the direction of the greatest variation of the data. This direction is in the least variation of the data. These two clusters would then be nearly perfectly separated from each other because of taking into account of their labels.

References :

1) B. Ghojogh, M. N. Samad, S. A. Mashhadi, T. Kapoor, W. Ali, F. Karray and M. Crowley, "Feature Selection and Feature Extraction in Pattern Analysis: A Literature Review", arXiv:1905.02845v1, 7 May 2019

2) B. Ghojogh,M. Crowley, "Unsupervised and Supervised Principal Component Analysis: Tutorial", arXiv:1906.03148v1, 1 Jun 2019

3) B. Ghojogh, F. Karray and M. Crowley, "Fisher and Kernel Fisher Discriminant Analysis: Tutorial", arXiv:1906.09436v1, 22 Jun 2019

Q 3 Nonlinear Dimensionality Reduction

3.1 Dataset

In [170]:
```
DataB_features =X1
import time
col_name_features = ["Feature "+str(i) for i in range(1,785)]
DataB_features_scaled_df = pd.DataFrame(data = DataB_features, columns = col_name
X_KERNEL_PCA_df = DataB_features
y_KERNEL_PCA_df = DataB_class
```

3.2 Practical Questions

3.2.1 Different Embedding Marks

1) Kernel PCA

In [171]:
```
X_KERNEL_PCA_df = DataB_features
y_KERNEL_PCA_df = DataB_class

from sklearn.decomposition import KernelPCA
start = time.time()
transformer = KernelPCA(n_components=2, kernel='rbf',random_state =42)
KernelPCA_components = transformer.fit_transform(X_KERNEL_PCA_df)
end = time.time()
KernelPCA_components_df = pd.DataFrame(data = KernelPCA_components, columns= ['K
KernelPCA_components_final_df = pd.DataFrame.join(KernelPCA_components_df,y_KERN
KernelPCA_components_final_df.rename_axis('index')
KernelPCA_components_final_df = KernelPCA_components_final_df.assign(new_index =
print("The time taken by KPCA is :",end - start)
KernelPCA_components_final_df.head()
```

```
The time taken by KPCA is : 0.7659530639648438

C:\Users\aksha\Anaconda3\lib\site-packages\sklearn\utils\extmath.py:516: Runtim
eWarning: invalid value encountered in multiply
  v *= signs[:, np.newaxis]
```

Out[171]:

	KernelPCA_component 1	KernelPCA_component 2	class	new_index
0	0.044795	-0.028970	0	0
1	-0.002986	0.001198	0	1
2	-0.013929	-0.009832	0	2
3	-0.004760	0.002459	0	3
4	0.010747	-0.004858	0	4

(2)Isomap

In [172]:
```
X_ISOMAP_df = DataB_features
y_ISOMAP_df = DataB_class
```

In [173]:
```
from sklearn.manifold import Isomap
start = time.time()
transformer = Isomap(n_components=2)
ISOMAP_components = transformer.fit_transform(X_ISOMAP_df)
end = time.time()
ISOMAP_components_df = pd.DataFrame(data = ISOMAP_components, columns= ['ISOMAP_
ISOMAP_components_final_df = pd.DataFrame.join(ISOMAP_components_df,y_ISOMAP_df)
ISOMAP_components_final_df.rename_axis('index')
ISOMAP_components_final_df = ISOMAP_components_final_df.assign(new_index = lambd
print("The time taken by ISOMAP is :",end - start)
ISOMAP_components_final_df.head()
```

```
The time taken by ISOMAP is : 7.3601460456848145
```

Out[173]:

	ISOMAP_component 1	ISOMAP_component 2	class	new_index
0	8706.049952	-344.127405	0	0
1	8858.821154	-333.866129	0	1
2	3954.252089	-926.795027	0	2
3	5791.867059	-643.647584	0	3
4	10700.083472	281.608819	0	4

(3)LLE

In [174]:
```
X_LLE_df = DataB_features
y_LLE_df = DataB_class
```

In [175]:
```
from sklearn.manifold import LocallyLinearEmbedding
start = time.time()
transformer = LocallyLinearEmbedding(n_components=2,random_state =42)
LLE_components = transformer.fit_transform(X_LLE_df)
end = time.time()
LLE_components_df = pd.DataFrame(data = LLE_components, columns= ['LLE_component
LLE_components_final_df = pd.DataFrame.join(LLE_components_df,y_LLE_df)
LLE_components_final_df.rename_axis('index')
LLE_components_final_df = LLE_components_final_df.assign(new_index = lambda z : 
print("The time taken by LLE is :",end - start)
LLE_components_final_df.head()
```

```
The time taken by LLE is : 6.125690937042236
```

