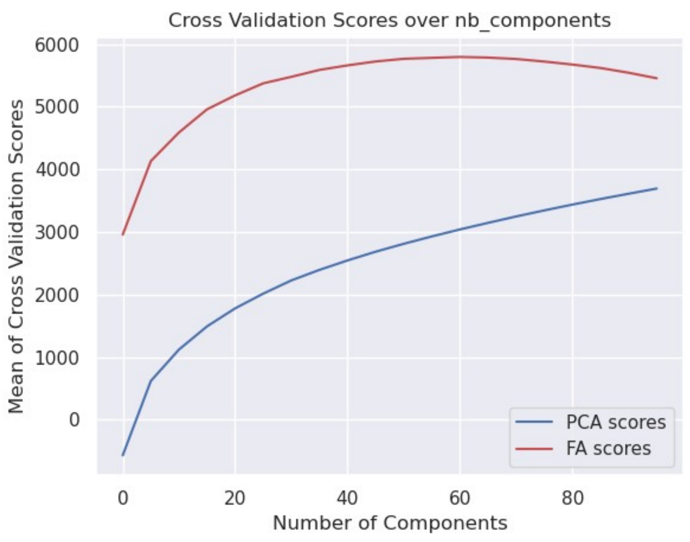
Principal Component Analysis (PCA) and Factor Analysis (FA) are both methods that can are used to reduce the dimensional complexity of data. Both reduction methods are similar in that they require standardization of the predictors, generation of a correlation matrix, and decomposition of that matrix into eigenvectors and their respective eignevalues and then use this information to rank each vector (representing a component) in order of descending eigenvalues. These eignevalues give us a measure of how much effect each respective component has on the variablity of the data. Finally, both methods are also similar in that they require a predictive model to be applied after the dimension reduction in order to actually reach predictions. In the case of this exercise, the predictive model that will be used following dimension reduction is Kmeans clustering (explained in section 2 below).

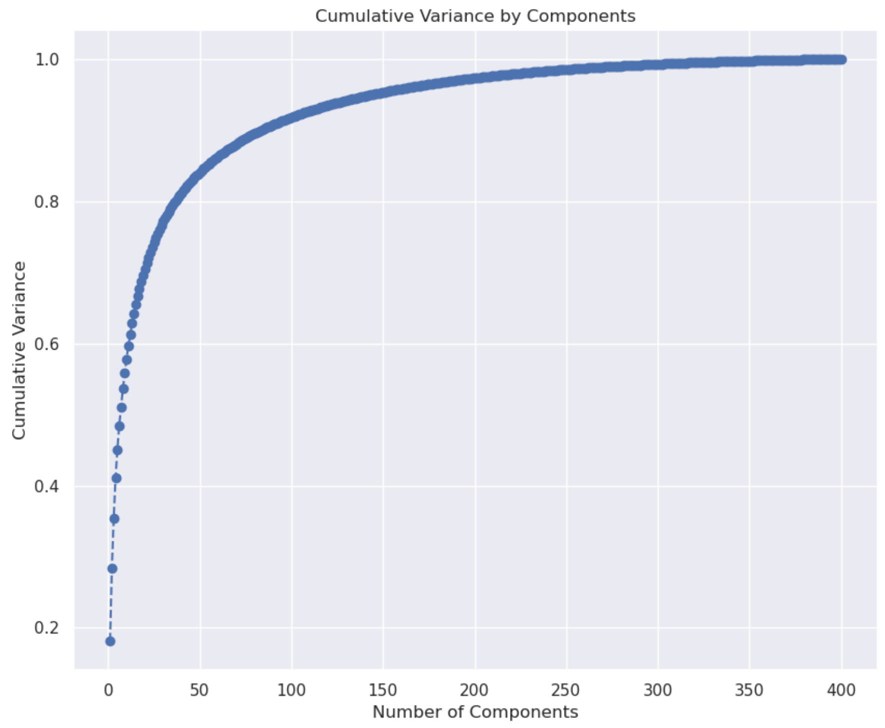
Before we jump into dropping components/reducing dimensions, we must first understand where FA and PCA are different and find a way to evaluate the two methods prior to applying our predictive model fit. To begin with, PCA treats each individual component/feature as fully independent which allows for isolating each feature and measuring its effect on the variance/spread in the data. FA, on the other hand, begins with an almost opposite assumption where it assumes an inherent relationship between several "similar" features and attempts to "clump" several features together as to end with an overall lower number of features (represented as part of bigger pieces). Of course, this difference is mainly felt when using the models in script where it becomes evident fairly quickly that the PCA approach provides far more control for the user especially given the explained variance ratio attribute that is part of sklearn's PCA() function. Instead of simply clumping features together, PCA cleanly presents to the user the cumulative variance available from an increasing number of components where the user is free to analyze the cumulative variance and decide on the number of principal components to use.

However, it is not enough to justify using PCA over FA merely because of ease of use or understanding. We aimed to find a valid metric to measure both methods applied on the same Olivetti Faces dataset and decided that reviewing the cross-validation score would be the best indication. We acknowledge that cross-validation does not work for unsupervised learning but this step is far before the learning actually happens: we are still only reducing dimensions until this point. In other words, we will not be able to use cross-validation to evaluate our predictive model (Kmeans) in the following sections of this exercise.



As can be seen from the figure above, PCA Cross Validation scores remain far lower than FA Cross Validation scores for a wide range of number of components (from 0 to 100). This means that once we calculate the cumulative variance for PCA, regardless of what number of components we reduce to, PCA will always be a better transform to the data than FA. Moreover, this specifically means that PCA will allow the Kmeans predictor to better generalize on never before seen data given that cross validation is just that: how well the model performs on unseen instances in the training phase. Note that the x-axis on this graph does not stretch to all 400 components, this is mainly because we do not expect to need more than 100 principal components for either method.

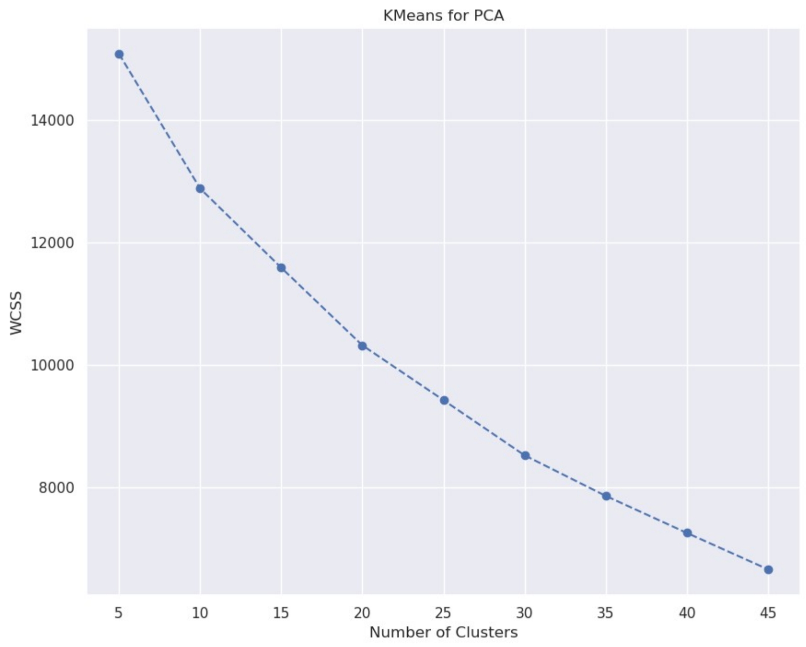
Using PCA, if not more than 100 principal components (hypothesis), then exactly how many components? And why? To answer these questions we refer to the cumulative variance of PCA, which is a measure of the total variance caused by a specified number of components, up to that number of components. To clarify, the cumulative variance of 10 components equals to the sum of the total variance caused by each of components 1 to 10, not that caused by component 10 independently. This may seem redundant but will serve as a helpful reminder when analyzing the figure below



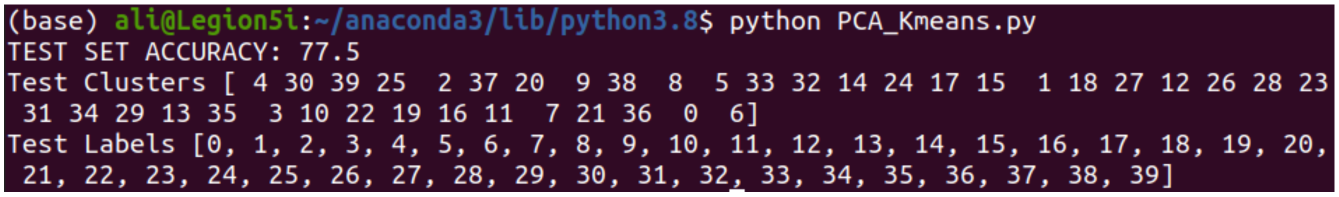
As shown in the figure above, 100 components put us well beyond the minimal variance threshold (generally 80% for the ML community). In other words, we do not need 100 components (and definitely do not need 400) when fitting predictive models to this data because more than 80% of the variance is caused by only 40-50 principal components. Therefore, using any number of components larger than 50 will have an extremely dismal, diminishing return where a lot more compute capacity will be required to deal with the added features with little to no improvement in the model’s ability to differentiate and classify/cluster based on this added loads of data. Therefore, the remainder of this exercise (prediction) will be done using 50 principal components as justified by the cumulative variance graph above.

Now that the data has undergone dimensional reduction, we can start using this newly transformed data to fit our prediction model to. Again, we are faced with a similar situation (as with number of components to reduce to) where we need to identify a certain number of clusters that would work best for this data. The assignment does state to stick with K=40 when performing clustering on the final train/test split which makes sense as the ideal goal would be to assign a cluster to each of the 40 individuals. Yet, there still remains a need to vary K and analyze its effect on the predictive capability.

To analyze K, we require a metric and as mentioned before, cross validation is now not an option seeing as we are no assumed to not know/have any labels. Thus, we revert to the “Within Cluster Sum of Squares” (WCSS) which is a measure of how far each instance is from the center of its cluster. Each cluster has multiple instances in it and each cluster has a cluster center, all these distances are summed and squared for each different K value as shown in the figure blow



In the case of this exercise, we know K=40 is the optimal cluster number given that we know the dataset has 40 different individuals. However, for the sake of learning more about the analysis of a Kmeans model applied to an unsupervised learning problem, we came to find that plotting the WCSS vs. K number of clusters as shown in the figure above is one method to arrive at the “best” K value. The method is specifically called the “Elbow Method” and involves the (fairly arbitrary) search for a “kink” in the graph or the point at which the gradient changes the most (sever change in slope, not sever slope). Using K=40 and clustering the 40 isolated test images results in the following predicted clusters and 77.5% prediction accuracy on the test set as shown in the screenshot below (PCA_Kmeans.py script attached in appendix).



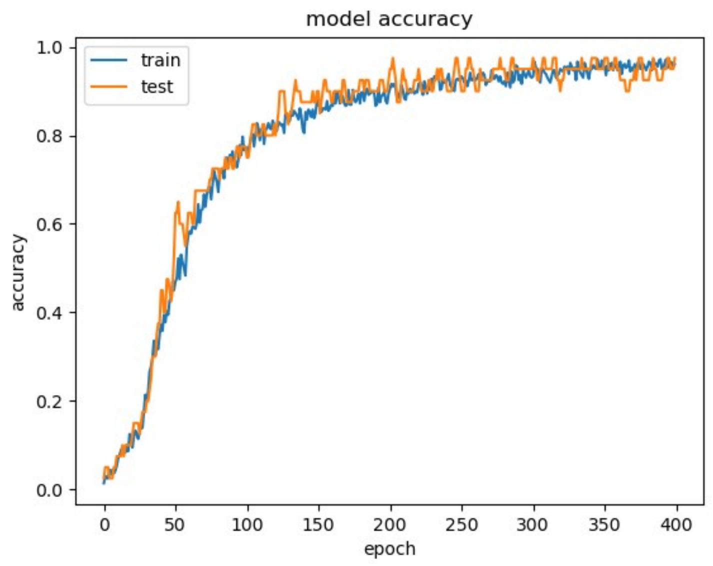
```
(base) ali@Legion5i:~/anaconda3/lib/python3.8$ python PCA_Kmeans.py
TEST SET ACCURACY: 77.5
Test Clusters [ 4 30 39 25  2 37 20  9 38  8  5 33 32 14 24 17 15  1 18 27 12 26 28 23
 31 34 29 13 35  3 10 22 19 16 11  7 21 36  0  6]
Test Labels [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
```

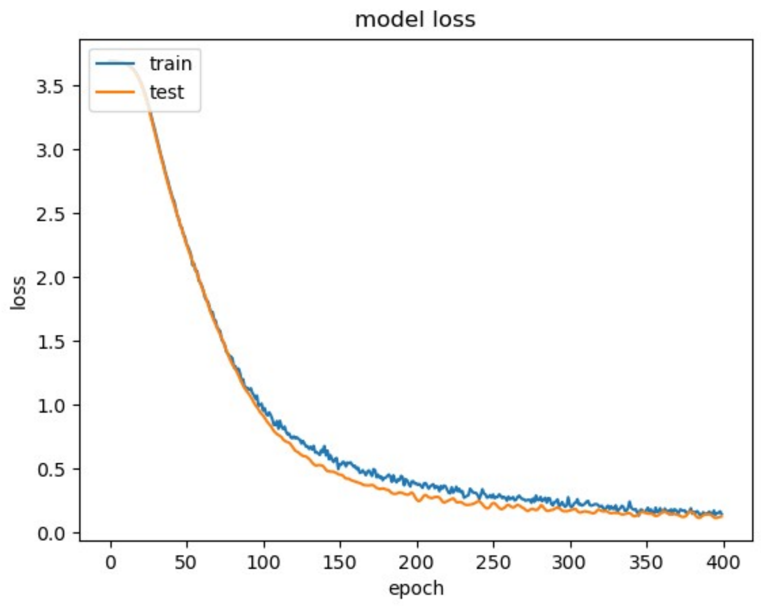
Finally, we deploy one of our convolution neural network (CNN) models from one of our special topics classes. This CNN was originally designed for the MNIST dataset but with all the test/train split work we have already done with the Olivetti datasets, it was extremely smooth importing the Olivetti Faces and working with them directly on the CNN model in Keras. A few major differences to note here:

- We use the same method to split the training and testing data (and respective labels) but we are importing the dataset in a completely different way: the first argument is now False which allows us to import the data as a bunch which comes as 3 sub objects, one of which is a ready 68x68 array of pixels called the image sub object. As we will see in later parts of this class, CNNs are a very special kind of Neural Network that requires a 2D image to work with, not a flattened image representation as we used with the Kmeans approach
- Here, the backbone of the script is almost entirely Keras whereas in the previous section it was almost entirely SkLearn. This brings us to the discussion of sequential blocks. Without getting into too much information out of the scope of this class:
    - Input layer is usually the instance/image itself with each pixel on the 68x68 grid acting as its own neuron firing in the first layer
    - The second layer shown in the script is a convolutional layer with 32 kernels of size (5x5). These 32 Kernels are different and are “spanned” over the image, convolving it and each resulting in its own feature map (32 feature maps corresponding to 32 kernels)
    - The third layer is a maxpooling layer which has two main advantages: it helps reduce overfitting by providing an abstracted form of the previous layer representation and it also reduces computational cost via reduced number of parameters
    - The fourth line is not a layer, it is a dropout coefficient applied to the layer in the previous line. The details of dropout are beyond the scope of this paper but it can be thought of as practicing to bat with a partial blindfold and then removing that blindfold on game day: you will be far better. Dropout acts as a blindfold that drops some of the inputs/neurons from the previous layer and thus forcing the following layer to learn “harder” based on a lacking representation. This is one of the most effective ways to fight overfitting
    - Finally we get to a normal (aka “dense”) layer that is flat and this layer must come before the output layer. The reason for this is that the output layer is usually used to classify something or give a “best guess” out of a list of classes. There is no way for an output layer to return a class if its input is coming directly from a convolutional layer simply because of the fact that only a densely connected, flat layer can provide all the output layer neurons with the inputs required to form the prediction
    - The output layer takes inputs from the dense layer before it, multiplies them by the weights of the connections and then makes a guess
- The interesting part of CNNs (or NNs in general) is that the learning does not stop after the first pass, in fact it just got started. The network will now evaluate its own guess, see how far it is from the goal (cost function), measure how each neuron effects the cost function and adjust the weights and biases accordingly (gradient descent).

For the sake of this experiment, we were able to reach a whopping 97.5% with an extremely shallow CNN. Deeper CNNs involve multiple 2D convolution layers, several dense layers to follow, and a myriad of dropout and maxpooling passes and are known to be far more capable on far more complex datasets (most recent CNNs can reach up to 99.5% accuracy on widely complex datasets such as CIFAR10). We hypothesize that the 97.5% accuracy limit seems to be from the relatively small size of the dataset. Neural Networks are more useful for datasets with 10s of thousands of samples where it can spend more time learning *faster*.

When it comes to neural networks, the fact that they are based on epochs (number of iterations to re-teach itself), the evaluation methods and metrics are extremely clear and streamlined. Apart from the final accuracy number printed, researchers tend to like to see how the model behaves over epochs, or in other words: how the model learns over time. For that, the two figures below show the accuracy and loss (respectively) plots for both the training and the test datasets. These figures aid us in finding out important information about our model that we may otherwise not be able to tell from the simple "97.5%" value such as weather our model is overfitting or underfitting and if we are suffering from too much noise or too a high a learning rate.


model accuracy
train
test
accuracy
epoch


model loss
train
test
loss
epoch