practical_5

June 29, 2021

1 Practical 5 - Dimensionality Reduction

[3]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.
 ↪set(context='talk',style='white',palette='colorblind')
import pickle
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
import scipy as sp
import copy
import pandas as pd
```

1.1 Task 0: Load and normalize count data

This practical uses the data set from https://www.nature.com/articles/s41586-018-0654-5. This is single cell transcriptomics data from ~25,000 cells from the cortex.

For each of these cells, the expression of several thousand genes was measured `['counts']`. In the original study, the authors were interested in clustering the cells into types.

We made a selection of 5000 cells and the 1000 most informative genes for run time reasons. We provide you with the original cell type labels determined by the authors for comparison `['clusters']`.

The following function will apply some preprocessing steps that are standard for transcriptomics data.

[5]:
```
def lognormalize_counts(tasic_dict):

    # normalize and logtransform counts
    counts = tasic_dict['counts']
    libsizes = counts.sum(axis=1)
    CPM = counts / libsizes * 1e+6
    logCPM = np.log2(CPM + 1)
    tasic_dict['logCPM'] = np.array(logCPM)

    return tasic_dict
```

[6]:
```
tasic_1k = lognormalize_counts(pickle.load(open('data/tasic_subset_1kselected.
↪pickle', 'rb')))
```

Have a look at `['counts']`, `['logPCM']` and `['clusters']` to get a better understanding of the data. Plot a histogram of the cell type labels provided by `['clusters']`.

[7]:
```
# Explore data
tasic_1k['counts']
```

[7]:
```
<5000x1000 sparse matrix of type '<class 'numpy.float64'>'
        with 937979 stored elements in Compressed Sparse Column format>
```

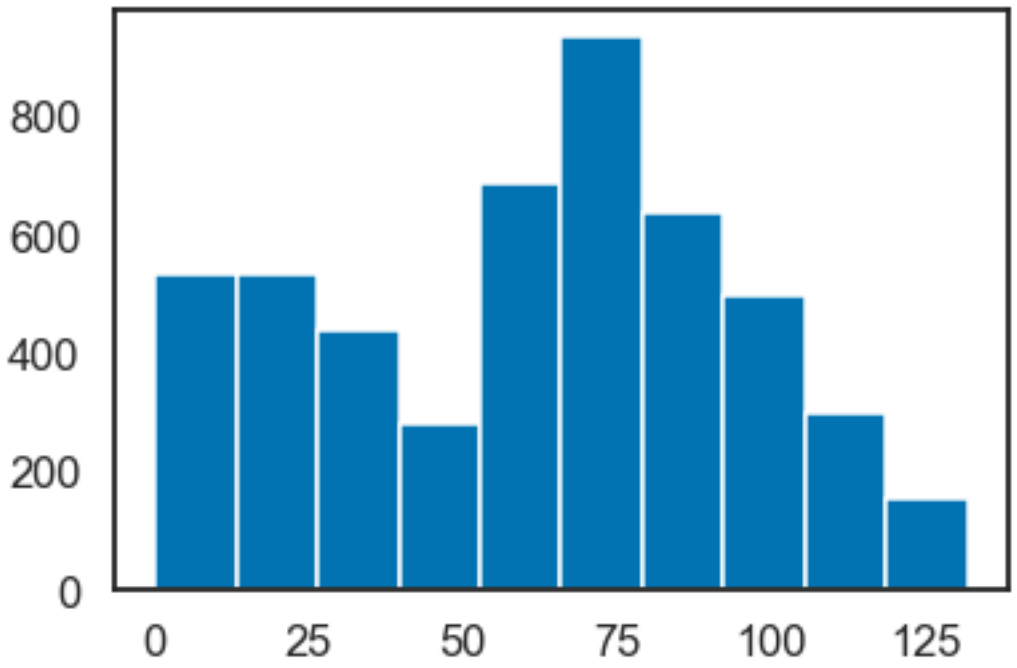
[8]:
```
tasic_1k['logCPM']
```

[8]:
```
array([[ 0.        ,  0.        ,  0.        , …,  0.        ,
         0.        , 14.48400138],
       [ 0.        ,  0.        ,  0.        , …,  0.        ,
         0.        , 14.74268238],
       [ 0.        ,  0.        ,  0.        , …,  0.        ,
         0.        , 14.86580424],
       …,
       [ 0.        ,  0.        ,  0.        , …,  0.        ,
        14.08885484,  2.79928049],
       [ 0.        ,  0.        ,  0.        , …,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , …,  0.        ,
         0.        ,  0.        ]])
```

[9]:
```
np.unique(tasic_1k['clusters'])
```

[9]:
```
array([  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,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,
        52,  53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,
        65,  66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,
        78,  79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,
        91,  92,  93,  94,  95,  96,  97,  98,  99, 100, 101, 102, 103,
       104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116,
       117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129,
       130, 131, 132])
```

[10]:
```
# Plotting
plt.hist(tasic_1k['clusters'])
plt.show()
```



1.2 Task 1: Linear dimensionality reduction with PCA

In this task, you will use Principal Component Analysis (PCA) to reduce the dimensionality of the dataset.

First, implement PCA "by hand". You can use eigenvalue/singular value decomposition from numpy/scipy but no `sklearn`-functions. Write a function that computes all possible principal components and returns them along with the fraction of variance they explain.

[11]:
```
def PCA_manual(_data):
    '''
    Function that performs PCA on the input data