Out[175]:

	LLE_component 1	LLE_component 2	class	new_index
0	-0.045258	-0.000862	0	0
1	-0.045060	-0.000827	0	1
2	-0.044573	-0.000925	0	2
3	-0.044700	-0.000871	0	3
4	-0.046829	-0.001124	0	4

(4) Laplacian Eigenmap

In [176]:
```
# (4) Laplacian Eigenmap
X_LEM_df = DataB_features
y_LEM_df = DataB_classX_LEM_df = DataB_features
y_LEM_df = DataB_class
```

In [177]:
```
from sklearn.manifold import SpectralEmbedding
start = time.time()
transformer = SpectralEmbedding(n_components=2,random_state =42)
LEM_components = transformer.fit_transform(X_LEM_df)
end = time.time()
LEM_components_df = pd.DataFrame(data = LEM_components, columns= ['LEM_component
LEM_components_final_df = pd.DataFrame.join(LEM_components_df,y_LEM_df)
LEM_components_final_df.rename_axis('index')
LEM_components_final_df = LEM_components_final_df.assign(new_index = lambda z : 
print("The time taken by LLE is :",end - start)
LEM_components_final_df.head()
```

```
The time taken by LLE is : 6.960597276687622
```

Out[177]:

	LEM_component 1	LEM_component 2	class	new_index
0	0.003748	-0.000115	0	0
1	0.003848	-0.000155	0	1
2	0.002052	-0.000082	0	2
3	0.002793	0.000009	0	3
4	0.004163	-0.000274	0	4

(5) t-SNE

In [178]:
```
X_TSNE_df = DataB_features
y_TSNE_df = DataB_class
```

In [179]:
```
from sklearn.manifold import TSNE
start = time.time()
transformer = TSNE(n_components=2,random_state =42)
TSNE_components = transformer.fit_transform(X_TSNE_df)
end = time.time()
TSNE_components_df = pd.DataFrame(data = TSNE_components, columns= ['TSNE_compon
TSNE_components_final_df = pd.DataFrame.join(TSNE_components_df,y_LEM_df)
TSNE_components_final_df.rename_axis('index')
TSNE_components_final_df = TSNE_components_final_df.assign(new_index = lambda z
print("The time taken by t-sne is :",end - start)
TSNE_components_final_df.head()
```

```
The time taken by t-sne is : 17.500461101531982
```

