Ch 3. Dimension Reduction V3

- 03장 프로그램 수정 및 설명.
- 전체 자료가 너무 많은 경우 일부를 추출하여 적합함. t-SNE, KernelPCA 등
  - 교재: 첫 10000개 사용
  - 권장: stratified sampling으로 각 숫자별 1000개씩 총 10000개를 TR에서 추출하여 사용
- CV는 강의에서 제외

In [1]:
```
import time
start_time = time.time()
```

In [2]:
```
# 라이브러리 불러오기
'''메인 라이브러리'''
import numpy as np
import pandas as pd
import os, time, pickle, gzip

'''시각화 관련 라이브러리'''
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()
import matplotlib as mpl

'''데이터 준비 관련 라이브러리'''
from sklearn import preprocessing as pp

%matplotlib inline
```

자료읽기

- 매번 읽지 않도록 csv파일로 저장. 각 행의 값으로 시각화 가능함

```
# 데이터 셋 로드
# 모든 X는 0~ 0.99609375 범위로 정규화 되어 있음. 확인:
np.max(X_train)
# y는 {0,..,9}인 십진 타겟
# 주의사항
# getcwd(): 현재 열린 파일의 위치이므로 D:/GHUB/HUNSLRN
# current_path = os.getcwd()
#  current_path = 'D:/GHUB/HUNSLRN'
# file = os.path.sep.join(['', 'datasets', 'mnist_data',
'mnist.pkl.gz']) # => /datasets/mnist_data/mnist.pkl.gz
# f = gzip.open(current_path+file, 'rb')
filename = 'D:/GHUB/DATASETS/mnist_data/mnist.pkl.gz'

f = gzip.open(filename, 'rb')
train_set, validation_set, test_set = pickle.load(f,
encoding='latin1')
f.close()

TRX, TRy = train_set[0], train_set[1]
```

```
VLX, VLy = validation_set[0], validation_set[1]
TSX, TSy = test_set[0], test_set[1]
```

In [3]:
```
# csv의 첫 행에 변수이름 포함함
TR0X = pd.read_csv('D:/GHUB/DATASETS/mnist_data/mnistTRX.csv')
VLX = pd.read_csv('D:/GHUB/DATASETS/mnist_data/mnistVLX.csv')
TSX = pd.read_csv('D:/GHUB/DATASETS/mnist_data/mnistTSX.csv')

# y는 주로 1-d np.array나 Series로 처리하는 것이 좋으므로, DataFrame으로 읽고 Series
TR0y = pd.read_csv('D:/GHUB/DATASETS/mnist_data/mnistTRy.csv')['y']
VLy = pd.read_csv('D:/GHUB/DATASETS/mnist_data/mnistVLy.csv')['y']
TSy = pd.read_csv('D:/GHUB/DATASETS/mnist_data/mnistTSy.csv')['y']


# 인덱스를 지정 (안해도 무방하지만 표본추출에 의한 혼란을 방지하기 위해 확보)
iTR0 = range(0,len(TR0X))
iVL = range(len(TR0X), len(TR0X)+len(VLX))
iTS = range(len(TR0X)+len(VLX), len(TR0X)+len(VLX)+len(TSX))

# 데이터 셋 구조 확인
nX = [TR0X.shape, VLX.shape, TSX.shape]
ny = [TR0y.shape, VLy.shape, TSy.shape]
pd.DataFrame(nX, index=['TR0X','VLX','TSX'], columns=['n','p'])
```

Out[3]:

	n	p
TR0X	50000	784
VLX	10000	784
TSX	10000	784

In [4]:
```
pd.DataFrame(ny, index=['TR0y','VLy','TSy'], columns=['n'])
```

Out[4]:

	n
TR0y	50000
VLy	10000
TSy	10000

In [5]:
```
# TR 요약. 표준편차=0 인 피쳐 많음
TR0X.describe()
```

Out[5]:

	x01	x02	x03	x04	x05	x06	x07	x08	x09	x10	.
count	50000.0	50000.0	50000.0	50000.0	50000.0	50000.0	50000.0	50000.0	50000.0	50000.0	.
mean	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.
std	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.
min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.
25%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.
50%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.
75%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.
max	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.

8 rows × 784 columns

In [6]:
```
tmp = TROX.describe()
tmp
```

Out[6]:

	x01	x02	x03	x04	x05	x06	x07	x08	x09	x10	.
count	50000.0	50000.0	50000.0	50000.0	50000.0	50000.0	50000.0	50000.0	50000.0	50000.0	.
mean	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.
std	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.
min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.
25%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.
50%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.
75%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.
max	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	.

8 rows × 784 columns

In [7]:
```
(tmp.loc['std',:]<=0.0).sum()  # TR기준 p=784중 67개의 표준편차가 0임
```

Out[7]:
```
67
```

In [8]:
```
tmp.columns[tmp.loc['std']<=0.0]
```

Out[8]:
```
Index(['x01', 'x02', 'x03', 'x04', 'x05', 'x06', 'x07', 'x08', 'x09', 'x10',
       'x11', 'x12', 'x17', 'x18', 'x19', 'x20', 'x21', 'x22', 'x23', 'x24',
       'x25', 'x26', 'x27', 'x28', 'x29', 'x30', 'x31', 'x32', 'x53', 'x54',
       'x55', 'x56', 'x57', 'x58', 'x83', 'x84', 'x85', 'x86', 'x112', 'x113',
       'x141', 'x142', 'x169', 'x477', 'x561', 'x645', 'x646', 'x672', 'x673',
       'x674', 'x700', 'x701', 'x702', 'x728', 'x729', 'x730', 'x731', 'x755',
       'x756', 'x757', 'x758', 'x759', 'x760', 'x781', 'x782', 'x783', 'x784'],
      dtype='object')
```

In [9]:
```
np.min(TROX.min()), np.max(TROX.max())
```

Out[9]:
```
(0.0, 0.99609375)
```

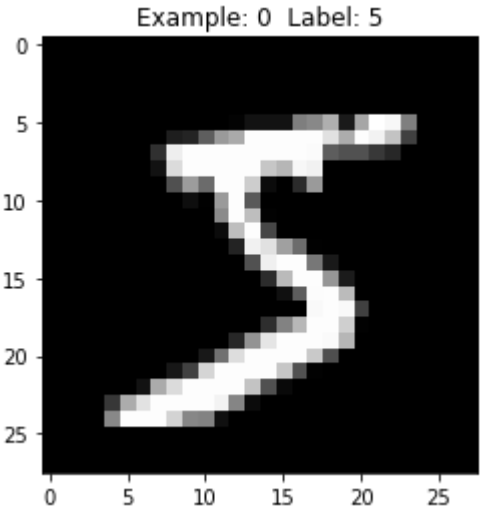
In [10]:
```
# 레이블 데이터 보기
TR0y.head()
```

Out[10]:
```
0    5
1    0
2    4
3    1
4    9
Name: y, dtype: int64
```

개별 이미지 시각화

In [11]:
```
def view_digit(example):
    label = TR0y.loc[example]
    image = TR0X.loc[example,:].values.reshape([28,28])
    plt.title('Example: %d  Label: %d' % (example, label))
    plt.imshow(image, cmap=plt.get_cmap('gray'))
    plt.show()
```

In [12]:
```
# 첫 번째 이미지 살펴보기
view_digit(0)
```



레이블 조정하기 (참고)

- 교재: 생성한 자료의 y가 문자형 {'0',..'9'} 이어서 `preprocessing.LabelBinarizer()` 로 변환했음
- `sklearn.preprocessing.[LabelBinarizer, LabelEncoder, OneHotEncoder]` : X와 y를 구분하여 사용함
  - `LabelEncoder` : for labels coding 0,..,K-1. y 순서형화
  - `OrdinalEncoder` : for features coding 0,..,K-1. 순서형화
  - `LabelBinarizer` : for labels,coding 0 and 1. 가변수화(Binarize labels in a one-vs-all fashion). 기본값: `sparse_output=False` (희소행렬로 저장안함)
  - `OneHotEncoder` : for features coding 0 and 1. 가변수화. 기본값: `sparse=True` 희소행렬로 저장
- pandas: X, y 구분없이 `astype('category')` 로 변수측도를 지정하고, `get_dummies(data, sparse=False, drop_first=False..)` 로 가변수화
- numpy: 가변수 함수가 없음. `keras.np_utils.to_categorical` 을 사용해야 함

```
def one_hot(series):
    label_binarizer = pp.LabelBinarizer()
    label_binarizer.fit(range(max(series)+1))
    return label_binarizer.transform(series)
def reverse_one_hot(originalSeries, newSeries):
    label_binarizer = pp.LabelBinarizer()
    label_binarizer.fit(range(max(originalSeries)+1))
    return label_binarizer.inverse_transform(newSeries)
# 레이블에 대한 원-핫 벡터 생성
# TRY: 가변수화한 타겟
TRY = one_hot(TRy)
VLY = one_hot(VLy)
TSY = one_hot(TSy)
# 첫번째 예제 데이터 숫자 5에 대한 원-핫 벡터 보기
TRY[0]
```

In [13]:
```
# LabelBinarizer로 가변수화
from sklearn.preprocessing import OneHotEncoder, LabelBinarizer
Tlb = LabelBinarizer()
TROY = Tlb.fit_transform(TROy)
TROY[:2]
```

Out[13]:
```
array([[0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

In [14]:
```
# OneHotEncoder를 사용하려면 Series를 DataFrame으로 변환해야 함
Tohe = OneHotEncoder(sparse=False)
TROY = Tohe.fit_transform(pd.DataFrame(TROy))
TROY[:2]
```

Out[14]:
```
array([[0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

층화추출한 자료

- nTR0 = 50,000, p=28^2=784 으로 상당히 큼
- 각 숫자마다 100개 씩 추출한 TRX, TRy으로 처리

In [15]:
```
# pd.DataFrame.sample 적용. MultiIndex (층:숫자, 원변수인덱스) 생성됨
TRX = TROX.groupby(TROy).apply(lambda x: x.sample(100, random_state=2018))

# MultiIndex (층:숫자, 원변수인덱스)에서 원변수인덱스를 추출하여 새 단일 인덱스로 재
iTR = TRX.index.to_frame()[1]
TRX.set_index(iTR, inplace=True)
TRy = TROy.iloc[iTR]
```

In [16]:
```
TRy.head()
```

Out[16]:
```
1907     0
5809     0
32846    0
8838     0
13044    0
Name: y, dtype: int64
```

In [17]:
```
TRX.head()   # 인덱스가 콩일되었는지 확인
```

Out[17]:

	x01	x02	x03	x04	x05	x06	x07	x08	x09	x10	...	x775	x776	x777	x778	x779
1																
1907	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0
5809	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0
32846	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0
8838	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0
13044	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0

5 rows × 784 columns

PCA

모델 적합/성분계산

In [18]:
```
# 주성분 분석
from sklearn.decomposition import PCA

n_components = 784   # 추출할 PC개수. 생략하면 모든 PC사용. [0,1]사이값이면 최소 누적
whiten = False       # True면 PCA sphering(;PC점수를  분산이 1이 되도록 화이트닝). D
random_state = 2018

Epca = PCA(n_components=n_components,   # 추출할 PC개수. 생략하면 모든 PC사용. [0,1]
           whiten=whiten,
           random_state=random_state)

# 반환값
# components_(nc, d),
# explained_variance_: Eigenvalue. PC별 분산 (분모는 n-1)
# singular_values_: SV. 2-norms of the n_components variables in the lower-dimension
# n_components_: k: 추출된 PC개수

# 주성분 점수
TRXTpca = Epca.fit_transform(TRX)
TRXTpca = pd.DataFrame(data=TRXTpca, index=iTR)
```

In [19]:
```
# 주성분점수간 상관행렬. 주성분은 서로 상관없으므로 I
TRXTpca.corr().round(3) # sns.heatmap(TRXT.corr())
```

Out[19]:

	0	1	2	3	4	5	6	7	8	9	...	774	775	776	777	778
0	1.0	0.0	0.0	0.0	-0.0	0.0	0.0	0.0	-0.0	0.0	...	0.0	0.000	-0.0	-0.000	-0.000
1	0.0	1.0	0.0	-0.0	0.0	0.0	0.0	0.0	0.0	-0.0	...	-0.0	0.000	0.0	0.000	-0.000
2	0.0	0.0	1.0	0.0	-0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	-0.000	0.0	-0.000	0.000
3	0.0	-0.0	0.0	1.0	0.0	0.0	-0.0	0.0	0.0	0.0	...	0.0	-0.000	-0.0	0.000	0.000
4	-0.0	0.0	-0.0	0.0	1.0	0.0	0.0	0.0	0.0	-0.0	...	-0.0	0.000	0.0	-0.000	-0.000
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
779	-0.0	-0.0	0.0	0.0	-0.0	0.0	-0.0	0.0	-0.0	-0.0	...	-0.0	0.001	-0.0	0.000	0.001
780	-0.0	0.0	-0.0	0.0	0.0	-0.0	0.0	-0.0	-0.0	-0.0	...	0.0	-0.001	0.0	-0.001	-0.001
781	-0.0	-0.0	-0.0	-0.0	0.0	0.0	0.0	-0.0	-0.0	-0.0	...	0.0	-0.000	0.0	-0.000	-0.001
782	-0.0	0.0	0.0	-0.0	-0.0	0.0	-0.0	0.0	-0.0	-0.0	...	0.0	-0.001	0.0	-0.000	-0.001
783	0.0	-0.0	-0.0	-0.0	0.0	0.0	0.0	0.0	-0.0	0.0	...	-0.0	0.001	-0.0	0.000	0.001

