APPM 3310 - Project Report
Support Vector Machines
Dr. Lyles
05/05/2017

Kyle Li, * Richard Moon, † Nicholas Renninger ‡
University of Colorado - Boulder

Support Vector Machines (SVMs) are a class of algorithms used in solving both linear and non-linear binary classification problems. Non-linear sets of data, like images, can be classified by an SVM by defining the sets of data with their quantitative distinguishing features using PCA. We seek to implement a MATLAB program for machine learning that would first process image data using an eigenface algorithm, and then classify the image data with an SVM. The data processed by the eigenfaces algorithm is used to train the SVM classifier, allowing the SVM to separate with maximal margin the different classes of image data with a hyperplane. We developed two separate codes: one to classify an inputted image as a discrete class, and another to determine regions of the eigenface space associated with each class tested. Our classification algorithm was successful, classifying with high accuracy test images using only 15 basis vectors from the eigenface (PCA) process. Our test data also yielded multiple, accurate classification regions based on the training data we gave our SVM. These classification regions could then be used to classify additional data we introduce to the SVM.

*SID: 104649224
†SID: 105675495
‡SID: 105492876

Contents

List of Figures

I. Attribution

Kyle Li designed the presentation. Richard Moon and Nicholas Renninger wrote code to implement the Eigenfaces and multi-class SVM algorithms, as well as wrote their respective sections of the presentation. All three contributed to preparing the report and project proposal.

II. Introduction

SVMs came to prominence in the 1990s as a leading machine learning algorithm and are best used in solving single or binary classification problems. While the theory behind other margin maximizing algorithms had been around for a few decades, several factors contributed to their rise in the 1990s. The sizeable jump in computing power and the introduction of the kernel trick for classifying non-linear data by Vapnik were both prime factors for the increased popularity of SVMs. In our particular application, we seek to implement a MATLAB library for machine learning called LIBSVM to do facial recognition. Before classification can occur, we must process image data from the (Yale image) database in order to reduce its dimensionality and reduce computational load. This dimensionality reduction is done by a specialty SVD to PCA algorithm for facial data, called the Eigenfaces algorithm. SVMs by design, produce a hyperplane that separates classes of data; any data that falls within a certain margin of the hyperplane gets classified accordingly.[4] The hyperplane itself changes to fit the training data that is fed to the SVM.

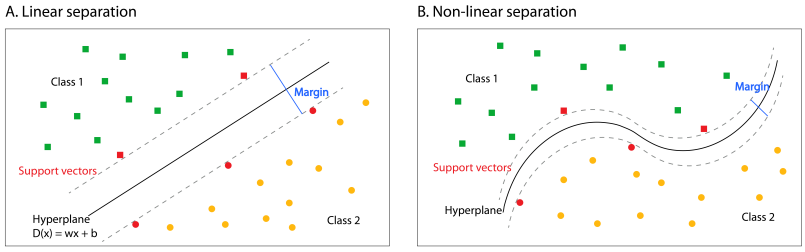


Figure 1: Figure depicting an SVM's ability to Classify both Linear and Non-Linear Data.

The math behind SVMs involves utilizing singular value decomposition to perform principle component analysis, utilizing an eigenvector basis to reduce data dimensionality, the kernel trick involving semi-positive definite matrices to represent inner products in higher dimension space, Euclidean minimization, minimization of norms, and using inner products to calculate norms.

First off, singular value decomposition (SVD) is the factorization of a matrix into its component parts. SVDs take advantage of the concept that any matrix A multiplied by its transpose A^T creates a symmetric matrix. Symmetric matrices have the nice property of forming an orthonormal basis for eigenvectors. A full SVD involves the factorization of matrix A into three component matrices, $U\Sigma V^T$, where Σ is a diagonal matrix of the roots of the eigenvalues (singular values).[8] For our particular project, principle component analysis (PCA) is a simple application of the SVD and is utilized to reduce the dimensionality of our facial data. Think of storing a size 500 x 500-pixel image. Encoding this matrix would require a great deal of memory if we decided to save every pixel value as a discrete value in memory. So, we utilize the SVD to try to find a good approximate matrix of the original image which is what we are doing with PCA, enabling us to decrease the computational load of this project. The aforementioned orthonormal basis of eigenvectors gives the ability to reduce dimensionality simply by choosing fewer basis vectors to represent each image.

The Kernel Trick is an algorithm that computes the inner product between two vectors without explicitly computing the coordinates of the vectors in the feature space of the kernel function. By mapping the inner product space to a non-linear space, like a quadratic cone in the polynomial kernel, the data may be linearly separable in this higher dimension. The special trick here is that two vectors $\vec{v}$ and $\vec{w}$ in the space $\mathbb{R}^N$, computes the inner product between $\vec{v}$ and $\vec{w}$, which implicitly, in a higher-dimensional feature space, calculates the support vectors and the hyperplane of the feature space with the largest margin, without explicitly calculating the new coordinates of $\vec{v}$ and $\vec{w}$ in $\mathbb{R}^M$. This greatly reduces computation time and needed memory to compute these operations. Positive semi-definite matrices define the new inner product, and as such are needed by the kernel trick because inner products cannot exist without all eigenvalues being real and non-negative. Thus, the kernel trick requires a positive definite kernel matrix which then defines each kernel function.

Euclidean minimization is the process of minimizing the Euclidean distance between a point and an n-dimensional subspace. For non-point subspaces, the minimized distance between the subspace and point

is the distance between said point and the point in the subspace where the resulting vector between the two points is orthogonal to the subspace. This is important to our problem because SVMs generate a hyperplane decision boundary to separate the data classes it processes. To ensure maximum accuracy we need to maximize the margin between the hyperplane and the support vectors, which originate from the data sets closest to the decision boundary. Geometrically the distance of the margin between two support vectors and the hyperplane is defined as

$$\frac{2}{||\vec{w}||}$$

where $\vec{w}$ is the normal vector to the hyperplane. As you can see based on the definition, to maximize the distance we must minimize the norm of matrix $\vec{w}$, which is where Euclidean minimization is needed.

III. Mathematical Formulation

In order to apply a SVM to facial recognition, we need a way to represent each image with quantitative features of the image that the SVM can read as a set of feature vectors and a set of labels for all of the classes. While we could simply feed the SVM raw data from each image and appropriate labels, this would be highly inefficient, as the SVM would need to work in the random subspace that the raw pixel values provide. Thus, we would prefer a way to process the data first; to both project the data onto a space where each feature vector contains more meaningful data about the image itself, as well as to reduce the number of feature vectors defining each image. For these reasons, we chose the eigenface algorithm in order to convert each image into a set of top quantitative feature vectors. The eigenface algorithm is used to define a face as a vector of weighted components of corresponding eigenfaces, as every face's distinctive features can be found by summing the proportionate amount of eigenfaces, which are the standard basis vectors that every face can be built from.

III.A. Eigenface Algorithm

For our project, we used n x n square images in the theoretical formulation of the eigenface algorithm. We used a database with m number of images. In order to change the basis into eigenface space, we need to find the eigenvalues and eigenvectors of the covariance matrix about the features of the images. To get this covariance matrix, each image must first be converted into a column vector of n^2 x 1 size, which we will call Γ_i where $1 \leq i \leq m$. We do this for all m images. We then find the value of the average human face, Ψ, which is just the mean of all the face column vectors:[7]

$$\Psi = \frac{1}{m}\sum_{i=1}^{m}\Gamma_i$$

Once we find this column vector, we can subtract the mean human face vector from each of the face column vectors Γ_i, which creates a column vector, Φ, which represents the distinctive features an image has compared to the average human face of the database:

$$\Phi_i = \Gamma_i - \Psi$$

We then make a matrix of all the Φ_i for all m images (a matrix containing all of the distinguishing features of each image), which we call A, where A is a n^2 x m matrix:

$$A = [\Phi_1\Phi_2...\Phi_m]$$

By subtracting the mean human face from each of the column vectors, each of resulting vectors in the A matrix can be considered random vectors, which allows us to create a covariance matrix C:[1]

$$C = AA^T$$

The covariance matrix of the features of each image can be found my multiplying the A matrix by its transpose, which we will call C, where C is an n^2 x n^2 matrix; however, calculating values from these large matrices led to the creation of algorithms like the eigenface algorithm which reduces the computational load

by reducing the number of explicit calculations needed to calculate eigenvalues of C. We know that the eigenvalues, λ_C, and eigenvectors, $\mathbf{u}$, of C can be found using the equation:

$$C\mathbf{u} = \lambda_C \mathbf{u}$$

However, this is a large matrix to do computations with (it is n^2 x n^2), so we instead look at the matrix G, a Gram matrix which is defined as:

$$G = (A^T A)$$

which is an m x m matrix, and we find that the eigenvectors, $\mathbf{v}$, and eigenvalues, λ_G, of the G matrix can be written as:

$$G\mathbf{v} = \lambda_G \mathbf{v} = A^T A\mathbf{v} = \lambda_G \mathbf{v}$$

which when the A matrix is multiplied to both sides becomes:

$$AA^T A\mathbf{v} = A\lambda_G \mathbf{v}$$

and because scalar multiplication is commutative becomes:

$$AA^T A\mathbf{v} = \lambda_G A\mathbf{v} = C\mathbf{u} = \lambda_1 \mathbf{u}$$

which means that $\mathbf{u} = A\mathbf{v}$ and $\lambda_G = \lambda_1$. This allows us to find and use the eigenvalues and eigenvalues of the m x m matrix G instead, which is computationally easier to deal with. It also limits the number of eigenvalues and eigenfaces of AA^T, as there can be up to n^2 distinct eigenvalues for the covariance matrix, with a corresponding number of eigenvectors, but using the G matrix of rank m the maximum number of distinct eigenvalues is m eigenvalues and eigenvectors. Each of these new basis eigenvectors is called an eigenface, and is a standard basis face that can be used to build up any face in this new eigenface space. With the eigenface algorithm, a k number of most significant eigenfaces (eigenfaces that correspond to the largest eigenvalues) can be chosen to define how many eigenfaces the user wants to use to describe each image. And by projecting each image into the eigenface space, we can now represent each matrix as a linear combination of the weight of each of these eigenfaces, where the weight w can be found from

$$\Phi_i = \sum_{j=1}^{k} w_j \mathbf{u_j}$$

$$w_j = \mathbf{u_j}^T \Phi_i$$

and the linear combination vector is represented as Ω which is a k x 1 column vector and $\Omega = [w_1 w_2 ... w_k]^T$.[7] We then make a matrix of all m of these coefficicent vectors into a k x m feature matrix. In relation to singular value composition, we can define the covariance of A as $\frac{A*A^T}{m}$, and let the singular decomposition of A be:[8]

$$A = P\Sigma Q^T$$

so the covariance matrix $\frac{AA^T}{m}$ would be:

$$\frac{AA^T}{m} = \frac{P\Sigma Q^T Q\Sigma^T P^T}{m} = \frac{P\Sigma I\Sigma^T P^T}{m} = \frac{P\Lambda P^T}{m}$$

where Λ is the diagonal matrix of eigenvalues, where $\lambda_{ii} = \frac{\sigma_{ii}^2}{m}$, with λ being the eigenvalue for AA^T and σ being the singular value for A, and P is the eigenfaces, which are the left singular vectors of A (also the orthonormal eigenvectors of AA^T).