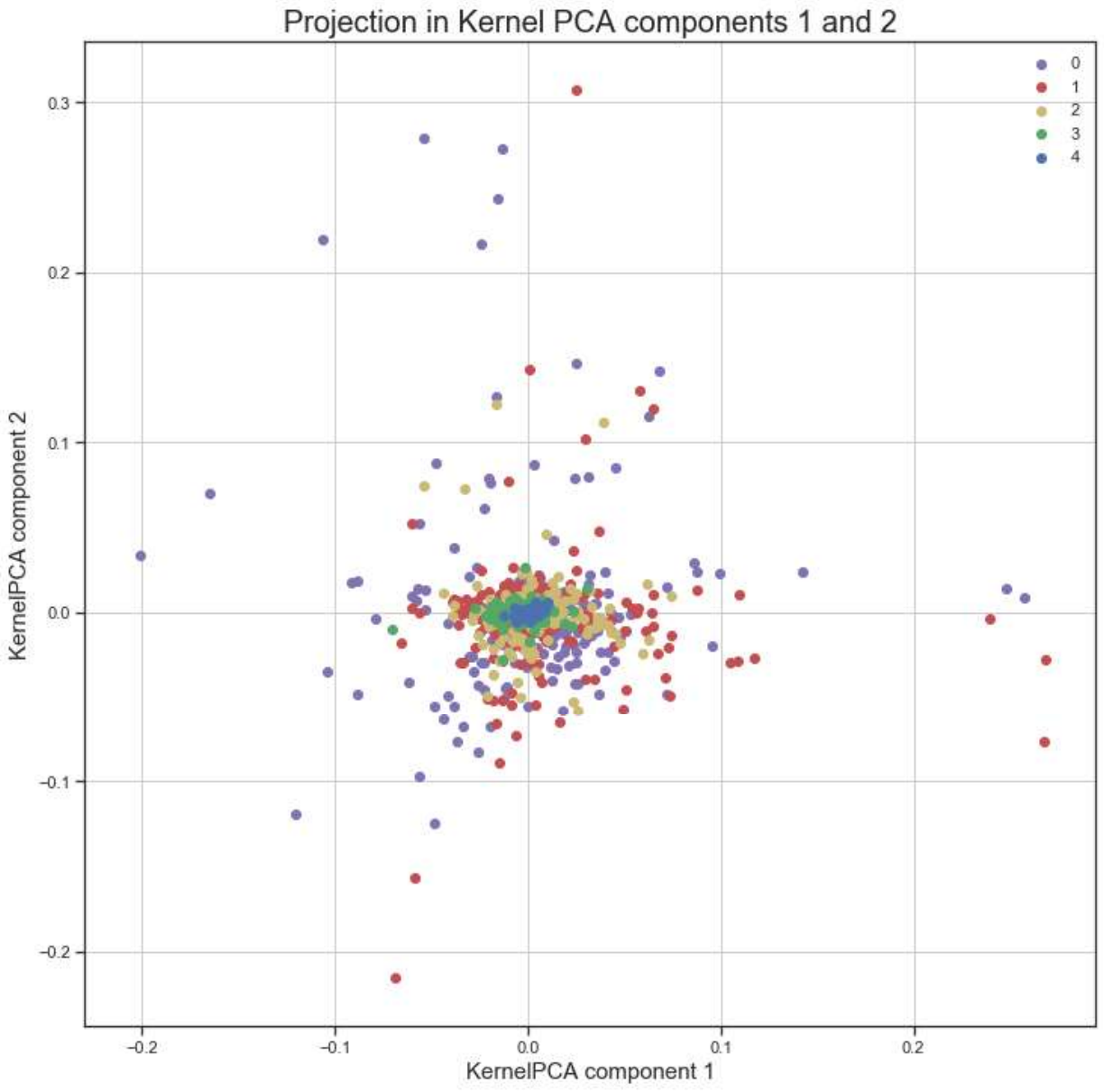
Out[179]:

	TSNE_component 1	TSNE_component 2	class	new_index
0	-53.996166	-7.566008	0	0
1	-54.534958	-8.797090	0	1
2	-43.846584	-0.181216	0	2
3	-44.933132	-3.096598	0	3
4	-60.373814	-7.939509	0	4