    input: (cells, genes)-shaped array of log transformed cell counts
    output:
        fraction_variance_explained: (genes,)-shaped array with the fraction of␣
↪variance explained by the individual PCs
        principal_components: (genes, genes)-shaped array containing the␣
↪principal components as columns
    '''
    ### NOTE: Make sure the function returns the PCs sorted by the fraction of␣
↪variance explained! ###
    ###       (First column of principal_components should hold the PC with the␣
↪highest variance   ###
```

```
    ###       explained -- fraction_variance_explained should also be sorted␣
↪accordingly)          ###

    # ---------------- INSERT CODE ----------------------
    # Step 1: Centering. Setting column (gene) mean to zero.
    data = copy.deepcopy(_data)
    centered = data - np.mean(data.T, axis=1)

    # Step 2: Covariance matrix
    C = np.cov(centered.T)

    # Step 3: Eigenvalue decomposition
    values, vectors = np.linalg.eig(C)

    # Step 4: Sort the values and vectors decending (argsort sorts ascending)
    idx = np.argsort(values)[::-1]

    # Step 5: Return (eigenvalues are 'fraction_variance_explained',␣
↪eigenvectors are 'principal_components')
    principal_components = vectors.T[idx]
    fraction_variance_explained = values[idx]

    # Checkpoint.
    #print(C - vectors.dot(np.diag(values)).dot(vectors.T))

    # ---------------- END CODE -------------------------

    return fraction_variance_explained, principal_components
```

[12]:
```
var_expl, PCs = PCA_manual(tasic_1k['logCPM'])
```

Before we explore the structure of the low-dimensional representation, we first want to know how much variance the first PCs explain:

- Plot the fraction of variance explained by the `n`-th PC vs. `n`
- Plot the cumulative fraction of variance explained by the first `n` PCs with largest eigenvalue vs. `n`

From the latter plot you should be able to see how many PCs you need to keep to explain at least `x`% of the variance.

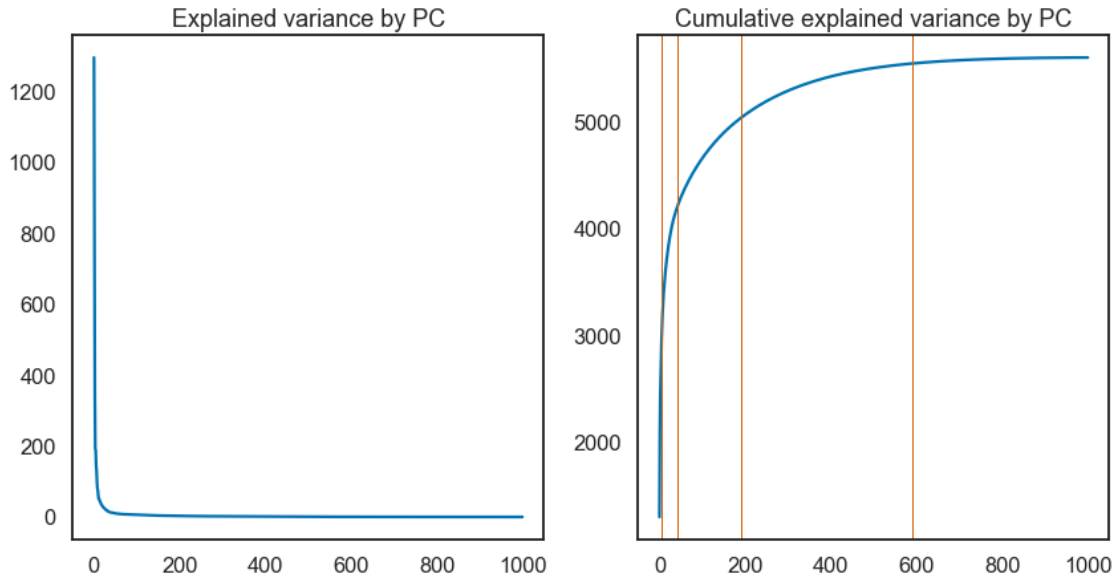
How many components do you need to keep to explain 50%, 75%, 90% and 99%, respectively? Indicate this in your plot.

[13]:
```
n_PCs = len(var_expl)
PC_ids = np.arange(1, n_PCs+1)

plt.figure(figsize=(14, 7))
```

```
plt.subplot(121)