784 rows × 784 columns

In [20]:
```
# 주성분점수의 분산 = S의 아이겐값
TRXTpca.var().round(3)
```

Out[20]:
```
0      5.278
1      3.636
2      3.463
3      2.996
4      2.550
       ...
779    0.000
780    0.000
781    0.000
782    0.000
783    0.000
Length: 784, dtype: float64
```

In [21]:
```
TRXTpca.cov().round(3)  # 대각원소가 아이겐값
```

Out[21]:

	0	1	2	3	4	5	6	7	8	9	...	774	775	776	777	77
0	5.278	0.000	0.000	0.000	-0.00	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	-0.0	-0.0	-0.
1	0.000	3.636	0.000	-0.000	0.00	0.0	0.0	0.0	0.0	-0.0	...	-0.0	0.0	-0.0	0.0	0.
2	0.000	0.000	3.463	0.000	-0.00	0.0	0.0	0.0	0.0	-0.0	...	0.0	-0.0	0.0	-0.0	0.
3	0.000	-0.000	0.000	2.996	0.00	0.0	-0.0	0.0	0.0	0.0	...	-0.0	0.0	-0.0	0.0	-0.
4	-0.000	0.000	-0.000	0.000	2.55	0.0	0.0	0.0	0.0	-0.0	...	-0.0	-0.0	-0.0	-0.0	-0.
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
779	-0.000	-0.000	-0.000	0.000	-0.00	-0.0	-0.0	0.0	-0.0	0.0	...	-0.0	0.0	-0.0	0.0	0.
780	-0.000	-0.000	-0.000	0.000	0.00	-0.0	-0.0	0.0	-0.0	-0.0	...	0.0	-0.0	0.0	-0.0	-0.
781	-0.000	-0.000	-0.000	-0.000	0.00	0.0	0.0	-0.0	-0.0	-0.0	...	0.0	-0.0	0.0	-0.0	-0.
782	-0.000	0.000	0.000	-0.000	-0.00	0.0	-0.0	0.0	-0.0	-0.0	...	0.0	-0.0	0.0	-0.0	-0.
783	0.000	-0.000	-0.000	-0.000	-0.00	0.0	0.0	0.0	-0.0	0.0	...	-0.0	0.0	-0.0	0.0	0.

784 rows × 784 columns

모델 설명력

In [22]:

```
# X개의 주성분으로 추출한 원본 데이터의 분산 비율
def summaryPCA(PCAobj):
  # pcid   = np.arange(1, PCAobj.n_components_+1)
  eigval = pd.DataFrame(PCAobj.explained_variance_)
  prop   = pd.DataFrame(PCAobj.explained_variance_ratio_)
  cumul  = pd.DataFrame(np.cumsum(prop)/prop.sum())
  #vrexp  = pd.DataFrame(np.c_[eigval, prop, cumul], columns=['ExpVar(eigval)','Prop
  vrexp = pd.concat([eigval, prop, cumul], axis=1).T
  vrexp.index = ['Eigenvalue','Prop','Cumulative']
  return vrexp

vrexp = summaryPCA(Epca)
vrexp.loc[:,[0, 1, 2, 9, 19, 49, 99, 199, 299, 399, 499, 599, 699]].round(3)
```

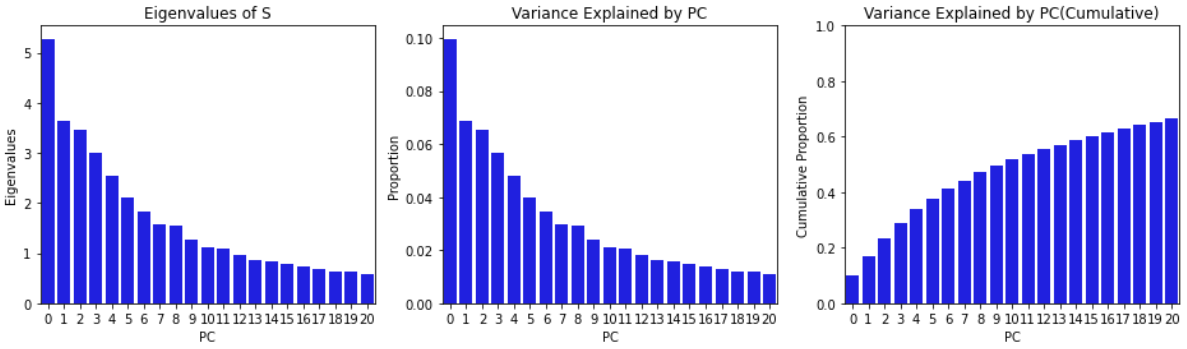
Out[22]:

	0	1	2	9	19	49	99	199	299	399	499	599	699
Eigenvalue	5.278	3.636	3.463	1.277	0.624	0.173	0.052	0.013	0.005	0.002	0.0	0.0	0.0
Prop	0.100	0.069	0.065	0.024	0.012	0.003	0.001	0.000	0.000	0.000	0.0	0.0	0.0
Cumulative	0.100	0.168	0.234	0.495	0.653	0.836	0.926	0.976	0.993	0.998	1.0	1.0	1.0

In [23]:

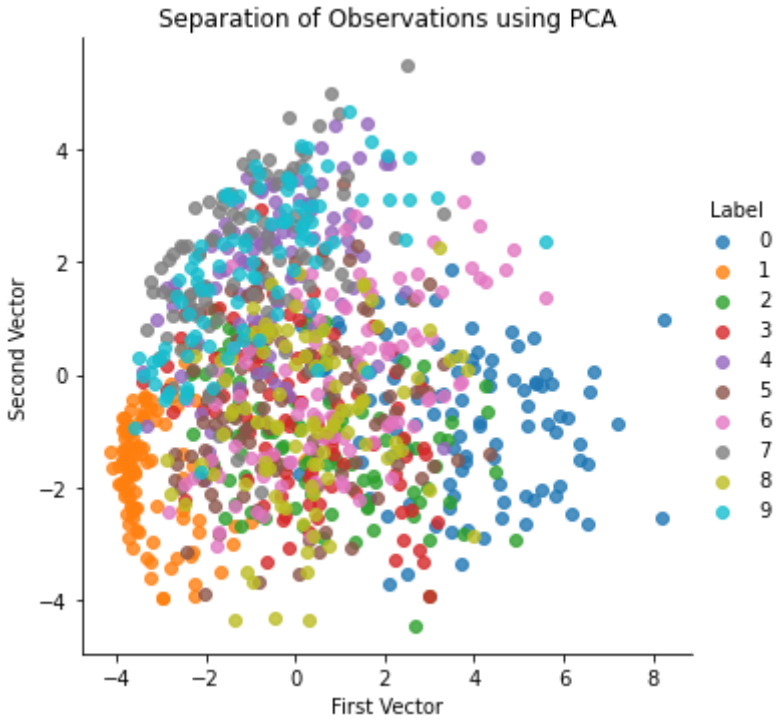
```
# sns.barplot(x:, y:, hue:, data, ...)
#  x, y, hue: data내 변수이름
#  x, y가 있으면 data는 wide-form, x, y가 없으면 long-form으로 간주
# color='blue,green,red,cyan,magenta,yellow,k(black),white' ; 색 지정

fig, ax = plt.subplots(ncols=3, figsize=(16, 4))
ev = sns.barplot(data=pd.DataFrame(vrexp.loc['Eigenvalue',:20]).T, color='b', ax=ax
ev.set(title='Eigenvalues of S', xlabel='PC', ylabel='Eigenvalues')
pr = sns.barplot(data=pd.DataFrame(vrexp.loc['Prop',:20]).T, color='b', ax=ax[1]);
pr.set(title='Variance Explained by PC', xlabel='PC', ylabel='Proportion')
```

```
cm = sns.barplot(data=pd.DataFrame(vrexp.loc['Cumulative',:20]).T, color='b', ax=ax
cm.set(title='Variance Explained by PC(Cumulative)', xlabel='PC', ylabel='Cumulative
```



In [24]:
```
# PC1, PC2 점수의 산점도
# xDF: XTh, yDF: y
# tempDF = [xDF의 첫번째 벡터(PC1 score), xDF의 두번째 벡터(PC2 score), y]
def scatterPlot(xDF, yDF, algoName):
    tempDF = pd.DataFrame(data=xDF.loc[:,0:1], index=xDF.index)  # 첫 2개의 성분
    tempDF = pd.concat( [tempDF, yDF], axis=1, join='inner')     # 인덱스를 키로 inn
    tempDF.columns = ['First Vector', 'Second Vector', 'Label']
    sns.lmplot(x='First Vector', y='Second Vector', hue='Label', data=tempDF, fit_re
    #sns.scatterplot(x='First Vector', y='Second Vector', hue='Label', data=tempDF)
    ax = plt.gca()  # Get the Current Axes instance. 현재 축 정보 저장
    ax.set_title('Separation of Observations using '+algoName)
```

In [25]:
```
# PC1 score vs PC2 score 산점도
scatterPlot(TRXTpca, TRy, 'PCA')
```

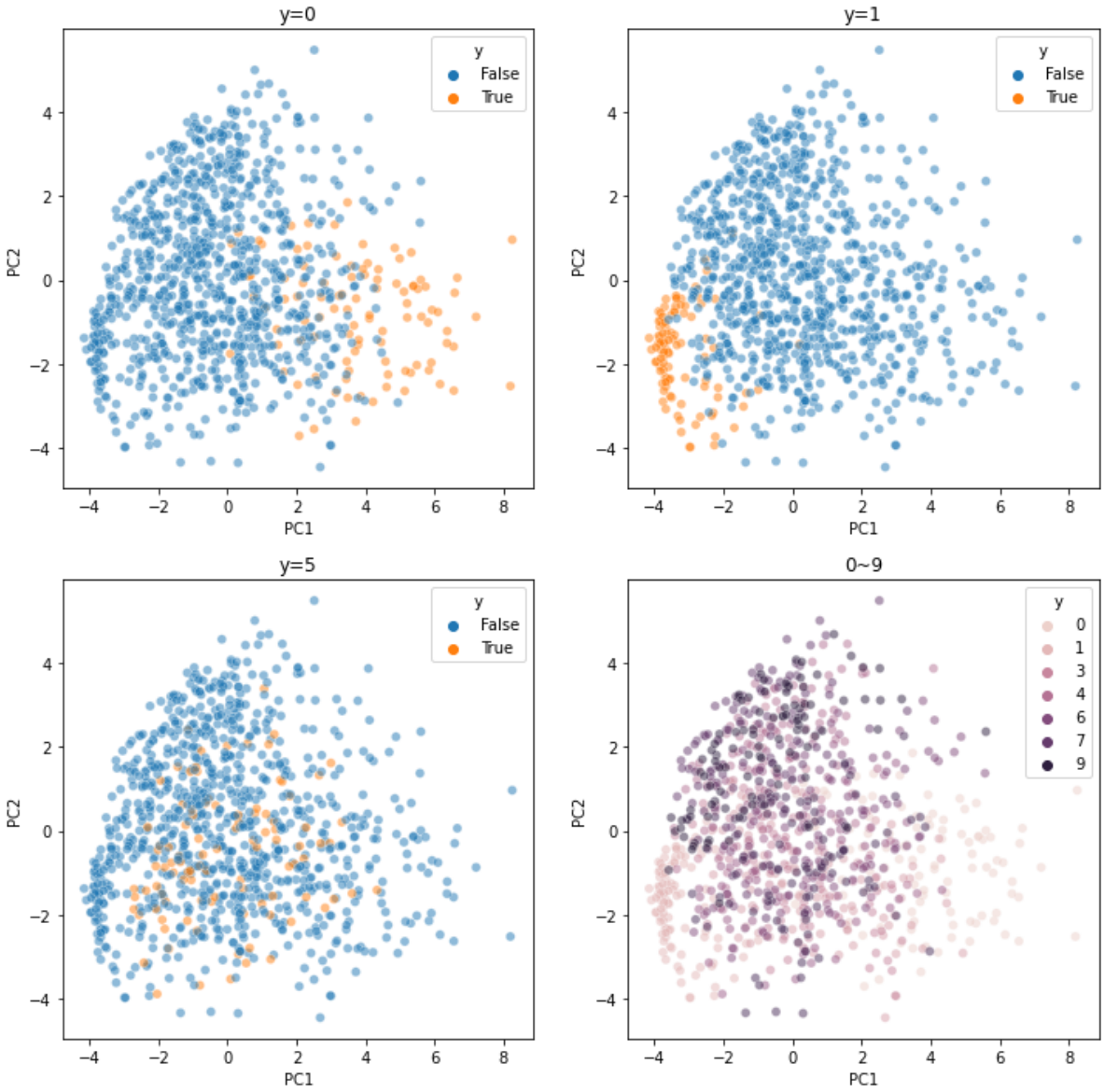


- 0, 1은 잘 구분이 되고, 5,6이 상대적으로 구분이 어려움

```
fig, axs = plt.subplots(nrows=4, ncols=3, figsize=(24,24))
d0 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1],
hue=TRy['y']==0, alpha=0.25, ax=axs[0,0]);
d0.set(xlabel='PC1', ylabel='PC2', title='0')
d1 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1],
hue=TRy['y']==1, alpha=0.25, ax=axs[0,1]);
```

```
d1.set(xlabel='PC1', ylabel='PC2', title='1')
d2 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1],
hue=TRy['y']==2, alpha=0.25, ax=axs[0,2]);
d2.set(xlabel='PC1', ylabel='PC2', title='2')
d3 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1],
hue=TRy['y']==3, alpha=0.25, ax=axs[1,0]);
d3.set(xlabel='PC1', ylabel='PC2', title='3')
d4 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1],
hue=TRy['y']==4, alpha=0.25, ax=axs[1,1]);
d4.set(xlabel='PC1', ylabel='PC2', title='4')
d5 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1],
hue=TRy['y']==5, alpha=0.25, ax=axs[1,2]);
d5.set(xlabel='PC1', ylabel='PC2', title='5')
d6 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1],
hue=TRy['y']==6, alpha=0.25, ax=axs[2,0]);
d6.set(xlabel='PC1', ylabel='PC2', title='6')
d7 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1],
hue=TRy['y']==7, alpha=0.25, ax=axs[2,1]);
d7.set(xlabel='PC1', ylabel='PC2', title='7')
d8 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1],
hue=TRy['y']==8, alpha=0.25, ax=axs[2,2]);
d8.set(xlabel='PC1', ylabel='PC2', title='8')
d9 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1],
hue=TRy['y']==9, alpha=0.25, ax=axs[3,0]);
d9.set(xlabel='PC1', ylabel='PC2', title='9')
dd = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1],
hue=TRy['y'], alpha=0.25, ax=axs[3,1]);
dd.set(xlabel='PC1', ylabel='PC2')
```

