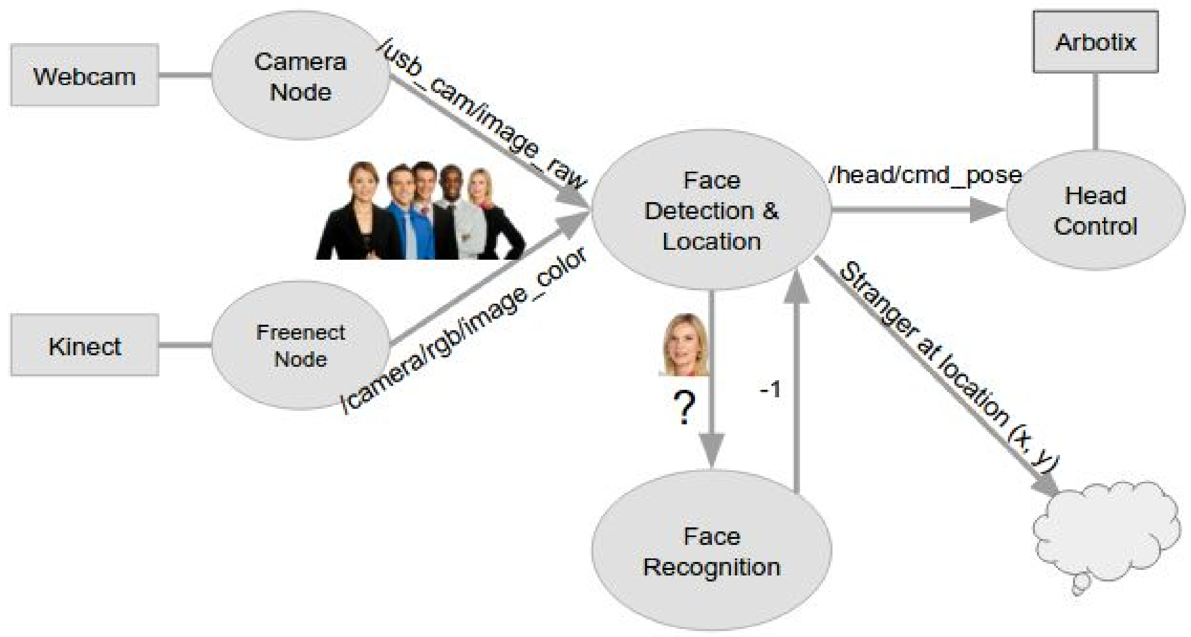


Diagram of the Face Detection/Recognition Node's interface

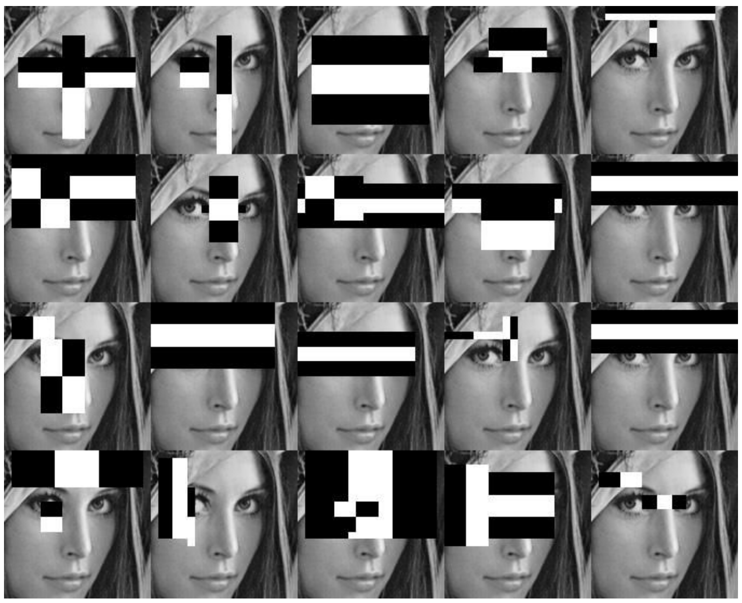
Face Detection and Recognition Algorithms