# Plot the variance explained of the n-th PC vs. n
# ---------------- INSERT CODE ----------------------

X = np.arange(len(PCs))
Y = var_expl[X]

plt.plot(X, Y)
plt.title("Explained variance by PC")

# ---------------- END CODE -------------------------


plt.subplot(122)

# Plot the cumulative variance explained for the n PCs with highest variance␣
 ↪explained vs. n
# Indicate how many components you need to keep to explain 50%, 75%, 90% and␣
 ↪99% in the plot.
# ---------------- INSERT CODE ----------------------

Y = np.cumsum(var_expl[X])

plt.plot(X, Y)
plt.title("Cumulative explained variance by PC")

total = Y[-1]
above50 = np.flatnonzero(Y > 0.5  * total)[0]
above75 = np.flatnonzero(Y > 0.75 * total)[0]
above90 = np.flatnonzero(Y > 0.9  * total)[0]
above99 = np.flatnonzero(Y > 0.99 * total)[0]

plt.axvline(above50, linewidth=1, color='r', label='50%')
plt.axvline(above75, linewidth=1, color='r', label='75%')
plt.axvline(above90, linewidth=1, color='r', label='90%')
plt.axvline(above99, linewidth=1, color='r', label='99%')

print("To explain 50% of the variance, you would need the first {:>3} principal␣
 ↪components".format(above50 + 1))
print("To explain 75% of the variance, you would need the first {:>3} principal␣
 ↪components".format(above75 + 1))
print("To explain 90% of the variance, you would need the first {:>3} principal␣
 ↪components".format(above90 + 1))
print("To explain 99% of the variance, you would need the first {:>3} principal␣
 ↪components".format(above99 + 1))
```

```
# ---------------- END CODE -------------------------
```

```
To explain 50% of the variance, you would need the first   5 principal
components
To explain 75% of the variance, you would need the first  43 principal
components
To explain 90% of the variance, you would need the first 193 principal
components
To explain 99% of the variance, you would need the first 590 principal
components
```



YOUR ANSWER HERE

Write a function to select the `n` PCs needed to explain at least `x`% of the variance and use this function to extract as many PCs as are needed to explain 75% of the variance.

[14]:
```
def select_PCs(
    variance_explained, principal_components, percent_variance=None):
    '''Function that selects the first n principal components necessary to␣
↪explain x% of the variance
    input:
        variance_explained: amount of variance explained by the individual PCs
        principal_components: contains the principal components as columns
        percent_variance: fraction of the variance, the all PCs that are kept␣
↪explain
    output:
```

```
        variance_explained_kept: individual amount of variance explained for␣
↪the remaining PCs
        principal_components_kept: remaining principal components, shape␣
↪(genes,n_PCs_kept)
    '''

    # ---------------- INSERT CODE ----------------------
    if(percent_variance is None):
        percent_variance = 0.75

    X = np.arange(len(principal_components))
    Y = np.cumsum(variance_explained[X])

    total = Y[-1]
    split = np.flatnonzero(Y > percent_variance  * total)[0]

    variance_explained_kept = variance_explained[0:split+1] # +1 because 'at␣
↪least'
    principal_components_kept = principal_components[0:split+1]

    # ---------------- END CODE ------------------------

    return variance_explained_kept, principal_components_kept
```

[15]:
```
_, PCs75 = select_PCs(var_expl, PCs, percent_variance=0.75)
```

To compute the representation of the data in this lower dimensional representation, write a function that compute the PC scores for each cell, i.e. that projects the original data matrix on the low-dimensional subspace provided by the first `n` PCs:

[16]:
```
def compute_PCA_scores(data, principal_components):
    '''Function that returns the PC scores for each data point
    input:
        data                --- (cells, genes)-shaped array of log transformed␣
↪cell counts
        principal_components --- contains the principal components as columns
    output:
        pc_scores           --- (cells, n_PCs_kept)-shaped array of PC scores
    '''