In [26]:
```
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(12,12))
d0 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1], hue=TRy==0, alpha=0.5,
d0.set(xlabel='PC1', ylabel='PC2', title='y=0')
d1 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1], hue=TRy==1, alpha=0.5,
d1.set(xlabel='PC1', ylabel='PC2', title='y=1')
d5 = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1], hue=TRy==5, alpha=0.5,
d5.set(xlabel='PC1', ylabel='PC2', title='y=5')
dd = sns.scatterplot(x=TRXTpca.loc[:,0], y=TRXTpca.loc[:,1], hue=TRy, alpha=0.5, ax
dd.set(xlabel='PC1', ylabel='PC2', title='0~9');
```

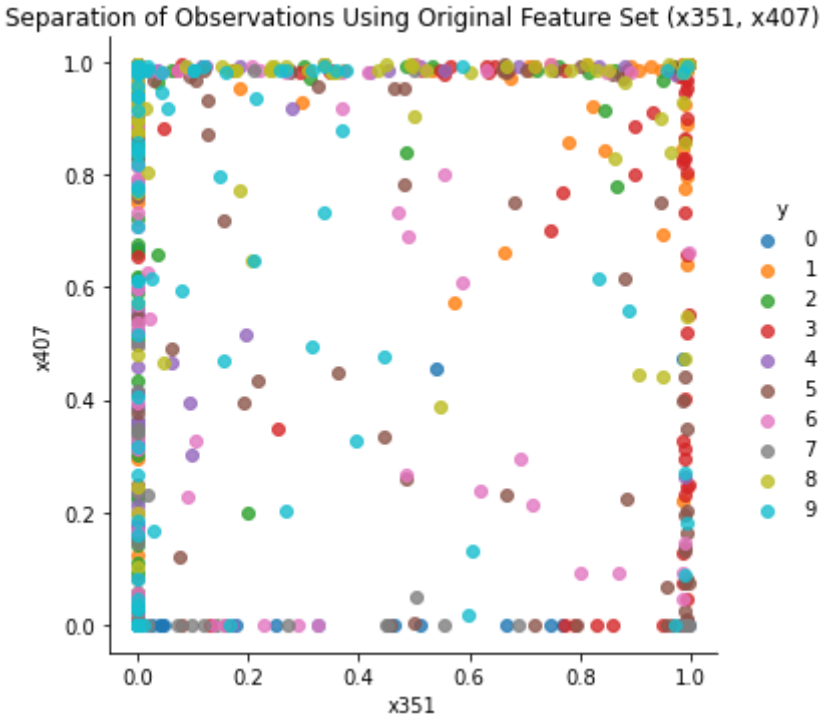


In [27]:
```
# x351, x407 값
TRX.iloc[:,[350,406]].head()
```

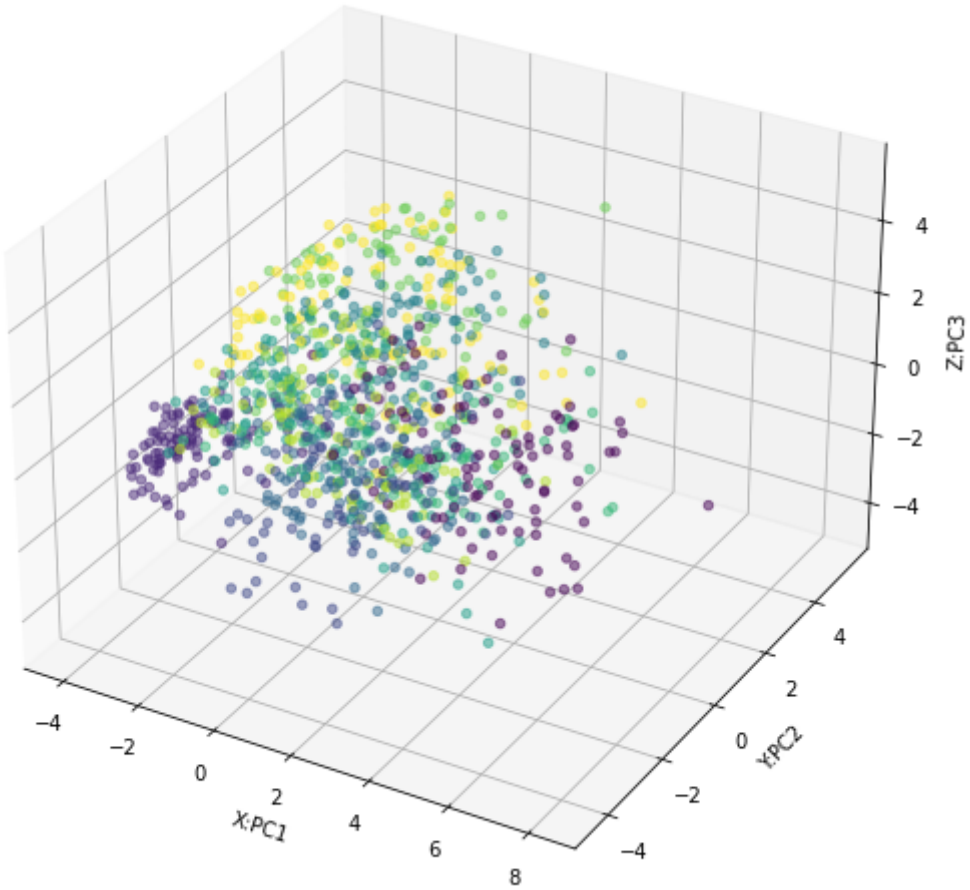
Out[27]:

1	x351	x407
1907	0.0	0.0
5809	0.0	0.0
32846	0.0	0.0
8838	0.0	0.0
13044	0.0	0.0

In [28]:
```
# x351, x407의 산점도: y 구분 불가능 (원변수 두 개만 가지고 y를 구분하는 것은 불가능
XX = pd.DataFrame(data=TRX.iloc[:,[350,406]], index=iTR)
XX = pd.concat([XX, TRy], axis=1, join='inner')
# XX.columns = ['x351', 'x407', 'y']
sns.lmplot(x='x351', y='x407', hue='y', data=XX, fit_reg=False)
ax = plt.gca()  # get the current axes
ax.set_title('Separation of Observations Using Original Feature Set (x351, x407)');
```



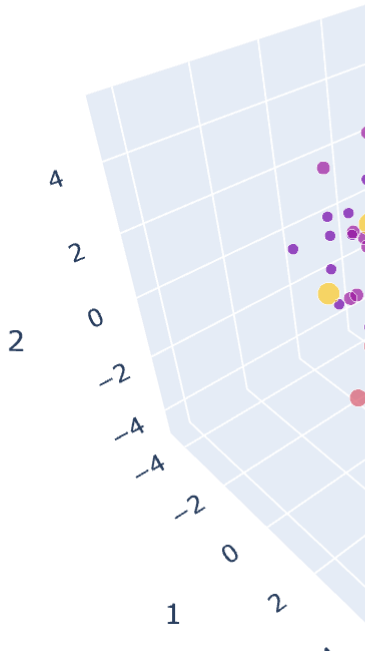
In [29]:
```
# 3D scatter: 첫 3개의 주성분으로 구분하기. 전체 분산의 23.4%만 설명됨
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize = (16, 9))
ax  = plt.axes(projection ='3d')
p   = ax.scatter3D(TRXTpca[0],TRXTpca[1],TRXTpca[2], alpha=0.5, c=TRy)
ax.set_xlabel('X:PC1')
ax.set_ylabel('Y:PC2')
ax.set_zlabel('Z:PC3');
```



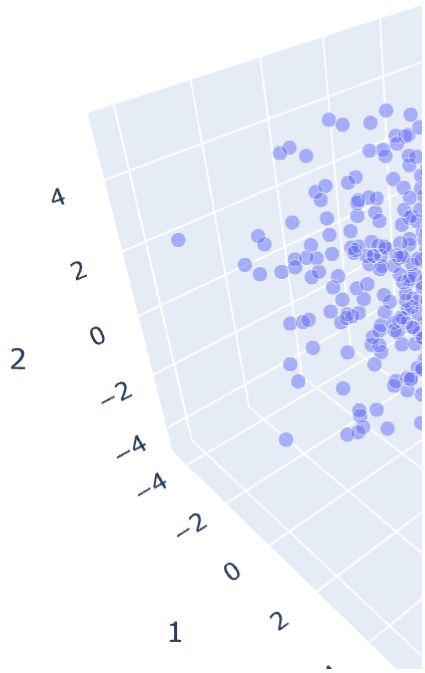
- Dynamic 3D scatterplot : `plotly` 설치해야 함. `tf` 환경 프롬프트에서 `pip install plotly` 로 설치하거나 노트북상에서 설치

```
! pip install plotly
```

In [30]:
```
import plotly.express as px
# 처음 3개의 주성분 점수에 대해 3D 산점도. 제일 구분 잘 되는 1 기준 으로 모든 관측값
fig = px.scatter_3d(TRXTpca, x=0, y=1, z=2, color=TRy, size=TRy,  size_max=10) #, o
fig.show()
```



In [31]:
```
fig = px.scatter_3d(TRXTpca, x=0, y=1, z=2,
                    color=TRy==1,     #
                    symbol=TRy==1,
                    size=1+(TRy==1).astype(int),
                    size_max=9,
                    opacity=0.5) #, size=0.1)
fig.show()
```



- 삭제 ``` # 층화추출된 주성분 점수에 대해 3D 산점도. 제일 구분 잘 되는 1 기준 으로 시각화 TRXT0 = TRXTpca.groupby(TRy).apply(lambda x:x.sample(100, random_state=2018)) iTR0 = TRXT0.index.to_frame()[1] TRXT0.set_index(iTR0, inplace=True) TRy0 = TRy.iloc[iTR0]

fig = px.scatter_3d(TRXT0, x=0, y=1, z=2, color=TRy0==1, # symbol=TRy0==1, size=1+(TRy0==1).astype(int), size_max=9, opacity=0.5) #, size=0.1) fig.show() ```

차원축소 Tuning

- 복원이 안되는 모델(; X의 근사값을 제공하지 않는 모델)은 성능평가가 어렵고, 주로 시각화나 탐색결과를 보고 평가함
- `inverse_transform` 을 제공하는 모델: X를 근사할 수 있으므로 복원오차로 모델 성능평가 가능
  - `PCA, IncrementalPCA, TruncatedSVD` : 최적 성분수 결정에 쓸 수 있지만 보통 scree plot이나 상황으로 판단함
  - `KernelPCA` : 초모수(alpha, gamma) 튜닝 가능
  - `FasICA, NMF`
- 성능평가기준:

- X와 Xh간 평균제곱오차(mse)나 Frobenius Norm(잔차제곱합) 사용가능
- 사용자가 성능평가함수(score)를 직접 작성해야 함

In [32]:
```
# Selecting kernel and hyperparameters for kernel PCA
# https://stackoverflow.com/questions/53556359/selecting-kernel-and-hyperparameters-
# https://scikit-learn.org/stable/modules/model_evaluation.html#implementing-your-ow

# GridSearchCV(X, y=None) 이면 비지도학습 CV
# scoring이 필요함
# 지도학습인 경우 myscorer = make_scorer(mean_squared_error, greater_is_better=False
# 비지도학습인 경우

from sklearn.model_selection import KFold, StratifiedKFold, GridSearchCV
from sklearn.metrics import mean_squared_error

# 1단계: CV분할
# SKF5 = StratifiedKFold(n_splits=5, shuffle=True, random_state=2018) # CV할 때 y가

KF5 = KFold(n_splits=5, shuffle=True, random_state=2018)


def negmse(estimator, X, y=None):
    XT = estimator.transform(X)           # 성분점수
    Xh = estimator.inverse_transform(XT)  # 성분점수로부터 Xh(X의 예측값) 반환. inve
    return -1* mean_squared_error(X, Xh) # GOF가 되도록 (-)
```

