Pattern Recognition Assignment No:2
Analysis & Results
18MCMI09
Krishnan.S

All the outputs for the code are shown here and not in the code file.

1.
In Principal Component Analysis (PCA), we try to reduce the dimensionality of the data set by projecting it on to axes that retain the maximum variances of the data. If we're given a two hundred dimensional data set, we want to reduce it two or three dimensional space so that it's easier to visualize and to do computations with. These axes are the principal components. The first principal component is a vector in the direction in which variance or spread of the data is maximum. The second principal component is in the direction of the the next most spread, and so forth.

These principal components are the eigen vectors of the Scatter Matrix of the data, corresponding to the top 'k' eigen values. If the data set is d-dimensional, it an be reduced to a 'k' dimensional space. The Scatter Matrix is given as

$$S = \sum_{k=1}^{n} (x_k - \mu)(x_k - \mu)^t$$

Where μ is the mean of the data, and,

$$Sv = \lambda v$$

λ is the eigenvalue and v is the corresponding eigen vector.
To implement this, the data matrix was centered by the mean found by using **np.mean()**, and then multiplied with its transpose to get S using **np.matmul()**. Then the **np.linalg.eig()** function was called on the scatter matrix to obtain the eigenvalues and eigenvectors, which is already sorted in the order of decreasing eigen values. The eigen vectors are multiplied with the data vector to get the projection Y.

In Fisher's Linear Discriminant analyis (LDA or FLD), we try to separate data points belonging to two classes by projecting them on to a single axis such that distance between their projected means is maximum and the variance in each class is minimum, so that they're close together. For two classes, there's only one axis. In this case, the axis maximizes the function

$$J(v) = \frac{|\mu_2^{'} - \mu_1^{'}|}{s_1^{'} + s_2^{'}}$$

After some math work where we derive what the means and variances are in the projected space, this leads us to

$$S_W = \sum_{x_i \in Class\,1} (x_i - \mu_1)(x_i - \mu_1)^t + \sum_{x_i \in Class\,2} (x_i - \mu_2)(x_i - \mu_2)^t$$

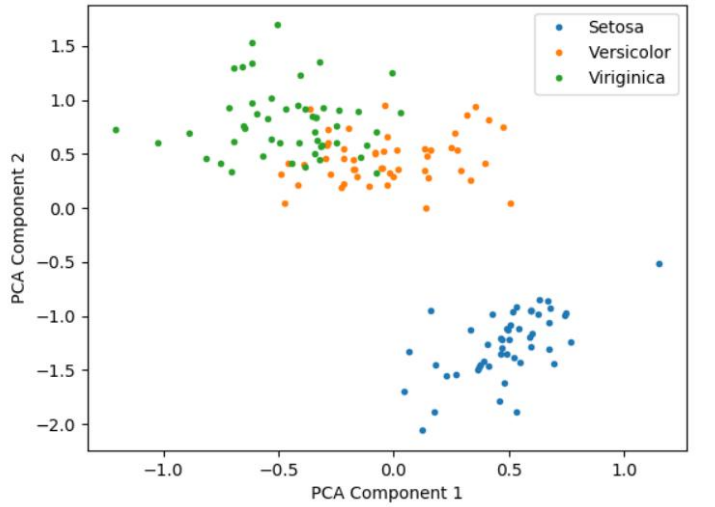
$$S_B = (\mu_1 - \mu_2)(\mu_1 - \mu_2)^t$$