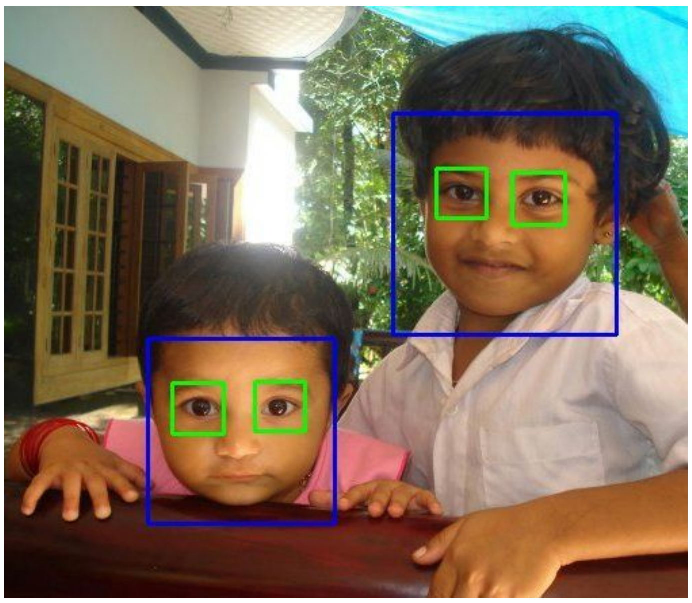
There are a number of algorithms for object detection and face recognition available in OpenCV. In order to decide which to use, I studied the papers listed in the final section of this document and chose the one best suited for this application. This section describes how each one works.

First, a Haar Cascade classifier is used to detect faces in images from the camera's video stream. OpenCV comes with Haar classifier models that are pre-trained to recognize faces and facial features such as eyes and mouths.

By manipulating OpenCV data types like Matrices, Points, and Rectangles, my Face Recognition package extracts faces detected by Jeeves and feeds them to one of three different face recognition algorithms: Eigenfaces, Fisherfaces, and Local Binary Pattern Histograms. These three methods are described here, as well as a couple other notable methods.

Haar Cascade Classifier

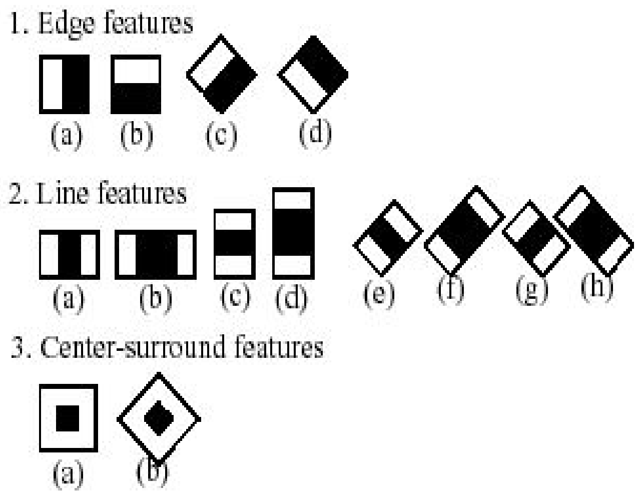


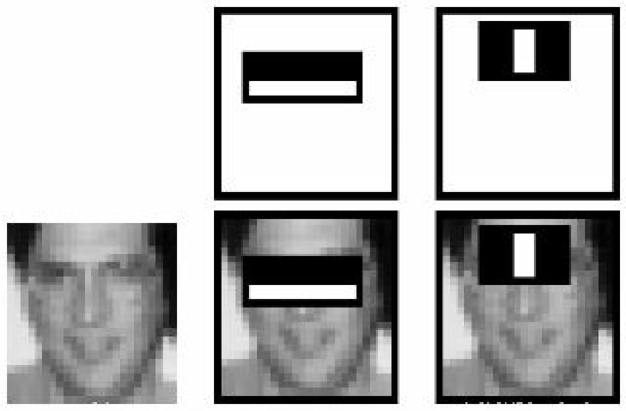


Haar Classifier: Using basic 2,3,4-rectangle objects to recognize common objects, such as a face

Training a Haar Feature-Based Cascade takes hundreds of positive and negative examples of the object you want to be able to recognize, in this case a human face.

During training, the algorithm applies every possible size and location of each of these basic rectangle features to every training image and calculate the error of each matchup. The set of rectangle features to be applied over all windows of your test image would number in the hundreds of thousands. The Viola-Jones paper hypothesizes that a small subset of rectangle features which most separate the positive and negative examples will be sufficient.





The Viola-Jones method [1] uses a variant of the Machine Learning algorithm called Adaptive Boosting for combining many weak classifiers into a strong predictive model. In this case the best rectangle features are chosen and represented as a weighted sum based on their predictive success among the training images. You repeat this process, focusing increasingly on the hardest-to-classify samples in the training set and tweaking the weights as you go along.