In [33]:
```
parampca ={'n_components':[100, 200, 300, 400, 500, 600, 700]}  # 7개의 성분수에 대하

Epca2 = PCA()
GSpca = GridSearchCV(Epca2, param_grid=parampca, cv=KF5, scoring=negmse)
%time GSpca.fit(TRX)
GSpca.score(TRX, TRy), GSpca.score(VLX, VLy), GSpca.score(TSX, TSy)
```

```
CPU times: total: 32.8 s
Wall time: 8.61 s
```

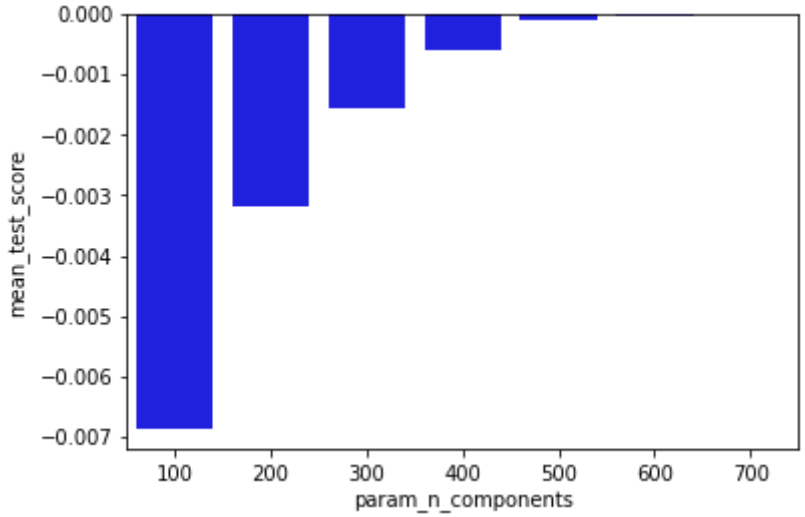
Out[33]:
```
(-1.6998993160134054e-31, -1.0719103993474914e-05, -9.653329642467029e-06)
```

In [34]:
```
CV = pd.DataFrame(GSpca.cv_results_)    # 7(params) x 5 (folds)
# n_components가 많을수록 성능이 좋기 때문에 지도학습에 비해 결과가 명확하지 않음.
# cv보다는 screeplot, min_eigenvalue > 0.7, 누적설명량 > 0.9 등의 기준이 더 현실적임
# CV.filter(regex='^params|^rank|test_score$').sort_values('rank_test_score')
CV[['params','param_n_components','mean_test_score','rank_test_score']].sort_values(
```

Out[34]:

	params	param_n_components	mean_test_score	rank_test_score
6	{'n_components': 700}	700	-0.000008	1
5	{'n_components': 600}	600	-0.000019	2
4	{'n_components': 500}	500	-0.000116	3
3	{'n_components': 400}	400	-0.000589	4
2	{'n_components': 300}	300	-0.001548	5
1	{'n_components': 200}	200	-0.003170	6
0	{'n_components': 100}	100	-0.006867	7

In [35]:
```
sns.barplot(x='param_n_components', y='mean_test_score', data=CV, color='b');
```



In [36]:
```
print(GSpca.best_score_)  # mean_test_score(negmse) at best_params
print(GSpca.best_params_)
print(GSpca.best_estimator_)
```

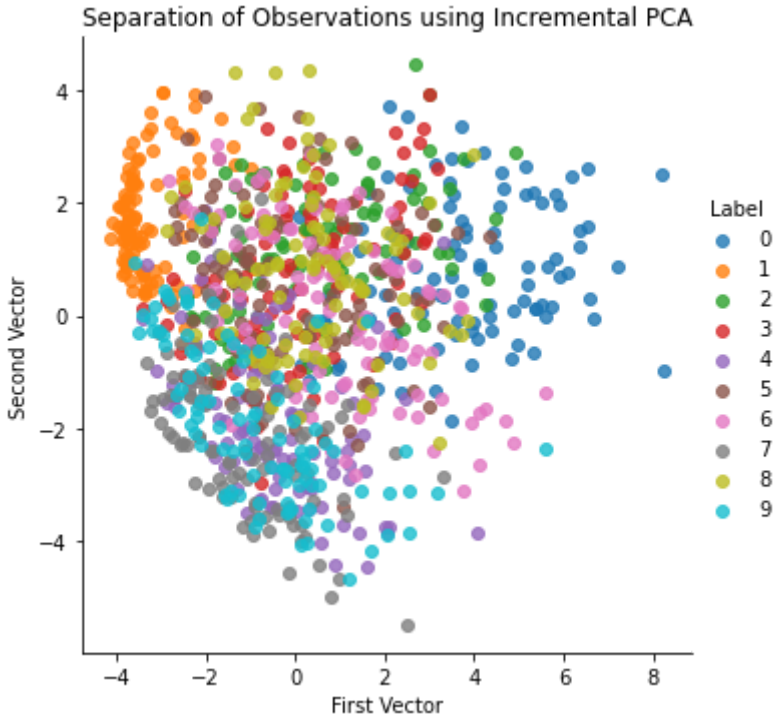
```
-8.472633312277276e-06
{'n_components': 700}
PCA(n_components=700)
```

Incremental PCA

- Incremental principal component analysis (IPCA): n이 너무 커서 메모리에 한번에 로딩할 수 없을 때 사용(`batch_size` 지정). PCA와 거의 동일한 결과(부호는 바뀔 수 있음)
- `batch_size=None` : 데이터 크기와 변수개수 x 5 등을 참고하여 설정함

In [37]:
```
# 점진적 PCA
from sklearn.decomposition import IncrementalPCA

n_components = 784  # 추출할 PC개수. 생략하면 모든 PC사용. [0,1]사이값이면 최소 누적
batch_size = None  #  None이면 데이터 크기를 고려하여 자동으로 설정함

Eipca = IncrementalPCA(n_components=n_components, batch_size=batch_size)

# 반환값
# components_(nc, d),
# explained_variance_: Eigenvalue. PC별 분산 (분모는 n-1)
# singular_values_: SV. 2-norms of the n_components variables in the lower-dimension
# n_components_: k: 추출된 PC개수

%time TRXTipca = Eipca.fit_transform(TRX)
TRXTipca = pd.DataFrame(data=TRXTipca, index=iTR)

VLXTipca = Eipca.transform(VLX)
VLXTipca = pd.DataFrame(data=VLXTipca, index=iVL)

print(TRXTipca.shape)
scatterPlot(TRXTipca, TRy, 'Incremental PCA')
```

```
CPU times: total: 781 ms
Wall time: 220 ms
(1000, 784)
```



Sparse PCA

- 희소(Sparse)행렬: 원소의 상당수가 0인 행렬 (예:DT행렬, 숫자인식 등의 이미지 픽셀). SparseMatrix 형식으로 저장하여 용량을 줄임
- 희소 표현(Sparse representation): 관련값(주로 가중치)들을 필요이상으로 커지지 않도록 규제(; 0에 가까워지도록 규제)하여 모형을 단순화. Lasso나 Ridge 회귀, 신경망의 Weight Decay처럼 계수를 축소

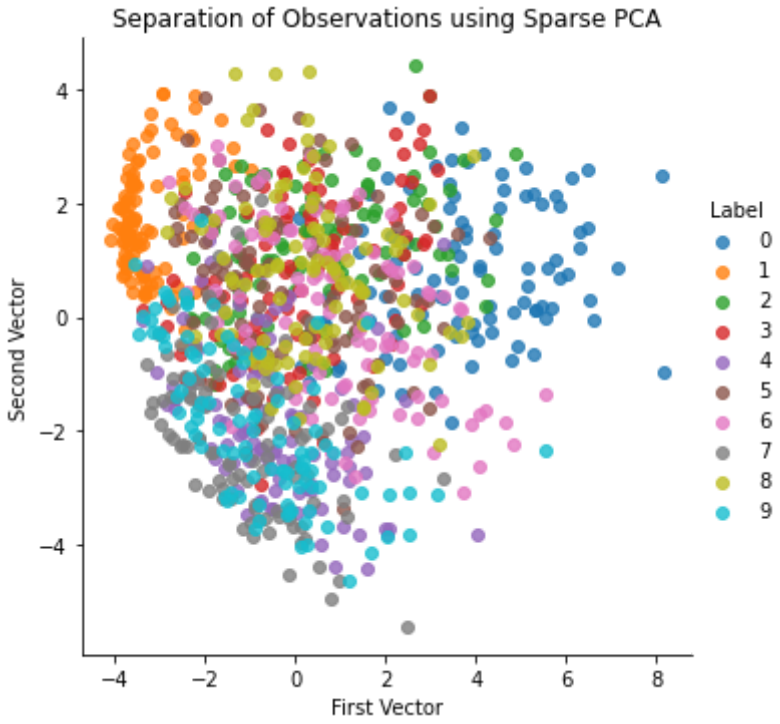
—

- 규제모수 `alpha` . `inverse_transform` 이 없어 sk로 튜닝불가. 지도학습과 연계해서 튜닝해야 함
- 느림.

In [38]:
```
# 희소 PCA
from sklearn.decomposition import SparsePCA

n_components = 100
alpha = 0.0001
random_state = 2018
n_jobs = -1

# 수정 사항:normalize_components='deprecated' 설정하면 경고 삭제됨 components 정규화
Espca = SparsePCA(n_components=n_components,
                  alpha=alpha,
                  random_state=random_state,
                  n_jobs=n_jobs)
                  #,normalize_components='deprecated'

%time Espca.fit(TRX)  # Espca.fit(TROX.loc[:10000,:])
%time TRXTspca  = Espca.transform(TRX)
TRXTspca  = pd.DataFrame(data=TRXTspca, index=iTR)
```

```
VLXTspca  = Espca.transform(VLX)
VLXTspca  = pd.DataFrame(data=VLXTspca, index=iVL)

print(TRXTspca.shape)
scatterPlot(TRXTspca, TRy, 'Sparse PCA')
```

```
CPU times: total: 2.59 s
Wall time: 9.55 s
CPU times: total: 62.5 ms
Wall time: 15 ms
(1000, 100)
```



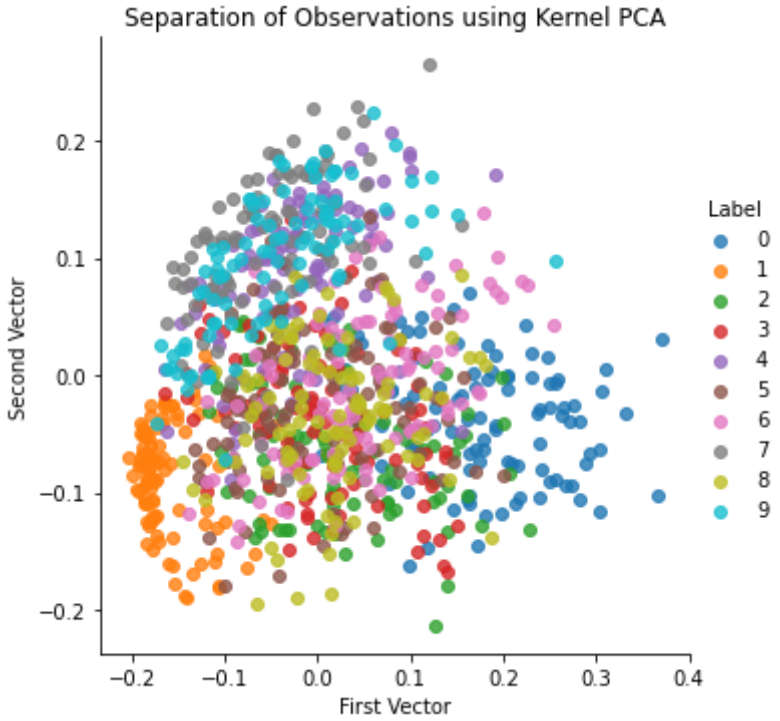
Kernel PCA

- 커널트릭을 이용하여 PCA를 비선형으로 확장
- 커널함수로 X의 개체간 유사성 계산. X가 선형으로 분리되지 않을 때 커널변환으로 고차원에서 선형분리 시도. 비선형자료에 효과적
- `inverse_transform` 가능: fit할때 `fit_inverse_transform=True` 지정해야 사용가능
- 초모수:
  - `kernel` :
  - `gamma` : 커널폭. 클수록 보수적. 기본값 1/p.
  - `alpha` : ridge regression 벌점계수