    # ---------------- INSERT CODE ----------------------

    pc_scores = principal_components.dot(data.T).T


    # ---------------- END CODE ------------------------
```

```
    return pc_scores
```

[17]:
```
tasic_1k['PCA_75'] = compute_PCA_scores(tasic_1k['logCPM'], PCs75)
```

Visualize the top 5 PCs as a pairwise scatterplot. Use one subplot for each pair of components.

Use the colors provided in `tasic_1k['clusterColors']` and the cluster information in `tasic_1k['clusters']` to color each data point according to its original cluster identity.

The colors indicate the family of the cell type:

- greenish colors: excitatory neurons
- orange colors: somatostatin positive interneurons
- pinkish colors: VIP-postive interneurons
- reddish colors: parvalbumin positive interneurons
- dark colors: non-neurons (glia etc)

What do you observe?

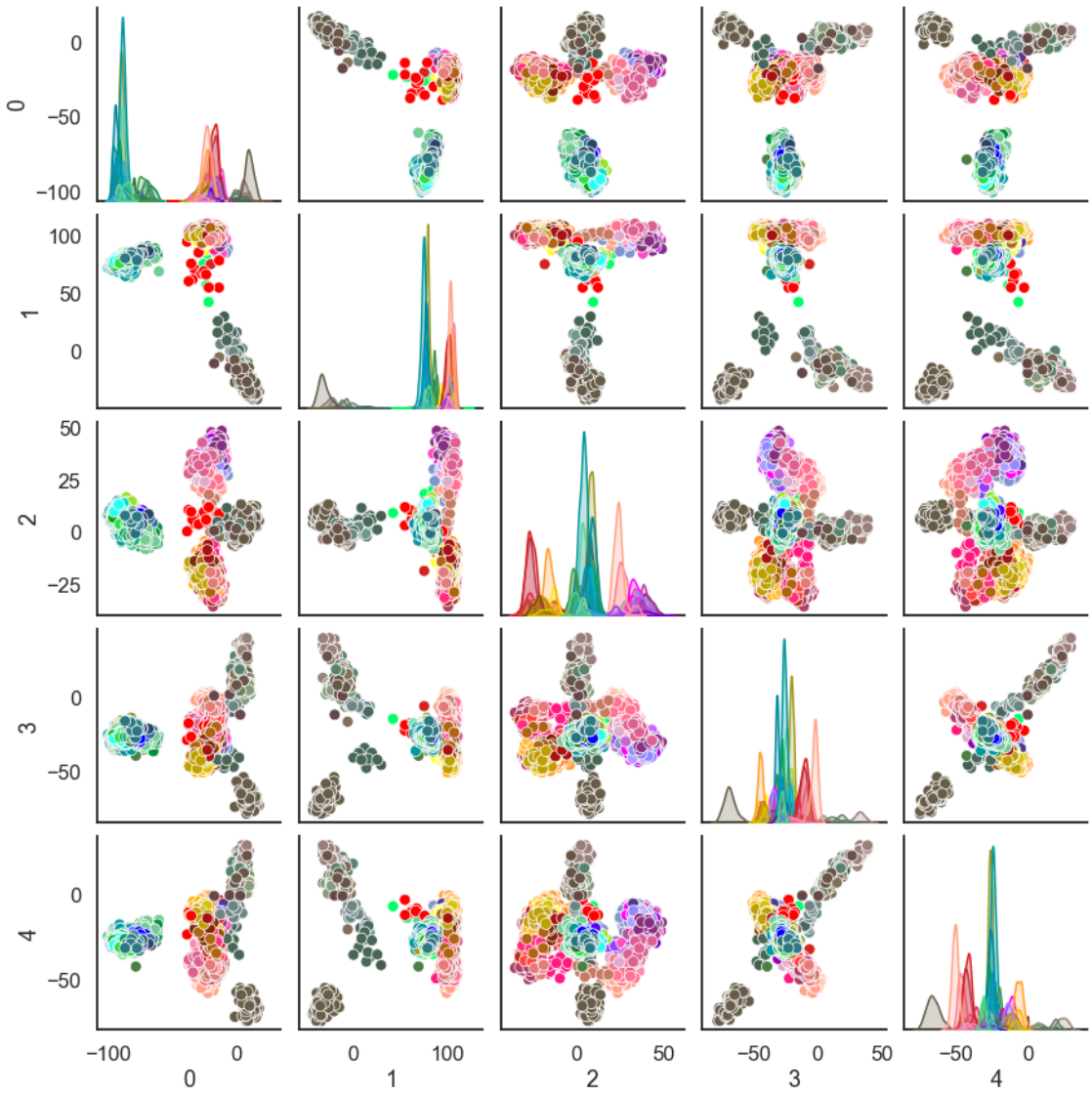
[18]:
```
def plot_PCs(data_transformed, color_per_datapoint):
    '''Function that plots the scores of the 10 pairs of the top 5 PCs against␣
↪each other.
        inputs:
            data_transformed    -- (cells, n_PCs_kept)-shaped array of PC scores
            color_per_datapoint -- (cells,)-shaped array of color strings, one␣
↪color for each cell
    '''