3.2 Plot and Compare

(1) Kernel PCA

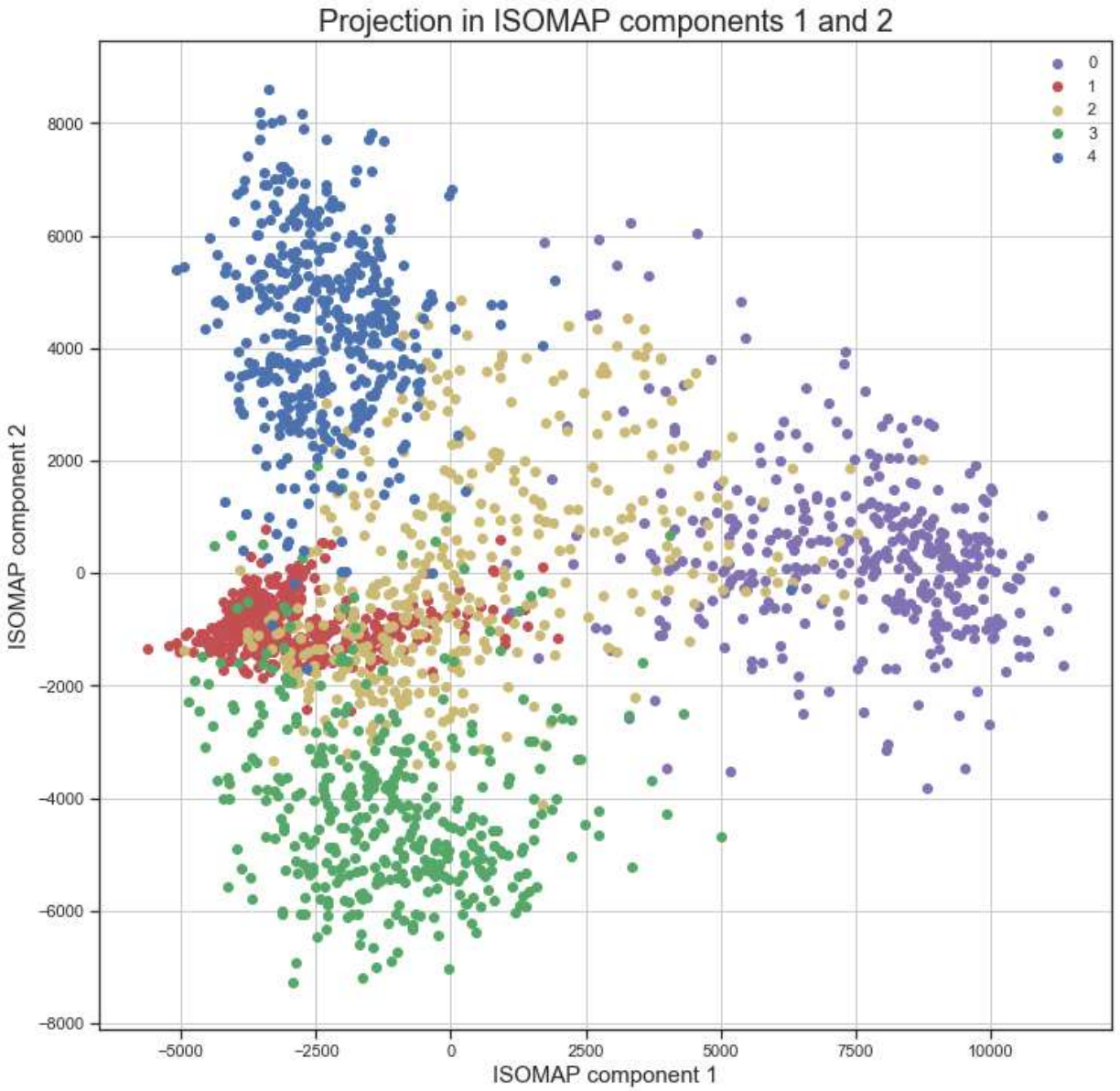
In [180]:
```
figure = plt.figure(figsize = (12,12))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('KernelPCA component 2', fontsize = 15)
ax.set_xlabel('KernelPCA component 1', fontsize = 15)
ax.set_title('Projection in Kernel PCA components 1 and 2', fontsize = 20)
class_colors = [0,1,2,3,4]
colors = ['m','r','y','g','b']
for class_color, color in zip(class_colors,colors):
    indicesTokeep = KernelPCA_components_final_df['class'] == class_color
    ax.scatter(x = KernelPCA_components_final_df.loc[indicesTokeep,'KernelPCA_co
               , c = color)
ax.legend(class_colors)
ax.grid()
```



- The kernel PCA here is using RBF kernel, since the kernel does not know much about the manifold on which the data lies, its performance on MNIST data seems poor.
- Kernel PCA tends to perform good on theoretical analysis, as the kernel to be used is decided beforehand and is independent of properties of data.
- As we can see from the above plot,Kernel PCA is not able to differentiate among different classes.