In [39]:
```
# 커널 PCA
from sklearn.decomposition import KernelPCA

n_components = 200
kernel = 'rbf'         # 'linear,poly,rbf,sigmoid,cosine,precomputed'
gamma = None           # 커널폭. 기본값(None) = 1/p
random_state = 2018
n_jobs = 1             # 병렬작업개수. 기본값(None)이면 1.

Ekpca = KernelPCA(n_components=n_components,
                  kernel=kernel,
```

```
                    gamma=gamma,
                    n_jobs=n_jobs,
                    random_state=random_state)

%time Ekpca.fit(TRX)      # Ekpca.fit(TROX)
%time TRXTkpca  = Ekpca.transform(TRX)

TRXTkpca  = pd.DataFrame(data=TRXTkpca, index=iTR)

VLXTkpca  = Ekpca.transform(VLX)
VLXTkpca  = pd.DataFrame(data=VLXTkpca, index=iVL)

print(TRXTkpca.shape)
scatterPlot(TRXTkpca, TRy, 'Kernel PCA')
```

```
CPU times: total: 781 ms
Wall time: 211 ms
CPU times: total: 250 ms
Wall time: 61 ms
(1000, 200)
```



In [40]:
```
paramkpca ={
    'gamma':np.linspace(0.01, 0.05, 5),
    'kernel': ['rbf','linear']
}
Ekpca2 = KernelPCA(n_components=200, fit_inverse_transform=True, n_jobs=n_jobs, ran
GSkpca = GridSearchCV(Ekpca2, param_grid=paramkpca, cv=KF5, scoring=negmse)

%time GSkpca.fit(TRX) # TRX의 일부만 사용하여 적합
GSkpca.score(TRX), GSkpca.score(VLX), GSkpca.score(TSX)
```

```
CPU times: total: 45 s
Wall time: 11.3 s
```

Out[40]:
```
(-0.04012729163405116, -0.04117531221542822, -0.04147315559415966)
```

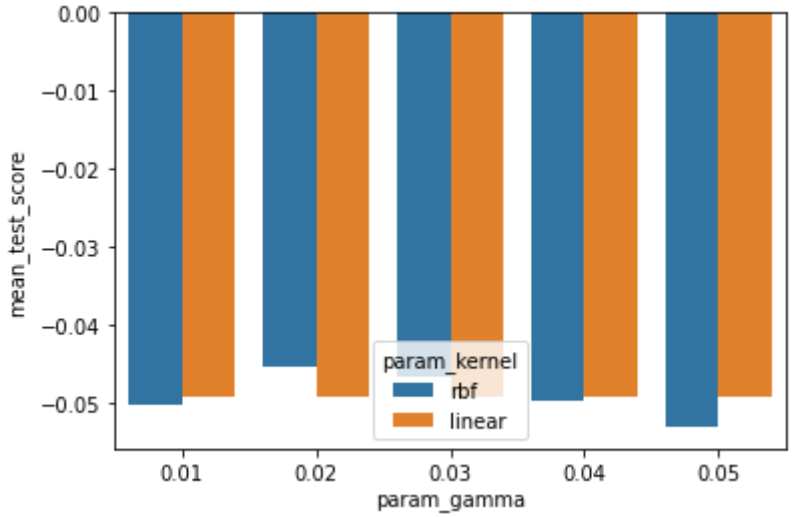
In [41]:
```
CV = pd.DataFrame(GSkpca.cv_results_)    # 10(params combination = 5개 gamma, 2개 커
# n_components가 많을수록 성능이 좋기 때문에 지도학습에 비해 결과가 명확하지 않음.
# 튜닝모수보다 성분수가 더 중요함
# CV.filter(regex='^params|^rank|test_score$').sort_values('rank_test_score')
CV.filter(regex='^params|test_score$').sort_values('rank_test_score')
```

Out[41]:

	params	split0_test_score	split1_test_score	split2_test_score	split3_test_score	split4_test_score
2	{'gamma': 0.02, 'kernel': 'rbf'}	-0.045202	-0.045304	-0.045994	-0.045729	-0.044710
4	{'gamma': 0.03, 'kernel': 'rbf'}	-0.046400	-0.046582	-0.047405	-0.047082	-0.045932
1	{'gamma': 0.01, 'kernel': 'linear'}	-0.049279	-0.049091	-0.048800	-0.048866	-0.049192
3	{'gamma': 0.02, 'kernel': 'linear'}	-0.049279	-0.049091	-0.048800	-0.048866	-0.049192
5	{'gamma': 0.03, 'kernel': 'linear'}	-0.049279	-0.049091	-0.048800	-0.048866	-0.049192
7	{'gamma': 0.04, 'kernel': 'linear'}	-0.049279	-0.049091	-0.048800	-0.048866	-0.049192
9	{'gamma': 0.05, 'kernel': 'linear'}	-0.049279	-0.049091	-0.048800	-0.048866	-0.049192
6	{'gamma': 0.04, 'kernel': 'rbf'}	-0.049400	-0.049662	-0.050608	-0.050251	-0.049028
0	{'gamma': 0.01, 'kernel': 'rbf'}	-0.050170	-0.050142	-0.050842	-0.050581	-0.049832
8	{'gamma': 0.05, 'kernel': 'rbf'}	-0.052644	-0.052989	-0.053987	-0.053637	-0.052389

In [42]:

```
sns.barplot(x='param_gamma', y='mean_test_score', hue='param_kernel', data=CV);
```



In [43]:
```
print(GSkpca.best_score_)  # mean_test_score(negmse) at best_params
print(GSkpca.best_params_)
print(GSkpca.best_estimator_)
```

```
-0.04538802390417373
{'gamma': 0.02, 'kernel': 'rbf'}
KernelPCA(fit_inverse_transform=True, gamma=0.02, kernel='rbf',
          n_components=200, n_jobs=1, random_state=2018)
```

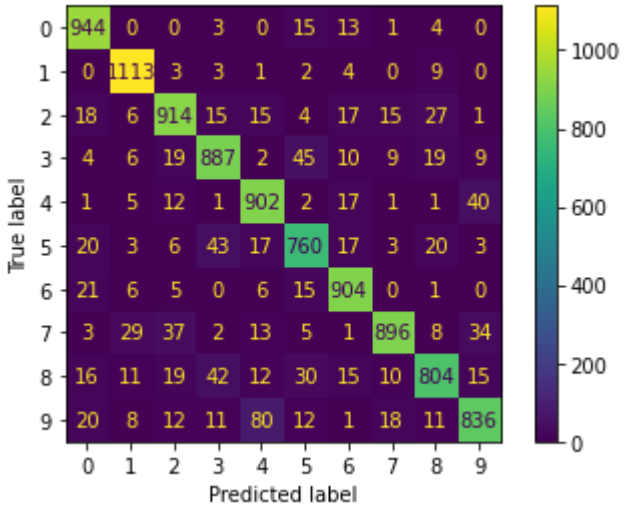
In [44]:
```
# 지도학습시 파이프라인에 차원축소를 포함시켜서 사용할 수 있음
# https://tekworld.org/2018/12/08/day-29-100-days-mlcode-kernel-pca-lle/#page-conten
# https://www.coursehero.com/file/ppvcbos/Then-it-uses-GridSearchCV-to-find-the-best

from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
PEglm  = Pipeline([('kpca', KernelPCA()),
                   ('glm', LogisticRegression())])

SKF5 = StratifiedKFold(n_splits=5, shuffle=True, random_state=2018)
param_grid = {
    'kpca__n_components':[100, 200],
    'kpca__gamma': np.linspace(0.01, 0.05, 5),
    'kpca__kernel': ['rbf', 'linear','sigmoid']}

GSglm = GridSearchCV(PEglm, param_grid, cv=SKF5)  # KPCA할 때 TRX의 일부만 사용해서
#%time GSglm.fit(TRX.loc[:10000,:], TRy.loc[:10000,'y'])
# 오류발생가능: column-vector y was passed when a 1d array was expected. change shap
# lbfgs failed to converge (status=1): STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
%time GSglm.fit(TRX, TRy) # column-vector y was passed when a 1d array was expected
print(GSglm.best_score_)
print(GSglm.best_params_)
```

```
CPU times: total: 1min 50s
Wall time: 27.7 s
0.873
{'kpca__gamma': 0.03, 'kpca__kernel': 'rbf', 'kpca__n_components': 200}
```

In [45]:
```
GSglm.best_estimator_.predict(TRX)
```

Out[45]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5, 1, 1,
       1, 1, 5, 1, 1, 1, 1, 1, 1, 3, 6, 1, 1, 6, 1, 1, 1, 1, 1, 1, 1, 5,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       8, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 6, 3, 3, 3, 3, 3, 3, 3,
       3, 3, 3, 3, 3, 3, 3, 3, 1, 3, 3, 3, 3, 3, 3, 3, 3, 5, 2, 3, 3, 3,
       3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 7, 3, 3, 3, 3, 3, 3, 3, 3,
       3, 5, 3, 3, 8, 8, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 3, 3, 5,
       3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
       3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 2, 4, 4, 4, 4, 4, 4, 4, 2, 4, 4,
       4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 8, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
       4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
       4, 4, 4, 1, 4, 4, 4, 4, 4, 9, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
       4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 9, 4, 4, 9, 5, 5, 5, 5, 5, 5,
       5, 5, 6, 5, 5, 5, 5, 5, 5, 5, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
       5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
       5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 2, 5, 0, 5, 5, 5,
       5, 5, 5, 2, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
       5, 5, 5, 6, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
       6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
       6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 2,
       6, 6, 6, 6, 6, 4, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
       6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 0, 6, 6, 6, 6, 6, 7, 7, 7, 7,
       7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 2, 7, 7, 7, 7,
       7, 7, 7, 8, 7, 7, 7, 7, 9, 9, 7, 9, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
       7, 3, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
       7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
       7, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 5, 8,
       5, 8, 8, 8, 3, 8, 8, 8, 8, 2, 8, 1, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
       8, 8, 3, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
       8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
       8, 8, 8, 8, 8, 8, 8, 8, 1, 8, 8, 8, 8, 8, 8, 8, 1, 8, 8, 8, 9, 9,
       9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
       9, 9, 9, 9, 3, 9, 9, 9, 9, 4, 9, 7, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
       9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 4, 9, 9, 9, 9, 9, 9,
       9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 4, 9, 9, 9, 9, 9,
       9, 9, 9, 9, 9, 9, 9, 9, 2, 9], dtype=int64)
```

In [46]:
```
from sklearn.metrics import ConfusionMatrixDisplay, RocCurveDisplay, PrecisionRecal
display = ConfusionMatrixDisplay.from_predictions(TSy, GSglm.best_estimator_.predict
```



In [47]:
```
GSglm.best_estimator_.score(TRX, TRy), GSglm.best_estimator_.score(VLX, VLy), GSglm.
```

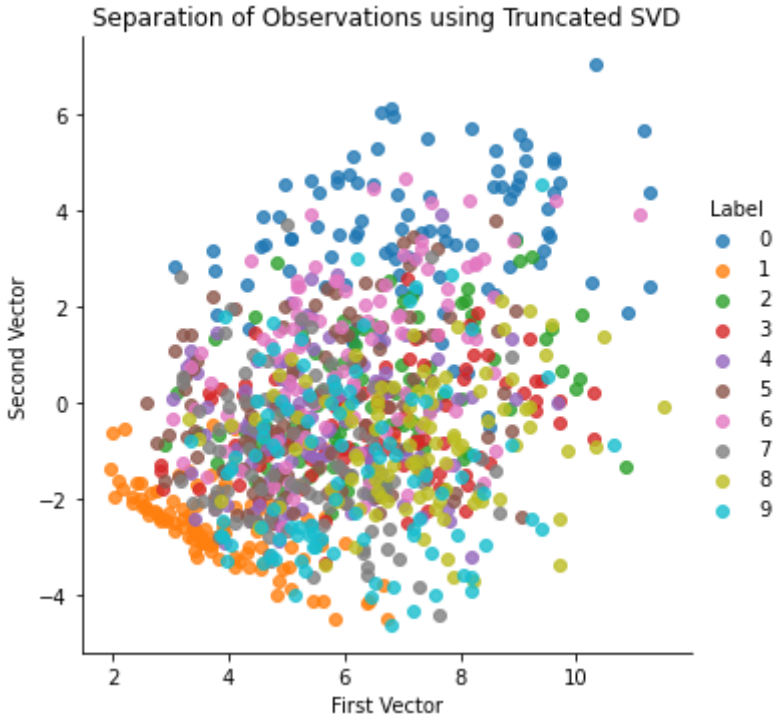
Out[47]:
```
(0.94, 0.9007, 0.896)
```

truncatedSVD

- `PCA` 와 유사한 결과. X를 직접 분해(X를 중심화하지 않고 진행)
- `inverse_transform` 가능

In [48]:
```
# 특잇값 분해
from sklearn.decomposition import TruncatedSVD

n_components = 200          # 기본값=2
algorithm = 'randomized'    # 'randomized, arpack'
n_iter = 5                  # randomized SVD의 n_iter. 기본값=5
random_state = 2018

Etsvd = TruncatedSVD(n_components=n_components,
                     algorithm=algorithm,
                     n_iter=n_iter,
                     random_state=random_state)


%time  TRXTtsvd  = Etsvd.fit_transform(TRX)
TRXTtsvd  = pd.DataFrame(data=TRXTtsvd, index=iTR)