    # ---------------- INSERT CODE ----------------------

    x = data_transformed[:, :5]
    colors = dict(zip(color_per_datapoint, color_per_datapoint))

    df = pd.DataFrame(x)
    df.insert(5, 'color', color_per_datapoint, True)

    g = sns.pairplot(df, hue='color', palette=colors)
    g._legend.remove()


    # ---------------- END CODE ------------------------
```

[19]:
```
color_per_datapoint = tasic_1k['clusterColors'][tasic_1k['clusters']]
plot_PCs(tasic_1k['PCA_75'], color_per_datapoint)
```

```
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
```

```
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
```

```
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
/Users/Nathanael/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:306: UserWarning: Dataset has 0 variance;
skipping density estimate.
  warnings.warn(msg, UserWarning)
```



YOUR ANSWER HERE

1.3 Task 2: Comparison with PCA implemented by sklearn

Use the PCA implementation of sklearn to check whether your PCA implementation is correct and obtain some insights into numerical precision of the algorithms underlying PCA implementations.

[20]:
```
def PCA_sklearn(data):
    '''
    Function that performs PCA on the input data, using sklearn

    input: (cells, genes)-shaped array of log transformed cell counts
    output:
        fraction_variance_explained: (genes,)-shaped array with the fraction of␣
↪variance explained by the individual PCs
        principal_components: (genes, genes)-shaped array containing the␣
↪principal components as columns
    '''

    # ---------------- INSERT CODE ----------------------
    pca = PCA().fit(data)
    fraction_variance_explained = pca.explained_variance_
    principal_components = pca.components_


    # ---------------- END CODE -------------------------

    return fraction_variance_explained, principal_components
```

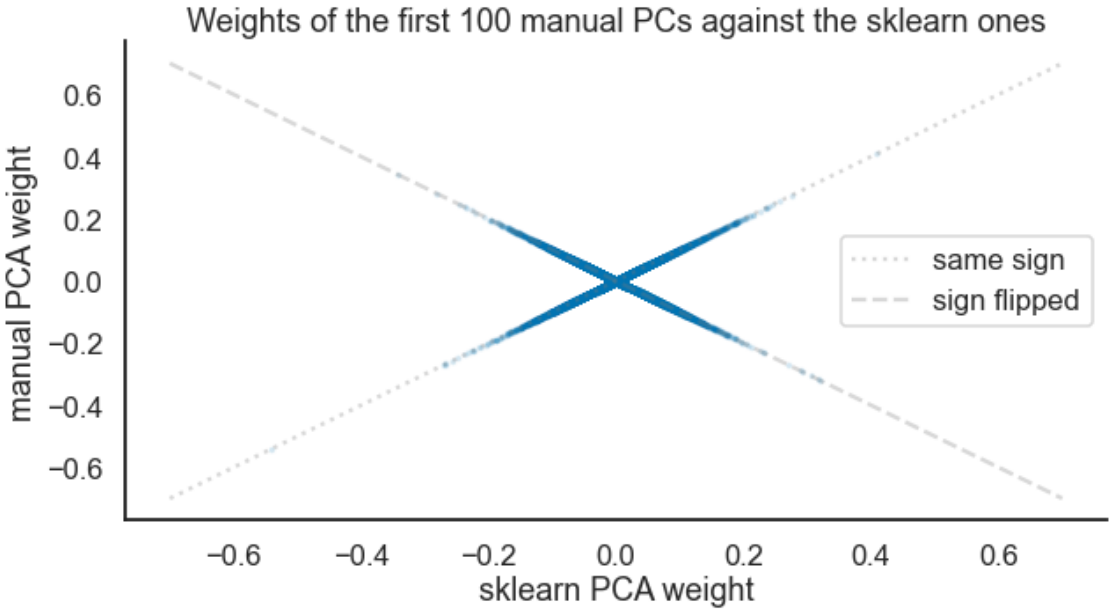
[21]:
```
# do sklearn-PCA on selected genes
var_expl_sklearn, PCs_sklearn = PCA_sklearn(tasic_1k['logCPM'])
# select components as before
_, PCs_sklearn75 = select_PCs(var_expl_sklearn, PCs_sklearn, 0.75)
# get PC scores
PCA_75_sklearn = compute_PCA_scores(tasic_1k['logCPM'], PCs_sklearn75)
```

To see if your manual PCA yielded the same PC weights as the sklearn PCA, we can just take the two matrices of principal components and plot their entries against each other. (Note: This again assumes they are sorted by variance explained and the order of dimensions in your weight matrix compared to the sklearn weight matrix is the same (change if necessary).)

Use the following plot to compare the results to your own implementation (here plotting the weights of the first 100 PCs against each other). What do you observe?

[22]:
```
n_evs_to_compare = 100

plt.figure(figsize=(10, 5))
plt.scatter(PCs_sklearn[:, :n_evs_to_compare].flatten(),
            PCs[:, :n_evs_to_compare].flatten(), s=5, alpha=0.1)
plt.plot([-.7, .7], [-.7, .7], ':', c='tab:gray', label='same sign', alpha=0.3)
```

```
plt.plot([-.7, .7], [.7, -.7], '--', c='tab:gray', label='sign flipped',␣
 ↪alpha=0.3)
plt.legend()
plt.xlabel('sklearn PCA weight')
plt.ylabel('manual PCA weight')
plt.title('Weights of the first %u manual PCs against the sklearn ones' %␣
 ↪(n_evs_to_compare))

sns.despine()
```



YOUR ANSWER HERE

Additional reading about the sign of PCs: https://stats.stackexchange.com/questions/88880/does-the-sign-of-scores-or-of-loadings-in-pca-or-fa-have-a-meaning-may-i-revers

1.4 Task 3: Nonlinear dimensionality reduction with t-SNE

In this task, you will use the nonlinear dimensionality reduction technique tSNE and look at visualizations of the data set. Plot the result of default t-SNE with the original cluster colors. For this and the following tasks, use the PCs explaining 75% of the variance `PCA_75_sklearn` you computed above.

[23]:
```
def plot_tsne(tsne_results, clusters=tasic_1k['clusters'], labels=['']):
    '''Plotting function for tsne results, creates one or multiple plots of␣
 ↪tSNE-transformed data.
       If the clustering is the original one (default), original cluster colors␣
 ↪will be used. Otherwise,
```

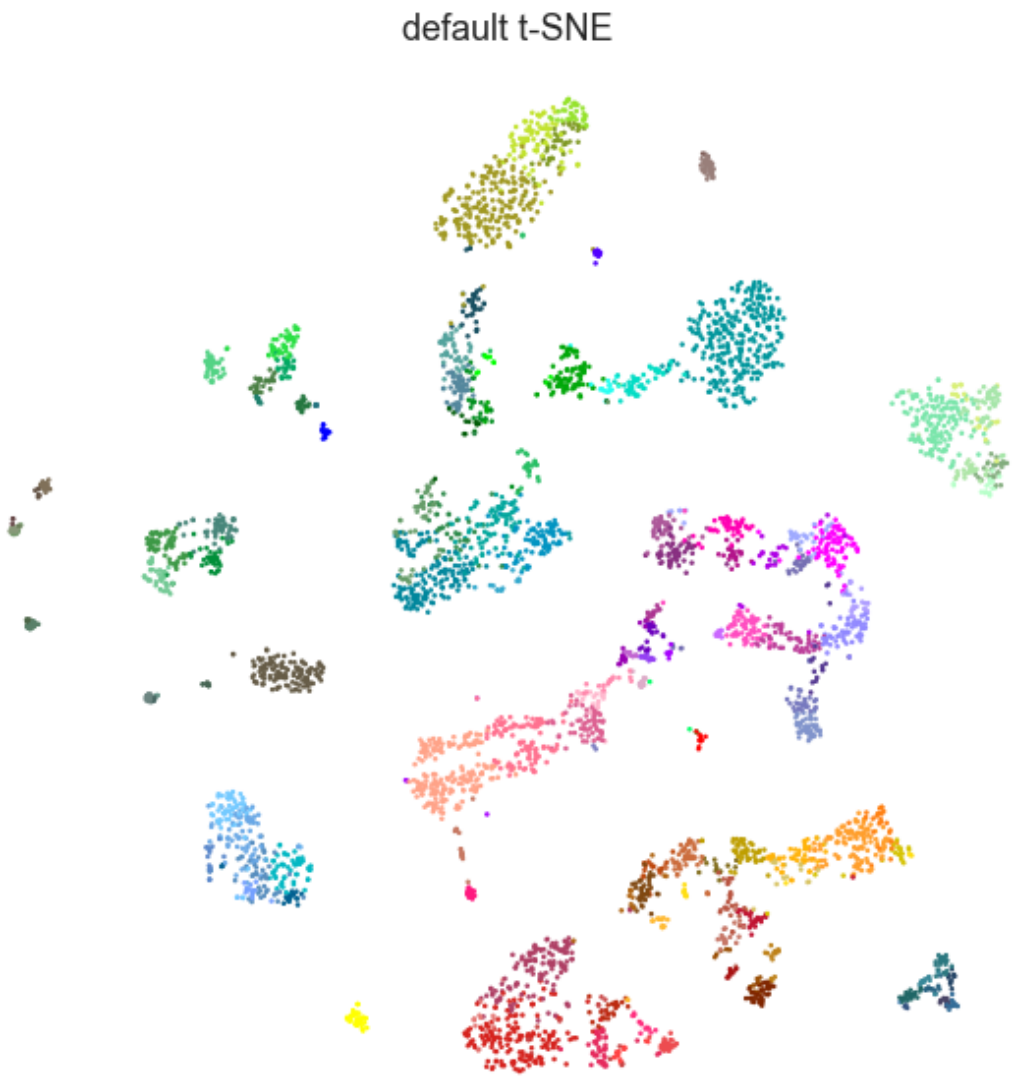
```
      colors will be a random permutation.

   input:
       tsne_results: (n, 2)-shaped array containing tSNE-transformed data or␣
↪list of such arrays
                     (output of the fit_transform function of sklearn tSNE)
       clusters: (n,)-shaped array containing cluster labels or list of such␣
↪arrays
       labels: optional, list of titles for the subplots
   '''

   if type(tsne_results) == list:  # make sure we can do both single and␣
↪multiple plots and are flexible regarding input
       num_plots = len(tsne_results)
   else:
       num_plots = 1
       tsne_results = [tsne_results]
   if type(clusters) == list:
       num_clusters = len(clusters)
       num_plots = num_plots * num_clusters
       tsne_results = tsne_results * num_clusters
   else:
       clusters = [clusters] * num_plots

   if len(labels) == 1:
       labels = labels * num_plots

   n_clusters = len(np.unique(clusters))      # ensure a long enough color␣
↪list even if we plot more than
   n_colors = len(tasic_1k['clusterColors'])  # the original number of clusters
   if n_clusters > n_colors:
       n_extra_colors = n_clusters - n_colors
       colors = np.concatenate((tasic_1k['clusterColors'],␣
↪tasic_1k['clusterColors'][:n_extra_colors]))
   else:
       colors = tasic_1k['clusterColors']

   fig, ax = plt.subplots(num_plots, 1, figsize=(10, num_plots*10))
   if num_plots == 1:
       if not np.all(tasic_1k['clusters'] == clusters[0]):
           current_colors = np.random.permutation(colors)
       else:
           current_colors = colors
       ax.scatter(tsne_results[0][:, 0], tsne_results[0][:, 1], s=1,␣
↪color=current_colors[clusters[0]])
       ax.set_title(labels[0])
```

```
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_axis_off()
    else:
        for i in range(num_plots):
            if not np.all(tasic_1k['clusters'] == clusters[i]):
                current_colors = np.random.permutation(colors)
            else:
                current_colors = colors
            ax[i].scatter(tsne_results[i][:, 0], tsne_results[i][:, 1], s=1,␣
↪color=current_colors[clusters[i]])
            ax[i].set_title(labels[i])
            ax[i].set_xticks([])
            ax[i].set_yticks([])
            ax[i].set_axis_off()
```

Run the following cells to set the random seed/random state, run tSNE and plot the results.

[24]:
```
# fit TSNE
tsne_default = TSNE(random_state=1)
tsne_results = tsne_default.fit_transform(PCA_75_sklearn)
```

[25]:
```
# Plotting

original_clusters = tasic_1k['clusters']
plot_tsne(tsne_results, original_clusters, labels=['default t-SNE'])
```



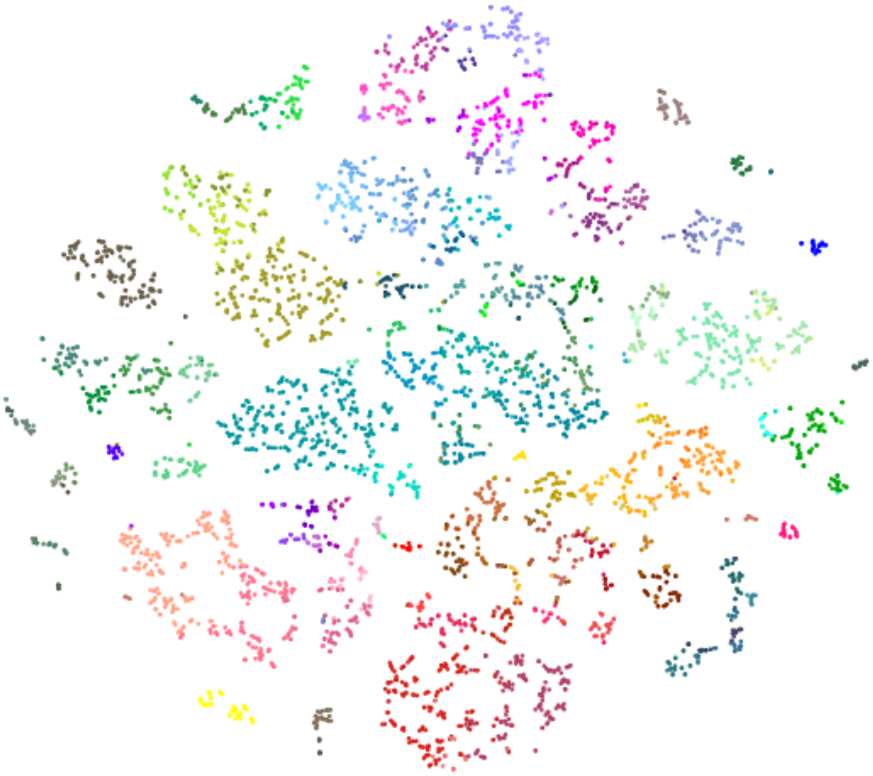
t-SNE has one main parameter called perplexity, which trades of local and global structure. Its default value is 30. Run the tSNE with some other perplexity values (e.g. 5, 100), plot the results next to each other and explain what you observe. In particular, compare with the PCA plot above.

[26]:
```
# try different perplexities

# ---------------- INSERT CODE ----------------------
tsne_p5 = TSNE(random_state=1, perplexity=5)
tsne_p5_results = tsne_p5.fit_transform(PCA_75_sklearn)

tsne_p100 = TSNE(random_state=1, perplexity=100)
tsne_p100_results = tsne_p100.fit_transform(PCA_75_sklearn)
```

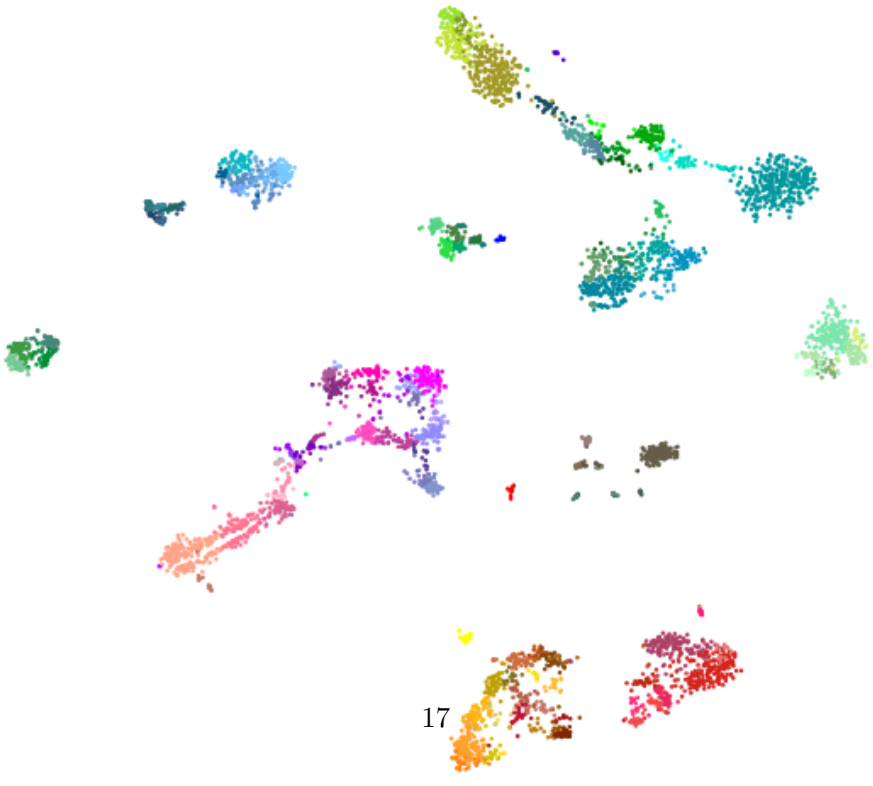
```
# ---------------- END CODE -------------------------
```

[27]:
```
# plot results

# ---------------- INSERT CODE ----------------------
plot_tsne([tsne_p5_results, tsne_p100_results], original_clusters,
          labels=['t-SNE, Perplexity: 5', 't-SNE, Perplexity: 50', 't-SNE,␣
 ↪Perplexity: 100'])


# ---------------- END CODE -------------------------
```

t-SNE, Perplexity: 5



t-SNE, Perplexity: 50



YOUR ANSWER HERE