The vast majority of those original hundreds of thousands of features are discarded. But the boosting process still leaves thousands of the most predictive features, producing a trained model made of a linear combination of weak classifiers. In OpenCV, this trained model is then saved into an XML file for use when you want to perform detection.

It would get computationally expensive to apply these thousands of features to every region at various sizes in every image in a video stream. So the solution is to come up with various stages containing only a few features to weed out regions of your image that definitely don't have a face. A smaller set of features goes through and rejects most of the sub-windows in your image. You go through a successive number of these stages, called a cascade, and the regions that pass through all of the stages are said to have a face.

It seems strange that something that looks like this can be used to intelligently recognize complex objects. But there already exist many well qualified classifiers that people created for lots of different objects. Automobiles, cell phones, different types of plants… you can find the XML files of classifiers that have already been trained to these objects.

OpenCV comes with a few sample classifiers for detecting faces and facial features. My face detection program references one of these XML files, converts each incoming image to the Matrix data type, and runs the Cascade Classifier algorithm on it. That produces a list of all the faces it finds, for a series of incoming images. And then I perform a little logic and calculations to choose a location in Jeeves' field of vision, and then I extract the face from that location and perform recognition on it. I also publish those coordinates to the appropriate topic for head tracking.

Correlation

This first face recognition method is a very simplistic pattern classification technique and only serves as a starting point in the Fisherfaces paper [3]. You take grayscale images of all the people you want to be able to recognize, and balance their darkness. Treating an image as a set of vectors, you find the difference between the test image and the learning images. The closest neighbor is your match.

There are major disadvantages to this approach, but the basic idea makes a lot of sense. However, using some understanding of the mathematical space that an image occupies gives us tools to make improvements on simply comparing all pixels of the raw training images.

Eigenfaces

A 2D image occupies a very high-dimensional vector space. That is, you can describe an image of N pixels as a vector in a N-dimensional space. So a collection of sample images maps to a collection of points in this space.

Eigenfaces uses a concept called Principal Component Analysis, which relies on finding those vectors which carry the most information and therefore represent the most variation between the images. The aim is to represent the training images with a smaller set of basis images, effectively achieving dimension reduction. And then you can represent test faces using this basis set.

You get this set by computing the covariance matrix S, which generalizes the idea of variance to multiple dimensions.

S's Eigenvectors have the same dimension as the training images, so each can be seen as an image and so are called Eigenfaces. Each represents information about how the images differ from the mean. The subset of those vectors with the largest variance are the ones used for facial recognition.

The algorithm is described step-by step as follows:

1. Subtract the mean vector from each vector in the dataset
2. Compute the covariance/scatter matrix S
3. Compute Eigenvalues λ_i and Eigenvectors $\mathbf{v}_i$ of S
4. "Eigenfaces" are a basis set of Eigenvectors with the largest corresponding Eigenvalues.

Mathematically, you can describe the eigenfaces of a set of pxq images in the following way:

$$X = \{\mathbf{x}_1, \mathbf{x}_2, \dots \mathbf{x}_m\} \longleftarrow\text{-- Collection of m images each in pxq=n-dimensional space}$$

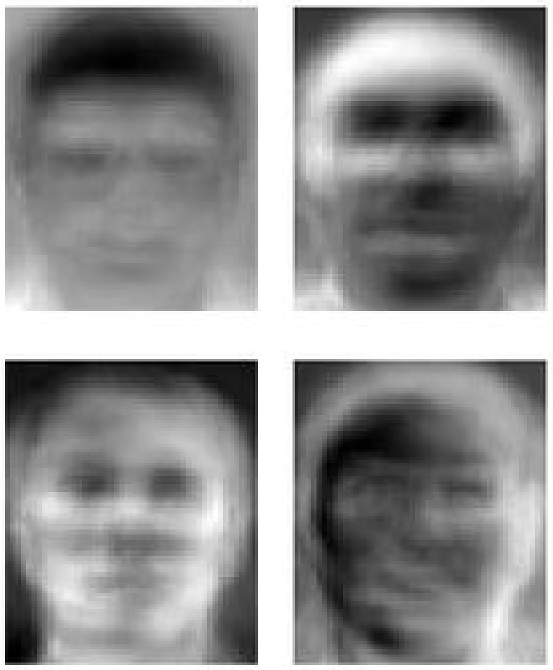
$$\text{mean } \mu = \Sigma \mathbf{x}_i / m \quad \longleftarrow\text{-- Average of all images}$$

$$S = (1/m)\ \Sigma\ (\mathbf{x}_i - \mu)(\mathbf{x}_i - \mu)^T$$

$$S\mathbf{v}_i = \lambda_i \mathbf{v}_i$$

$$W = \{\mathbf{v}_1, \mathbf{v}_2, \dots \mathbf{v}_k\}, \text{ where } \mathbf{v}_i \text{ is an Eigenface}$$

Here, S is a nxn transform matrix created from the training images and their distance from the average. The most significant eigenvectors of S are called Eigenfaces, because all of the eigenvectors are the same dimension (n) as the training images meaning they can also be displayed as an image:



Eigenfaces computed using training images from the AT&T Face Database

The largest eigenvalues correlate to the most indicative eigenvectors. Each eigenface represents how much the training images vary in that vector's direction.