VLXTtsvd  = Etsvd.transform(VLX)
VLXTtsvd  = pd.DataFrame(data=VLXTtsvd, index=iVL)

print(TRXTtsvd.shape)
scatterPlot(TRXTtsvd, TRy, 'Truncated SVD')
```

```
CPU times: total: 625 ms
Wall time: 155 ms
(1000, 200)
```



Random Projection

- Johson-Lindenstrauss Lemma: 고차원 공간의 X의 점들은 점들간 거리가 거의 보존되는 방식으로 낮은 차원에 표현가능
- 선형 차원축소
- `GRP`, `SRP`
- `inverse_transform` 불가

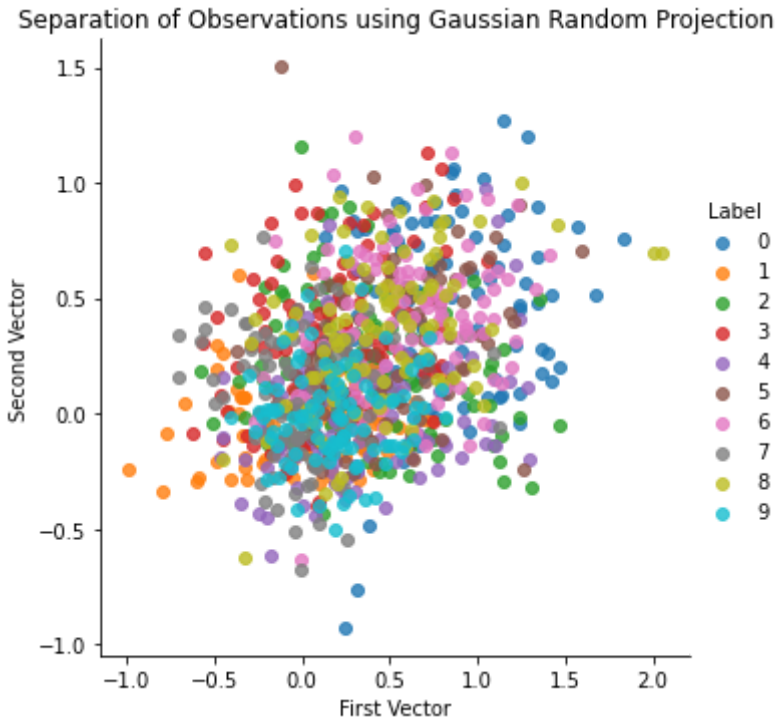
GRP: Gaussian RandomProjection

- 초모수:
  - eps: 값이 작을수록 투사한 저차원의 차원수가 높아짐(;정교한 임베딩). 클수록 저차원의 차원수가 낮아짐(;느슨한 임베딩)
- `inverse_transform` 불가

In [49]:
```
# 가우시안 랜덤 투영
from sklearn.random_projection import GaussianRandomProjection

n_components = 'auto'   # n에 따라 자동으로 설정하거나 사용자가 지정
eps = 0.5               # 기본값 0.1. 값이 작을수록 정교하게 축소됨(;n_components가
random_state = 2018

Egrp = GaussianRandomProjection(n_components=n_components,
                                eps=eps,
                                random_state=random_state)

# Mtsvd.fit(TRX.loc[:10000,:])
%time  TRXTgrp  = Egrp.fit_transform(TRX)
TRXTgrp  = pd.DataFrame(data=TRXTgrp, index=iTR)

VLXTgrp  = Egrp.transform(VLX)
VLXTgrp  = pd.DataFrame(data=VLXTgrp, index=iVL)
```

```
print(TRXTgrp.shape)   # Egrp.n_components_)
scatterPlot(TRXTgrp, TRy, 'Gaussian Random Projection')
```

```
CPU times: total: 31.2 ms
Wall time: 26 ms
(1000, 331)
```

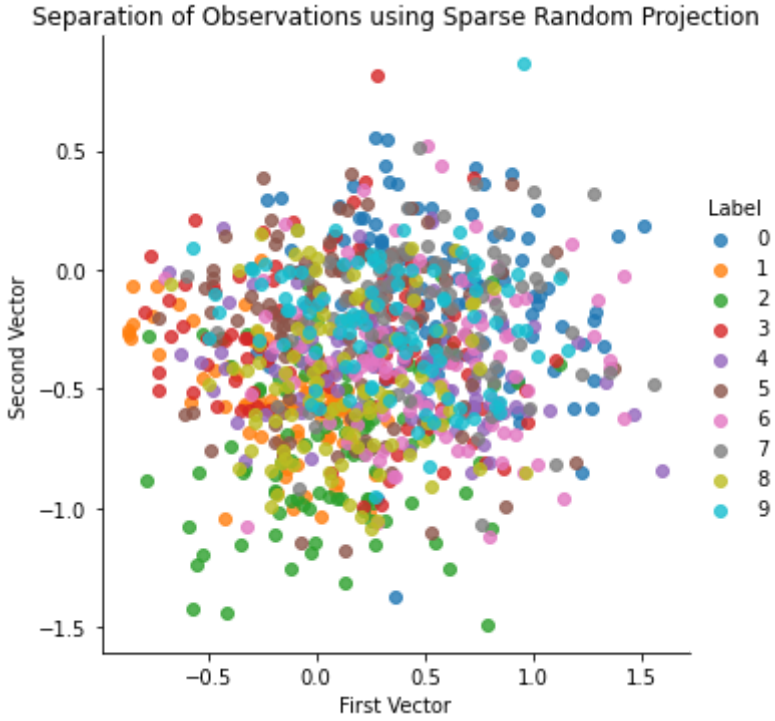


Sparse Random Projection

- GRP 계산부담(속도, 메모리)을 완화시킨 방법
- 초모수
  - `eps` : 값이 작을수록 투사한 저차원의 차원수가 높아짐(;정교한 임베딩). 클수록 저차원의 차원수가 낮아짐(;느슨한 임베딩)
- `inverse_transform` 불가

In [50]:
```
# 희소 랜덤 투영
from sklearn.random_projection import SparseRandomProjection

n_components = 'auto'   # n에 따라 자동으로 설정하거나 사용자가 지정
density = 'auto'        # density='auto, 0~1사이 숫자': 랜덤투사행렬에서 0이 아닌 원
eps = 0.5               # 기본값 0.1. 값이 작을수록 정교하게 축소됨(;n_components가
dense_output = False    # 기본값 False
random_state = 2018
Esrp = SparseRandomProjection(n_components=n_components,
                              density=density,
                              eps=eps,
                              dense_output=dense_output,
                              random_state=random_state)

%time TRXTsrp  = Esrp.fit_transform(TRX)
TRXTsrp  = pd.DataFrame(data=TRXTsrp, index=iTR)

VLXTsrp  = Esrp.transform(VLX)
VLXTsrp  = pd.DataFrame(data=VLXTsrp, index=iVL)

print(TRXTsrp.shape)  # Esrp.n_components_)
scatterPlot(TRXTsrp, TRy, 'Sparse Random Projection')
```

```
CPU times: total: 15.6 ms
Wall time: 27 ms
(1000, 331)
```



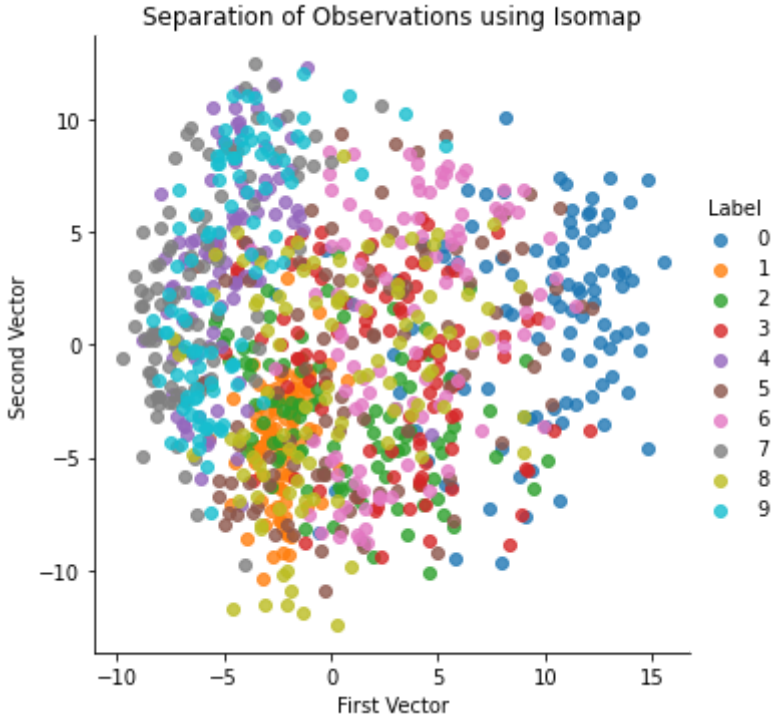
Isomap

- 지역성을 반영한 비선형 차원축소
- 원공간에서의 거리가 가급적 동일하게 유지되도록(quasi-isometric) 저차원으로 차원축소
- 지역 그래프로 표현된 측지거리(두 점간 최단경로의 선가중치의 합계)를 다차원척도법 또는 Kernel PCA로 차원축소함
- 초모수
  - `n_neighbors=5` : 각 점마다 고려할 인접이웃수
- 측지거리행렬이 필요하므로 오래 걸림
- `inverse_transform` 없음

In [51]:
```
# Isomap
from sklearn.manifold import Isomap

n_neighbors = 50     # 기본값 5
n_components = 100
n_jobs = 4

Eimap = Isomap(n_neighbors=n_neighbors,
               n_components=n_components,
               n_jobs=n_jobs)

%time Eimap.fit(TRX)   # Eimap.fit(TRX.loc[0:5000, :])
%time TRXTimap  = Eimap.transform(TRX)
TRXTimap  = pd.DataFrame(data=TRXTimap, index=iTR)

VLXTimap  = Eimap.transform(VLX)
VLXTimap  = pd.DataFrame(data=VLXTimap, index=iVL)

# frobenius_norm(K(D)-K(D_fit))/n
```

```
print(TRXTimap.shape)
scatterPlot(TRXTimap, TRy, 'Isomap')
```

```
CPU times: total: 3.19 s
Wall time: 1.33 s
CPU times: total: 1.34 s
Wall time: 232 ms
(1000, 100)
```



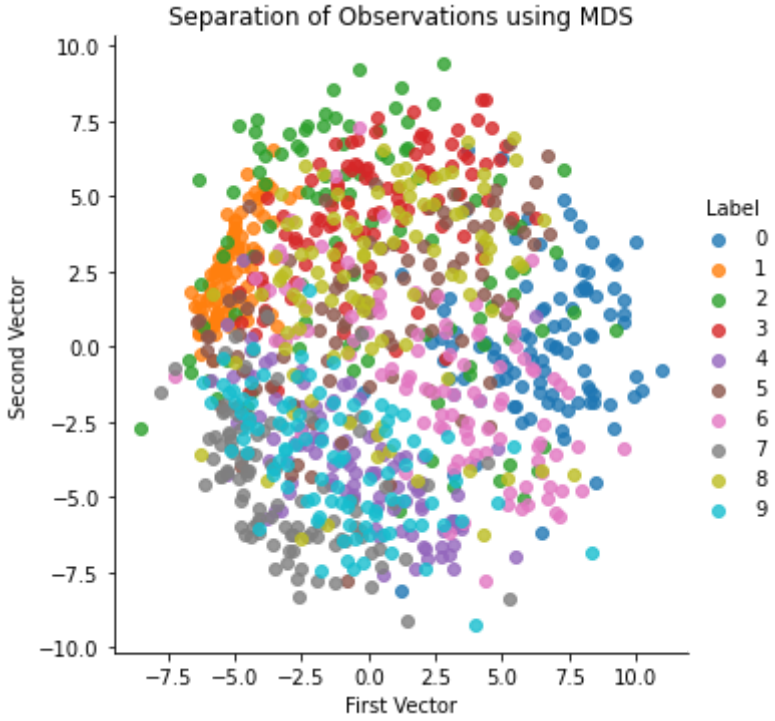
MDS

- 모든 점들간 거리행렬이 필요함. 일부 점만 사용해도 상당히 오래 걸림
- 모든 점들을 사용하면 메모리 때문에 오류. Unable to allocate 9.31 GiB for an array with shape (50000, 50000) and data type float32
- 별도의 `transform()` 이 없음(; 성분값을 구하려면 `fit_transform` 사용). VL이나 TS에 대해 `transform` 불가 (교재에도 없음)

In [52]:
```
# 다차원 스케일링
from sklearn.manifold import MDS

n_components = 3   # 차원수. 기본값=2
n_init = 12        # SMACOF 실행시 초기화 시도 횟수. 기본값=4
max_iter = 1200    # SMACOF 반복횟수 기본값=300.
metric = True       # metric MDS 여부. 기본값=True(metric), False(non-metric)
n_jobs = 4
random_state = 2018

Emds = MDS(n_components=n_components,
           n_init=n_init,
           max_iter=max_iter,
           metric=metric,
           n_jobs=n_jobs,
           random_state=random_state)

#TRXTmds  = Mmds.fit_transform(TRX)
#TRXTmds  = pd.DataFrame(data=TRXTmds, index=iTR)
```

```
%time TRXTmds  = Emds.fit_transform(TRX)         # Emds.fit_transform(TRX.loc[0:1000
TRXTmds  = pd.DataFrame(data=TRXTmds, index=iTR) # [0:1001])

print(TRXTmds.shape)
scatterPlot(TRXTmds, TRy, 'MDS')
```

```
CPU times: total: 469 ms
Wall time: 3min 14s
(1000, 3)
```



In [53]:
```
TRXTmds
```

Out[53]:

	0	1	2
1			
1907	8.044772	-0.767625	2.967085
5809	2.738215	0.305797	6.420265
32846	9.568431	1.542003	-0.440080
8838	8.323983	3.463551	-1.183085
13044	7.912080	1.899037	-3.627605
...	...	...	...
5459	-2.084064	-5.025164	3.359866
34983	-2.329189	-5.082287	2.092079
16622	-5.100073	-1.441358	-0.007218
32504	0.465715	-1.916147	-7.251204
27565	-2.394219	-3.072374	2.996577

1000 rows × 3 columns

In [54]:
```
TRXTmds.describe()
```

Out[54]:

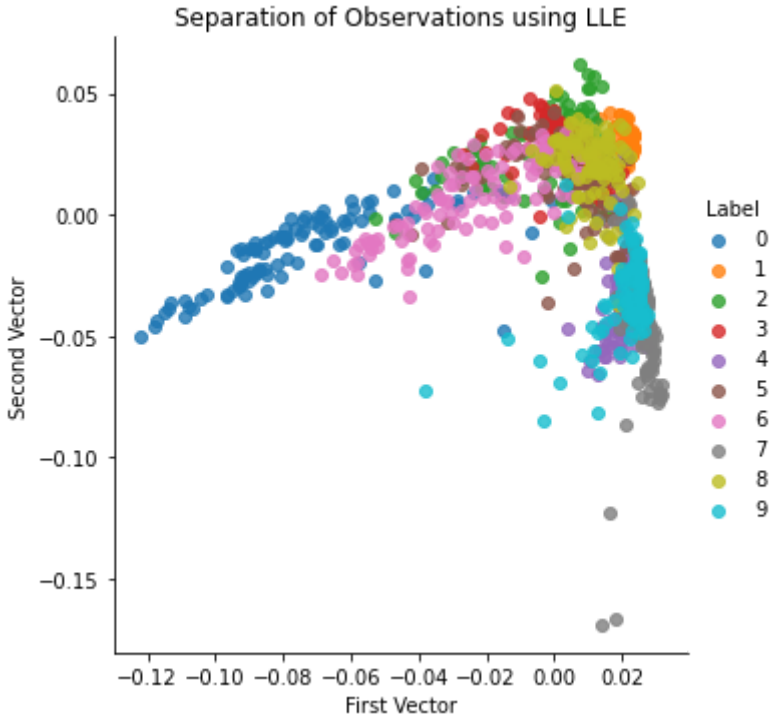
	0	1	2
count	1000.000000	1.000000e+03	1.000000e+03
mean	0.000000	1.136868e-16	7.105427e-18
std	4.133707	3.985912e+00	4.027532e+00
min	-8.560754	-9.228784e+00	-9.271709e+00
25%	-3.560996	-3.241589e+00	-3.123880e+00
50%	-0.481706	1.949082e-01	1.685357e-02
75%	3.132785	3.072825e+00	3.140393e+00
max	10.970958	9.404455e+00	9.174987e+00

LLE(Local Linear Embedding)

- 지역성을 반영한 비선형 차원축소 (Isomap과 유사)
- 모수
  - `n_neighbors=5` : 각 점마다 고려할 인접이웃수
  - `reg=1e-3` : 규제화 모수
  - `method='standard, hessian, modified, ltsa'` :
- 오래 걸림.
- `inverse_transform()` 없음

In [55]:
```
# 지역 선형 임베딩
from sklearn.manifold import LocallyLinearEmbedding

n_neighbors  = 10      # 기본값=5
n_components = 3      # 기본값=2
method = 'modified'   # 기본값='standard'
n_jobs = 4
random_state = 2018

Elle = LocallyLinearEmbedding(n_neighbors=n_neighbors,
                              n_components=n_components,
                              method=method,
                              random_state=random_state,
                              n_jobs=n_jobs)

%time Elle.fit(TRX)   # Elle.fit(TRX.loc[0:5000, :])
TRXTlle  = Elle.transform(TRX)
TRXTlle  = pd.DataFrame(data=TRXTlle, index=iTR)

VLXTlle  = Elle.transform(VLX)
VLXTlle  = pd.DataFrame(data=VLXTlle, index=iVL)

print(TRXTlle.shape)
scatterPlot(TRXTlle, TRy, 'LLE')
```

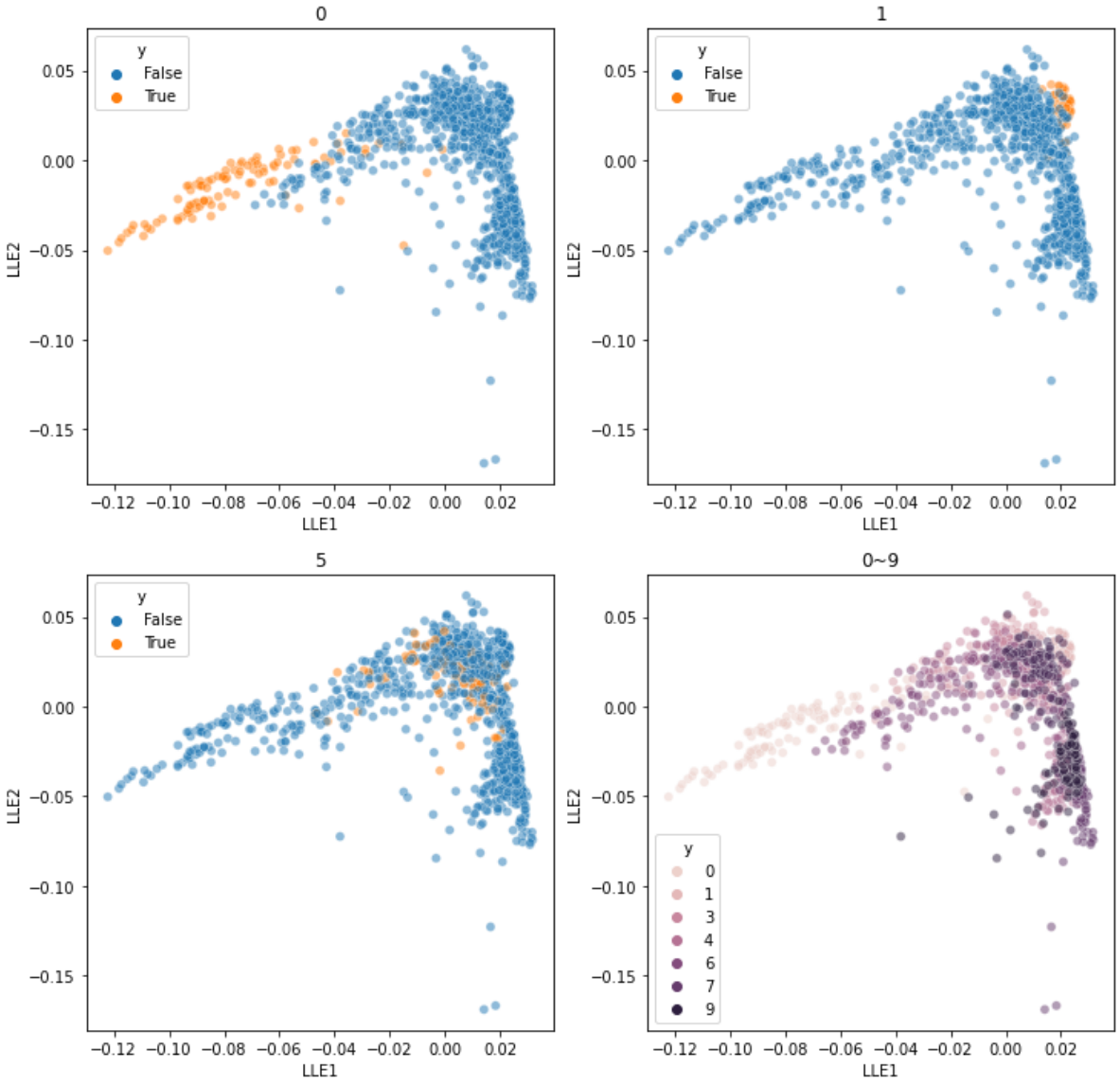
```
CPU times: total: 2.38 s
Wall time: 560 ms
(1000, 3)
```



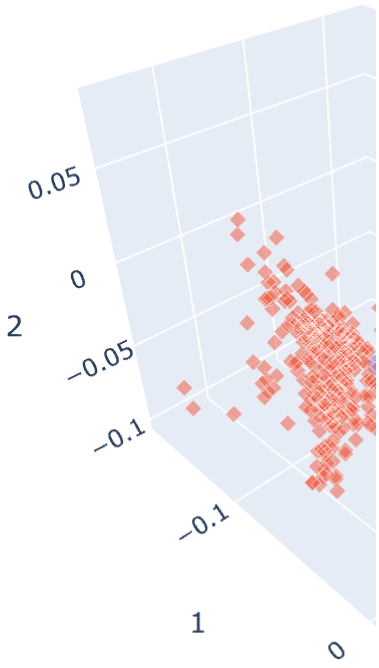
In [56]:
```
Elle.embedding_
```

Out[56]:
```
array([[-0.08459401, -0.01064842,  0.05350467],
       [-0.04761774,  0.01032803,  0.04513811],
       [-0.0902233 , -0.02674481,  0.01107022],
       ...,
       [ 0.02269126, -0.00960735,  0.01517015],
       [ 0.01088642, -0.04613278, -0.04219271],
       [ 0.02380921, -0.02091187,  0.02998032]])
```

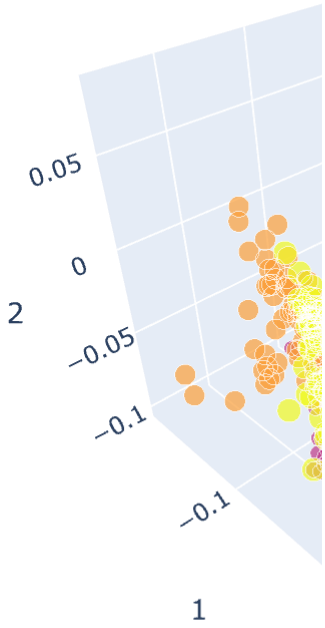
In [57]:
```
# y=0, y=1, y=5, y에 대해 성분값 산점도
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(12,12))
d0 = sns.scatterplot(x=TRXTlle.loc[:,0], y=TRXTlle.loc[:,1], hue=TRy==0, alpha=0.5,
d0.set(xlabel='LLE1', ylabel='LLE2', title='0')
d1 = sns.scatterplot(x=TRXTlle.loc[:,0], y=TRXTlle.loc[:,1], hue=TRy==1, alpha=0.5,
d1.set(xlabel='LLE1', ylabel='LLE2', title='1')
d5 = sns.scatterplot(x=TRXTlle.loc[:,0], y=TRXTlle.loc[:,1], hue=TRy==5, alpha=0.5,
d5.set(xlabel='LLE1', ylabel='LLE2', title='5')
dd = sns.scatterplot(x=TRXTlle.loc[:,0], y=TRXTlle.loc[:,1], hue=TRy, alpha=0.5, ax
dd.set(xlabel='LLE1', ylabel='LLE2', title='0~9');
```



In [58]:
```
# y=0
fig = px.scatter_3d(TRXTlle, x=0, y=1, z=2,
                    color=TRy==0,     #
                    symbol=TRy==0,
                    size=1+(TRy==0).astype(int),
                    size_max=9,
                    opacity=0.5) #, size=0.1)
fig.show()
```



In [59]:
```
# import plotly.express as px
# 처음 3개의 LLE 성분 점수에 대해 3D 산점도
fig = px.scatter_3d(TRXTlle, x=0, y=1, z=2, color=TRy, size=TRy,  size_max=10) #, o
fig.show()
```



t-SNE(t-Stochastic Neighbor Embedding)

- 고차원 자료의 점간 유사성을 유지되도록 저차원에 표현 (고차원과 저차원의 분포간 Kullback-Leibler 거리를 최소화)
- 점간 거리와 분포를 계산하므로 느림. 실행마다 결과가 다를 수 있음. 주로 고차원자료의 시각화에 사용
- p가 너무 크면(30~50 이상) 적용이 어려움. PCA로 구한 주성분점수에 대해 적용하면 효율적
- `inverse_transform()` 불가. `transform()` 없음(; 자료마다 모델을 적용해야 함)
- How to tune hyperparameters of tSNE
- How to Use t-SNE Effectively

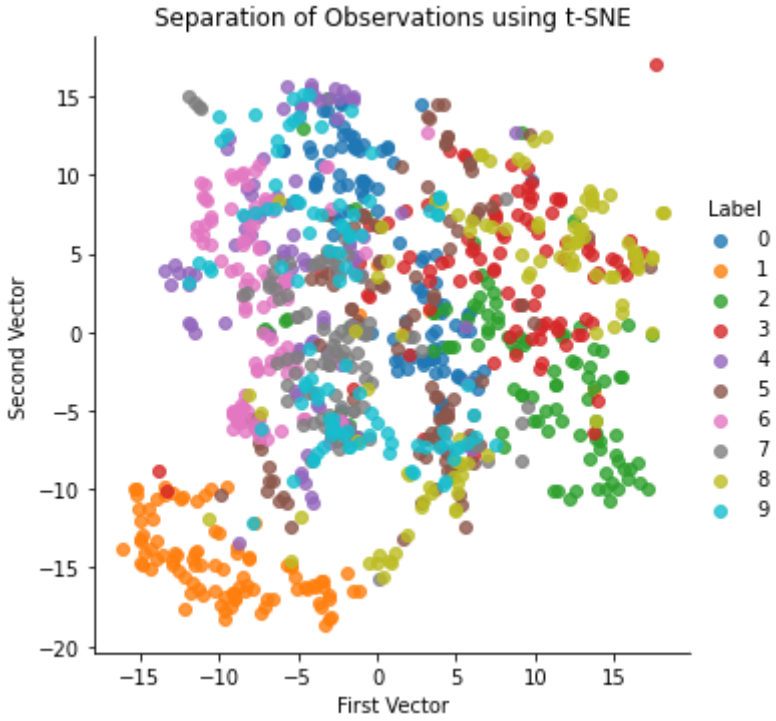
In [60]:
```
TRX.iloc[:10,:9]
```

Out[60]:

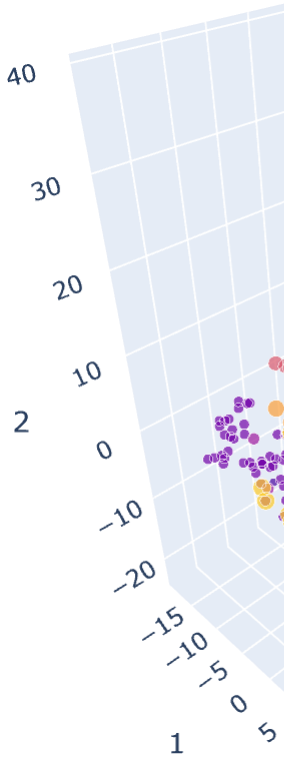
	x01	x02	x03	x04	x05	x06	x07	x08	x09
1									
1907	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5809	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
32846	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
8838	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
13044	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
6468	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3016	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5342	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4852	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7183	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

In [61]:
```
# t-분포 확률적 임베딩
from sklearn.manifold import TSNE