(2) Isomap

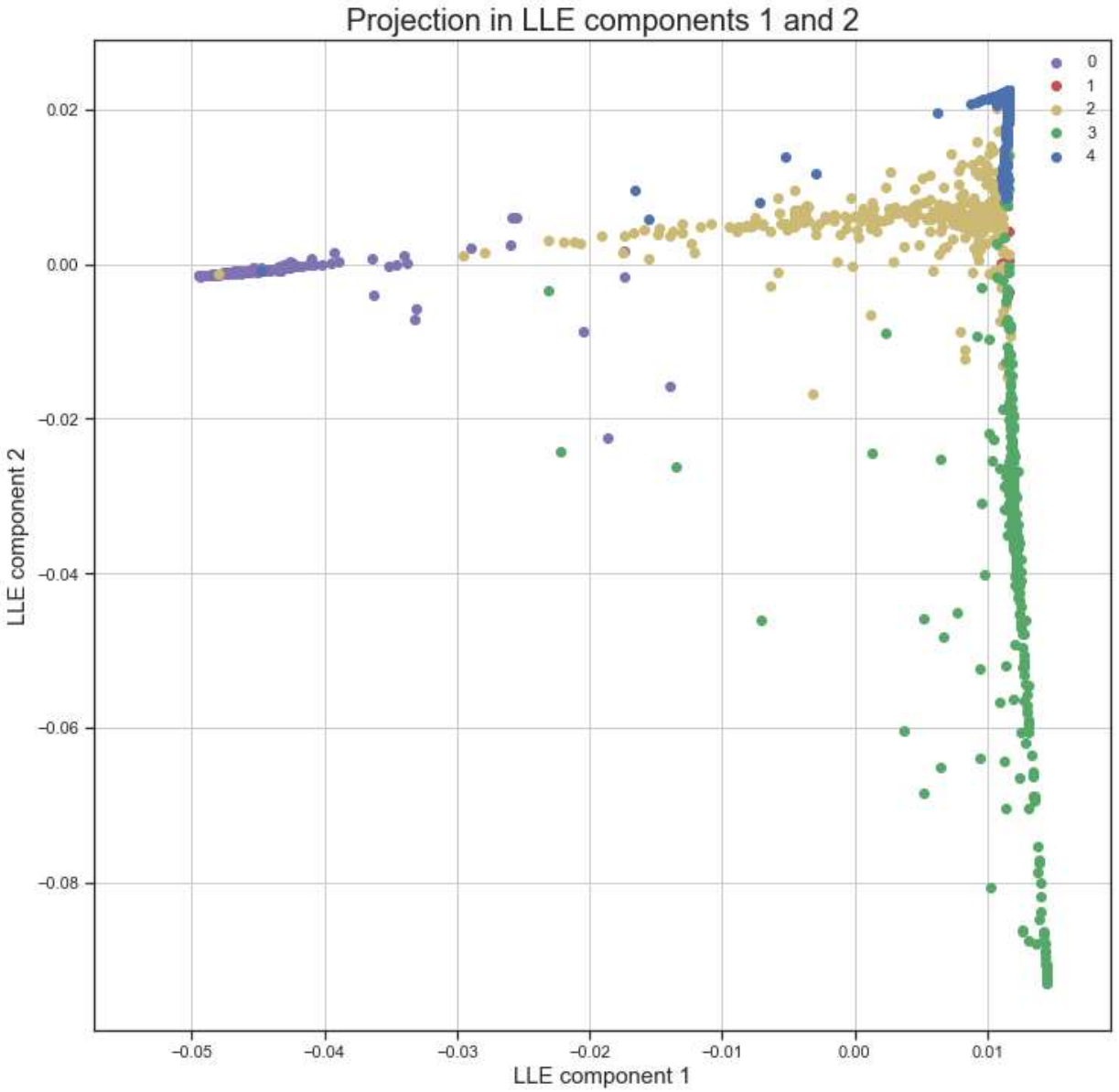
In [181]:
```
figure = plt.figure(figsize = (12,12))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('ISOMAP component 2', fontsize = 15)
ax.set_xlabel('ISOMAP component 1', fontsize = 15)
ax.set_title('Projection in ISOMAP components 1 and 2', fontsize = 20)
class_colors = [0,1,2,3,4]
colors = ['m','r','y','g','b']
for class_color, color in zip(class_colors,colors):
    indicesTokeep = ISOMAP_components_final_df['class'] == class_color
    ax.scatter(x = ISOMAP_components_final_df.loc[indicesTokeep,'ISOMAP_componen
               , c = color)
ax.legend(class_colors)
ax.grid()
```



- Isomap uses a k-nearest neighbor graph(to use geodesic distance, here k =5) and applies MDS on that graph, thus. It tries to use local distances among nearby points and hence is usually able to unfold the manifold well.
- from the above projections we can see it looks like "octopus" shaped, which can be expected from the theoretical background as it tries to connect a KNN graph.
- It can be seen from the above graph, points 2,3 and 1 share some overlap. In addition to this classes, 4,0 and 3 look well separated.