You choose the number of Principal Components or Eigenfaces to use by arbitrarily setting a threshold for the total variance (represented as a sum of your eigenvalues), and choosing the smallest number k which crosses that threshold.

A linear combination of this subset of eigenfaces can then be used to represent any of the training images or a test image. You're of course losing information by discarding all but the principal components, but you're not losing much information because by definition you're keeping the ones which represent the most variance among the training data. That is, we keep the eigenfaces with the largest eigenvalues.

In performing recognition, you take the eigenface projection of your test image and compare it to the representation of each of the test images to find your match. (Every face is represented by a linear combination of the eigenfaces, use RMS on the coefficients to find the shortest "distance").

A major downside to this algorithm is that it requires a pretty controlled environment for taking pictures because it's pretty sensitive to lighting changes, facial expressions, etc. Plus, PCA works by maximizing the variance among ALL the data. So it does not consider a group of training samples for person A, person B, etc. And also, adding more training faces won't necessarily help accuracy, and in some cases may reduce the quality of the covariance matrix.

Linear Subspace Method

Both Correlation and Eigenfaces fail to consider different lighting environments, and the fact that a face is a surface that exists in physical space regardless of lighting. They don't take into account that you need a combination of images of one person in order to produce a good model; they're essentially methods to find one training image that's closest to the image under test.

The question of lighting is not a trivial one. It might be the most important question when recognizing complicated objects. Consider this animation of this woman's face under different lighting angles:

The animated image can be seen by clicking here
Or see the full video by artist Nacho Guzman here: https://vimeo.com/63602119

It is a mesmerizing animation because of how well it tricks our brains' natural ability to recognize facial features. Consider how you might explain to a computer that it's the same face, even with how completely different the pixels in each frame are, and how much difficulty even a human has with processing the animation.

The other method detailed in the paper which invents Fisherfaces is called Linear Subspaces [4]. Linear Subspace modeling treats the person as a Lambertian Surface, which is an ideal surface that scatters light equally in all directions. So the apparent brightness reflecting off a surface is the same regardless of the angle of view. This concept is actually used extensively in computer graphics to model diffuse light spreading on polygon surfaces.

Lambertian Reflectance

$$I_D = (\mathbf{L}\cdot\mathbf{N})*I_L = |L||N|\cos(a)*I_L = \cos(a)*I_L$$

I_D = surface brightness in any direction
I_L = source light intensity
$\mathbf{L}$ = unit vector in direction of light
$\mathbf{N}$ = vector normal to surface

Projection onto image

$$E_p = a_p\mathbf{N}_p^T\mathbf{s}$$

E = image intensity at point p
a_p = albedo (reflection coefficient) at p
$\mathbf{N}$ = normal vector at p
$\mathbf{s} = I_L * L$

The left equation describes the concept of a Lambertian surface, the main takeaway being that the surface brightness is the same in all directions and only relies on the light source intensity and angle, and not the observer's angle.

The right equation says that we can describe the light intensity at all points as seen from the camera in terms of the lighting conditions and surface characteristics at each point. Therefore, if we have three images of the same object viewed under linearly independent light sources, we can recover the surface normal and albedo at each point. This means that we can reconstruct the face under arbitrary lighting conditions simply as a linear combination of those three images.

This 3D surface is said to lie in a linear subspace of the original high-dimensional image space, with the three training images constituting a basis in this subspace. (each basis vector has the same dimensionality of the training images).

The main advantage is that it works under a wide variety of lighting conditions. The disadvantages are that you need to take training images under highly controlled conditions, and it's not good at handling self-shadowing of facial features and facial expressions.

Fisherfaces

The next method is Fisherfaces, named after the mathematician whose work Linear Discriminant Analysis is based off.

LDA maximizes between-class to within-class scatter as opposed to maximizing overall scatter between samples like Eigenfaces does. Eigenfaces fails to consider any classes, so a lot of discriminative data gets lost when ignoring the least indicative eigenvectors. What if a lot of the variance comes from an external source that's not part of the actual face, such as lighting and shadowing?

$$S_B = \sum_{i=1}^{c} N_i(\mu_i - \mu)(\mu_i - \mu)^T$$

$$S_W = \sum_{i=1}^{c} \sum_{x_j \in X_i} (x_j - \mu_i)(x_j - \mu_i)^T$$

$$W_{opt} = \arg\max_W \frac{|W^T S_B W|}{|W^T S_W W|}$$

From docs.opencv.org's FaceRecognizer tutorials page

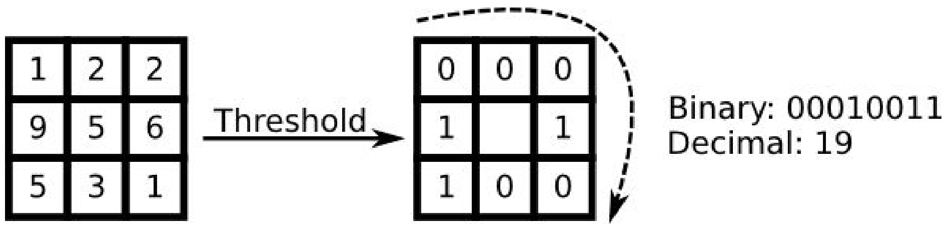
The math for Fisherfaces is similar to the Eigenfaces approach, except that you calculate your overall scatter matrix of N samples between all classes (S_B), where each u_i is the average image *of each class*. And S_W is the sum of all the scatter matrices of each class, for a total of c classes. And then you look for a projection W that maximizes the variance between classes and minimum variance within classes. This isn't the end of the story, you actually perform PCA on the data to project samples onto a smaller space first, and some other stuff, but what I have here is the concept in broad strokes.

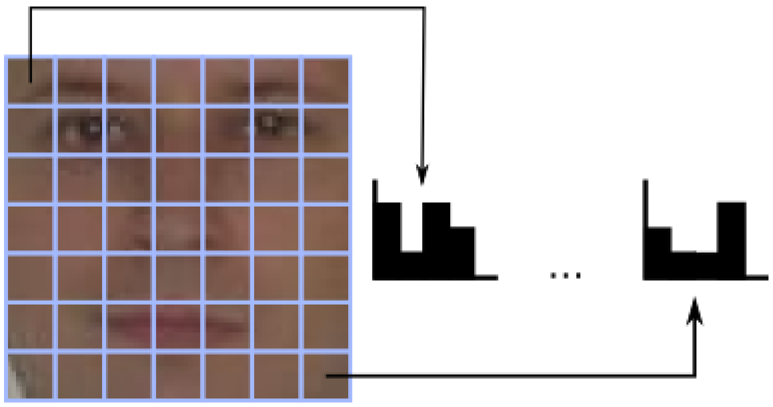
The resulting W projects onto a space defined by the number of classes, not the number of total images and finds the distance of your test image to the nearest class. All told, LDA works by finding the features to discriminate between the classes of people, so it builds a representation of each person in a class, as opposed to Eigenfaces which finds the best match among all the training images without regard to other images of that same person.

With a good set of training images for each person, this method isn't as sensitive to lighting as eigenfaces and in tests has a fairly higher rate of success.

Local Binary Pattern Histograms

The final algorithm offered by OpenCV is Local Binary Pattern Histograms (LBPH). As opposed to the previous examples, which takes the approach of considering the entire face holistically and treating it as a high-dimensional vector while attempting to project it onto a lower-dimensional space containing only the most useful information.



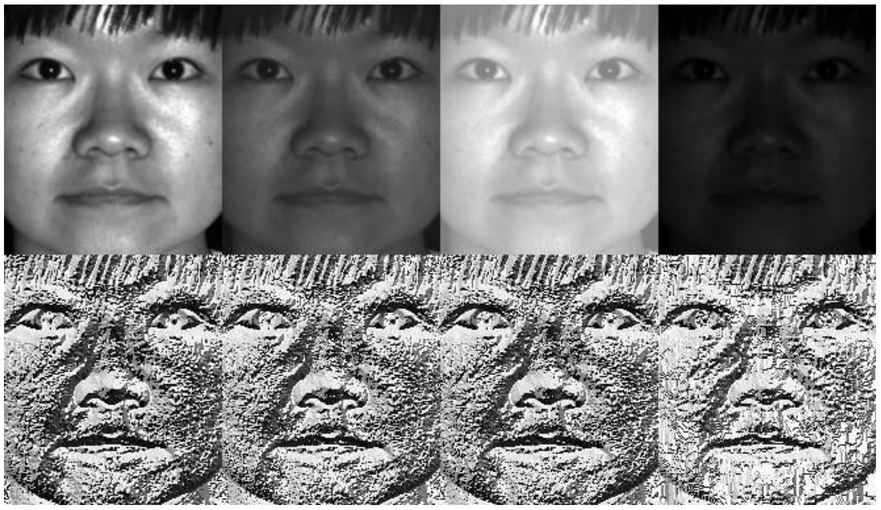


By creating histograms of each individual cell, LBPH focuses on local features instead of the face as a whole.

LBPH attempts to only describe localized features of an object and has roots in texture analysis. The algorithm is executed by dividing an image into nxn cells and extracting a histogram from each.

- For each pixel in a cell, threshold its neighbors against the center pixel's intensity.
- Surrounding 8-bit number is the cell's "value". Do so for all pixels in a cell.
- Cell's histogram is of the frequency of each value.
- Concatenate the histograms of all the cells to get a vector representing the entire image.

Since LBPH is computing the value of each pixel based on how it compares to its immediate neighbors, the way this method represents faces is robust against variations in overall intensity:



For a robot like Jeeves who will be moving around in various environments, this is important because the lightning conditions are not guaranteed to be very consistent.

ROS Package

Face Recognition Implementation with OpenCV

OpenCV has a class called `FaceRecognizer` which can be used to implement Eigenfaces, Fisherfaces, or LBPH. Each of the three available types are derived from the same class, and FaceRecognizer inherits from the Algorithm class. This makes it easy to switch between each one in your code to test and compare results.

- Eigenfaces (createEigenFaceRecognizer())
- Fisherfaces (createFisherFaceRecognizer())
- Local Binary Patterns Histograms (createLBPHFaceRecognizer())

The main member functions that FaceRecognizer provides are:

Training: `int train(vector<Mat> src, vector<int> labels)`

Prediction `int predict(Mat src)`

`void predict(Mat src, int& label, double& confidence)`

Loading/Saving: `void save (const string& filename)`

`void load(const string& filename)`

Updating(LBPH only): `update(vector<Mat> src, vector<int> labels)`

Training is done by providing a group of cropped face images in the OpenCV data type Mat, as well as providing an array of integers corresponding to the person in each image. Jeeves' FR package has a model that's trained to four people:

0 = Chris Dean
1 = Dr Perkowski
2 = Dean Su
3 = Mathias Sundari

There were 50 training images for each person, all cropped and balanced grayscale photos, and they included various poses and facial expressions.

When performing a prediction the confidence threshold is set. The threshold is the maximum distance a test image can be from one in the training model in order for it to be considered a match. Calling the prediction returns an integer corresponding to the person the test image is matched to, or -1 if there is no match. The program's threshold is currently set at 80 which performed well in experimentation.

Jeeves has the previously mentioned model in the FR package that's trained to recognize four individuals. It exists as a YAML file and is loaded by the node during initialization.

One thing to note is that Eigenfaces and Fisherfaces assume all of your training images are of equal size. I had to manually resize all of my training images to 100x100 pixels just to get my program to work with those algorithms. It will fail even if just one of the images is 99x100 by accident.

LBPH doesn't face that problem. Plus, it's the only algorithm which can support updating the trained model with additional images. To do that with FisherFaces and EigenFaces requires starting from scratch, because the transformation itself (the covariance matrix) is generated by combining the training images.

For these reasons I chose to use LBPH for Jeeves. Maintainability is a goal, and being able to update the model with new images is just not possible with the other algorithms. The 200 initial training images are not stored in Jeeves' repository and do not ever need to be referenced again because the model doesn't need to begin from scratch while updating. If MCECS Bot team members want to give an individual more training images or even add a new person altogether, the `update` method allows for that without disturbing the original data.

Face Recognition Node

The program which launches this node is written in C++ and consists of some initialization in the `main` function, including setting up the node, models, and publisher/subscribers and then waits in a loop for incoming images from the subscribed topic. When an image message is received, it gets passed to a callback function which is where most of the work gets done such as image processing, prediction, and head tracking calculations.

The code is described in detail below. Not every line is shown, just the important features of the program. Code such as error handling and minor details are left out of this document to improve readability and be more to the point in explaining how Face Recognition is achieved in the context of a robot running ROS.

Function Headers and Globals

```
/** Function Headers */
void ImageReceivedCallback( const sensor_msgs::ImageConstPtr& msg );
void ImageReceivedCallback2( Mat frame );

/** Global variables */
CascadeClassifier my_cascade; // The Haar Cascade Classifier
Ptr<FaceRecognizer> model = createLBPHFaceRecognizer(); // FR Model
ros::Publisher pub_yaw; // Creates topics to publish the face location
ros::Publisher pub_pitch;
```

Main Function

```
int main(int argc, char **argv)
{
  // Initialize node
  ros::init(argc, argv, "facial_recognition");
  ros::NodeHandle n;

  // Load LBPH model and Cascade files
  model->load("src/facial_recognition/people/model.yaml");
  model->set("threshold", 100.0);
  my_cascade.load( cascade_filename )

  // Set up Publisher
  pub_yaw   = n.advertise<std_msgs::Float32>("/head/cmd_pose_yaw", 100);
  pub_pitch = n.advertise<std_msgs::Float32>("/head/cmd_pose_pitch", 100);

  // Subscribe to third party usb_cam_node
  image_transport::ImageTransport it(n);
  image_transport::Subscriber subscribe = it.subscribe("/usb_cam/image_raw", 10,
ImageReceivedCallback);

  // Subscribe to Kinect images
  image_transport::ImageTransport itrans(n);
  image_transport::Subscriber sub = itrans.subscribe("/camera/rgb/image_color", 10,
ImageReceivedCallback);

  // Wait for incoming images
  ros::spin();
  return 0;
}
```

ImageReceivedCallback Function

This function is called when a message is received that contains the ROS standard image type. It must be converted to the OpenCV Mat type before we can work with it. The converted image is then passed to the second callback function.

The CV Bridge library used in this project was specially designed to foster compatibility between ROS and OpenCV, and it's used here to make the conversion.

```
// Image received from freenect topic gets converted to a Mat
// and then passed to the primary function
void ImageReceivedCallback(const sensor_msgs::ImageConstPtr& msg)
{
  // Create an OpenCV image type
  cv_bridge::CvImagePtr frame_ptr;

  // Convert the message to type cv::Mat
    frame_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::RGB8)

  // Send the converted cv::MAT image to the the main callback
  ImageReceivedCallback2(frame_ptr->image);
}
```

ImageReceivedCallback2 Function

This is the main workhorse of the program. It contains a couple hundred lines of code, so many of the tedious details have been omitted here to give a more concise picture of how the face recognition is implemented using OpenCV.

This function keeps track of the most prominent (biggest) face it sees in each image, and stores the location in a circular array of the most recent 15 faces. The published location is an average of these locations, converted to radians for head tracking. This is done to promote smooth movement of the head tracking and prevent Jeeves from suddenly jerking his head around when his attention shifts from one person to another.

```
void ImageReceivedCallback2(Mat frame)
{
  static Point recent_locations[MAX_FRAMES];  // Array of points corresponding to found faces
  static int count = 0;                       // Current index of recent_locations
  std::vector<Rect> faces;                    // Dynamic array of rectangles
  Point average_location( 0, 0);
  int biggest = 0;
  int valid_count = 0;
  Mat grayframe;  // Grayscale version of the passed-in frame

  // Convert frame to grayscale and equalize the histogram
  cvtColor( frame, grayframe, CV_BGR2GRAY );
  equalizeHist( grayframe, grayframe );

  // Use the Haar Cascade Classifier to check the frame for faces
  my_cascade.detectMultiScale( grayframe, faces, 1.1, 2, 0|CV_HAAR_SCALE_IMAGE, Size(30, 30) );

// Go through the list of faces in this image and store the best one in recent_locations[i]
// And for each person seen, perform face recognition.
  for( size_t i = 0; i < faces.size(); i++ )
  {
    // Find the point at the center of the Rect and the face size
    -OMITTED-

    // Extract face from location described by Rect, crop inward a bit, and equalize
    Rect faces_i = faces[i];
    Mat person = grayframe(faces_i);
    Mat person_resized;
    resize(person, person_resized, Size(120, 120), 1.0, 1.0, INTER_CUBIC);
    person_resized = person_resized(Rect(10, 20, 100, 100));  // Crop the face closer
    equalizeHist( person_resized, person_resized );

    // Predict!
    int prediction = model->predict(person_resized);
    cout<< "PREDICTION: " << prediction << endl;

    // Replace the entry in recent_locations if current size is bigger than previous
    // Draw ellipse on frame
    -OMITTED-

    // Draw text with the guess on the frame
    string person_name;
    switch(prediction){
      case -1: person_name = "Stranger";
              break;
      case  0: person_name = "Chris";
              cout << "HELLO CHRIS" << endl;
              break;
      case  1: person_name = "Perkowski";
              cout << "HELLO PERKOWSKI" << endl;
              break;
      case  2: person_name = "Dean Su";
              cout << "HELLO DEAN" << endl;
              break;
      case  3: person_name = "Mathias";
              cout << "HELLO MATHIAS" << endl;
              break;
    }
    putText(frame, person_name, Point(faces[i].x, faces[i].y), FONT_HERSHEY_PLAIN, 1.0,
CV_RGB(0,255,0), 2.0);
  } // Repeat this loop until all faces in this image have been checked

  // Count the number of valid frames among recent_locations
  // Calculate center of x and y
  -OMITTED-

  // Decide whether there are enough valid frames to go ahead with publishing a head location
  if(valid_count > (MAX_FRAMES/2))
  {

    // Message type instances
    std_msgs::Float32 yaw;
    std_msgs::Float32 pitch;

    // Convert to radians and limit angle to Jeeves' neck constraints
    // Round off to prevent micro-movements
    -OMITTED-

    // Publish to Pitch/Yaw topic
    pub_yaw.publish(yaw);
    pub_pitch.publish(pitch);

    // Publish to pub_coordinates topic
    coordinates.x = yaw.data;
    coordinates.y = pitch.data;
    pub_coordinates.publish(coordinates);
  }

  // Cycle through circular array of most recent points
  count++;
  if(count >= MAX_FRAMES) count = 0;
}
```

Currently Face Recognition prompts no action from Jeeves besides displaying the person's name on the web page video display. However, this information is made available to any future high-level nodes that might perform intelligent interaction with people near Jeeves.

Performance

In order to test performance, I tested the model using five images of each of the four individuals, plus five strangers. These are all images that were not part of the training set. Below are the images that were used and the resulting confusion matrices. I tested recognition with a few different levels of confidence thresholds.

The confidence threshold is defined as the RMS average of the Euclidean distances for each point (or histogram values in the case of LBPH). So for a grayscale image with each pixel encoded as a byte value, you can set a threshold between 0 and 255. A threshold of zero would only return a match for a test face if the image is exactly the same as one in the training set, and a threshold of 255 would always return a match.

A lower threshold means it will only recognize a person if the test image is a very close match, otherwise it will report it is a stranger. A high threshold will have the opposite problem: it will think strangers are people it recognizes because it is easier to meet the criteria for a match.

In the confusion matrices, the rows represent the actual person in the image, and the count in each column represents who the face recognition software guessed it was.

Threshold = 30 (very low)

	Chris	Perkowski	Dean Su	Mathias	Stranger

Chris					5
Perkowski		2			3
Dean Su			2		3
Mathias				4	1
Stranger					5

Threshold = 80 (best performance)

	Chris	Perkowski	Dean Su	Mathias	Stranger
Chris	5				
Perkowski		5			
Dean Su			5		
Mathias				5	
Stranger	1				4

Threshold = 125 (too high)

	Chris	Perkowski	Dean Su	Mathias	Stranger
Chris	5				
Perkowski		5			
Dean Su			5		
Mathias				5	
Stranger	2	2		1	

As expected, the face recognition performs poorly when the threshold is too low or high. The most accurate prediction is around 80, which is where Jeeves currently has his FR threshold set.

However, depending on Jeeves' future desired behavior it might actually be better to set a lower threshold. This is because the Face Recognition node is processing all of the images from a video stream, so it will be testing the same person many times per second. Since there would be many chances for Jeeves to recognize that person, it might be best to minimize the number of false positives while increasing confidence that any matches it does find are true.

Acknowledgements

OpenCV tutorials by Philipp Wagner (www.bytefish.de) were incredibly helpful for researching the FR algorithms and while working on the FR project for Jeeves.

The OpenCV face recognition documentation page was very instructive and was the source of a number of images included here:
http://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec_tutorial.html

[1] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, 2001, pp. I-511-I-518 vol.1.

[2] M. Turk and A. P. Pentland, "Eigenfaces for Recognition," in *Journal of Cognitive Neuroscience*, vol. 3, no. 1, pp. 71-86, Jan. 1991.

[3] P. N. Belhumeur, J. P. Hespanha and D. J. Kriegman, "Eigenfaces vs. Fisherfaces: recognition using class specific linear projection," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 7, pp. 711-720, Jul 1997.

[4] T. Ahonen, A. Hadid, and M. Pietikäinen. Face recognition with local binary patterns. In The 8th European Conference on Computer Vision. Springer, 2004.