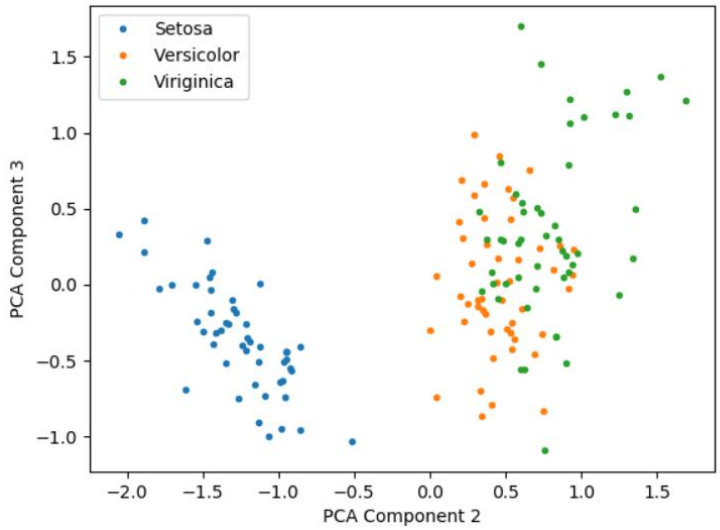
$$S_W^{-1} S_B v = \lambda v$$

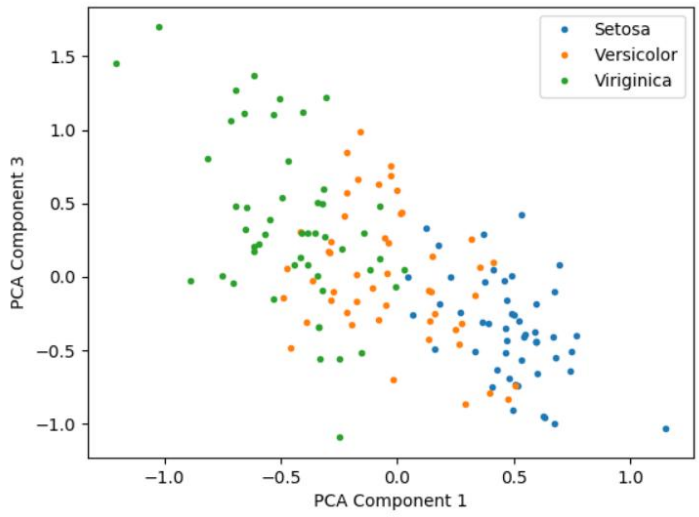
Where S_W is the within class variance and S_B is the between class variance. Again, the eigen vector obtained from this equation is the projection space onto which we project our data points.
To implement this, the means of the two classes were found. Then S_W by computing the product of the mean centered data matrix and its transpose of both classes and then adding them. The inverse was found using **np.linalg.pinv()**. S_B was computer by multiplying the vector difference of the two by it's transpose. The eigenvalues and vectors were found and then multiplied with the input data matrix to get the points in the projection space.

Examples are shown in the next couple of questions.

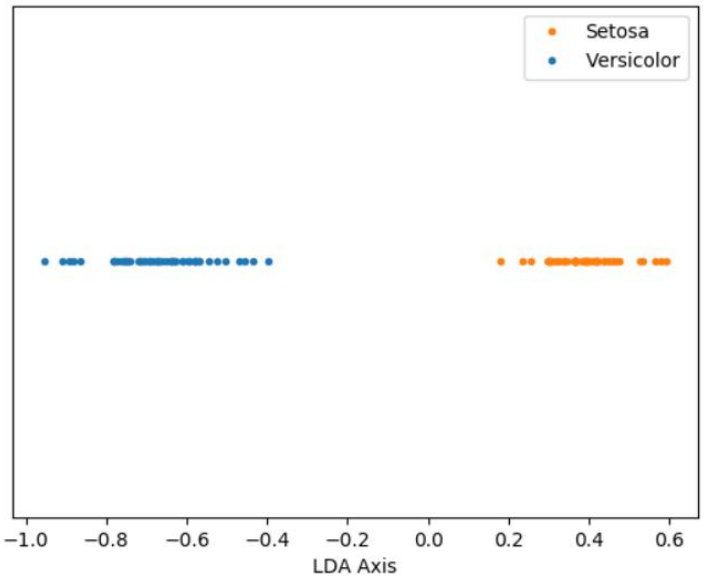
2.
The functions implemented in the previous question were used to find the projections, and matplotlib was used to visualize the points. The plotting function was written such that one can specify what components they want to view. This question wants us to apply PCA and LDA on the IRIS dataset.
The PCA Plots: (PC1 vs PC2, PC2 vs PC3, PC1 vs PC3)