n_components = 3      # 저차원 차원수. 기본값=2
learning_rate = 300   # 기본값=200. [10(과적합)~1000(저적합)]
perplexity = 10       # 기본값=30. 보통 5-50사이 시도. 인접이웃수 함수. n이 클수록 큰
early_exaggeration = 10 # 값이 클수록 저차원 공간내 군집간 공간이 넓어짐. 최종 결과0
init = 'random'       # 초기값 지정. 'random, pca'
random_state = 2018

Etsne = TSNE(n_components=n_components,
             learning_rate=learning_rate,
             perplexity=perplexity,
             early_exaggeration=early_exaggeration,
             init=init,
             random_state=random_state)


# 9개 주성분 점수로 계산
%time TRXTtsne  = Etsne.fit_transform(TRXTpca.iloc[:, :9])
TRXTtsne  = pd.DataFrame(data=TRXTtsne, index=iTR)

print(TRXTtsne.shape)
scatterPlot(TRXTtsne, TRy, 't-SNE')
```

```
CPU times: total: 1min
Wall time: 8.42 s
(1000, 3)
```

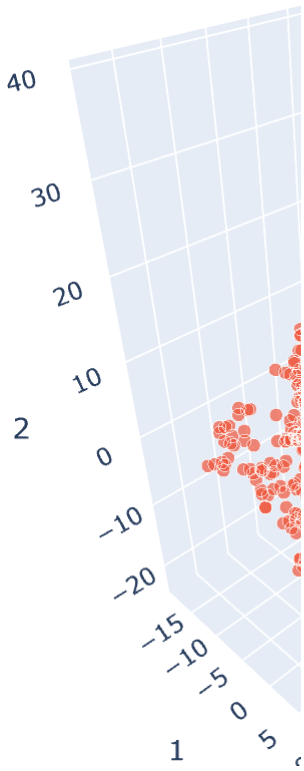


In [62]:
```
# LLE성분1~3
TMPy = TRy.loc[TRXTtsne.index]
fig = px.scatter_3d(TRXTtsne, x=0, y=1, z=2, color=TMPy, size=TMPy+1, size_max=8)
fig.show()
```



```
# LLE성분1~3. 숫자 0 기준
```

In [63]:
```
fig = px.scatter_3d(TRXTtsne, x=0, y=1, z=2, color=(TRy==0), size=(TRy==0)+1, size_
fig.show()
```



MiniBatch Dictionary Learning

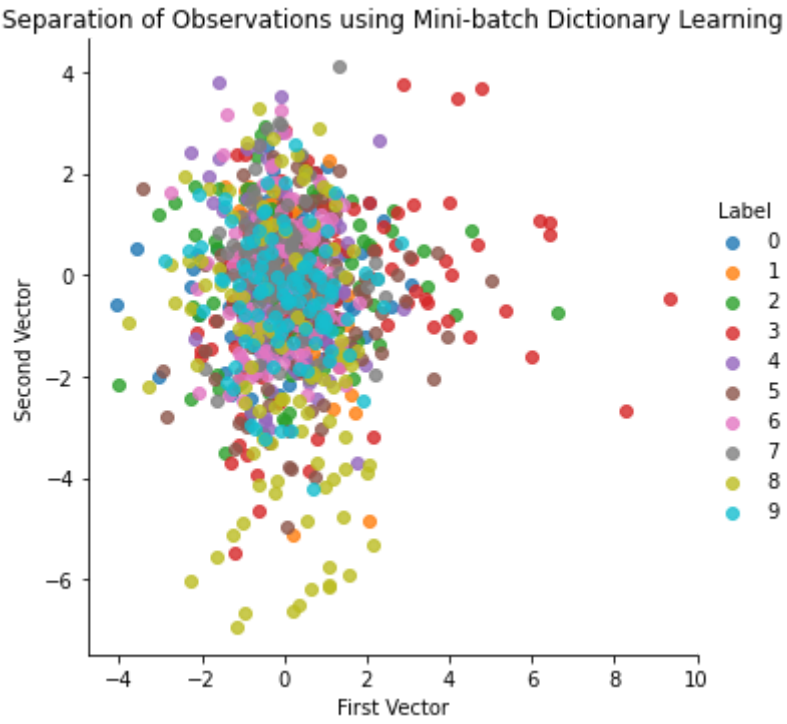
- X를 잘 표현할 수 있는 사전(딕셔너리, 대표점들의 집합)을 생성. 희소 표현을 학습
- 대표점들은 0과 1로만 구성된 벡터. 원 점들은 이런 사전내 대표점들의 선형결합으로 재구성
- 대표점의 차원이 d, 대표점 개수가 m일때
  - m<d이면 과소완전 (undercomplete) 사전. 차원축소에 해당
  - m>d이면 과대완전 (overcomplete) 사전
- 오래 걸림. `batch_size`를 작게하면 빨라짐

In [64]:
```
# 미니-배치 사전 학습
from sklearn.decomposition import MiniBatchDictionaryLearning

n_components = 50  # 딕셔너리에 포함될 원소의 수
alpha = 1          # 희소성 조절모수. 기본값=1.
batch_size = 3     # 미니배치에 포함될 점의 수. 기본값=3
n_iter = 1000      # 기본값=1000
random_state = 2018

Edl = MiniBatchDictionaryLearning(
        n_components=n_components,
```

```
        alpha=alpha,
        batch_size=batch_size,
        n_iter=n_iter,
        random_state=random_state)

# EdI.fit(TRX.loc[:, :100]). # Not working
%time TRXTdI = EdI.fit_transform(TRX)
TRXTdI = pd.DataFrame(data=TRXTdI, index=iTR)

VLXTdI = EdI.transform(VLX)
VLXTdI = pd.DataFrame(data=VLXTdI, index=iVL)

print(TRXTdI.shape)
scatterPlot(TRXTdI, TRy, 'Mini-batch Dictionary Learning')
```

```
C:₩Users₩BigD₩anaconda3₩envs₩tf29₩lib₩site-packages₩sklearn₩decomposition₩_dict_lear
ning.py:2314: FutureWarning:

'n_iter' is deprecated in version 1.1 and will be removed in version 1.3. Use 'max_i
ter' instead.
```

```
CPU times: total: 39 s
Wall time: 11.3 s
(1000, 50)
```



ICA(Independent Component Analysis)

- 독립성분추출: PCA는 서로 상관관계가 0인 성분(first-order)을 추출. ICA는 독립성분(second-order)을 추출. Blind source separation(Cocktail party problem)
- `inverse_transform()` 가능

In [65]:
```
# 독립 성분 분석
from sklearn.decomposition import FastICA

n_components = 20       # 추출할 성분수. 기본값=None (모든 변수)
algorithm = 'parallel'  # parallel, deflation
whiten = True           # 공분산행렬을 항등행렬화(Whitening)할 지 여부. 기본값=True
```

```
max_iter = 1000          # 기본값=200
tol = 1e-4               # 기본값 1e-4
random_state = 2018

Eica = FastICA(n_components=n_components,
               algorithm=algorithm,
               whiten=whiten,
               max_iter=max_iter,
               tol =
               random_state=random_state)

%time Eica.fit(TRX)
TRXTica = Eica.fit_transform(TRX)
TRXTica = pd.DataFrame(data=TRXTica, index=iTR)

VLXTica = Eica.transform(VLX)
VLXTica = pd.DataFrame(data=VLXTica, index=iVL)

scatterPlot(TRXTica, TRy, 'Independent Component Analysis')
```

```
  Input In [65]
    random_state=random_state)
                 ^
SyntaxError: invalid syntax
```

In [ ]:
```
# ICA성분1~3. 숫자 0 기준
fig = px.scatter_3d(TRXTica, x=0, y=1, z=2, color=(TRy==1), size=(TRy==1)+1, size_r
fig.show()
```

In [ ]:
```
print("--- %s seconds ---" % (time.time() - start_time))
```

- Note: All have `fit, fit_transform, transform, get_params, set_params`
- `sklearn.decomposition`

Name	Purpose	All have fit, fit_transform, transform, get_params, set_params
`DictionaryLearning([…])`	Dictionary learning	No `inverse` . `components_` :(K,p)
`MiniBatchDictionaryLearning([…])	Mini-batch dictionary learning	No inverse. `components_` :(K,p)
`PCA([n_comp, copy, …])`	Principal component analysis (PCA)	`inverse_transform. components_, explained_variance_, explained_variance_ratio_, singular_values_, n_comp_`
`IncrementalPCA([n_comp, …])`	Incremental PCA (IPCA)	`inverse_transform` . same as PCA
`SparsePCA([n_comp, …])`	Sparse PCA (SparsePCA).	No `inverse` . `components_` , `n_comp`
`KernelPCA([n_comp, …, fit_inverse_transform=False])`	Kernel Principal component analysis (KPCA)	`inverse_transform()` . `lambdas_` , alphas_,
`FastICA([n_comp, …])`	FastICA: a fast algorithm for Independent Component Analysis	`inverse_transform()` . `components_`

Name	Purpose	All have fit, fit_transform, transform, get_params, set_params
`TruncatedSVD([n_comp, …])`	Dimensionality reduction using truncated SVD (aka LSA)	`inverse_transform()` . PCA와 유사
`FactorAnalysis([n_comp, …])`	Factor Analysis (FA).	No inverse. `components_` , `noise_variance_`
`LatentDirichletAllocation([…])`	Latent Dirichlet Allocation with online variational	No inverse
`MiniBatchSparsePCA([…])	Mini-batch Sparse PCA	No inverse. `components_` , `n_comp_`
`NMF([n_comp, init, …])`	Non-Negative Matrix Factorization (NMF).	`inverse_transform()` . `components_` , `n_comp_` , `reconstruction_err_`
`SparseCoder(dictionary, *[, …])`	Sparse coding	

- sklearn.random_projection.

Name	Purpose	No inverse_transform. All n*comp*, components_
`GaussianRandomProjection([…])`	Reduce dimensionality through Gaussian random projection.	`n_comp_` , `components_`
`SparseRandomProjection([…])`	Reduce dimensionality through sparse random projection.	`n_comp_` , `components_`

- sklearn.manifold

Name	Purpose	No inverse*transform. embedding* for all
`Isomap([, n_neighbors, …])`	Isomap Embedding	`embedding_, reconstruction_error, kernel_pca_, nbrs_, dist_matrix_`
`LocallyLinearEmbedding([, …])`	Locally Linear Embedding	`embedding_, reconstruction_error`
`MDS([n_comp, metric, n_init, …])`	Multidimensional scaling.	`embedding_, stress_, dissimilarity_matrix_`
`SpectralEmbedding([n_comp, …])`	Spectral embedding for non-linear dimensionality reduction.	`embedding_, affinity_matrix_, n_neighbors_`
`TSNE([n_comp, perplexity, …])`	t-distributed Stochastic Neighbor Embedding.	`embedding_, kl_divergence`

In [ ]: