Neural data analysis:

Neural Data Analysis

Lecturer: Prof. Dr. Philipp Berens

Tutors: Sarah Strauss, Ziwei Huang

Summer term 2022

Student names: Clarissa Auckenthaler, Marina Dittschar

Coding Lab 9: Transcriptomics

In [1]:
```
# Prepare

%matplotlib notebook

import numpy as np
import pylab as plt
import seaborn as sns
import pandas as pd
import pickle
import scipy
from scipy import sparse

# I recommend using openTSNE for experiments with t-SNE
# https://github.com/pavlin-policar/openTSNE
# conda install -c conda-forge openTSNE
from openTSNE import TSNE
```

In [2]:
```
# LOAD HARRIS ET AL DATA

# Load gene counts
data = pd.read_csv('harris-data/expression.tsv.gz', sep='\t')
genes = data.values[:,0]
cells = data.columns[1:-1]
counts = data.values[:,1:-1].transpose().astype('int')
data = []

# Kick out all genes with all counts = 0
genes = genes[counts.sum(axis=0)>0]
counts = counts[:, counts.sum(axis=0)>0]
print(counts.shape)

# Load clustering results
data = pd.read_csv('harris-data/analysis_results.tsv', sep='\t')
clusterNames, clusters = np.unique(data.values[0,1:-1], return_inverse=True)

# Load cluster colors
data = pd.read_csv('harris-data/colormap.txt', sep='\s+', header=None)
clusterColors = data.values

# Note: the color order needs to be reversed to match the publication
clusterColors = clusterColors[::-1]

# Taken from Figure 1 - we need cluster order to get correct color order
clusterOrder = ['Sst.No', 'Sst.Npy.C', 'Sst.Npy.Z', 'Sst.Npy.S', 'Sst.Npy.M',
                'Sst.Pnoc.Calb1.I', 'Sst.Pnoc.Calb1.P', 'Sst.Pnoc.P', 'Sst.Erbb4.R',
```

```
                  'Sst.Erbb4.C', 'Sst.Erbb4.T', 'Pvalb.Tac1.N', 'Pvalb.Tac1.Ss',
                  'Pvalb.Tac1.Sy', 'Pvalb.Tac1.A', 'Pvalb.C1ql1.P', 'Pvalb.C1ql1.C',
                  'Pvalb.C1ql1.N', 'Cacna2d1.Lhx6.R', 'Cacna2d1.Lhx6.V', 'Cacna2d1.Ndnf.N',
                  'Cacna2d1.Ndnf.R', 'Cacna2d1.Ndnf.C', 'Calb2.Cry', 'Sst.Cry', 'Ntng1.S',
                  'Ntng1.R', 'Ntng1.C', 'Cck.Sema', 'Cck.Lmo1.N', 'Cck.Calca', 'Cck.Lmo1.Vip
                  'Cck.Lmo1.Vip.C', 'Cck.Lmo1.Vip.T', 'Cck.Ly', 'Cck.Cxcl14.Calb1.Tn',
                  'Cck.Cxcl14.Calb1.I', 'Cck.Cxcl14.S', 'Cck.Cxcl14.Calb1.K',
                  'Cck.Cxcl14.Calb1.Ta', 'Cck.Cxcl14.V', 'Vip.Crh.P', 'Vip.Crh.C1', 'Calb2.V
                  'Calb2.Vip.I', 'Calb2.Vip.Nos1', 'Calb2.Cntnap5a.R', 'Calb2.Cntnap5a.V',
                  'Calb2.Cntnap5a.I']

reorder = np.zeros(clusterNames.size) * np.nan
for i,c in enumerate(clusterNames):
    for j,k in enumerate(clusterOrder):
        if c[:len(k)]==k:
            reorder[i] = j
            break
clusterColors = clusterColors[reorder.astype(int)]
```

```
(3663, 17965)
```

1. Data inspection

Before we do tSNE visualisation or other advanced methods on the data, we first want to have a closer look on the data and plot some statistics. For most of the analysis we will compare the data to a Poisson distribution.

1.1. Relationship between expression mean and fraction of zeros

The higher the average expression of a gene, the smaller fraction of cells will show a 0 count.

(2pts.)

In [3]:
```
print(f"Counts shape: {counts.shape}")
print("Genes shape: ",genes.shape)
print("Cells shape: ", cells.shape)
```

```
Counts shape: (3663, 17965)
Genes shape:  (17965,)
Cells shape:  (3663,)
```

In [4]:
```
np.unique(counts[:,0])
```

Out[4]:
```
array([ 0,  1,  2,  3,  4, 15])
```

In [5]:
```
# Compute the average expression for each gene
# Compute the fraction of zeros for each gene

expression_mean = np.mean(counts, axis=0)
zero_fraction   = np.sum(counts == 0, axis=0)/counts.shape[0]
print("Expression mean: ", expression_mean)
print("Zero fraction: ", zero_fraction)
```

```
Expression mean:  [8.57220857e-02 5.46000546e-04 2.73000273e-04 ... 8.80425880e-01
 7.64400764e-02 1.91100191e-03]
Zero fraction:  [0.92410592 0.999454   0.999727   ... 0.47583948 0.92929293 0.998089  ]
```

In [6]:
```
# Compute the Poisson prediction
# (what is the expected fraction of zeros in a Poisson distribution with a given mean?)
```

```
print(f"Range of expression mean: [{np.min(expression_mean)},{np.max(expression_mean)}]")
print(f"Range of zero fraction: [{np.min(zero_fraction)},{np.max(zero_fraction)}]")
```

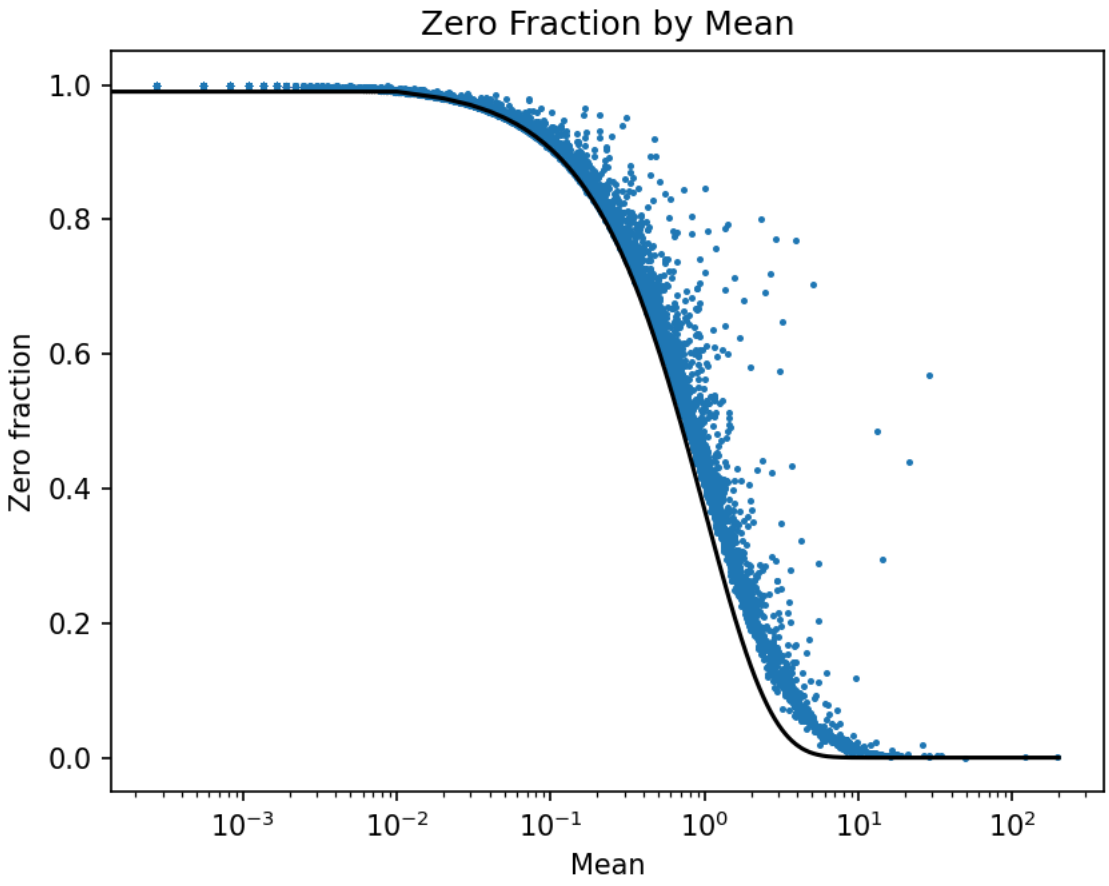
```
Range of expression mean: [0.000273000273000273,194.15615615615616]
Range of zero fraction: [0.0,0.9997269997269997]
```

In [7]:

```
mean = np.arange(0, 200, .01)
expected = np.exp(-mean)
```

In [8]:

```
plt.scatter(expression_mean, zero_fraction, s=2)
plt.plot(mean, expected, color="k")
plt.title("Zero Fraction by Mean")
plt.xscale("log")
plt.xlabel("Mean")
plt.ylabel("Zero fraction")
```

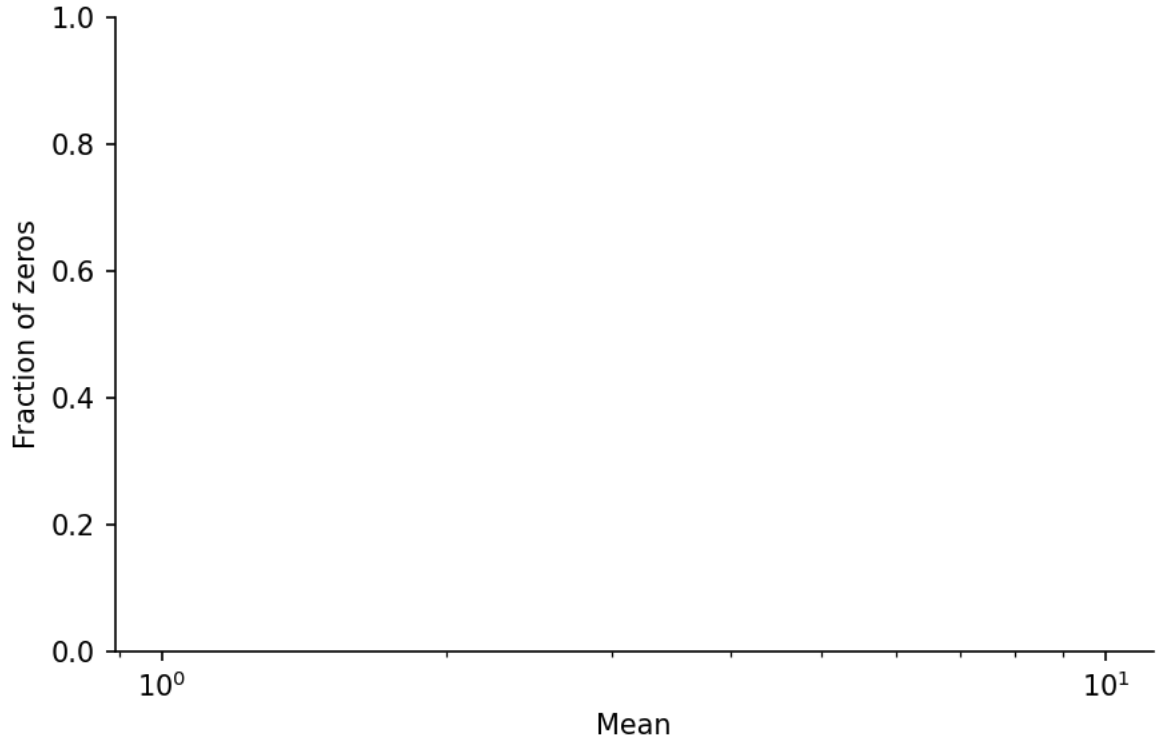


Out[8]:

```
Text(0, 0.5, 'Zero fraction')
```

In [9]:

```
# plot the data and the Poisson prediction

plt.figure(figsize=(6,4))

plt.xscale('log')
plt.xlabel('Mean')
plt.ylabel('Fraction of zeros')


sns.despine()
plt.tight_layout()
```



1.2. Mean-variance relationship

If the expression follows Poisson distribution, then the mean should be equal to the variance.

(1pt.)

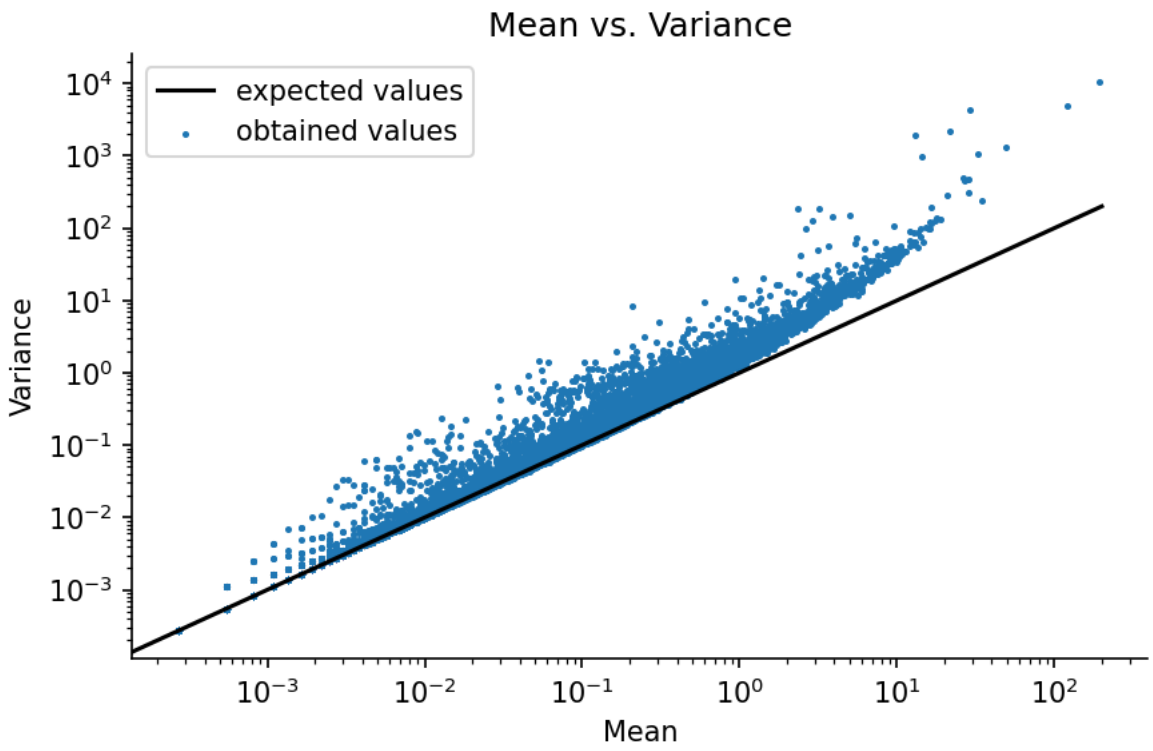
In [10]:
```
# Compute the variance of the expression counts of each gene

expression_var = np.std(counts, axis=0)*np.std(counts, axis=0)
print(expression_var.shape)
```

```
(17965,)
```

In [11]:
```
expression_mean.shape
```

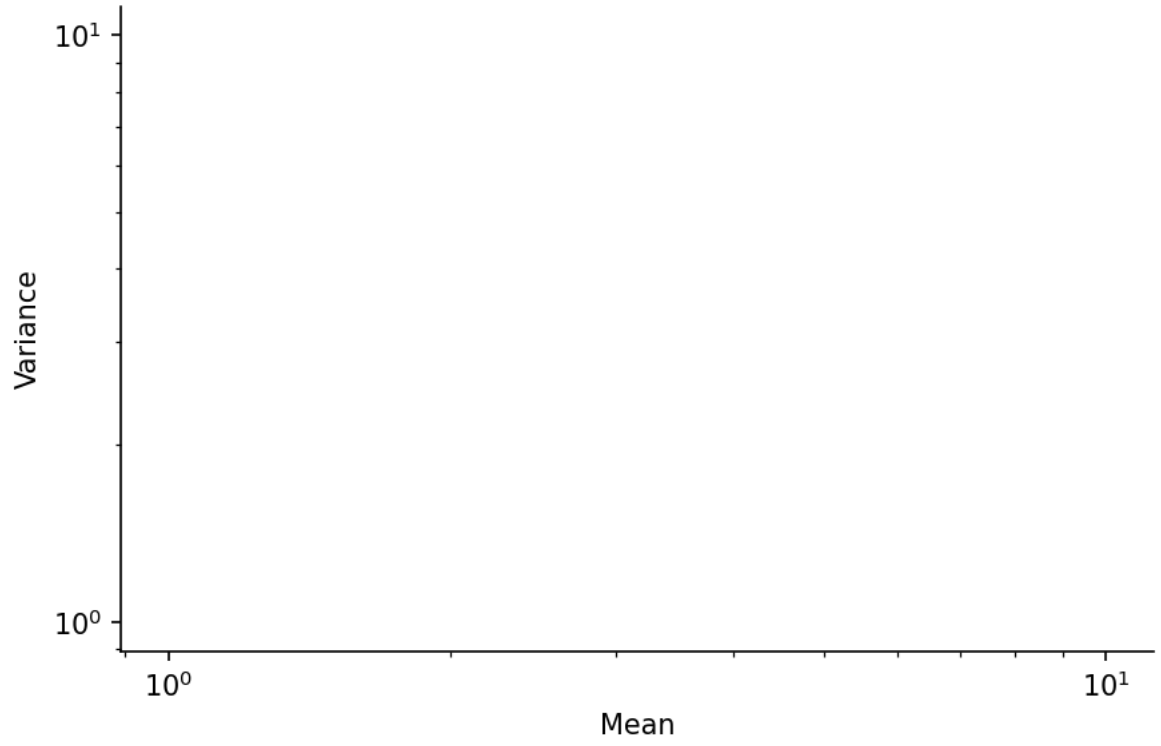
Out[11]:
```
(17965,)
```

In [13]:
```
plt.figure(figsize=(6,4))

plt.scatter(expression_mean, expression_var, s=2, label= "obtained values")
plt.plot(mean, mean, color="k", label = "expected values")
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Mean')
plt.ylabel('Variance')
plt.title("Mean vs. Variance")
plt.legend()


sns.despine()
plt.tight_layout()
```



In [14]:

```
# Plot the mean-variance relationship on a log-log plot
# Plot the Poisson prediction as a line

plt.figure(figsize=(6,4))

plt.xscale('log')
plt.yscale('log')
plt.xlabel('Mean')
plt.ylabel('Variance')


sns.despine()
plt.tight_layout()
```



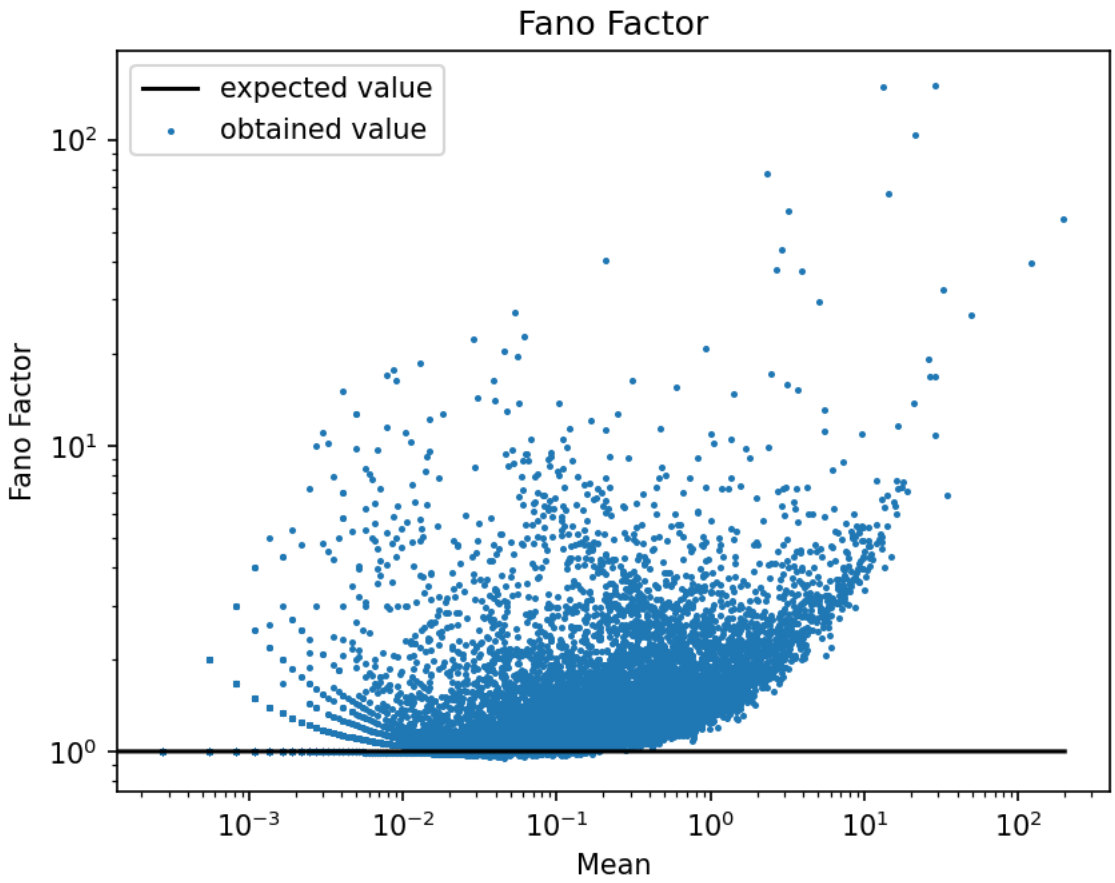
1.3. Relationship between the mean and the Fano factor

If the expression follows the Poisson distribution, then the Fano factor (variance/mean) should be equal to 1 for all genes.

(1pt.)

In [15]:

```
# Compute the Fano factor for each gene and make a scatter plot
# of expression mean vs. Fano factor in log-log coordinates.
# Plot a Poisson prediction as line

# Use the same style of plot as above.

plt.scatter(expression_mean, expression_var/expression_mean, s=2, label="obtained value")
plt.yscale("log")
plt.xscale("log")
plt.plot(mean, np.tile([1], mean.shape), color="k", label="expected value")
plt.xlabel("Mean")
plt.ylabel("Fano Factor")
plt.legend()
plt.title("Fano Factor")
```



Out[15]:
```
Text(0.5, 1.0, 'Fano Factor')
```

1.4. Histogram of sequencing depths

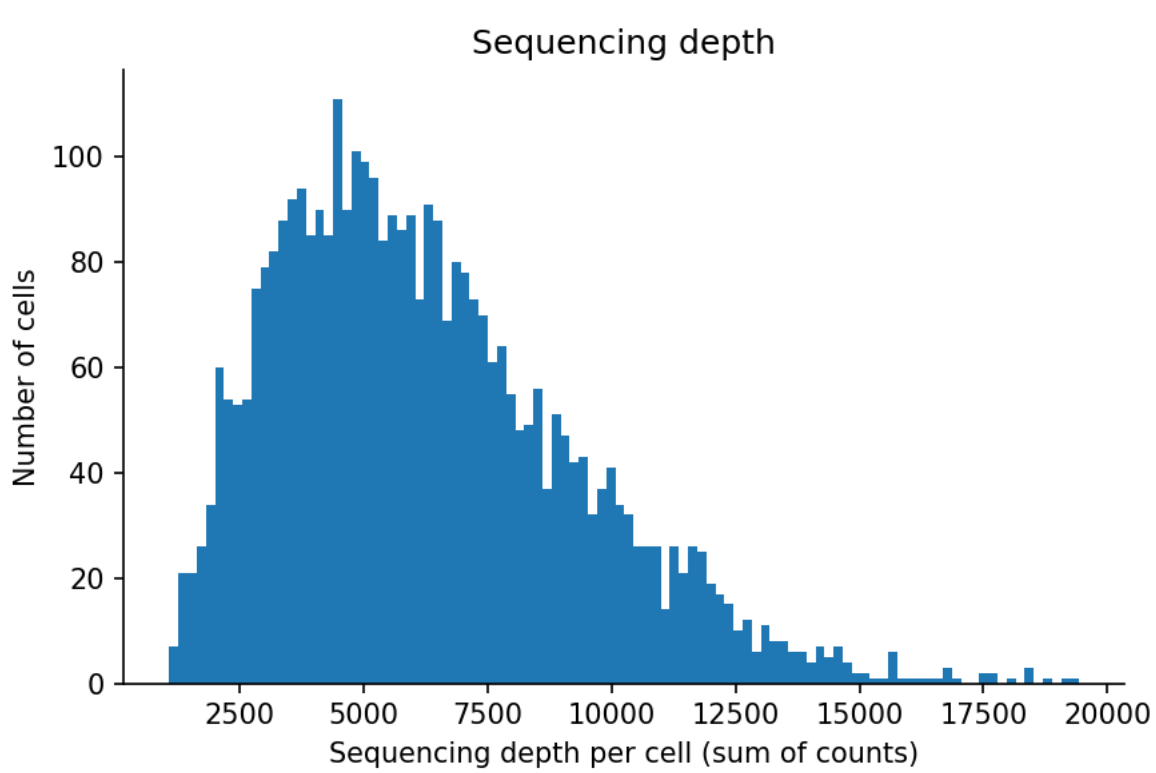
Different cells have different sequencing depths (sum of counts across all genes) because the efficiency can change from droplet to droplet due to some random experimental factors.

(1pt.)

In [16]:
```
counts.shape
```

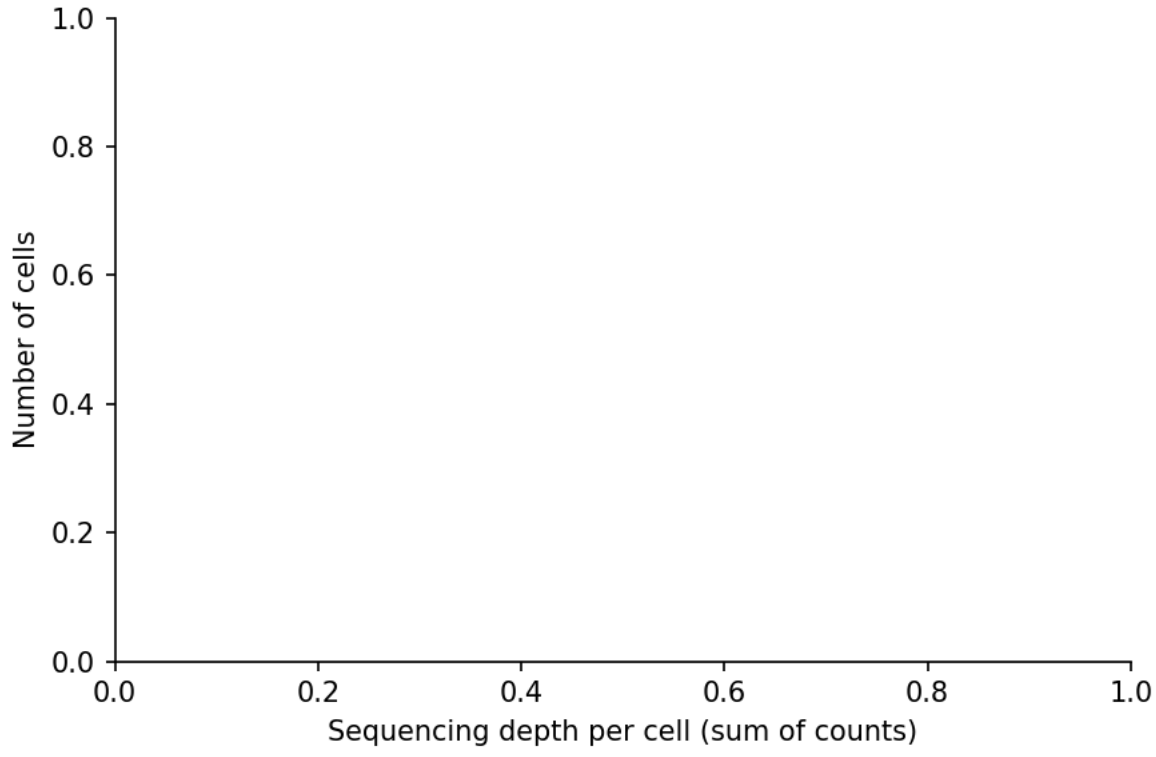
Out[16]:
```
(3663, 17965)
```

In [17]:
```
depth = np.sum(counts, axis=1)
plt.figure(figsize=(6,4))

plt.hist(depth, bins=100)
plt.xlabel('Sequencing depth per cell (sum of counts)')
plt.ylabel('Number of cells')
plt.title("Sequencing depth")

sns.despine()
plt.tight_layout()
```



In [18]:

```
# Make a histogram of sequencing depths across cells.
# Sequencing depth of each cell is the sum of all counts of this cell

plt.figure(figsize=(6,4))


plt.xlabel('Sequencing depth per cell (sum of counts)')
plt.ylabel('Number of cells')

sns.despine()
plt.tight_layout()
```



1.5. Fano factors after normalization

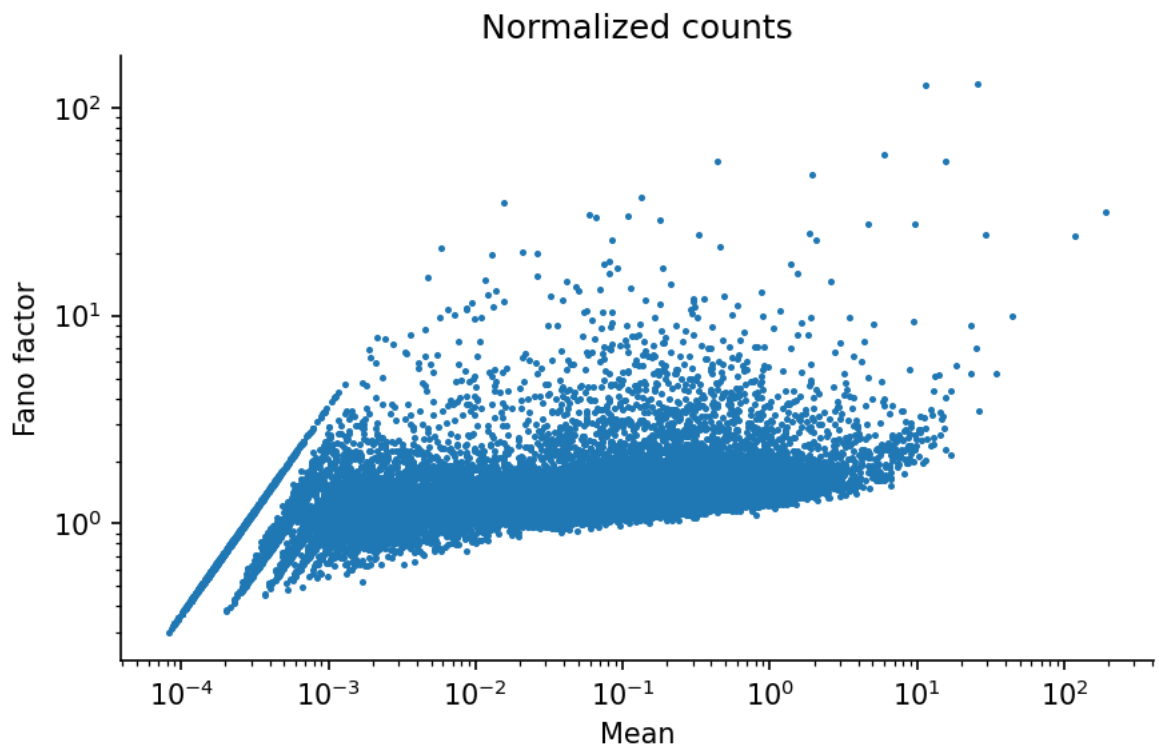
After normalization by sequencing depth, Fano factor should be closer to 1 (i.e. variance even more closely following the mean). This can be used for feature selection.

(1pt.)

In [19]:
```
print(depth.shape)
counts.shape
```

Out[19]:
```
(3663,)
(3663, 17965)
```

In [20]:
```
# Normalize counts by the sequencing depth of each cell and multiply by the median sequenc
# Then make the same expression vs Fano factor plot as above
norm_counts = counts/depth[...,np.newaxis]
norm_counts = norm_counts*np.median(depth)
norm_var = np.var(norm_counts, axis=0)
fano = norm_var/np.mean(norm_counts, axis=0)
plt.figure(figsize=(6,4))
plt.scatter(np.mean(norm_counts, axis=0), fano, s=2)
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Mean')
plt.ylabel('Fano factor')
plt.title('Normalized counts')

#plt.scatter(mu, fano, s=1)

sns.despine()
plt.tight_layout()
```



In [21]:
```
genes
```

Out[21]:
```
array(['Xkr4', 'Gm1992', 'Sox17', ..., 'PISD', 'DHRSX', 'CAAA01147332.1'],
      dtype=object)
```

In [22]:
```
# Find top-10 genes with the highest normalized Fano factor
# Print them sorted by the Fano factor starting from the highest
# Gene names are stored in the `genes` array
top_genes = genes[np.argsort(fano)[::-1][:10]]
print("Top 10 genes with the highest normalized Fano factor: ")
print(top_genes)
```

```
Top 10 genes with the highest normalized Fano factor:
['Sst' 'Npy' 'Vip' 'Cck' 'Cpne2' 'Pcp4' 'Ptpn23' 'Pdzd9' 'Malat1' 'Armc2']
```

2. Low dimensional visualization

Here we look at the influence of variance-stabilizing transformations on PCA and t-SNE.

2.1. PCA with and without transformations

Square root is a variance-stabilizing transformation for the Poisson data. Log-transform is also often used in the transcriptomic community. Look at the effect of both.

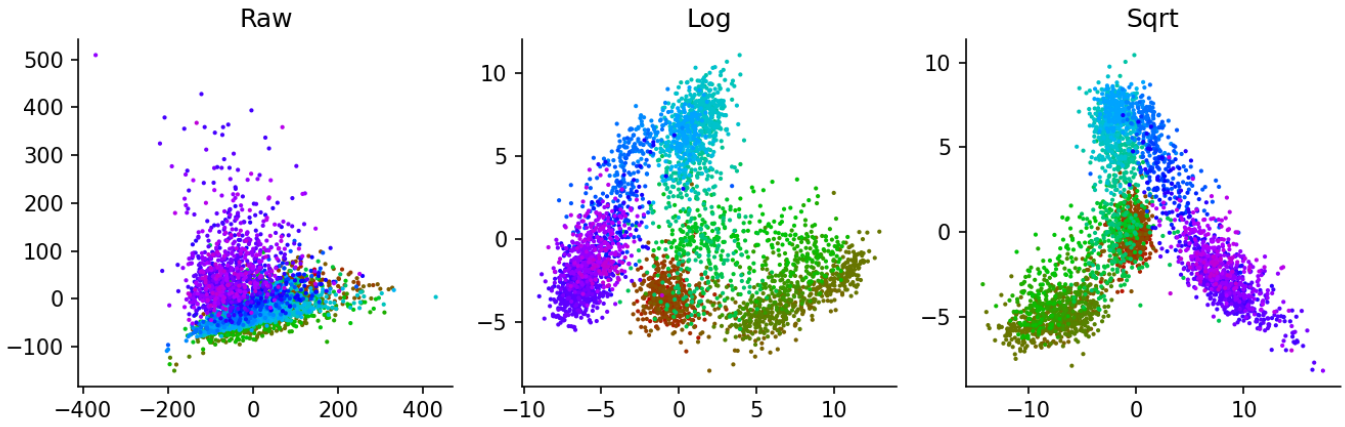
(1pt.)

In [23]:
```
# Transform the counts into normalized counts (as above)
# (Normalize counts by the sequencing depth of each cell and multiply by the median sequer
# Select all genes with the normalized Fano factor above 3 and remove the rest
counts_above = norm_counts[:, fano>3]
counts_above.shape
```

Out[23]:
```
(3663, 707)
```

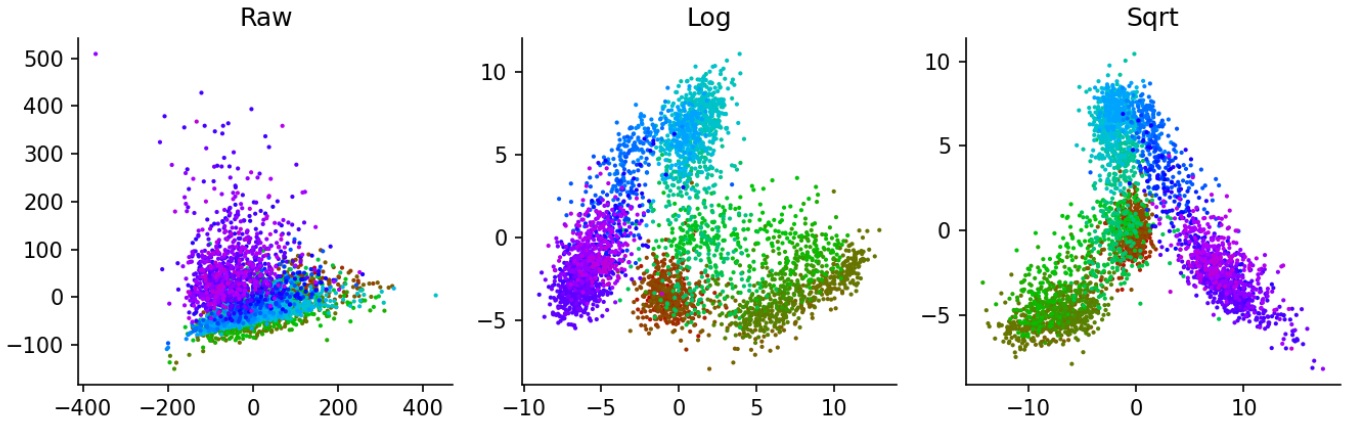
In [24]:
```
# Perform PCA three times: on the resulting matrix as is,
# after np.log2(X+1) transform, and after np.sqrt(X) transform

from sklearn.decomposition import PCA

pca1 = PCA(n_components = 50).fit_transform(counts_above)
pca2 = PCA(n_components = 50).fit_transform(X= np.log2(counts_above+1))
pca3 = PCA(n_components = 50).fit_transform(X= np.sqrt(counts_above))
```

In [25]:
```
plt.figure(figsize=(9,3))

plt.subplot(131)
plt.scatter(pca1[:,0], pca1[:,1], s=1, c=clusterColors[clusters])
plt.title('Raw')

plt.subplot(132)
plt.scatter(pca2[:,0], pca2[:,1], s=1, c=clusterColors[clusters])
plt.title('Log')

plt.subplot(133)
plt.scatter(pca3[:,0], pca3[:,1], s=1, c=clusterColors[clusters])
plt.title('Sqrt')

sns.despine()
plt.tight_layout()
```



In [26]:

```
# Plot the results

plt.figure(figsize=(9,3))

plt.subplot(131)
plt.scatter(pca1[:,0], pca1[:,1], s=1, c=clusterColors[clusters])
plt.title('Raw')

plt.subplot(132)
plt.scatter(pca2[:,0], pca2[:,1], s=1, c=clusterColors[clusters])
plt.title('Log')

plt.subplot(133)
plt.scatter(pca3[:,0], pca3[:,1], s=1, c=clusterColors[clusters])
plt.title('Sqrt')

sns.despine()
plt.tight_layout()
```



2.2. tSNE with and without transformations

Do these transformations have any effect on t-SNE?

(1pt.)

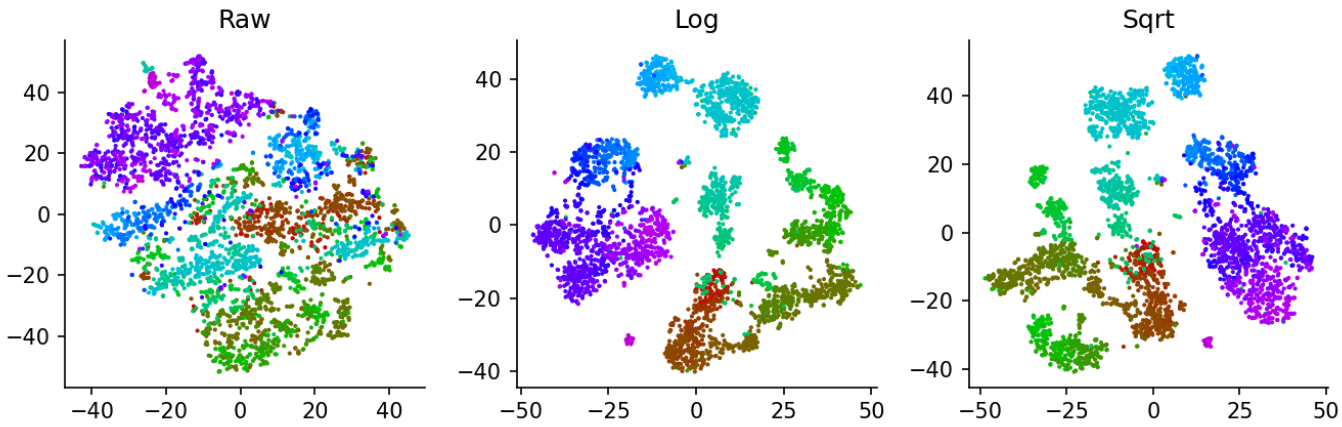
In [27]:

```
# Perform tSNE three times: on the resulting matrix as is,
# after np.log2(X+1) transform, and after np.sqrt(X) transform

# Apply t-SNE to the 50 PCs

# Use default settings of openTSNE
```

```
# You can also use sklearn if you want

tsne1 = TSNE().fit(pca1)
tsne2 = TSNE().fit(pca2)
tsne3 = TSNE().fit(pca3)
```

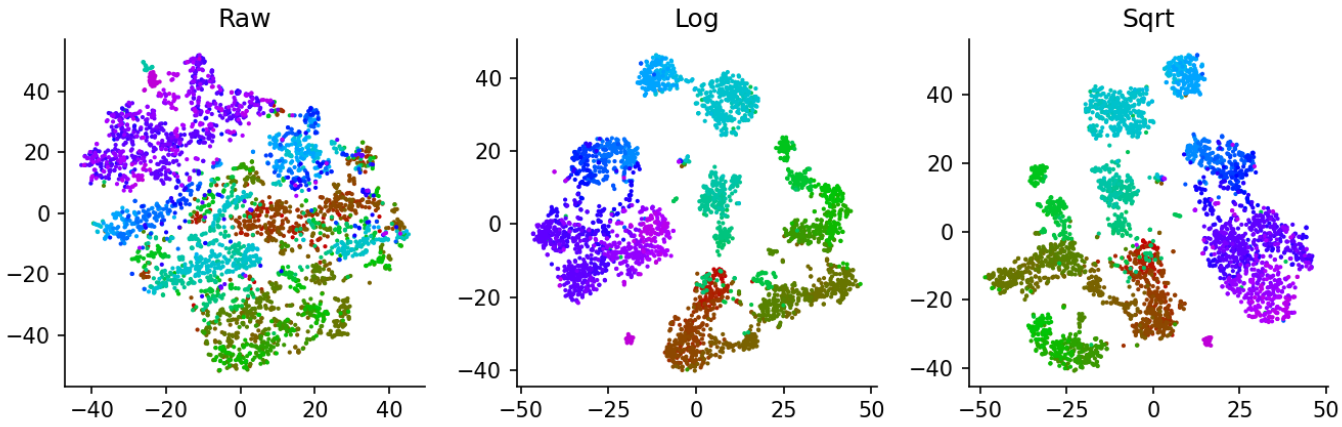
In [28]:

```
plt.figure(figsize=(9,3))

plt.subplot(131)
plt.scatter(tsne1[:,0], tsne1[:,1], s=1, c=clusterColors[clusters])
plt.title('Raw')

plt.subplot(132)
plt.scatter(tsne2[:,0], tsne2[:,1], s=1, c=clusterColors[clusters])
plt.title('Log')

plt.subplot(133)
plt.scatter(tsne3[:,0], tsne3[:,1], s=1, c=clusterColors[clusters])
plt.title('Sqrt')

sns.despine()
plt.tight_layout()
```



In [29]:

```
# Plot the results

plt.figure(figsize=(9,3))

plt.subplot(131)
plt.scatter(tsne1[:,0], tsne1[:,1], s=1, c=clusterColors[clusters])
plt.title('Raw')

plt.subplot(132)
plt.scatter(tsne2[:,0], tsne2[:,1], s=1, c=clusterColors[clusters])
plt.title('Log')

plt.subplot(133)
plt.scatter(tsne3[:,0], tsne3[:,1], s=1, c=clusterColors[clusters])
plt.title('Sqrt')

sns.despine()
plt.tight_layout()
```



2.3. Leiden clustering

This dataset is small and can be clustered in many different ways. We will apply Leiden clustering (closely related to the Louvain clustering), which is standard in the field and works well even for very large datasets.

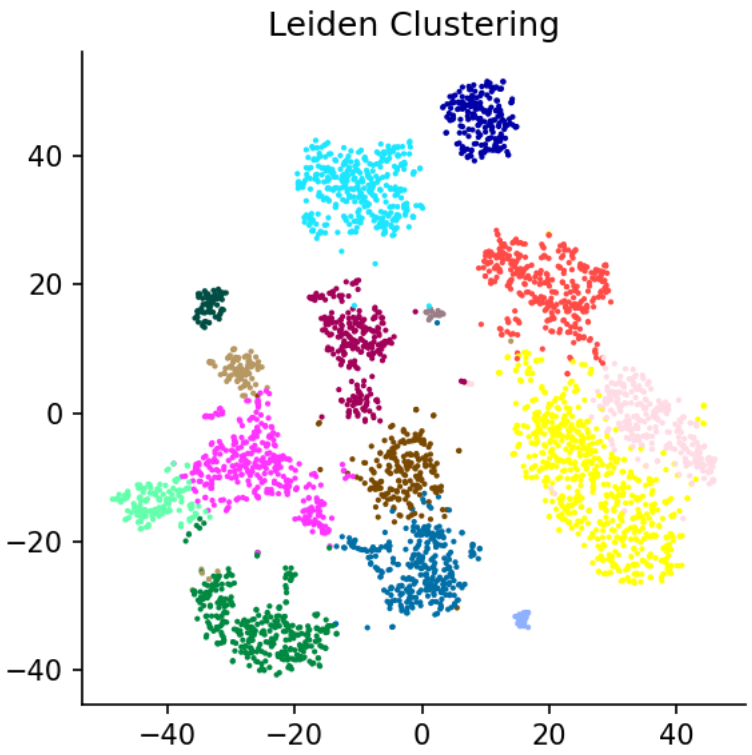
(1pt.)

In [30]:
```
# To run this code you need to install leidenalg and igraph
# conda install -c conda-forge python-igraph leidenalg

import igraph as ig
from sklearn.neighbors import NearestNeighbors, kneighbors_graph
import leidenalg as la
```

In [31]:
```
# Define some contrast colors

clusterCols = ["#FFFF00", "#1CE6FF", "#FF34FF", "#FF4A46", "#008941", "#006FA6", "#A30059"
        "#FFDBE5", "#7A4900", "#0000A6", "#63FFAC", "#B79762", "#004D43", "#8FB0FF", "#997
        "#5A0007", "#809693", "#FEFFE6", "#1B4400", "#4FC601", "#3B5DFF", "#4A3B53", "#FF2
        "#61615A", "#BA0900", "#6B7900", "#00C2A0", "#FFAA92", "#FF90C9", "#B903AA", "#D16
        "#DDEFFF", "#000035", "#7B4F4B", "#A1C299", "#300018", "#0AA6D8", "#013349", "#008
        "#372101", "#FFB500", "#C2FFED", "#A079BF", "#CC0744", "#C0B9B2", "#C2FF99", "#001
        "#00489C", "#6F0062", "#0CBD66", "#EEC3FF", "#456D75", "#B77B68", "#7A87A1", "#788
        "#885578", "#FAD09F", "#FF8A9A", "#D157A0", "#BEC459", "#456648", "#0086ED", "#886
        "#34362D", "#B4A8BD", "#00A6AA", "#452C2C", "#636375", "#A3C8C9", "#FF913F", "#938
        "#575329", "#00FECF", "#B05B6F", "#8CD0FF", "#3B9700", "#04F757", "#C8A1A1", "#1E6
        "#7900D7", "#A77500", "#6367A9", "#A05837", "#6B002C", "#772600", "#D790FF", "#9B9
        "#549E79", "#FFF69F", "#201625", "#72418F", "#BC23FF", "#99ADC0", "#3A2465", "#922
        "#5B4534", "#FDE8DC", "#404E55", "#0089A3", "#CB7E98", "#A4E804", "#324E72", "#6A3
        "#83AB58", "#001C1E", "#D1F7CE", "#004B28", "#C8D0F6", "#A3A489", "#806C66", "#222
        "#BF5650", "#E83000", "#66796D", "#DA007C", "#FF1A59", "#8ADBB4", "#1E0200", "#5B4
        "#C895C5", "#320033", "#FF6832", "#66E1D3", "#CFCDAC", "#D0AC94", "#7ED379", "#012
clusterCols = np.array(clusterCols)

# Construct kNN graph with k=15

A = kneighbors_graph(pca3, 15)

# Transform it into an igraph object

sources, targets = A.nonzero()
G = ig.Graph(directed=False)
G.add_vertices(A.shape[0])
edges = list(zip(sources, targets))
G.add_edges(edges)
```

In [32]:
```
# Run Leiden clustering


partition = la.find_partition(G, la.RBConfigurationVertexPartition, resolution_parameter=1
```

In [33]:
```
# Plot the results

plt.figure(figsize=(4,4))
plt.scatter(tsne3[:,0], tsne3[:,1], s=1, c=clusterCols[partition.membership])
plt.title("Leiden Clustering")
sns.despine()
plt.tight_layout()
```



2.4. Change the clustering resolution

The number of clusters can be changed by modifying the resolution parameter.

(1pt.)

In [34]:
```
# How many clusters did we get?
# Change the resolution parameter to yield 2x more and 2x fewer clusters
# Plot all three results as tSNE overlays (as above)
print(f"The number of clusters is: {np.max(partition.membership)}")
```

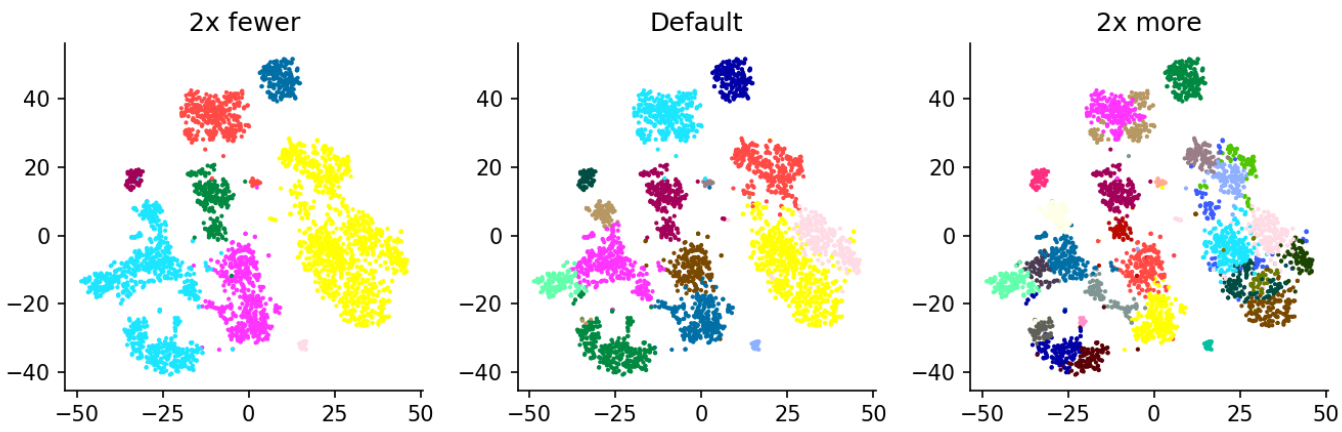
```
The number of clusters is: 14
```

In [46]:
```
partitionx2 = la.find_partition(G, la.RBConfigurationVertexPartition, resolution_parameter
print(f"The number of clusters is: {np.max(partitionx2.membership)}")
partitionx0_5 = la.find_partition(G, la.RBConfigurationVertexPartition, resolution_paramet
print(f"The number of clusters is: {np.max(partitionx0_5.membership)}")
```

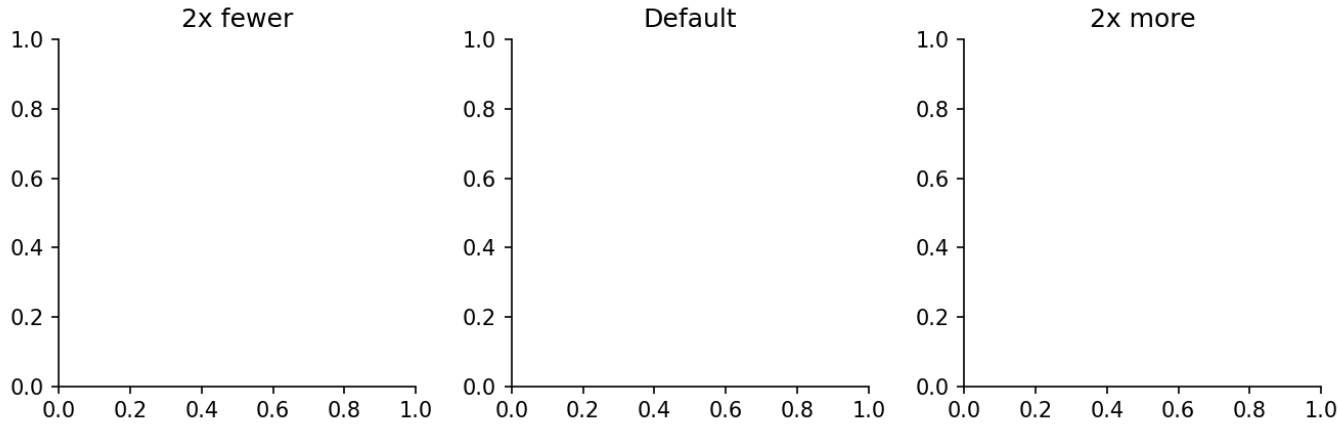
```
The number of clusters is: 7
The number of clusters is: 28
```

In [ ]:
```
#print(np.max(partition.membership))

partition2 = None
#print(np.max(partition2.membership))

partition3 = None
#print(np.max(partition3.membership))
```

In [47]:
```
plt.figure(figsize=(9,3))

plt.subplot(131)
plt.scatter(tsne3[:,0], tsne3[:,1], s=1, c=clusterCols[partitionx2.membership])
plt.title('2x fewer')

plt.subplot(132)
plt.scatter(tsne3[:,0], tsne3[:,1], s=1, c=clusterCols[partition.membership])
plt.title('Default')

plt.subplot(133)
plt.scatter(tsne3[:,0], tsne3[:,1], s=1, c=clusterCols[partitionx0_5.membership])
plt.title('2x more')

sns.despine()
plt.tight_layout()
```



In [48]:
```
plt.figure(figsize=(9,3))

plt.subplot(131)
#plt.scatter(tsne3[:,0], tsne3[:,1], s=1, c=clusterCols[partition2.membership])
plt.title('2x fewer')

plt.subplot(132)
#plt.scatter(tsne3[:,0], tsne3[:,1], s=1, c=clusterCols[partition.membership])
plt.title('Default')

plt.subplot(133)
#plt.scatter(tsne3[:,0], tsne3[:,1], s=1, c=clusterCols[partition3.membership])
plt.title('2x more')

sns.despine()
plt.tight_layout()
```


2x fewer
Default
2x more
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0