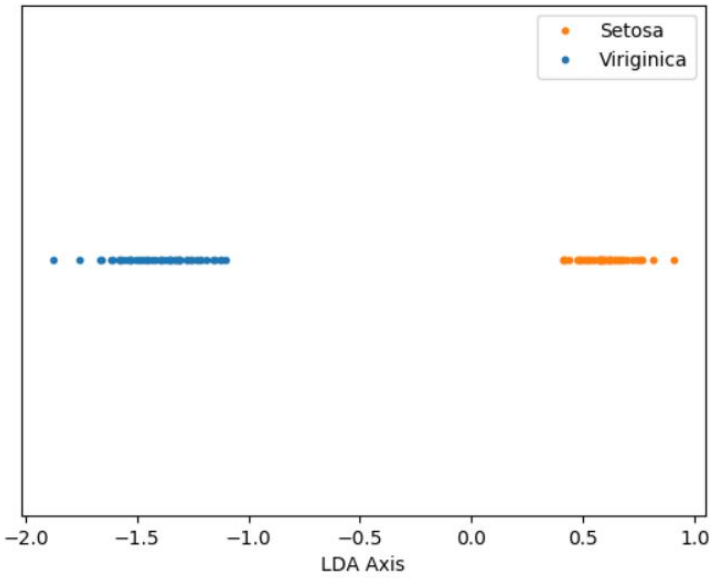


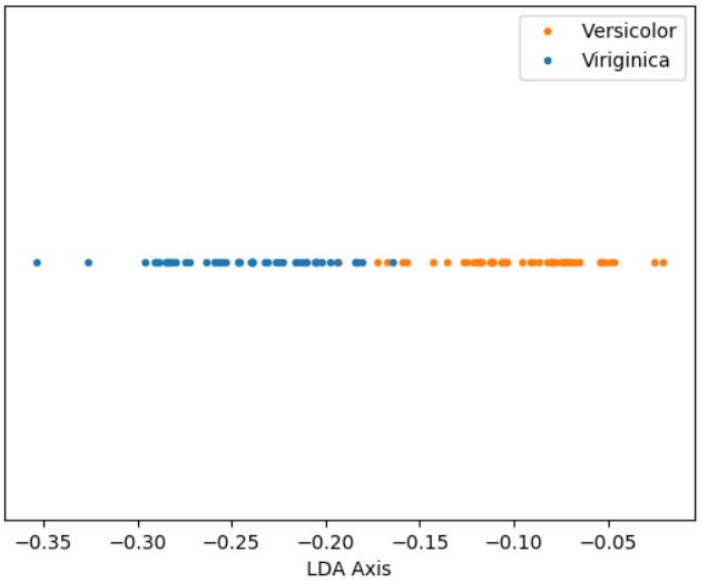




The LDA Plots: (Setosa vs Versicolor, Setosa vs Virginica, Versicolor vs Virginica)







The PCA plots show the entire distribution of the data along the projected axes. The legend shows which point belongs to which classes. The different classes are clustered together, particularly Versicolor and Virginica in the first two plots. PC1, PC2 and PC3 are in order of decreasing variance.

The LDA plots show the projections between each of the two classes. The LDA axis separates the points from the two classes. The first two comparisons have clear distinctions between one one another, but Versicolor and Virginica have some overlap between them, and they are so clustered together in the PCA plots as well. Their natural distributions of features must be close to one another.

The eigenvalues and the eigen vectors obtained were

$$\lambda = \begin{bmatrix} 629.5 & 36.09 & 11.87 & 3.52 \end{bmatrix}$$

$$V = \begin{bmatrix} 0.361 & -0.656 & -0.58 & 0.31 \\ -0.082 & -0.729 & 0.596 & -0.324 \\ 0.856 & 0.175 & 0.072 & -0.479 \\ 0.358 & 0.0747 & 0.0549 & 0.751 \end{bmatrix}$$

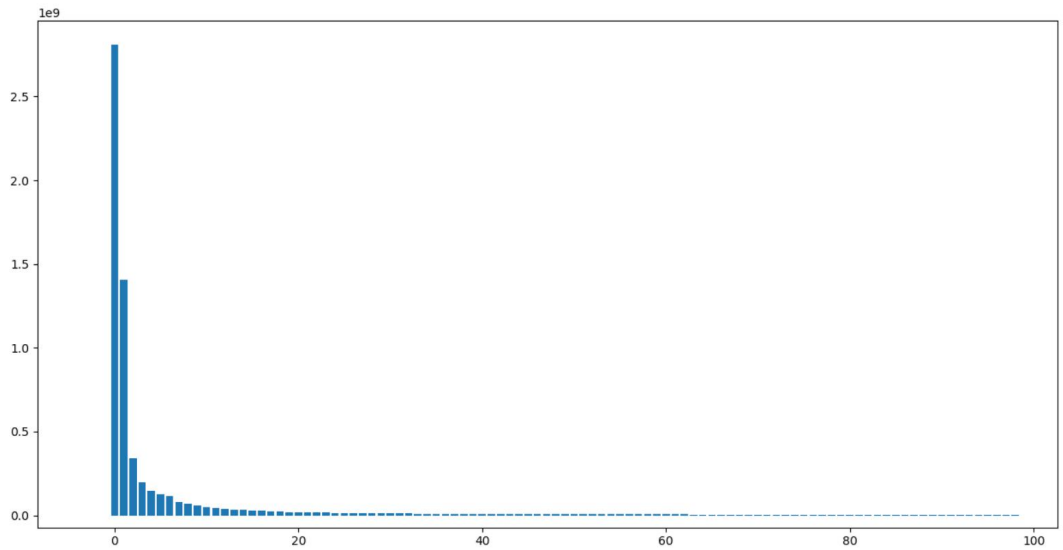
The eigenvectors are the row vectors of V, not the column vectors. The first three rows correspond to the PC1, PC2 and PC3 axes.

3.

The UCI Arcene Dataset contains 900 data vectors having a dimension of 10,000 each, so it's really necessary to reduce the dimension if we are to visualize the data. Performing PCA and LDA took around forty five minutes total with my computer since the size of the data is so large, particularly the number of features.

There are two classes of individuals, ones that are healthy and ones that have cancer. The dataset is put up there to figure out if we can distinguish those that have cancer and those that don't from the given features.

The Scree Plot is given below.



The x axis represents the kth eigen value, which there are 10,000 of, and they y axis is the eigenvalue. There's a sharp drop off right after the first eigen value which is at approximately 2.8×10^9. A lot of the

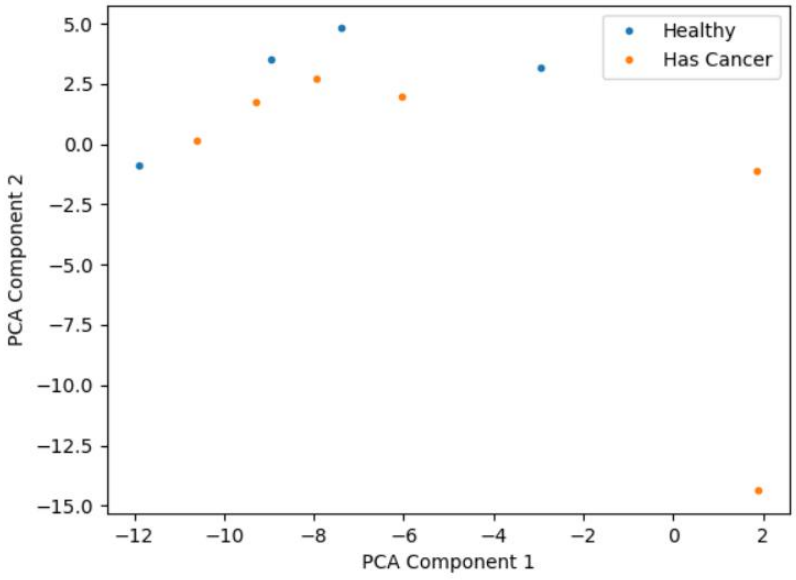
eigen values after the 60ish mark are really close to zero. The first eigenvalue captures about **44.2%** of the variance in the dataset.
To compute how many eigenvalues are needed to get different percentages of the total variance, I trialed different numbers and got the below results.

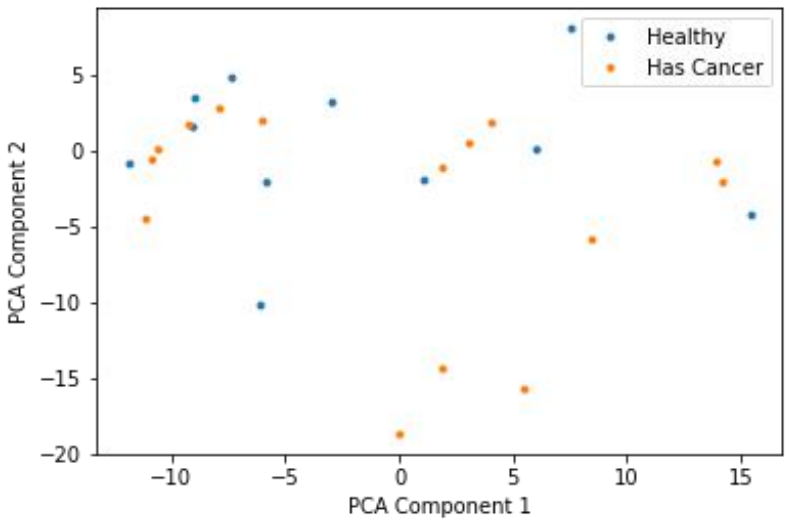
```
In [40]: print(sum(EVals[:10])/sum(EVals)*100)
         print(sum(EVals[:19])/sum(EVals)*100)
         print(sum(EVals[:42])/sum(EVals)*100)
         print(sum(EVals[:84])/sum(EVals)*100)

         (85.41175360374284+0j)
         (90.24912586877858+0j)
         (95.02815776303324+0j)
         (99.06535942634873+0j)
```

The first 10 eigenvalues give us 85% of the variance, the first 19 gives us 90% of the variance, the first 42 gives us 95% of the variance and the first 84 values gives us 99% of the variance. The number of eigenvalues required almost doubles for each five percent increase.

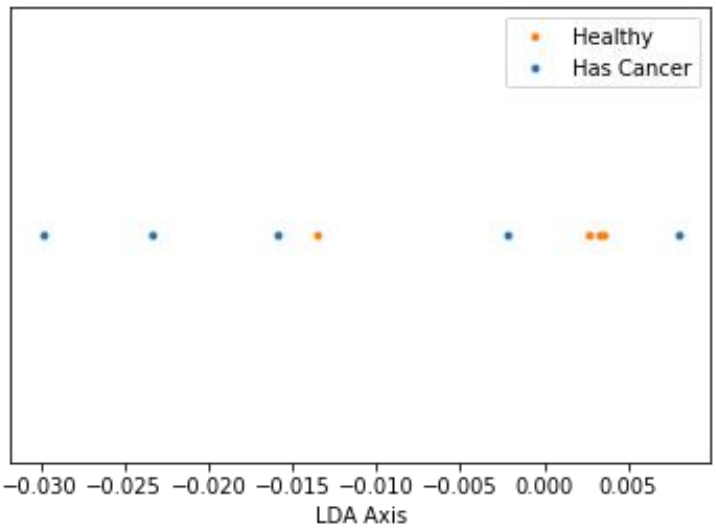
For the PCA and LDA Plots I took 10% and 25% of the total dataset.

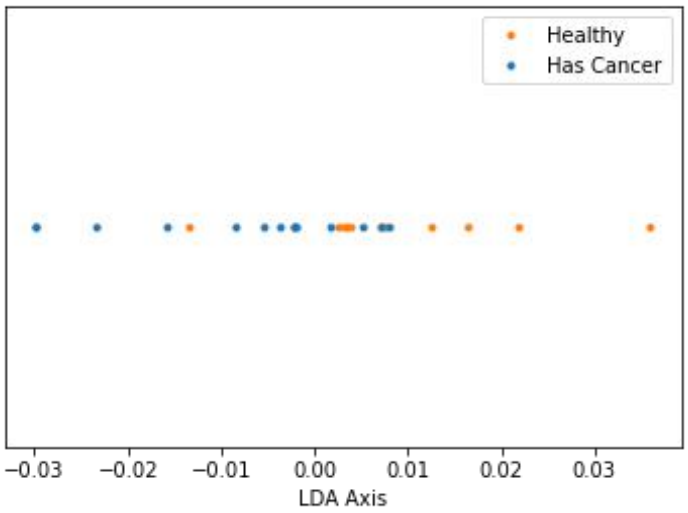
The PCA Plots: (10%, 25%)





The LDA Plots: (10%, 25%)

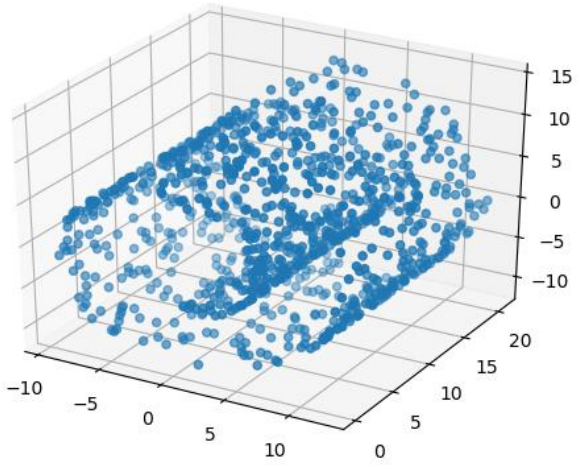




The thing with the PCA components is that we really don't know what the component axes are really describing; they could be a single feature or a combination of features, and in this case it's definitely a large combination of those 10,000 features. If a medical diagonstician were to look at the features produced by the data and that these certain points show a lot of variance, then they might be able to infer what features are actually changing and causing this variance. For example, there's a large variance in those data vectors that belong to healthy people, it's more sparse as compared to the ones that have cancer. The ones that have cancer show the same breadth of variance, but they are more clustered together. That means they must have some features that have values that are close to one another, I.e whatever those cancer cells cause.
The LDA plots show us that there's a large overlap between the two classes, that there's going to be a large tradeoff or classification error. The features of people who don't or do have cancer have a large overlap. We'd probably be able to distinguish better with a non linear technique.

4.
The Swiss Roll Dataset is used to test different dimensionality reduction algorithms. A set of points are generated in 2D and then projected on to the 3D space using a smooth function, which results in that swiss roll like shape. The points lie on a manifold and the variance is along the manifold.

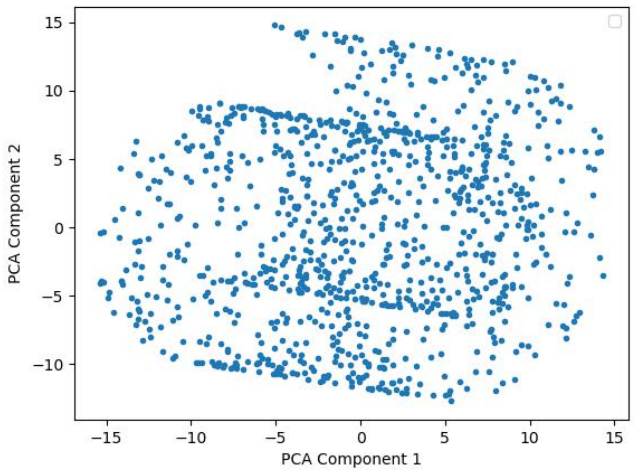
To generate the dataset, I used the function **make_swiss_roll()** from the scikit learn dataset module. I generated about a thousand points. The plot is given below.



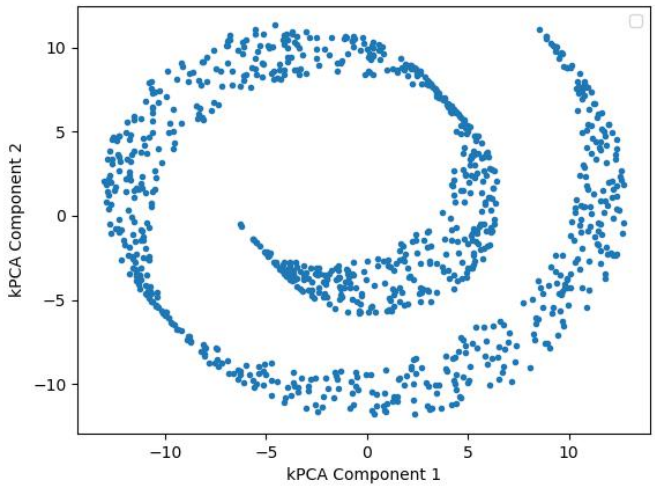
The points get more translucent as we move away from the center of the plot. It curls from the inside to the out.
kPCA and LLE are techniques that are used with non linear datasets such as the swiss roll datasets. Ordinary PCA does not deal with non linear datasets well since it tries to project them on to a linear surface. What kernel PCA does is that it projects the points to a higher dimension and then finds the components in that dimension, so it fits non linear data really well. LLE tries to approximate the non linear dataset by constructing a number of local linear patches that add up to the global. It approximates the geodesic distances across the roll.

I used the functions **KernelPCA()** and **LocallyLinearEmbedding()** from the scikit learn decomposition and manifold modules respectively. Linear and radial basis function kernels were used for kPCA. For LLE, I tried it twice, once with k = 5 nearest neighbours and one with k = 150 neighbours. Plots are shown below for the projections on the first two components of PCA, kPCA and LLE with k=5, k=150.

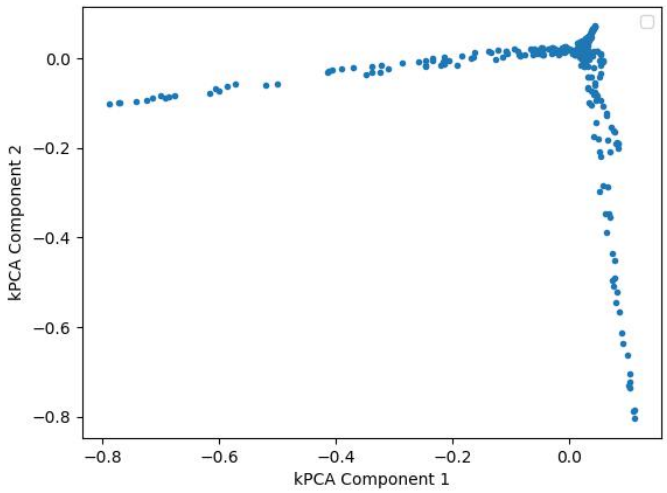
PCA Plot:



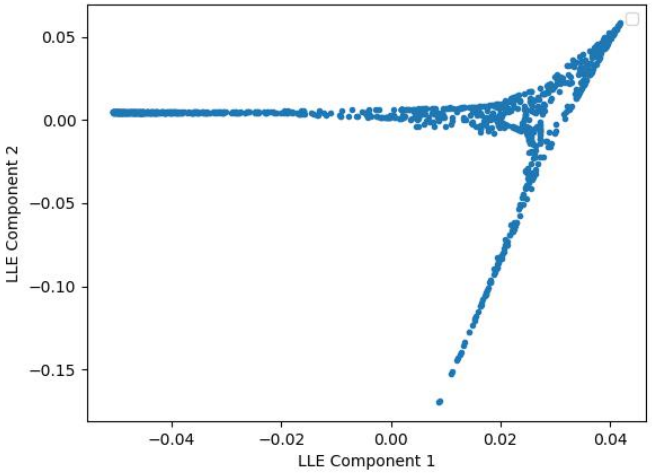
kPCA Plot (Linear Kernel):



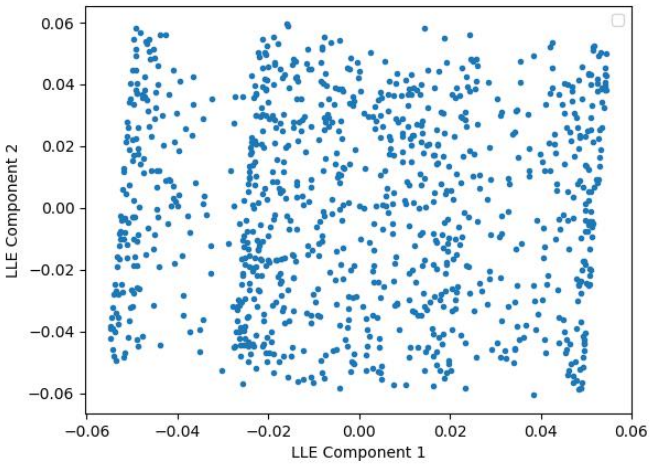
kPCA Plot (RBF Kernel):



LLE Plot (k=5):



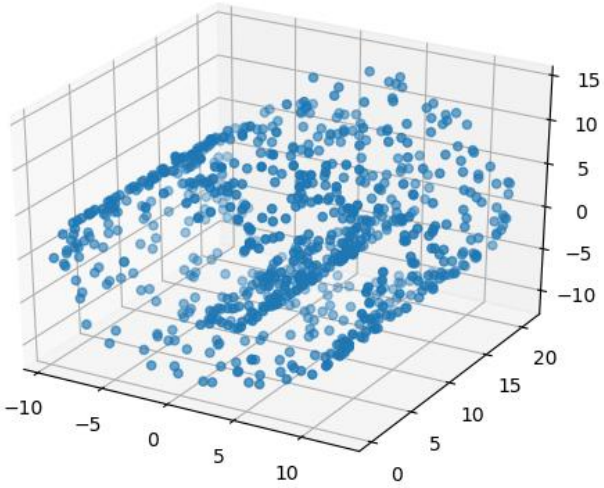
LLE Plot (k=150):



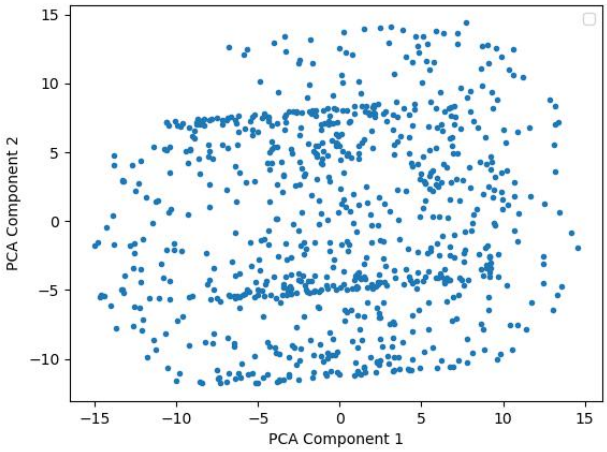
Out of the above five approaches to reducing the swiss roll, the Kernel PCA with the RBF kernel and LLE with k = 5 neighbours works the best. They actually capture the variance of the data as it would have been in the 2D plane on which the points lie, the variance on the manifold when it is flattened. LLE seems to be reproducing the variance more perfectly than kPCA with RBF, the points seem to be aligned more smoothly.
PCA and kPCA with a linear kernel almost reproduce the swiss roll as it is, the variance is that distance in the 3D space from the center of the swiss roll to the curve, wheras it really should be along the manifold. LLE with a large value for k produces the same thing since the linear approximations are much more fewer as compared to a small k. As k approaches the number of points in the space, it becomes a linear function just like PCA.

The next thing to do was to try puncturing the data set, which I did by removing some parts of the swiss roll data set array at a whim. I took out random sets of different sizes and then tried the algorithms again. The punctured roll and the plots are shown below.

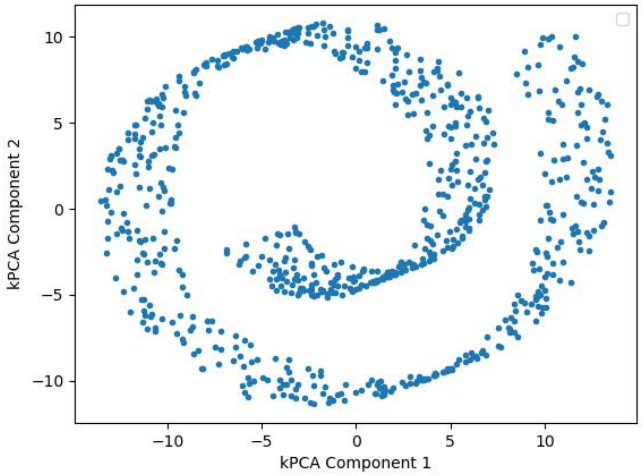
The plots look different because I had to randomize the swiss roll again.



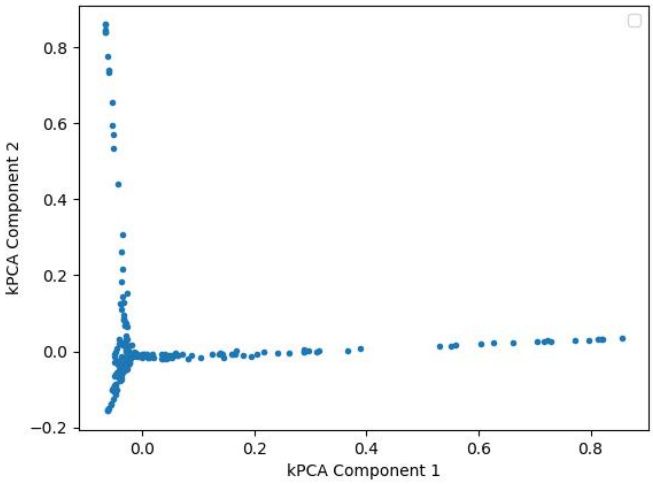
PCA Plot:



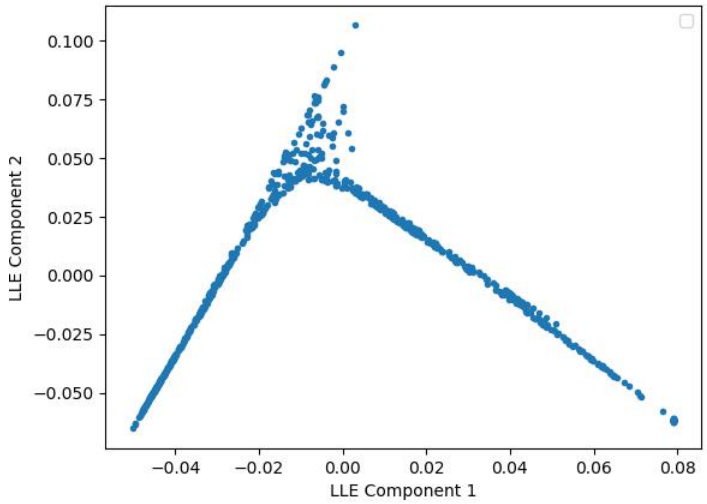
kPCA Plot (Linear Kernel):



kPCA Plot (RBF Kernel):



LLE Plot (k=5):



Even with punctures in the data set, the LLE and kPCA (with RBF) algorithms manage to represent the variation well. I assume that with even bigger punctures they would stop working properly.