(3) LLE

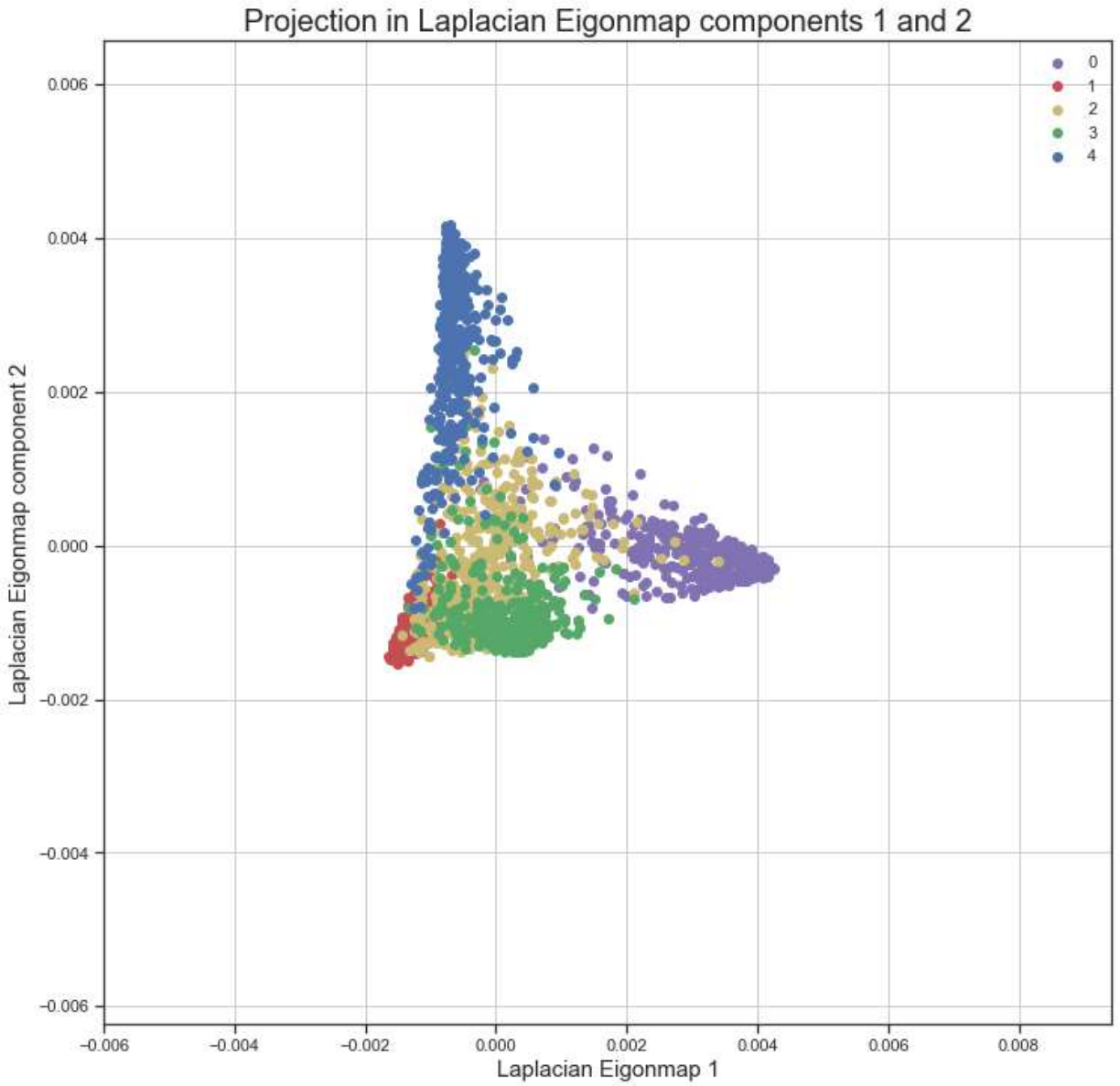
In [182]:
```
figure = plt.figure(figsize = (12,12))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('LLE component 2', fontsize = 15)
ax.set_xlabel('LLE component 1', fontsize = 15)
ax.set_title('Projection in LLE components 1 and 2', fontsize = 20)
class_colors = [0,1,2,3,4]
colors = ['m','r','y','g','b']
for class_color, color in zip(class_colors,colors):
    indicesTokeep = LLE_components_final_df['class'] == class_color
    ax.scatter(x = LLE_components_final_df.loc[indicesTokeep,'LLE_component 1'],y
               , c = color)
ax.legend(class_colors)
ax.grid()
```



- The result of LLE is almost symmetric it can be attributed as optimization of uses the constraint of unit covariance. (It assumes the dense distribution of data points in original space, and does not perform well in presence of outliers in the original data)
- Classes 0,2,3 and 4 looks well separated from each other, whereas there is substantial overlap among class 1 and classes 2,3 and 4.
- LLE is sensitive to outliers and noise. Datasets have a varying density and it is not always possible to have a smooth manifold. In these cases, LLE gives a poor result. (Reference: https://blog.paperspace.com/dimension-reduction-with-lle/ (https://blog.paperspace.com/dimension-reduction-with-lle/))

(4) Laplacian eigonmap

In [183]:
```
figure = plt.figure(figsize = (12,12))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('Laplacian Eigonmap component 2', fontsize = 15)
ax.set_xlabel('Laplacian Eigonmap 1', fontsize = 15)
ax.set_title('Projection in Laplacian Eigonmap components 1 and 2', fontsize = 2
class_colors = [0,1,2,3,4]
colors = ['m','r','y','g','b']
for class_color, color in zip(class_colors,colors):
    indicesTokeep = LEM_components_final_df['class'] == class_color
    ax.scatter(x = LEM_components_final_df.loc[indicesTokeep,'LEM_component 1'],y
               , c = color)
ax.legend(class_colors)
ax.grid()
```