From the math above, we have created a feature matrix, A, of all m images of n^2 x 1 size, in the normal image space, and projected them into the eigenface space, where a k number of eigenvalues and corresponding eigenvectors were chosen at the user's digression, and the corresponding eigenvectors define the eigenface basis and space. This is the new feature space, and the new feature vector of the image data is k x m.

III.B. Support Vector Machine Algorithm

Support vector machines need two training input vectors, the first input vector being the feature vectors that represents each of the data points in feature space which we call $\mathbf{x}$, and the classifications for each of the feature data, which we call y. Because SVMs are binary classifiers, the classifications are treated as 1 or -1 (either the image data is or is not the desired class). The support vectors of a support vector machine are the vectors closest to the margin of the hyperplane that if removed, will change the length of the margin. They also represent vectors which have positive α_i values. The line that decides whether a data point is one class or the other (this is affected by the magnitude of the margin) is defined by the *discriminant function* $f(\mathbf{x})$:

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$

where b is the bias of the discriminant. The hyperplane is placed where the Euclidean minimized distance between the each of the support vectors and the margin is maximized. The hyperplane can be defined by all the $\mathbf{x}$ vectors which satisfy the equation:

$$\mathbf{w}^T\mathbf{x} - b = 0$$

where $\mathbf{w}$ is the vector normal to the hyperplane (a weight vector), b is the offset the hyperplane has from the origin. Now assume that the weight vector, $\mathbf{w}$, can be expressed as linear combination of the training data points $\mathbf{x}$:

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i \mathbf{x}_i$$

$$f(\mathbf{x}) = \sum_{i=1}^{m} \alpha_i \mathbf{x}_i^T\mathbf{x} + b$$

$\mathbf{w}$ is normal to the hyperplane as the minimized Euclidean distance from a point and a line is always orthogonal to the line. It is also worth mentioning that a hyperplane, of a given space of dimension n, is a subspace with dimensionality 1 less then that of the main space, n-1. Now, noticing that $\alpha_i\mathbf{x}_i^T\mathbf{x}$ is simply the definition of the inner product, we can redefine the discriminant function, which determines the line that divides positive and negative classes, with a kernel function $K(\mathbf{x}, \mathbf{x}')$ that redefines the inner product on some new feature space:[6]

$$f(\mathbf{x}) = \sum_{i=1}^{m} K(\mathbf{x}, \mathbf{x}') + b$$

$$K(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T\phi(\mathbf{x}')$$

where $\phi(\mathbf{x}$ is a non-linear function mapping to a new feature space. For the Gaussian or Radial Basis Function (RBF) kernel used in this application, the kernel function is defined as:

$$K(\mathbf{x}, \mathbf{x}') = exp(-\gamma||\mathbf{x} - \mathbf{x}'||^2)$$

$$||\mathbf{x}|| = \sqrt{\mathbf{x}^T\mathbf{x}}$$

where γ controls the width of the Gaussian and thus the flexibility of the resulting classifier. Increasing γ too much can cause over-fitting problems. The *margin* of the decision line can be defined as the distance between the decision line and the closest data points (the support vectors). Mathematically, it follows from the formulas above the the margin M can be defined in terms of a weight vector as:

$$M(\mathbf{w}) = \frac{1}{2}\hat{\mathbf{w}}^T(\mathbf{x}_+ - \mathbf{x}_-)$$

Now we want to define the size of the margin based on our previous formulation, and thus we define the points $\mathbf{x}_-$ and $\mathbf{x}_+$ as points equidistant to the margin on either the negative of positive side of the decision line, respectively. Using our discriminant formula defined above, we get that:

$$f(\mathbf{x}_-) = \mathbf{w}^T\mathbf{x}_- + b = -\rho$$
$$f(\mathbf{x}_+) = \mathbf{w}^T\mathbf{x}_+ + b = \rho$$

where ρ is just some constant distance > 0. Setting $\rho = 1$, as it simply scales with the margin (essentially is a unit), fixing the value of the discriminant function at points closest to the hyperplane (the support vectors), and adding the two equations above and dividing by $||\mathbf{w}||$ we get:

$$M(\mathbf{w}) = \frac{1}{2}\hat{\mathbf{w}}^T(\mathbf{x}_+ - \mathbf{x}_-) = \frac{1}{||\mathbf{w}||}$$

Thus, the maximum-margin classification algorithm works by maximizing the margin $M(\mathbf{w}) = \frac{1}{||\mathbf{w}||}$ which is equivalent to minimizing the quantity $||\mathbf{w}||^2$. Thus the problem becomes the constrained optimization problem, the derivation for which is beyond the scope of this document:

$$minimize\, \frac{1}{2}||\mathbf{w}||^2 + C\sum_i \zeta_i$$

$$subject\, to:\ y_i(\mathbf{w}^t\mathbf{x}_i + b) \geq 1 - \zeta_i\, for\, i = 1, ..., m\, \&\zeta_i > 0$$

where y_i is the label (positive or negative class) for each data point $\mathbf{x}_i$, and ζ_i is a slack variable that allows an outlier data point to be within the margin, in order for the margin with respect to the rest of the data points to be maximized. The constant $C > 0$ is set to control how important maximizing the margin and minimizing the slack are relative to each other. This gives the user a control over the trade-off between large margins with some decrease in overall accuracy, or perfect accuracy with narrow margins that allows for little slack. However, this form of optimization problem, named the "Primal" form the Lagrangian optimization problem, is challenging to solve. Instead, the more useful "Dual" form of the optimization problem is given using the method of Lagrange Multipliers as:

$$maximize \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} y_i y_j \alpha_i \alpha_j K(\mathbf{x}_i^T\mathbf{x}_j)$$

$$subject\, to: \sum_{i=1}^{n} y_i\alpha_i = 0,\ 0 \leq \alpha_i \leq C$$

With the Dual form of the Lagrangian optimization in hand, the weight vector can be put in terms of an expansion about the input data points $\mathbf{x_i}$ and y_i:

$$\mathbf{w} = \sum_{i=m}^{m} \alpha_i y_i \mathbf{x}_i$$

Using this relation leads directly to the calculation of the **support vectors**, the points $\mathbf{x}_i$, which correspond to any vectors where $\alpha > 0$.[6] Usually for feature vectors that require a non-linear hyperplane to classify the vectors, a kernel function can be used in the equations above to solve for the support vectors (and thus the decision boundary) of the model data. For our specific application, we used the Gaussian (RBF) Kernel, which defines the inner product of the feature vectors in such a way that the feature space of the kernel is of infinite dimensions, which allows data of large dimensionality to be classified in the infinite dimension feature space.

IV. Examples and Results

IV.A. Implementation and Algorithms

The implementation of a facial recognition algorithm required the use of both the PCA and SVMs in order to achieve an efficient and effective algorithm for identifying faces and predicting classifications. These algorithms were implemented in MATLAB using a mix of built-in libraries (like the `fitcsvm` set of functions) as well as the popular LIBCSVM for actually identifying faces. The MATLAB code was split up into two large main functions, `MainMultiClassSVM.m` and `classifyNoOverlap.m`. `MainMultiClassSVM.m` is a script that uses a database of images, along with a set of user-inputted desired classes to classify, to calculate regions of the eigenface space that are the most likely to be associated with a certain class, as calculated probabilistically with the SVM. `classifyNoOverlap.m` is a script that takes user input on which image in the database to identify, after which the classification algorithm finds all of the other images in the database that match the predicted class of the chosen image.

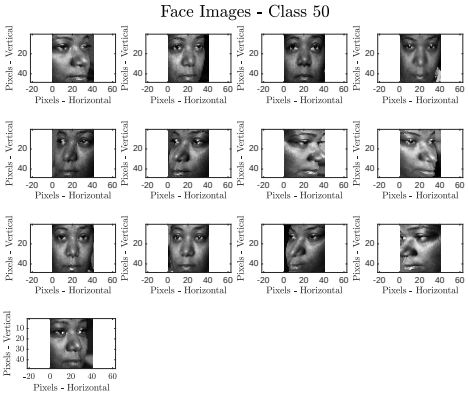


Figure 2: Example of all faces in the database for one class (person). *There are 13 examples for each class, with the 50th class shown above.*

In order to classify images, a database of faces needed to be used. The database we chose to use was the Yale Images Database,[5] which had a variety of faces pre-processed for our use. Each of the images in the Yale database was already cropped to a standardized resolution of 48 by 40 pixels, and each image was centered on the head. This centering is critical to the eigenfaces algorithm, as otherwise the basis elements it extracts will not be lined up in a standardized way with each face. Thusly, the information about a face contained in each eigenface basis element will be drastically reduced and scattering will occur in the target space after each image is projected.[7]

For our application, we chose to use a subset of the database containing 68 different classes, where each class in this instance is a different person. For each class, there were 13 different images of the class subject under various lighting conditions and with different poses of the person's face used in each image, as seen in Figure 2. This broad range of conditions for each class allows for a diverse eigenface basis to be calculated, where an incoming image projected into this new basis will see expected classification, despite lighting and positional variances that normally upset facial recognition algorithms.

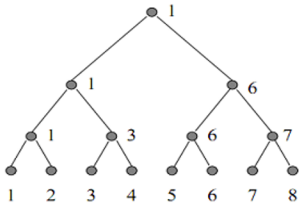
Figure 3: Example of the principle eigenvectors in the eigenface basis, reformatted into image form. *Each of these images represent a basis element in the new eigenface space - an "eigenface". This means that any new face can be represented as a linear combination of these eigenfaces*

Once the database has been loaded into MATLAB, the first part of the analysis processed the database into a meaningful set of weights in the eigenface basis using the PCA process described in §III. By first performing an SVD, the eigenvectors of the covariance matrix were determined and then used to project each of the images into the "face space". Once the images were projected into the new, more informative basis, the user-chosen number of basis vectors were actually used for the classification (this is actually the PCA part of this algorithm). The weights (or coordinates) of each image in the new basis, with the number of basis elements determined by the PCA, are called *feature vectors* in the new space. These feature vectors represent how much of each eigenface basis element each image is made of, and represent the "features" of each image in the new basis.[7] The principle eigenvectors in this new face space represent the new basis elements, and thus each image can be made of a linear combination of these new basis elements, called "eigenfaces", in the new basis (shown in Figure 3).

One key thing to note here is that the face chosen by the user for identification is not included in the training data in any way so that the classification made by the SVM of the chosen face is made without any class bias. This means that the eigenface basis is

calculated without the chosen image, and when the chosen image is projected into the eigenface basis and when training data sets are chosen, it is as if the program had never seen it before, more closely replicating real-world classification applications. This means that our program could be given *any* image of a class contained in the database, and the program would be able to accurately classify it.

Once the database is imported, and the images are projected with a limited number of eigenface basis elements into the new basis, the SVM can begin to classify the data. In our algorithm, the SVM was trained with two classes worth of feature vectors calculated from the database (in our case it would use 12 images from each class), each feature vector containing the chosen number of principle eigenface basis elements. An SVM model is trained by giving it as inputs this training data as well as the class label for each image. In our case, the first class we fed the SVM was always labeled as a positive class, for reasons we will now expand on. As an SVM is a binary classification algorithm, classifying a potential of 68 different classes is obviously not possible with a standard SVM algorithm.[3]



The binary tree structure for 8 classes face recognition. For a coming test face, it is compared with each two pairs, and the winner will be tested in an upper level until the top of the tree. The numbers 1-8 encode the classes. By bottom-up comparison of each pair, the unique class number will finally appear on the top of the tree.

Figure 4: Details of the winner-take-all algorithm employed in a multi-class SVM.[3]

For this reason, a multi-class SVM algorithm was used in our application in order to determine which class the chosen face best matched. The multi-class SVM algorithm is essentially a winner-take-all binary tree competition type algorithm. This means that each time the SVM was trained on two classes, it would determine whether the first class it was given had a higher probability than the second class it was given of being the correct class. If the first class had the higher probability of being the correct class as chosen by the SVM, it would move on to be classified against another class. Likewise, if the first class had a lower probability of being the correct class, the second class would move on to be classified against another class. This process repeats itself until the SVM classifies the two classes out of the 68 with the highest probability of being the chosen image and finally, chooses an overall class with the highest probability of being the chosen class. This process is shown visually in Figure 4.

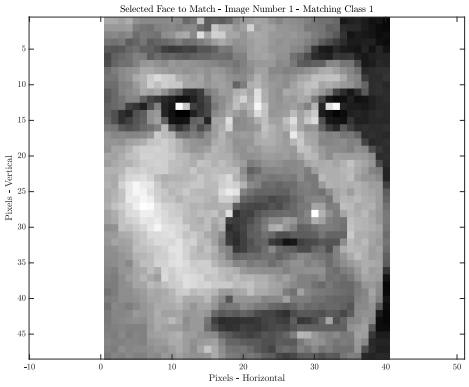
For both `MainMultiClassSVM.m` and `classifyNoOverlap.m`, this winner-take-all algorithm is employed, just with slightly different motivations. In the `classifyNoOverlap.m` program, the code directly computes which class from the database most accurately matches input image, and then displays all of the examples of the class in the database. However, for `MainMultiClassSVM.m`, the code actually computes regions in the eigenface basis that are most likely to be the inputted classes. By employing a multi-class SVM, and creating a mesh of test data points over the entire the eigenface basis space, the SVM can classify at each point in space which class the SVM would classify a new image as if it were to be projected onto that exact point in the new basis. This again is done by comparing each of the SVM models to one another at each point in the space and determining which class has the highest probability for classification for each point in the mesh.

IV.B. Results

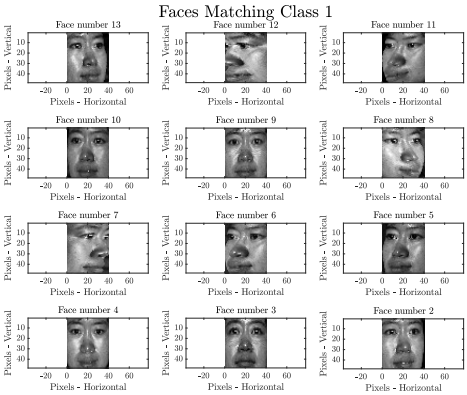
Figures 5 - 8 were obtained by running `classifyNoOverlap.m` on the given Yale database, with different classes chosen to be classified. In general, as long as the SVM could classify the face at all using the maximal number of feature vectors (748 for our data), the accuracy of the classification could be preserved using only 15 feature vectors, as shown it the classification examples shown in Figures 5 - 7. As long as at least 15 feature vectors were used, the SVM only misclassified a minimal number of the input images. The 15 feature vector number was determined through empirical testing and was used as a heuristic throughout the remainder of the testing. The SVM used for classification utilized a RBF Kernel with $\gamma = 1/numFeatureVecs$ for classification, due to the highly non-linear nature of the facial feature data.

However, there were some images in the database that our SVM method simply could not classify, regardless of the number of feature vectors used (i.e. even using every feature vector, the SVM still could not accurately distinguish some classes). One example is any member of class 37, shown in Figure 8 with all 748 feature vectors used for classification. Even with all feature vectors used, the SVM returned an incorrect classification of image 377, classifying the input image (which was originally a member of class 29)

as a member of class 37. No matter how many feature vectors were chosen, the classifier could not make the distinction between these two classes. Some further kernel parameter tuning may have helped with this task, but regardless the similarity between class 29 and class 37 remains a challenging problem.

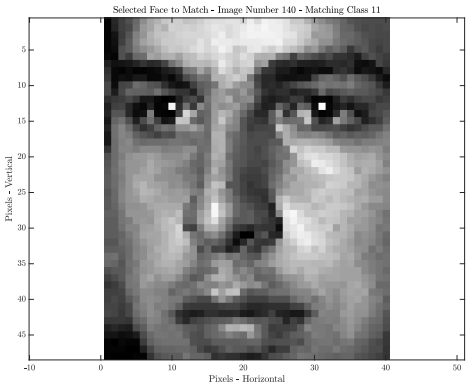


(a) Chosen Face to Classify

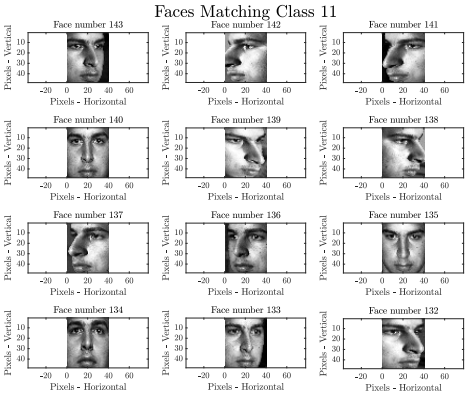


(b) Images from Class 1 chosen to match image 1

Figure 5: Image # 1 Correctly Classified as Member of Class 1

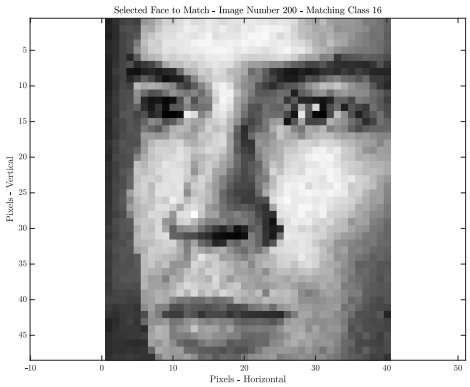


(a) Chosen Face to Classify

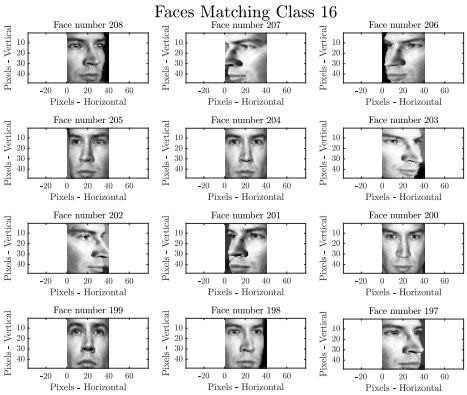


(b) Images from Class 11 chosen to match image 140

Figure 6: Image # 140 Correctly Classified as Member of Class 11

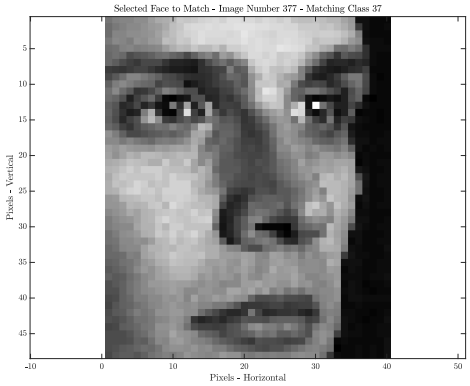


(a) Chosen Face to Classify

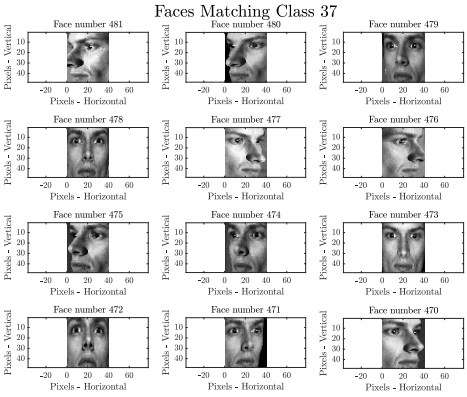


(b) Images from Class 16 chosen to match image 200

Figure 7: Image # 200 Correctly Classified as Member of Class 16



(a) Chosen Face to Classify

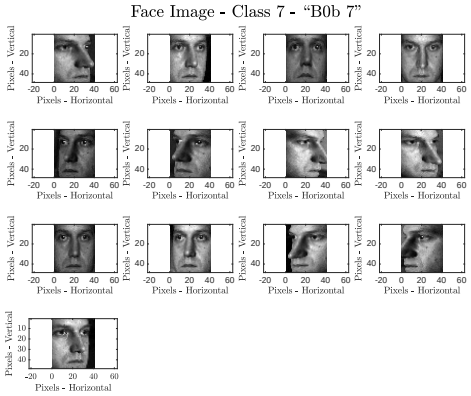


(b) Images from Class 37 chosen to match image 377

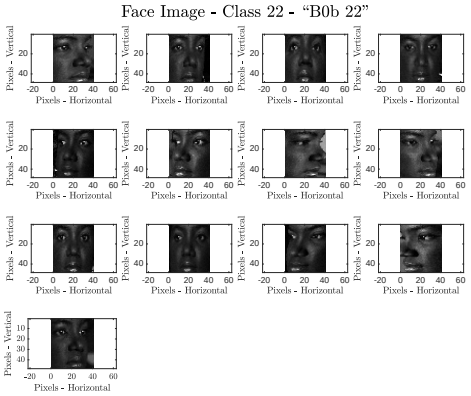
Figure 8: Image # 377 Incorrectly Classified as Member of Class 37

Figures 9 - 10 were obtained by running `MainMultiClassSVM.m` on the given Yale database, asking the program to classify regions in the eigenface space for 4 classes: 7, 22, 24, and 46. The faces used in this classification example are shown in Figures 9a - 9d. For this example, only two feature vectors were used, such that the margin and non-linear hyperplane separating the classes of data could easily be visualized in 2-D. For practical classification purposes, more than two eigenface basis elements would be used to significantly increase the accuracy of the SVM.

The final results of this simulation are shown in Figure 10, where the distinct classification regions calculated by a multi-class SVM algorithm (as described in Figure 4) are shown to be bounded by non-linear decision boundaries. The infinite dimensional RBF (Gaussian) kernel utilized allows for the use of a non-linear decision boundary, which allows for a much larger margin in lower dimensional space. The tuning parameters of the SVM are automatically optimized through cross-validation and objective function minimization routines built into the `fitcsvm.m` library of functions in MATLAB. Figure 11 shows the power of more advanced change of basis techniques, like the Fischerfaces algorithm demonstrated in the figure.[2] Compared to Figure 10, the scatter in Figure 11 between different images is much greater, and the scatter between images of the same class is greatly reduced. This means that each region of the eigenvector basis is much more specifically mapped to a class when using the fischerfaces algorithm than when using the rudimentary eigenface algorithm.

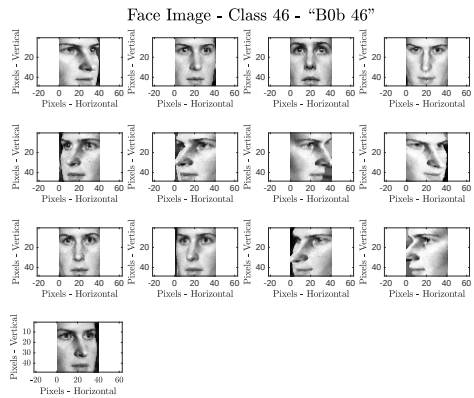


(a) All Examples of Class 7 in Database



(b) All Examples of Class 22 in Database

(c) All Examples of Class 24 in Database



(d) All Examples of Class 46 in Database

Figure 9: The four classes used to develop regional classifications in `MainMultiClassSVM.m`

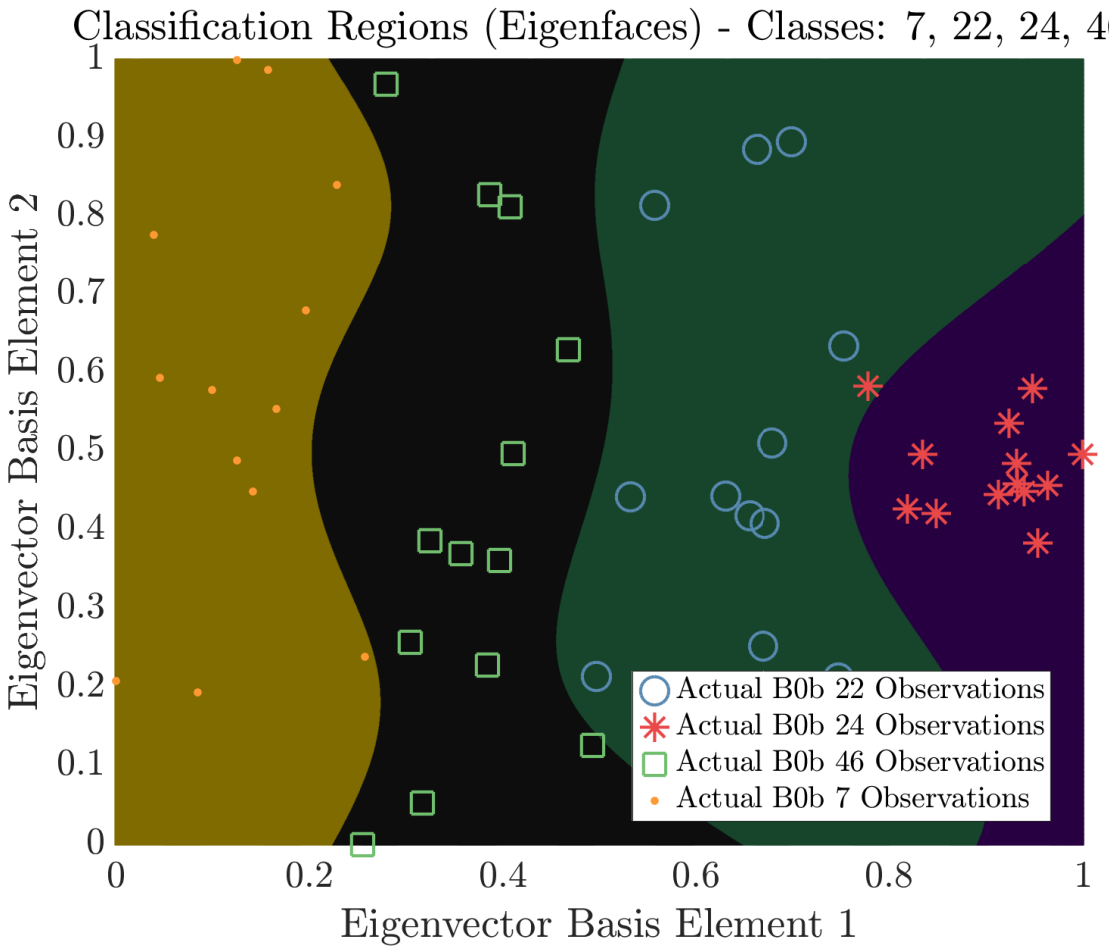


Figure 10: Plot showing the classification regions for each of the four classes. *Each region represents how the SVM would classify a new image projected into the two-dimensional eigenface basis. A new image landing anywhere in the region associated with each class would be classified as a member of the owning class for that region. For example, if a new image were to be projected into the eigenface basis and have coordinates of (0.4, 0.5), the plot above shows that the SVM would classify this new image as a member of the "B0b 46" class (class 46 in the database).*

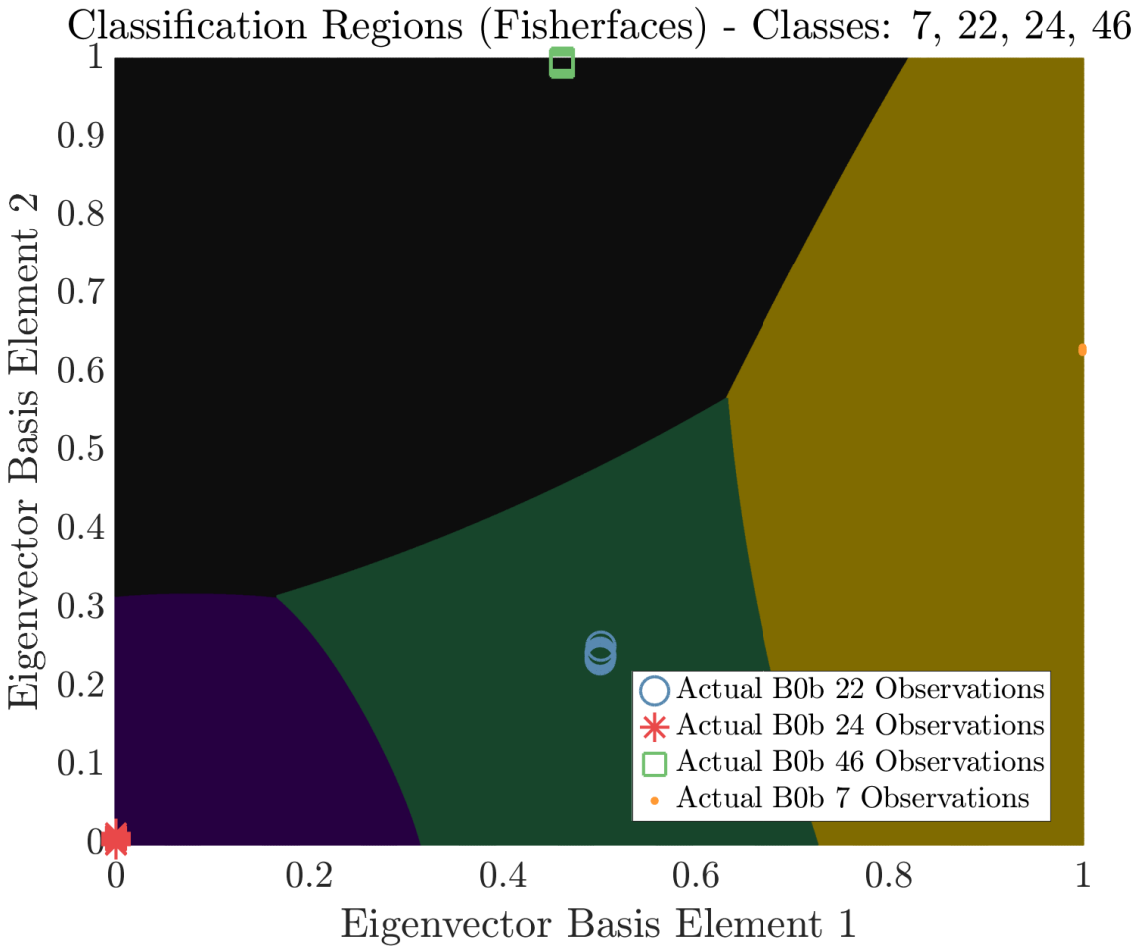


Figure 11: Plot showing the classification regions for each of the four classes using the Fischerfaces algorithm. *Compared to the Eigenfaces change of basis technique, one can see that the inter-class scatter has been drastically reduced while the intra-class scatter has been greatly increased. This much more advanced change of basis algorithm makes classification obviously much easier.*

V. Discussion and Conclusions

We found that the eigenface algorithm that is used for facial recognition ended up defining and creating a feature space with feature vectors representing the distinctive traits of each face built by the corresponding eigenvectors, which allows for each image to be redefined with a lower dimension coordinate vector. However, we also found that support vector machines can do this feature space and feature vector math implicitly using the kernel trick, as long as the data inputted into the SVM is written as distinctive features in any space, the SVM would be able to do the equivalent of finding the coordinates of the support vectors and the optimal hyperplane with corresponding decision boundary without having to actually compute them. This would allow the SVM to classify images accurately without necessarily converting to a feature space first. We also found when using the eigenface basis as our feature space and the weighted coefficient vector as the feature vector, we could use the first 15 eigenfaces out of the total 748 eigenfaces available and still have the SVM accurately calculate the class of the image. However, we did find limitations in the eigenface + SVM algorithm; the SVM would sometimes mis-classify faces that we ourselves could not tell if the face was the same face as the class. For this reason, should we continue our learning in this topic, we would investigate more into the fisherfaces change of basis algorithm that we did implement, or maybe try implementing the one vs all algorithm rather than the one vs one algorithm used for multi-class SVM classification.

References

[1]Trivedi, S. (2013, March 19). Face Recognition using Eigenfaces and Distance Classifiers: A Tutorial. Retrieved April 27, 2017, from http://onionesquereality.wordpress.com/2009/02/11/face-recognition-using-eigenfaces-and-distance-classifiers-a-tutorial/

[2]Belhumeur, P. N., Hespanha, J. P., & Kriegman, D. J. (1996). Eigenfaces vs Fisherfaces: Recognition Using Class Specific Linear Projection. Retrieved April 27, 2017, from http://www.face-rec.org/algorithms/LDA/belhumeur96eigenfaces.pdf

[3]Guo, G., Li, S. Z., & Chan, K. L. (n.d.). Face Recognition by Support Vector Machines. Retrieved April 27, 2017, from https://www.semanticscholar.org/paper/Face-Recognition-by-Support-Vector-Machines-Guo-Li/4ac61814d0f624ebda190b240ede72f0b156ff22

[4]Trivedi, S. (2009, April 17). Face Recognition/Authentication Using Support Vector Machines. Retrieved April 27, 2017, from https://onionesquereality.wordpress.com/2009/04/17/face-recognitionauthentication-using-support-vector-machines/

[5]The Normalized Yale Face Database. (2000, October 4). Retrieved April 28, 2017, from http://vismod.media.mit.edu/vismod/classes/mas622-00/datasets/

[6]Ben-Hur, A., & Weston, J. (2010). A Users Guide to Support Vector Machines. Retrieved April 27, 2017, from http://www.cs.colostate.edu/ asa/pdfs/howto.pdf/

[7]Pentland, A., & Turk, M. (1991). Eigenfaces for Recognition. Retrieved April 27, 2017, from http://www.vision.jhu.edu/teaching/vision08/Handouts/"case_study_pca1.pdf"/

[8]Gibiansky, A. (2013, May 29). Cool Linear Algebra: Singular Value Decomposition. Retrieved May 2, 2017, from http://andrew.gibiansky.com/blog/mathematics/cool-linear-algebra-singular-value-decomposition/

Acknowledgments

We would like to thank Dr. Lyles for all of her help explaining the foundational material for this project. Without her insight at the beginning of this project, it would never have been successful.

Appendix A: MATLAB Code

`classifyNoOverlap.m` - Select Image for Classification, Finds All Matching Images of Same Class, Plots and Saves Plots

```
%%% This script is designed to classify images using the eigenfaces
%%% algorithm to compute a new facial recognition basis for all incoming
%%% images. An image that needs to be classified is projected into this new
%%% basis, with a configurable number of basis vectors, and then classified
%%% with an SVM. The SVM here is bases on the LIBCSVM library availible at
%%% https://www.csie.ntu.edu.tw/~cjlin/libsvm/
%%%
%%%
%%% Requires that the included data file 'yaleData.mat' in the following
%%% relative directory from the working directory of this code:
%%% '../Data/yaleData.mat'
%%%
%%%
%%% This code is based off of some starter code written by Jaime I.
%%% Cervantes at https://github.com/JaimeIvanCervantes/FaceRecognition
%%%
%%%
%%% Author: Nicholas Renninger
%%% Last Updated: May 4, 2017
%%% Date Created: Aprl 17, 2017


% Housekeeping
close all
clear
clc

%% Setup and User Input
MARKERSIZE = 9;
MAX_MULTICLASS_COMPARE_ITER = 1000;
FONTSIZE = 24;
set(0, 'defaulttextinterpreter', 'latex');
saveLocation = '../../../Figures/Face_Matching/';
shouldSaveFigures = true;

% select how many principle eigenvectors  of the eigenface basis to use for
% classfication
NUM_EIG_FEATURES = 15;


disp('Loading database please wait...')

% loading in data set
load '../Data/yaleData.mat'
faceData = yaleData;

disp('Database active.')

% number of classes and number of images per
numClasses = 68;
numImagesPerClass = 13;
```

```
52
53 % get user input
54 faceToMatch = input(['Enter the face # in the database you want to find', ...
55                      ' matches for: ']);
56
57
58
59 %% Eigenfaces Algorithm (SVD -> PCA)
60
61 % Read images in T matrix
62 [nRow, nCol, totalNumImages] = size(faceData);
63
64 % T is a matrix containing the reshaped vectors for each image
65 faceMatrix = reshape(faceData, [nRow*nCol, totalNumImages]);
66
67 % remove selected face from the training data to make sure the SVM is not
68 % classifying an exact match of the thing its trying to classify...
69 % confusing huh. Also remove one other image from each other class
70 % corresponding to the selected image's class index. i.e. if you selected
71 % the 15th image, delete every 2nd image from each class in the image
72 % database
73 indicesToDelete = mod(faceToMatch, numImagesPerClass):numImagesPerClass: ...
74                       numClasses * numImagesPerClass;
75 indicesToDelete = indicesToDelete( indicesToDelete > 0 );
76 % A(:, indicesToDelete) = [];
77 faceMatrix(:, indicesToDelete) = [];
78
79 % update how many images there are, just deleted on image from each class
80 totalNumImages = totalNumImages - numClasses;
81 numImagesPerClass = numImagesPerClass - 1;
82
83 % phi is the mean of the entire set of training images
84 phi = mean(faceMatrix, 2);
85
86 % make a matrix with M colums, with each column being phi to subtract off
87 % the average features of each
88 psi = repmat(phi, 1, totalNumImages);
89
90 % substract mean to get a matrix of the distinguishing features (each row)
91 % of each face (each face is a col vec of A)
92 A = faceMatrix - psi;
93
94 % Reshape the selected face
95 selectedFace = reshape(faceData(:, :, faceToMatch), [nRow*nCol 1]);
96 distinguishingFeatures = selectedFace - phi;
97
98
99 % calculate the SVD matrix C = A'*A, which is the transpose of the
100 % covariance. Use A' * A to save a ton of computation time, as the
101 % eigenvectors of  A' * A are the same as the much larger matrix A * A'
102 C = A'*A;
103
104 % Obtaining eigenvalues and eigenvectors of C = A'*A
105 [eigVecs, eigValMat] = eig(C);
106
```

```
% Obtaning more relevant eigenvalues and eigenvectors
eigVals = diag(eigValMat);

principle_evals = [];
principle_evecs = [];

% perform PCA by ordering the eig vals and vecs by their importance
for i = totalNumImages:-1:numClasses + 1
    principle_evals = [principle_evals, eigVals(i)];
    principle_evecs = [principle_evecs, eigVecs(:,i)];
end

% Obtaining the eigenvectors
U = A * principle_evecs;

% Obtaining PCA weights, multiply each eigenvector of U: u_i by the vector
% containing the distinguishing features of each input image: phi_i
Wpca = U' * A;


%% Classification with SVM

% Obtain the weights of the normalized selected face
selectedFaceWeights = U' * distinguishingFeatures;

% Setting the SVM parameters. Uses an rbf kernel
K = 1e9;
gamma = 1/NUM_EIG_FEATURES;
inputParams = ['-t ' int2str(2) ' -c ' int2str(K) ' -b ' int2str(1) ...
               ' -g ' int2str(gamma)];

% prevArray starts with an array containing each class. Winner array is the
% classes that are selected in the binary tree
prevArr = 1:numClasses;
winnerArr = [];

% This section of the code computes a binary SVM tree, by solving a 2 class
% problem with SVM for every 2 classes (1 and 2, 3 and 4, 5 and 6 etc). The
% that you selected is classified according to each 2-class problem and the
% class selected goes on to compete with the other classes selected.

currIter = 1;
CLASS = NaN;

while currIter < MAX_MULTICLASS_COMPARE_ITER

    winnerArr = [];

    for winRep = 1:2:length(prevArr)

        % Selects the two classes to train the SVM
        if winRep >= length(prevArr)
            i = prevArr(winRep) ;
            j = prevArr(winRep - 1) ;
        else
```

```
        i = prevArr(winRep)   ;
        j = prevArr(winRep + 1)   ;
    end

    % Selects the features of the 2 classes
    feature = [Wpca(1:NUM_EIG_FEATURES, ...
                    numImagesPerClass*i-(numImagesPerClass-1): ...
                    numImagesPerClass*i), ...
               Wpca(1:NUM_EIG_FEATURES, ...
                    numImagesPerClass*j-(numImagesPerClass-1): ...
                    numImagesPerClass*j)]';

    [numObservations, numFeatures] = size(feature);


    % scale each feature vector to a value between 0 and 1
    for k = 1:numObservations
        features_normed(k, :) = (feature(k, :) - min(feature(k, :))) / ...
                                ( max(feature(k, :)) - min(feature(k, :))
                                    );
    end


    % Assigns the labels for each class
    for m1 = 1:numImagesPerClass
       label(m1) = 1;
    end

    for n1 = numImagesPerClass+1:2*numImagesPerClass
       label(n1) = -1;
    end


    % The SVM is trained
    model = svmtrain(label', features_normed, inputParams);

    % The face that the user selected is classified to any of the two
    % classes
    guessLab(1) = 1;
    featuresToLookFor = selectedFaceWeights(1:NUM_EIG_FEATURES)';

    % scale each feature vector to a value between 0 and 1
    featuresToLookFor = (featuresToLookFor - min(featuresToLookFor)) / ...
                        (( max(featuresToLookFor)) - min(featuresToLookFor
                            ));

    predLabel = svmpredict(guessLab', featuresToLookFor, model);

    % plot SVs
    %{
    figure
    hold on
    for i = 1:2
        feature(1:13, i) = (feature(1:13, i)) / ...
                            max(feature(1:13, i));
```

```
215
216             feature(13:26, i) = (feature(13:26, i)) / ...
217                                 ( max(feature(13:26, i)));
218
219         end
220
221         plot(feature(1:13, 1), feature(1:13, 2), 'bo', 'markersize',
               MARKERSIZE)
222         plot(feature(13:26, 1), feature(13:26, 2), 'rx', 'markersize',
               MARKERSIZE)
223         plot(model.SVs(:, 1), model.SVs(:, 2), 'kp', 'markersize', MARKERSIZE)
224         title(sprintf('Class 1: %d, Class 2: %d', i, j))
225         xlim([-6, 6])
226         ylim([-4, 4])
227         hold off
228         %}
229
230         % A winner class is selected
231         if predLabel == 1
232             winnerArr = [winnerArr i];
233         elseif predLabel == -1
234             winnerArr = [winnerArr j];
235         end
236
237         if winnerArr > 1
238             for c1 = 2:length(winnerArr)
239                 if winnerArr(c1) == winnerArr(c1 - 1)
240                     winnerArr(c1) = [];
241                 end
242             end
243         end
244
245     end
246
247     prevArr = winnerArr;
248
249    % if the multi class algorithm has selected only one winner return it
250    if length(winnerArr) < 2
251        % This is the class that was selected
252        CLASS = winnerArr;
253        break
254    end
255
256    currIter = currIter + 1;
257 end
258
259 % check if the multi-class algorithm ever actually selected a face
260 if isnan(CLASS)
261     error('The SVM never returned an absolute classification.')
262 end
263
264 disp('The class that matches your face is:')
265 disp(CLASS)
266
267 %% Plot the selected face
```

```
titleString = sprintf('Selected Face to Match - Image Number %d - Matching Class %d', ...
                        faceToMatch, CLASS);
saveTitle = cat(2, saveLocation, sprintf('%s.pdf', titleString));

figure('name', titleString)

imagesc(reshape(selectedFace, nRow, nCol));
colormap gray;
title(titleString)
xlabel('Pixels - Horizontal')
ylabel('Pixels - Vertical')
set(gca, 'FontSize', round(FONTSIZE * 0.5))
set(gca, 'defaulttextinterpreter', 'latex')
set(gca, 'TickLabelInterpreter', 'latex')
axis equal

% setup and save figure as .pdf
saveMeSomeFigs(shouldSaveFigures, saveTitle)


%% setup plot saving
titleString = sprintf('Faces Matching Class %d', CLASS);
saveTitle = cat(2, saveLocation, sprintf('%s.pdf', titleString));

hFig = figure('name', titleString);
scrz = get(groot, 'ScreenSize');
set(hFig, 'Position', scrz)

numImgPRow = ceil(sqrt(numImagesPerClass));
if numImgPRow^2 == numImagesPerClass
    subPlotX = numImgPRow;
    subPlotY = numImgPRow;
elseif numImgPRow * (numImgPRow - 1) == numImagesPerClass
    subPlotX = numImgPRow;
    subPlotY = numImgPRow - 1;
end

for i = 1:numImagesPerClass

    subTitStr = sprintf('Face number %d', ...
                        winnerArr*(numImagesPerClass + 1) - i + 1);


    subplot(subPlotX, subPlotY, i)
    imagesc(reshape(faceMatrix(:, ...
                               winnerArr*numImagesPerClass - i + 1), ...
                    nRow, nCol));
    colormap gray;

    title(subTitStr)
    xlabel('Pixels - Horizontal')
    ylabel('Pixels - Vertical')
    axis equal
    set(gca, 'FontSize', round(FONTSIZE * 0.5))
```

```
322     set(gca, 'defaulttextinterpreter', 'latex')
323     set(gca, 'TickLabelInterpreter', 'latex')
324 
325 end
326 
327 h = gca;
328 leg = h.Legend;
329 titleStruct = h.Title;
330 set(titleStruct, 'FontWeight', 'bold')
331 set(leg, 'FontSize', round(FONTSIZE * 0.8))
332 mtit(titleString, 'Fontsize', FONTSIZE, 'FontWeight', 'bold');
333 
334 % setup and save figure as .pdf
335 saveMeSomeFigs(shouldSaveFigures, saveTitle)
```

```
1
2  %% Find Multiple Class Boundaries Using Binary SVM
3  % AUTHOR: Nick Renninger, Richard Moon
4  % LAST UPDATED: April 28, 2017
5  % DESCRIPTION: This code implements PCA (eigenfaces), Fisherfaces (linear
6  % discriminant analysis) and an SVM to classify regions of the face space
7  % as characteristic of certain classes.
8
9  % Housekeeping
10 close all
11 clear
12 clc
13
14 %% Setup
15
16 %%%%%%%%%%%%%%%%%%%%%%%% Plot Setup %%%%%%%%%%%%%%%%%%%%%%%%
17 set(0, 'defaulttextinterpreter', 'latex');
18 saveLocation = '../../../Figures/';
19 LINEWIDTH = 2;
20 MARKERSIZE = 18;
21 FONTSIZE = 24;
22 colorVecsOrig = [0.294118 0 0.509804; % indigo
23                  0.1 0.1 0.1; % orange red
24                  1 0.843137 0; % gold
25                  0.180392 0.545098 0.341176; % sea green
26                  0.662745 0.662745 0.662745]; % dark grey
27
28 % de-saturate background colors. saturation must be from 0-1
29 saturation = 0.5;
30 colorVecs = colorVecsOrig * saturation;
31
32
33 markers = {'o','*','s','.','x','d','^','+','v','>','<','p','h'};
34
35 shouldSaveFigures = true;
36
37 % Choose simple, pose variation, or illumination variation dataset
38 numClasses = input(['Please select the number of classes to compare ', ...
39                     '(recommended is 4 classes: 7, 22, 24, 46): \n']);
40
41 for i = 1:numClasses
42
43     % enter the class number (ranges from 1-68 for pose.mat)
44     whichClasses(i) = input('Input one class you want to compare: \n');
45
46 end
47
48 whichClasses = sort(whichClasses);
49 dataset = 2; % automatically choose pose.mat data set
50
51 if dataset == 1
52
53     load '../Data/simple.mat'
54     %Define variables
```

```
55        numClasses = 200;% Number of classes
56        numImagesPerClass = 3;% Number of images per class
57
58    elseif dataset == 2
59
60        load '../Data/yaleData.mat'
61        face = yaleData;
62        numImagesPerClass = 13;% Number of images per class
63    else
64        disp('Input not valid');
65    end
66
67    testNum = input(['Please select whether to use Fischer Faces:', ...
68                     '\n (1) Use Eigenfaces only', ...
69                     '\n (2) Use Fischerfaces and Eigenfaces (more accurate)\n']);
70
71    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
72    % Principal Component Analysis (PCA)
73    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
74
75    disp('Loading please wait...')
76
77    % Read images in T matrix
78    [nRow, nCol, maxNumImgPerClass, maxNumClasses] = size(face);
79
80    maxNumImages = maxNumImgPerClass * maxNumClasses;
81
82    totalNumImages = numClasses * numImagesPerClass;
83
84    % T is a matrix containing the reshaped vectors for each image
85    faceMatrix = reshape(face, [nRow*nCol, maxNumImages]);
86
87    % phi is the mean of the entire set of training images
88    phi = mean(faceMatrix, 2);
89
90    % make a matrix with maxNumImages colums, with each column being phi to
91    % subtract off the average features of each
92    psi = repmat(phi, 1, maxNumImages);
93
94    % substract mean
95    A = faceMatrix - psi;
96
97    % calculate the SVD matrix C = A'*A
98    C = A'*A;
99
100   % Obtaining eigenvalues and eigenvectors of C = A'A
101   [V, D] = eig(C);
102
103   % Obtaning more relevant eigenvalues and eigenvectors
104   eval = diag(D);
105
106   % initialize
107   p_eval = [];
108   p_evec = [];
109
```

```
% sorting eigenvalues from largest to smallest. Takes away zero eigenvalues
for i = maxNumImages:-1:maxNumClasses+1
    p_eval = [p_eval, eval(i)];
    p_evec = [p_evec, V(:,i)];
end

% Obtaining the eigenvectors of A*A'
U = A * p_evec;

% Obtaining PCA weights, multiply each eigenvector of U: u_i by the vector
% containing the distinguishing features of each input image: phi_i
Wpca = U' * A;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Fisher's Linear Discriminant Analysis
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Obtaining Sb and Sw
cMean = zeros(maxNumImages - maxNumClasses, maxNumImages - maxNumClasses);
Sb = zeros(maxNumImages - maxNumClasses, maxNumImages - maxNumClasses);
Sw = zeros(maxNumImages - maxNumClasses, maxNumImages - maxNumClasses);

pcaMean = mean(Wpca, 2);

for i = 1:maxNumClasses
    cMean = mean(Wpca(:, numImagesPerClass*i - (numImagesPerClass-1) : ...
                         numImagesPerClass*i), 2);
    Sb = Sb + (cMean-pcaMean)*(cMean-pcaMean)';
end

Sb = numImagesPerClass*Sb;

for i = 1:maxNumClasses
    cMean = mean(Wpca(:, numImagesPerClass*i - (numImagesPerClass-1) : ...
                         numImagesPerClass*i), 2);
    for j = numImagesPerClass*i - (numImagesPerClass-1):numImagesPerClass*i
        Sw = Sw + (Wpca(:,j) - cMean) * (Wpca(:,j) - cMean)';
    end
end

% Obtaining Fisher eigenvectors and eigenvalues
[Vf, Df] = eig(Sb, Sw);

% Calculating weights
Df = fliplr(diag(Df));
Vf = fliplr(Vf);

% Calculating fisher weights
Wf = Vf' * Wpca;

%% Choose to use Fischer Faces or not

% pick out indicies of chosen class images to compare
```

```
imagesPerClassVec = 1:1:13;
class_idxs = [];

% get the indicies of the selected classes
for i = 1:length(whichClasses)

    class_idxs = [class_idxs, (whichClasses(i) - 1) * ...
                              numImagesPerClass + imagesPerClassVec];
end

% select eigen or fischer faces
if testNum == 1 % use only eigen faces

    % define the features of the selected classes using 2 eigenvectors (1:2)
    features = Wpca(1:2, class_idxs)';
    face_basis = 'Eigenfaces';

else % or use fischer faces as well

    % define the features of the selected classes using 2 eigenvectors (1:2)
    features = Wf(1:2, class_idxs)';
    face_basis = 'Fisherfaces';
end

[~, numFeatures] = size(features);

% scale each feature vector to a value between 0 and 1
for i = 1:numFeatures
    features_normed(:, i) = (features(:, i) - min(features(:, i))) / ...
                            ( max(features(:, i)) - min(features(:, i)) );
end


disp('Loaded. Extracting Features and Projecting to new basis...')
%% Extracting and Labeling Data

% extract two principle eigenvalue weight vectors from the PCA weight
% matrix to form two columns of data. First column corresponds to the
% principle weight of each image in the eigenface space. Second column
% corresponds to the 2nd principle weight of each image in the egenface
% space.


% initialize
classes = cell(1, maxNumClasses);
Y = cell(1, maxNumImages);

% create class names
for i = 1:maxNumClasses
    classes{i} = sprintf('B0b %d', i);
end

% label each image into the class it is
for i = 1:numImagesPerClass:maxNumImages

```

```
    % label numImagesPerClass Images as same class, then move on to the
    % next label
    classes_idx = (i + numImagesPerClass - 1) / numImagesPerClass;
    Y(i:i + numImagesPerClass - 1) = classes(classes_idx);

end

% only pull out the labels relevant to the selected classes
Y = Y(class_idxs)';

%% Scatter Plot of the data.
titleString = sprintf('Scatter Plot - %s - Classes', face_basis);
for i = 1:numClasses
    titleString = sprintf('%s %d', titleString, whichClasses(i));
end

% setup plot saving
saveTitle = cat(2, saveLocation, sprintf('%s.pdf', titleString));


hFig = figure('name', titleString);
scrz = get(groot, 'ScreenSize');
set(hFig, 'Position', scrz)

g = gscatter(features_normed(:,1), features_normed(:,2), Y, ...
         linspecer(numClasses));


h = gca;
leg = h.Legend;
titleStruct = h.Title;
set(titleStruct, 'FontWeight', 'bold')
set(gca, 'FontSize', FONTSIZE)
set(leg, 'FontSize', round(FONTSIZE * 0.7))
set(gca, 'defaulttextinterpreter', 'latex')
set(gca, 'TickLabelInterpreter', 'latex')
for i = 1:numClasses
    g(i).LineWidth = LINEWIDTH;
    g(i).MarkerSize = MARKERSIZE;
    g(i).Marker = markers{i};
end

titleString = sprintf('Scatter Plot (%s) - Classes:', face_basis);
for i = 1:numClasses
    if i == numClasses
        titleString = sprintf('%s %d', titleString, whichClasses(i));
    else
        titleString = sprintf('%s %d,', titleString, whichClasses(i));
    end
end

title(titleString);
xlabel('Eigenvector Basis Element 1');
ylabel('Eigenvector Basis Element 2');
set(leg, 'Location', 'best', 'Interpreter', 'latex')
```

```
xlim([0, 1])
ylim([0, 1])
grid on


% setup and save figure as .pdf
saveMeSomeFigs(shouldSaveFigures, saveTitle)

disp('Done. Plotting projected data and beginning classification...')


%% Build SVM Models

% For each class:
%
% # Create a logical vector (|indx|) indicating whether an observation is
% a member of the class.
% # Train an SVM classifier using the predictor data and |indx|.
% # Store the classifier in a cell of a cell array.
%
% It is good practice to define the class order.
SVMModels = cell(numClasses, 1);
classes = unique(Y);
rng(1); % For reproducibility

for j = 1:numel(classes)
    % Create binary classes for each classifier
    indx = strcmp(Y, classes{j});

    % fit a model to each binary set of data and store the resultant model
    SVMModels{j} = fitcsvm(features_normed, indx, ...
                           'ClassNames', [false true], ...
                           'OptimizeHyperparameters', 'auto',...
                           'HyperparameterOptimizationOptions', ...
                           struct('AcquisitionFunctionName', ...
                           'expected-improvement-plus'), ...
                           'Standardize', true, ...
                           'KernelFunction', 'rbf', ....
                           'BoxConstraint', 1);
end

%%
% |SVMModels| is a numClasses-by-1 cell array, with each cell containing a
% |ClassificationSVM| classifier.  For each cell, the positive class is
% defined by the Y vector.
%%
% Define a fine grid within the plot, and treat the coordinates as new
% observations from the distribution of the training data.  Estimate the
% score of the new observations using each classifier.
mesh_scale_factor = 1e5;
mesh_steps = totalNumImages / mesh_scale_factor;
[x1Grid, x2Grid] = meshgrid(min(features_normed(:,1)):mesh_steps: ...
                            max(features_normed(:,1)), ...
                            min(features_normed(:,2)):mesh_steps: ...
                            max(features_normed(:,2)));
```

```
xGrid = [x1Grid(:), x2Grid(:)];
N = size(xGrid, 1);
Scores = zeros(N, numel(classes));

for j = 1:numel(classes)
    [~, score] = predict(SVMModels{j}, xGrid);
    % Second column contains positive-class scores
    Scores(:, j) = score(:, 2);
end

%%
% Each row of |Scores| contains numClasses scores.  The index of the
% element with the largest score is the index of the class to which the new
% class observation most likely belongs.
%%

% Associate each new observation with the classifier that gives it the
% maximum score.
[~, maxScore] = max(Scores,[],2);

%%
% Color in the regions of the plot based on which class the
% corresponding new observation belongs.

titleString = sprintf('Classification Regions - %s - Classes', face_basis);
for i = 1:numClasses
    titleString = sprintf('%s %d', titleString, whichClasses(i));
end

% setup plot saving
saveTitle = cat(2, saveLocation, sprintf('%s.pdf', titleString));


hFig = figure('name', titleString);
scrz = get(groot, 'ScreenSize');
set(hFig, 'Position', scrz)

g(1:numClasses) = gscatter(xGrid(:,1), xGrid(:,2), maxScore,...
                           colorVecs(1:numClasses, :));
hold on
g(numClasses+1:2*numClasses) = gscatter(features_normed(:,1), ...
                                        features_normed(:,2), ...
                                        Y, linspecer(numClasses));
h = gca;
leg = h.Legend;
titleStruct = h.Title;
set(titleStruct, 'FontWeight', 'bold')
set(gca, 'FontSize', FONTSIZE)
set(leg, 'FontSize', round(FONTSIZE * 0.8))
set(gca, 'defaulttextinterpreter', 'latex')
set(gca, 'TickLabelInterpreter', 'latex')
for i = numClasses+1:2*numClasses
    g(i).LineWidth = LINEWIDTH;
    g(i).MarkerSize = MARKERSIZE;
    g(i).Marker = markers{i - numClasses};
```

```
385 end
386
387 titleString = sprintf('Classification Regions (%s) - Classes:', face_basis);
388 for i = 1:numClasses
389     if i == numClasses
390         titleString = sprintf('%s %d', titleString, whichClasses(i));
391     else
392         titleString = sprintf('%s %d,', titleString, whichClasses(i));
393     end
394 end
395
396 title(titleString);
397 xlabel('Eigenvector Basis Element 1');
398 ylabel('Eigenvector Basis Element 2');
399 set(leg, 'Location', 'best', 'Interpreter', 'latex')
400 grid on
401 xlim([0, 1])
402 ylim([0, 1])
403 hold off
404
405 for i = 1:numClasses
406 %     leg_string{i} = sprintf('%s Region', classes{i});
407     leg_string{i} = sprintf('Actual %s Observations', ...
408                                         classes{i});
409 end
410
411 legend(g(numClasses+1:2*numClasses), leg_string, 'Location', 'best', '
    interpreter', 'latex');
412
413
414 % setup and save figure as .pdf
415 saveMeSomeFigs(shouldSaveFigures, saveTitle)
416
417 disp('Classification Completed. Plotting Results and Exiting.')
418
419 %% Plot the selected classes
420
421 for i = 1:numClasses
422     titleString = sprintf('Face Image - Class %d - B0b %d', ...
423                           whichClasses(i), whichClasses(i));
424
425     % setup plot saving
426     saveTitle = cat(2, saveLocation, sprintf('%s.pdf', titleString));
427
428     hFig = figure('name', titleString);
429     scrz = get(groot, 'ScreenSize');
430     set(hFig, 'Position', scrz)
431
432     for j = 1:numImagesPerClass
433
434         subplot(round(sqrt(numImagesPerClass)), ...
435                 round(sqrt(numImagesPerClass)), j)
436         imagesc(reshape(faceMatrix(:, (whichClasses(i) - 1) *
                maxNumImgPerClass + j), ...
437                         nRow, nCol));
```

```
438         colormap gray;
439
440         xlabel('Pixels - Horizontal')
441         ylabel('Pixels - Vertical')
442         axis equal
443         set(gca, 'FontSize', round(FONTSIZE * 0.5))
444
445     end
446     h = gca;
447     leg = h.Legend;
448     titleStruct = h.Title;
449     set(titleStruct, 'FontWeight', 'bold')
450     set(leg, 'FontSize', round(FONTSIZE * 0.8))
451     set(gca, 'defaulttextinterpreter', 'latex')
452     set(gca, 'TickLabelInterpreter', 'latex')
453     titleString = sprintf('Face Image - Class %d - ``B0b %d"', ...
454                           whichClasses(i), whichClasses(i));
455     mtit(titleString, 'Fontsize', FONTSIZE, 'FontWeight', 'bold');
456
457
458
459     % setup and save figure as .pdf
460     saveMeSomeFigs(shouldSaveFigures, saveTitle)
461 end
```

Helper Functions and Formatting Tools

```
function saveMeSomeFigs(shouldSaveFigures, saveTitle)

    %%% saveMeSomeFigs(shouldSaveFigures, saveTitle)
    %%%
    %%% Takes a boolean toggle (shouldSaveFigures) and a string with the
    %%% save name (saveTitle), including the path, that you want to save
    %%% the current figure as. Example saveTitle looks like the following:
    %%%
    %%%     saveTitle = '../Figures/Velocity vs Time.pdf' - this would save
    %%%     a figure named 'Velocity vs Time.pdf' (as a .pdf) to a folder
    %%%     called 'Figures' up one directoy from the code's working
    %%%     directory.
    %%%
    %%% Uses the gca function to pull the current figure, normalize
    %%% and scale it to the deafult paper size, and save it as a .pdf.
    %%%
    %%%
    %%% Examples function call:
    %%%
    %%% x = linspace(0, 2*pi, 100);
    %%% y = sin(x);
    %%% plot(x, y)
    %%% title('saveMeSomeFigs Test')
    %%%
    %%% saveTitle = 'saveMeSomeFigs Test Plot.pdf';
    %%% shouldSaveFigures = true;
    %%% saveMeSomeFigs(shouldSaveFigures, saveTitle)
    %%%
    %%% Last Modified: 5/3/2017
    %%% Date Created: 2/10/2017
    %%% Author: Nicholas Renninger

    % setup and save figure as .pdf
    if shouldSaveFigures
        curr_fig = gcf;
        set(curr_fig, 'PaperOrientation', 'landscape');
        set(curr_fig, 'PaperUnits', 'normalized');
        set(curr_fig, 'PaperPosition', [0 0 1 1]);
        [fid, errmsg] = fopen(saveTitle, 'w+');

        if fid < 1 % check if file is already open.
            error('Error Opening File in fopen: \n%s', errmsg);
        end

        fclose(fid);
        print(gcf, '-dpdf', saveTitle);
    end

end
```

```
% function lineStyles = linspecer(N)
% This function creates an Nx3 array of N [R B G] colors
% These can be used to plot lots of lines with distinguishable and nice
% looking colors.
```

```
5  %
6  % lineStyles = linspecer(N);  makes N colors for you to use: lineStyles(ii,:)
7  %
8  % colormap(linspecer); set your colormap to have easily distinguishable
9  %                      colors and a pleasing aesthetic
10 %
11 % lineStyles = linspecer(N,'qualitative'); forces the colors to all be
       distinguishable (up to 12)
12 % lineStyles = linspecer(N,'sequential'); forces the colors to vary along a
       spectrum
13 %
14 %% Examples demonstrating the colors.
15 %
16 % LINE COLORS
17 % N=6;
18 % X = linspace(0,pi*3,1000);
19 % Y = bsxfun(@(x,n)sin(x+2*n*pi/N), X.', 1:N);
20 % C = linspecer(N);
21 % axes('NextPlot','replacechildren', 'ColorOrder',C);
22 % plot(X,Y,'linewidth',5)
23 % ylim([-1.1 1.1]);
24 %
25 % SIMPLER LINE COLOR EXAMPLE
26 % N = 6; X = linspace(0,pi*3,1000);
27 % C = linspecer(N)
28 % hold off;
29 % for ii=1:N
30 %     Y = sin(X+2*ii*pi/N);
31 %     plot(X,Y,'color',C(ii,:),'linewidth',3);
32 %     hold on;
33 % end
34 %
35 % COLORMAP EXAMPLE
36 % A = rand(15);
37 % figure; imagesc(A); % default colormap
38 % figure; imagesc(A); colormap(linspecer); % linspecer colormap
39 %
40 %   See also NDHIST, NHIST, PLOT, COLORMAP, 43700-cubehelix-colormaps
41 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
42 % by Jonathan Lansey, March 2009-2013 ? Lansey at gmail.com                  %
43 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
44 %
45 %% credits and where the function came from
46 % The colors are largely taken from:
47 % http://colorbrewer2.org and Cynthia Brewer, Mark Harrower and The
       Pennsylvania State University
48 %
49 %
50 % She studied this from a phsychometric perspective and crafted the colors
51 % beautifully.
52 %
53 % I made choices from the many there to decide the nicest once for plotting
54 % lines in Matlab. I also made a small change to one of the colors I
55 % thought was a bit too bright. In addition some interpolation is going on
56 % for the sequential line styles.
```

```
%
%
%%

function lineStyles=linspecer(N,varargin)

if nargin==0 % return a colormap
    lineStyles = linspecer(128);
    return;
end

if ischar(N)
    lineStyles = linspecer(128,N);
    return;
end

if N<=0 % its empty, nothing else to do here
    lineStyles=[];
    return;
end

% interperet varagin
qualFlag = 0;
colorblindFlag = 0;

if ~isempty(varargin)>0 % you set a parameter?
    switch lower(varargin{1})
        case {'qualitative','qua'}
            if N>12 % go home, you just can't get this.
                warning('qualitiative is not possible for greater than 12 items, please reconsider');
            else
                if N>9
                    warning(['Default may be nicer for ' num2str(N) ' for clearer colors use: whitebg(''black''); ']);
                end
            end
            qualFlag = 1;
        case {'sequential','seq'}
            lineStyles = colorm(N);
            return;
        case {'white','whitefade'}
            lineStyles = whiteFade(N);return;
        case 'red'
            lineStyles = whiteFade(N,'red');return;
        case 'blue'
            lineStyles = whiteFade(N,'blue');return;
        case 'green'
            lineStyles = whiteFade(N,'green');return;
        case {'gray','grey'}
            lineStyles = whiteFade(N,'gray');return;
        case {'colorblind'}
            colorblindFlag = 1;
        otherwise
            warning(['parameter ''' varargin{1} ''' not recognized']);
```

```
110     end
111 end
112 % *.95
113 % predefine some colormaps
114 set3 = colorBrew2mat({[141, 211, 199];[ 255, 237, 111];[ 190, 186, 218];[ 251,
     128, 114];[ 128, 177, 211];[ 253, 180, 98];[ 179, 222, 105];[ 188, 128,
    189];[ 217, 217, 217];[ 204, 235, 197];[ 252, 205, 229];[ 255, 255,
    179]}');
115 set1JL = brighten(colorBrew2mat({[228, 26, 28];[ 55, 126, 184]; [ 77, 175,
    74];[ 255, 127, 0];[ 255, 237, 111]*.85;[ 166, 86, 40];[ 247, 129, 191];[
    153, 153, 153];[ 152, 78, 163]}'));
116 set1 = brighten(colorBrew2mat({[ 55, 126, 184]*.85;[228, 26, 28];[ 77, 175,
    74];[ 255, 127, 0];[ 152, 78, 163]}),.8);
117
118 % colorblindSet = {[215,25,28];[253,174,97];[171,217,233];[44,123,182]};
119 colorblindSet = {[215,25,28];[253,174,97];[171,217,233]*.8;[44,123,182]*.8};
120
121 set3 = dim(set3,.93);
122
123 if colorblindFlag
124     switch N
125         %     sorry about this line folks. kind of legacy here because I used
              to
126         %     use individual 1x3 cells instead of nx3 arrays
127         case 4
128             lineStyles = colorBrew2mat(colorblindSet);
129         otherwise
130             colorblindFlag = false;
131             warning('sorry unsupported colorblind set for this number, using
                 regular types');
132     end
133 end
134 if ~colorblindFlag
135     switch N
136         case 1
137             lineStyles = { [  55, 126, 184]/255};
138         case {2, 3, 4, 5 }
139             lineStyles = set1(1:N);
140         case {6 , 7, 8, 9}
141             lineStyles = set1JL(1:N)';
142         case {10, 11, 12}
143             if qualFlag % force qualitative graphs
144                 lineStyles = set3(1:N)';
145             else % 10 is a good number to start with the sequential ones.
146                 lineStyles = cmap2linspecer(colorm(N));
147             end
148         otherwise % any old case where I need a quick job done.
149             lineStyles = cmap2linspecer(colorm(N));
150     end
151 end
152 lineStyles = cell2mat(lineStyles);
153
154 end
155
156 % extra functions
```

```
157 function varIn = colorBrew2mat(varIn)
158 for ii=1:length(varIn) % just divide by 255
159     varIn{ii}=varIn{ii}/255;
160 end
161 end
162
163 function varIn = brighten(varIn,varargin) % increase the brightness
164
165 if isempty(varargin),
166     frac = .9;
167 else
168     frac = varargin{1};
169 end
170
171 for ii=1:length(varIn)
172     varIn{ii}=varIn{ii}*frac+(1-frac);
173 end
174 end
175
176 function varIn = dim(varIn,f)
177     for ii=1:length(varIn)
178         varIn{ii} = f*varIn{ii};
179     end
180 end
181
182 function vOut = cmap2linspecer(vIn) % changes the format from a double array
    to a cell array with the right format
183 vOut = cell(size(vIn,1),1);
184 for ii=1:size(vIn,1)
185     vOut{ii} = vIn(ii,:);
186 end
187 end
188 %%
189 % colorm returns a colormap which is really good for creating informative
190 % heatmap style figures.
191 % No particular color stands out and it doesn't do too badly for colorblind
    people either.
192 % It works by interpolating the data from the
193 % 'spectral' setting on http://colorbrewer2.org/ set to 11 colors
194 % It is modified a little to make the brightest yellow a little less bright.
195 function cmap = colorm(varargin)
196 n = 100;
197 if ~isempty(varargin)
198     n = varargin{1};
199 end
200
201 if n==1
202     cmap =  [0.2005    0.5593    0.7380];
203     return;
204 end
205 if n==2
206      cmap =  [0.2005    0.5593    0.7380;
207               0.9684    0.4799    0.2723];
208           return;
209 end
```

```
210
211 frac=.95; % Slight modification from colorbrewer here to make the yellows in
        the center just a bit darker
212 cmapp = [158, 1, 66; 213, 62, 79; 244, 109, 67; 253, 174, 97; 254, 224, 139;
        255*frac, 255*frac, 191*frac; 230, 245, 152; 171, 221, 164; 102, 194, 165;
         50, 136, 189; 94, 79, 162];
213 x = linspace(1,n,size(cmapp,1));
214 xi = 1:n;
215 cmap = zeros(n,3);
216 for ii=1:3
217     cmap(:,ii) = pchip(x,cmapp(:,ii),xi);
218 end
219 cmap = flipud(cmap/255);
220 end
221
222 function cmap = whiteFade(varargin)
223 n = 100;
224 if nargin>0
225     n = varargin{1};
226 end
227
228 thisColor = 'blue';
229
230 if nargin>1
231     thisColor = varargin{2};
232 end
233 switch thisColor
234     case {'gray','grey'}
235         cmapp =
                [255,255,255;240,240,240;217,217,217;189,189,189;150,150,150;115,115,115;82,82

236     case 'green'
237         cmapp =
                [247,252,245;229,245,224;199,233,192;161,217,155;116,196,118;65,171,93;35,139,6

238     case 'blue'
239         cmapp =
                [247,251,255;222,235,247;198,219,239;158,202,225;107,174,214;66,146,198;33,113

240     case 'red'
241         cmapp =
                [255,245,240;254,224,210;252,187,161;252,146,114;251,106,74;239,59,44;203,24,29

242     otherwise
243         warning(['sorry your color argument ' thisColor ' was not recognized'
                ]);
244 end
245
246 cmap = interpomap(n,cmapp);
247 end
248
249 % Eat a approximate colormap, then interpolate the rest of it up.
250 function cmap = interpomap(n,cmapp)
251     x = linspace(1,n,size(cmapp,1));
252     xi = 1:n;
```

```
253      cmap = zeros(n,3);
254      for ii=1:3
255          cmap(:,ii) = pchip(x,cmapp(:,ii),xi);
256      end
257      cmap = (cmap/255); % flipud??
258  end
```

```
1  %MTIT            creates a major title in a figure with many axes
2  %
3  %                MTIT
4  %                - creates a major title above all
5  %                  axes in a figure
6  %                - preserves the stack order of
7  %                  the axis handles
8  %
9  %SYNTAX
10 %    ------------------------------------------------------------------------

11 %                P = MTIT(TXT,[OPT1,...,OPTn])
12 %                P = MTIT(FH,TXT,[OPT1,...,OPTn])
13 %
14 %INPUT
15 %    ------------------------------------------------------------------------

16 %     FH :       a valid figure handle            [def: gcf]
17 %    TXT :       title string
18 %
19 % OPT    :       argument
20 % ---------------------------------------------
21 %   xoff :       +/- displacement along X axis
22 %   yoff :       +/- displacement along Y axis
23 %   zoff :       +/- displacement along Z axis
24 %
25 %                title modifier pair(s)
26 % ---------------------------------------------
27 %    TPx :       TVx
28 %                see: get(text) for possible
29 %                     parameters/values
30 %
31 %OUTPUT
32 %    ------------------------------------------------------------------------

33 % par    :       parameter structure
34 %   .pos :       position of surrounding axis
35 %    .oh :       handle of last used axis
36 %    .ah :       handle of invisible surrounding axis
37 %    .th :       handle of main title
38 %
39 %EXAMPLE
40 %    ------------------------------------------------------------------------

41 %        subplot(2,3,[1 3]);              title('PLOT 1');
```

```
%       subplot(2,3,4);                 title('PLOT 2');
%       subplot(2,3,5);                 title('PLOT 3');
%       axes('units','inches',...
%            'color',[0 1 .5],...
%            'position',[.5 .5 2 2]);   title('PLOT 41');
%       axes('units','inches',...
%            'color',[0 .5 1],...
%            'position',[3.5 .5 2 2]);  title('PLOT 42');
%       shg;
%       p=mtit('the BIG title',...
%            'fontsize',14,'color',[1 0 0],...
%            'xoff',-.1,'yoff',.025);
%% refine title using its handle <p.th>
%       set(p.th,'edgecolor',.5*[1 1 1]);

% created:
%       us      24-Feb-2003             / R13
% modified:
%       us      24-Feb-2003             / CSSM
%       us      06-Apr-2003             / TMW
%       us      13-Nov-2009 17:38:17

%
```

```
function        par=mtit(varargin)

                defunit='normalized';
        if      nargout
                par=[];
        end

% check input
        if      nargin < 1
                help(mfilename);
                return;
        end
        if      isempty(get(0,'currentfigure'))
                disp('MTIT> no figure');
                return;
        end

                vl=true(size(varargin));
        if      ischar(varargin{1})
                vl(1)=false;
                figh=gcf;
                txt=varargin{1};
        elseif  any(ishandle(varargin{1}(:)))           &&...
                ischar(varargin{2})
                vl(1:2)=false;
                figh=varargin{1};
                txt=varargin{2};
        else
                error('MTIT> invalid input');
        end
```

```
95                  vin=varargin(vl);
96                  [off,vout]=get_off(vin{:});
97
98  % find surrounding box
99                  ah=findall(figh,'type','axes');
100         if      isempty(ah)
101                 disp('MTIT> no axis');
102                 return;
103         end
104                 oah=ah(1);
105
106                 ou=get(ah,'units');
107                 set(ah,'units',defunit);
108                 ap=get(ah,'position');
109         if      iscell(ap)
110                 ap=cell2mat(get(ah,'position'));
111         end
112                 ap=[    min(ap(:,1)),max(ap(:,1)+ap(:,3)),...
113                         min(ap(:,2)),max(ap(:,2)+ap(:,4))];
114                 ap=[    ap(1),ap(3),...
115                         ap(2)-ap(1),ap(4)-ap(3)];
116
117 % create axis...
118                 xh=axes('position',ap);
119 % ...and title
120                 th=title(txt,vout{:});
121                 tp=get(th,'position');
122                 set(th,'position',tp+off);
123                 set(xh,'visible','off','hittest','on');
124                 set(th,'visible','on');
125
126 % reset original units
127                 ix=find(~strcmpi(ou,defunit));
128         if      ~isempty(ix)
129         for     i=ix(:).'
130                 set(ah(i),'units',ou{i});
131         end
132         end
133
134 % ...and axis' order
135                 uistack(xh,'bottom');
136                 axes(oah);                                      %#ok
137
138         if      nargout
139                 par.pos=ap;
140                 par.oh=oah;
141                 par.ah=xh;
142                 par.th=th;
143         end
144 end
145 %
```

```
146 function        [off,vout]=get_off(varargin)
147
```

```
% search for pairs <.off>/<value>

                off=zeros(1,3);
                io=0;
        for     mode={'xoff','yoff','zoff'};
                ix=strcmpi(varargin,mode);
        if      any(ix)
                io=io+1;
                yx=find(ix);
                ix(yx+1)=1;
                off(1,io)=varargin{yx(end)+1};
                varargin=varargin(xor(ix,1));
        end
        end
                vout=varargin;
end
%
```