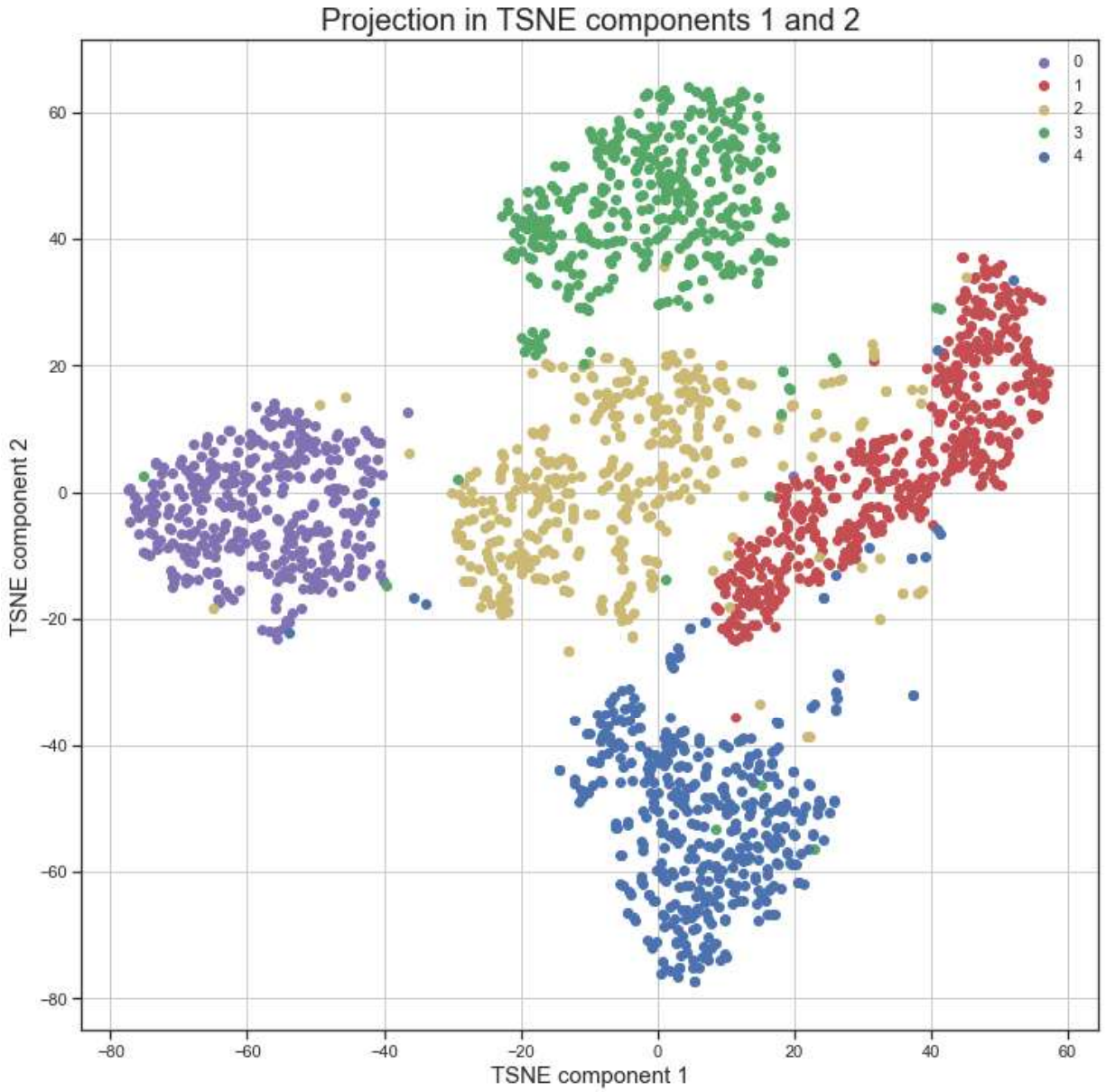


From the above plot and theoretical background on Laplacian eigenmaps, we have the following observations :

- Datapoints sharing similar features look very close to each other in the lower dimensional space, this can be attributed to one-directional cost function used for the Laplacian eigenmaps algorithm, which penalizes cost if we do not maintain the high similarity(in projected space) among the most similar points(from the original space).
- The cluster of data looks good and compact.
- it can be visualized that classes 0,3 and 4 looks separated from the rest of the classes.

(5)T-sne

In [184]:
```
figure = plt.figure(figsize = (12,12))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('TSNE component 2', fontsize = 15)
ax.set_xlabel('TSNE component 1', fontsize = 15)
ax.set_title('Projection in TSNE components 1 and 2', fontsize = 20)
class_colors = [0,1,2,3,4]
colors = ['m','r','y','g','b']
for class_color, color in zip(class_colors,colors):
    indicesTokeep = TSNE_components_final_df['class'] == class_color
    ax.scatter(x = TSNE_components_final_df.loc[indicesTokeep,'TSNE_component 1'
               , c = color)
ax.legend(class_colors)
ax.grid()
```



As expected, t-SNE separates classes well, following are the observations from the above plot and theory :

- Most Classes form spherical/ t-distribution patches, which can be attributed to the use of t distribution in the theoretical framework for t-SNE.
- Classes are separated well, i.e. there is a significant distance among the patches for different classes, thus it can be understood as t-SNE deals with the probabilities and also these are further multiplied(probabilities) by a factor of 4.
- There is very little overlap shared among different categories.
- all classes; 0,1,2,3,4 looks well separated in t-SNE transformed 2-d plot with very few data points outside the class patches.

Comparison among Manifold methods:

1. Most of the manifold learning methods such as isomap, LLE, LE and t-sne are able to differentiate between different classes of MNIST data except RBF kernel-based Kernel PCA. This can be attributed to the fact that the kernel used by KPCA(here in this question) is Radial Basis function and is independent of data features, whereas the rest of techniques used here use a data-driven kernel and hence are comparatively more accurate for visualizing different classes present in the data.
2. From time calculation performed above, it can be seen that KPCA(approx 0.8 seconds) takes almost one-tenth of time compared to the time taken by isomap (approx 7 seconds), LLE (approx 6 seconds), LE (approx 7 seconds). Furthermore, t-SNE(approx 18 seconds) takes almost as much as thrice time compared to isomap, LLE and LE. (Time taken by ISOMAP and LLE depends on the number of neighbors, here these algorithms are performed using k = 5)
3. t-SNE vs KPCA: A) Time taken by t-SNE(approx 18 seconds) is substantially more than time taken by KPCA (approx 0.8 seconds), as t-SNE based on iteratively placing points together on lower dimension space(by converting pairwise distances to probabilities) and does not have a convex cost function, Whereas KPCA has an optimized solution (based on the use of Kernel). B) t-SNE is able to differentiate among the different classes of data well as can be seen from

the above plots. On the other hand, KPCA does not seem to be able to distinguish the classes and the projected data looks scattered for this dataset. C) t-SNE is able to maintain the distance between various class patches (as t-SNE uses T distribution to reduce overlap), whereas the classes in KPCA projected space to exhibit a lot of overlap.

D. Trade-off while deciding the best methods :

Time: Time taken by these algorithms is diffrenent if computational power is limited, a method which takes less time to execute would be preferred. Therefore in such a scenario, Isomap, LE, and LLE would be better choice.
Visualization / seprability: The best visualization in terms of separability is provided by t-SNE and hence, in situations where it is more important to visualize irrespective of the computational time taken by the algorithm, t-SNE would be the most preferred method.

In [ ]: