CodingLab7

June 8, 2025

Neural Data Science

Lecturer: Dr. Jan Lause, Prof. Dr. Philipp Berens

Tutors: Jonas Beck, Fabio Seel, Julius Würzler

Summer term 2025

Student names: *FILL IN YOUR NAMES HERE*

LLM Disclaimer: *Did you use an LLM to solve this exercise? If yes, which one and where did you use it? [Copilot, Claude, ChatGPT, etc.]*

1 Coding Lab 7 : Transcriptomics

[2]:
```
import numpy as np
import pylab as plt
import pandas as pd
import matplotlib.pyplot as plt

# We recommend using openTSNE for experiments with t-SNE
# https://github.com/pavlin-policar/openTSNE
from openTSNE import TSNE

%matplotlib inline

%load_ext jupyter_black

%load_ext watermark
%watermark --time --date --timezone --updated --python --iversions --watermark␣
  ↪-p sklearn
```

```
Last updated: 2025-06-08 09:54:26CEST

Python implementation: CPython
Python version       : 3.10.13
IPython version      : 8.21.0

sklearn: 1.3.2
```

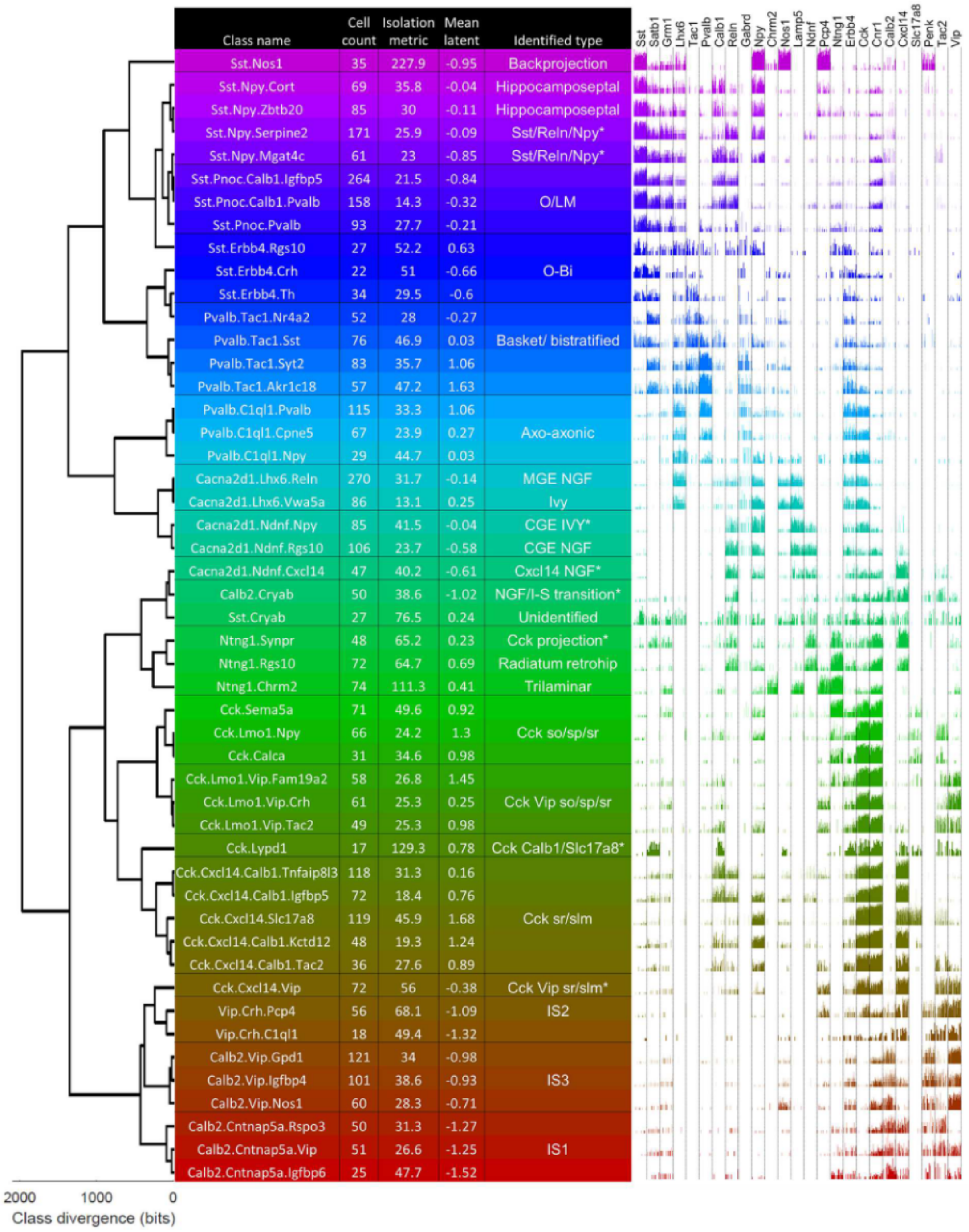
```
numpy     : 1.26.2
pandas    : 2.2.3
matplotlib: 3.8.0
openTSNE  : 1.0.2

Watermark: 2.5.0
```

[3]:
```
plt.style.use("../matplotlib_style.txt")
```

2 Introduction

In this notebook you are going to work with transcriptomics data, in particular single-cell RNA sequencing (scRNA-seq) data from the paper by Harris et al. (2018). They recorded the transcriptomes of 3,663 inhibitory cells in the hippocampal area CA1. Their analysis divided these cells into 49 fine-scale clusters coresponding to different cell subtypes. They asigned names to these cluster in a hierarchical fashion according to strongly expressed gene in each clusters. The figure below shows the details of their classification.

You will first analyze some of the most relevant statistics of UMI gene counts distributions, and afterwards follow the standard pipeline in the field to produce a visualization of the data.



Class name	Cell count	Isolation metric	Mean latent	Identified type
Sst.Nos1	35	227.9	-0.95	Backprojection
Sst.Npy.Cort	69	35.8	-0.04	Hippocamposeptal
Sst.Npy.Zbtb20	85	30	-0.11	Hippocamposeptal
Sst.Npy.Serpine2	171	25.9	-0.09	Sst/Reln/Npy*
Sst.Npy.Mgat4c	61	23	-0.85	Sst/Reln/Npy*
Sst.Pnoc.Calb1.Igfbp5	264	21.5	-0.84	
Sst.Pnoc.Calb1.Pvalb	158	14.3	-0.32	O/LM
Sst.Pnoc.Pvalb	93	27.7	-0.21	
Sst.Erbb4.Rgs10	27	52.2	0.63	
Sst.Erbb4.Crh	22	51	-0.66	O-Bi
Sst.Erbb4.Th	34	29.5	-0.6	
Pvalb.Tac1.Nr4a2	52	28	-0.27	
Pvalb.Tac1.Sst	76	46.9	0.03	Basket/ bistratified
Pvalb.Tac1.Syt2	83	35.7	1.06	
Pvalb.Tac1.Akr1c18	57	47.2	1.63	
Pvalb.C1ql1.Pvalb	115	33.3	1.06	
Pvalb.C1ql1.Cpne5	67	23.9	0.27	Axo-axonic
Pvalb.C1ql1.Npy	29	44.7	0.03	
Cacna2d1.Lhx6.Reln	270	31.7	-0.14	MGE NGF
Cacna2d1.Lhx6.Vwa5a	86	13.1	0.25	Ivy
Cacna2d1.Ndnf.Npy	85	41.5	-0.04	CGE IVY*
Cacna2d1.Ndnf.Rgs10	106	23.7	-0.58	CGE NGF
Cacna2d1.Ndnf.Cxcl14	47	40.2	-0.61	Cxcl14 NGF*
Calb2.Cryab	50	38.6	-1.02	NGF/I-S transition*
Sst.Cryab	27	76.5	0.24	Unidentified
Ntng1.Synpr	48	65.2	0.23	Cck projection*
Ntng1.Rgs10	72	64.7	0.69	Radiatum retrohip
Ntng1.Chrm2	74	111.3	0.41	Trilaminar
Cck.Sema5a	71	49.6	0.92	
Cck.Lmo1.Npy	66	24.2	1.3	Cck so/sp/sr
Cck.Calca	31	34.6	0.98	
Cck.Lmo1.Vip.Fam19a2	58	26.8	1.45	
Cck.Lmo1.Vip.Crh	61	25.3	0.25	Cck Vip so/sp/sr
Cck.Lmo1.Vip.Tac2	49	25.3	0.98	
Cck.Lypd1	17	129.3	0.78	Cck Calb1/Slc17a8*
Cck.Cxcl14.Calb1.Tnfaip8l3	118	31.3	0.16	
Cck.Cxcl14.Calb1.Igfbp5	72	18.4	0.76	
Cck.Cxcl14.Slc17a8	119	45.9	1.68	Cck sr/slm
Cck.Cxcl14.Calb1.Kctd12	48	19.3	1.24	
Cck.Cxcl14.Calb1.Tac2	36	27.6	0.89	
Cck.Cxcl14.Vip	72	56	-0.38	Cck Vip sr/slm*
Vip.Crh.Pcp4	56	68.1	-1.09	IS2
Vip.Crh.C1ql1	18	49.4	-1.32	
Calb2.Vip.Gpd1	121	34	-0.98	
Calb2.Vip.Igfbp4	101	38.6	-0.93	IS3
Calb2.Vip.Nos1	60	28.3	-0.71	
Calb2.Cntnap5a.Rspo3	50	31.3	-1.27	
Calb2.Cntnap5a.Vip	51	26.6	-1.25	IS1
Calb2.Cntnap5a.Igfbp6	25	47.7	-1.52	

2.1 Load data

Download the data from ILIAS, move it to the `data/` directory and unzip it there. The read counts can be found in `counts`, with rows corresponding to cells and columns to genes. The cluster assignments for every individual cell can be found in `clusters`, along with the colors used in the publication in `clusterColors`.

```
[4]: # LOAD HARRIS ET AL DATA

     # Load gene counts
     data = pd.read_csv("../data/nds_cl_7/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("../data/nds_cl_7/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("../data/nds_cl_7/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.F",
         "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.Vip.G",
         "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)
```

3 Task 1: Data inspection

Before we use t-SNE or any other advanced visualization 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.

3.0.1 1.1. Relationship between expression mean and fraction of zeros

Compute actual and predicted gene expression. The higher the average expression of a gene, the smaller fraction of cells will show a 0 count. Plot the data and explain what you see in the plot.

(3 pts)

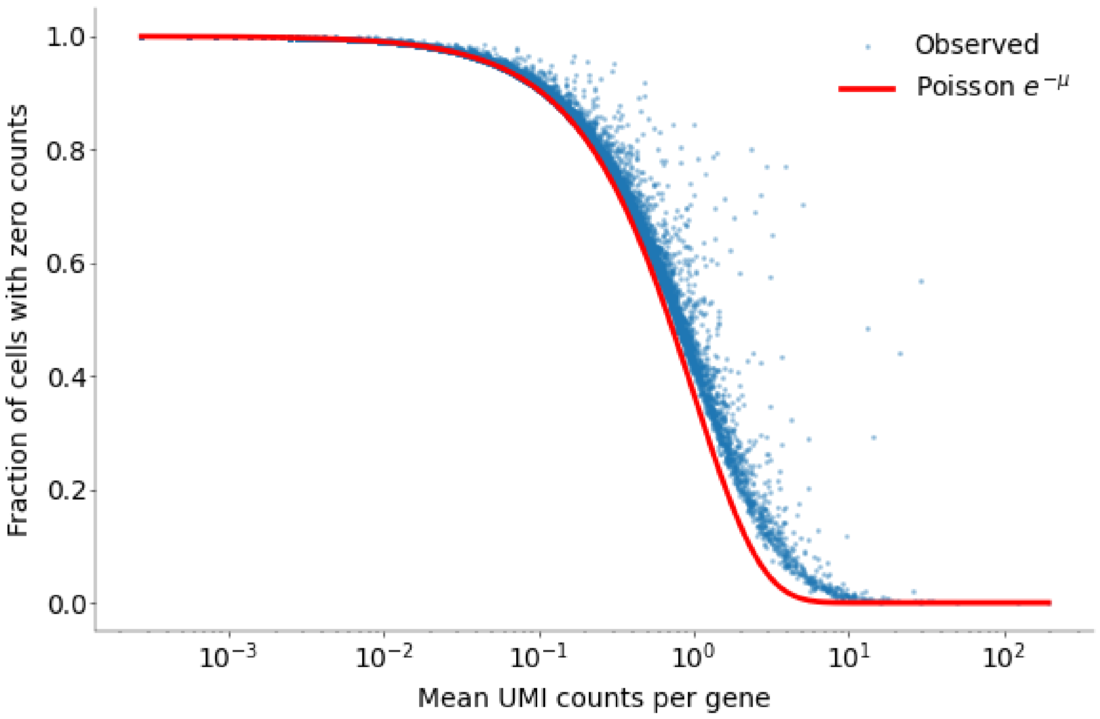
[5]:
```
# -----------------------------------------------------
# Compute actual and predicted gene expression (1 pt)
# -----------------------------------------------------

# Compute the average expression for each gene
gene_means = counts.mean(axis=0)

# Compute the fraction of zeros for each gene
n_cells = counts.shape[0]
frac_zeros = (counts == 0).sum(axis=0) / n_cells
```

[6]:
```
# Compute the Poisson prediction
# (what is the expected fraction of zeros in a Poisson distribution with a␣
 ↪given mean?)
# (expected fraction of zeros in a Poisson distribution with mean   is e^{- })
poisson_pred = np.exp(-gene_means)

poisson_pred
```

[6]:
```
array([0.91784928, 0.99945415, 0.99972704, …, 0.4146063 , 0.92640843,
       0.99809082])
```

[7]:
```
# -------------------------------------------------
# plot the data and the Poisson prediction (1 pt)
# -------------------------------------------------

fig, ax = plt.subplots(figsize=(6, 4))

# scatter observed mean vs zero-fraction
ax.scatter(gene_means, frac_zeros, s=5, alpha=0.3, label="Observed")

# plot Poisson curve (sorted for a smooth line)
idx = np.argsort(gene_means)
ax.plot(gene_means[idx], poisson_pred[idx], "r-", lw=2, label=r"Poisson␣
 ↪$e^{-\mu}$")

ax.set_xscale("log")
ax.set_xlabel("Mean UMI counts per gene")
ax.set_ylabel("Fraction of cells with zero counts")
ax.legend()
plt.tight_layout()
plt.show()
```

```
/var/folders/76/g6ys7mkj75zg7zyn8m093rk40000gn/T/ipykernel_82853/2298918199.py:1
8: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



Explanation (1 pt) …

The red curve shows the theoretical Poisson zero-fraction

$$P(X = 0) \;=\; e^{-\mu}$$

where ( ) is the gene's mean UMI count. For very lowly expressed genes ($\mu \ll 1$), the observed fraction of zeros closely follows this Poisson expectation, reflecting only sampling noise. However, once mean expression rises ($\mu \gtrsim 1$), the **observed** zero-fraction exceeds $e^{-\mu}$. This "excess" zeros is due to technical dropouts (failed transcript capture or detection) and true cell-to-cell variability, which introduce overdispersion beyond Poisson. Consequently, single-cell RNA-seq data are better modeled with a negative-binomial or zero-inflated distribution to account for both the mean–variance relationship and elevated zero rates.

3.0.2 1.2. Mean-variance relationship

If the expression follows Poisson distribution, then the mean should be equal to the variance. Plot the mean-variance relationship and interpret the plot.

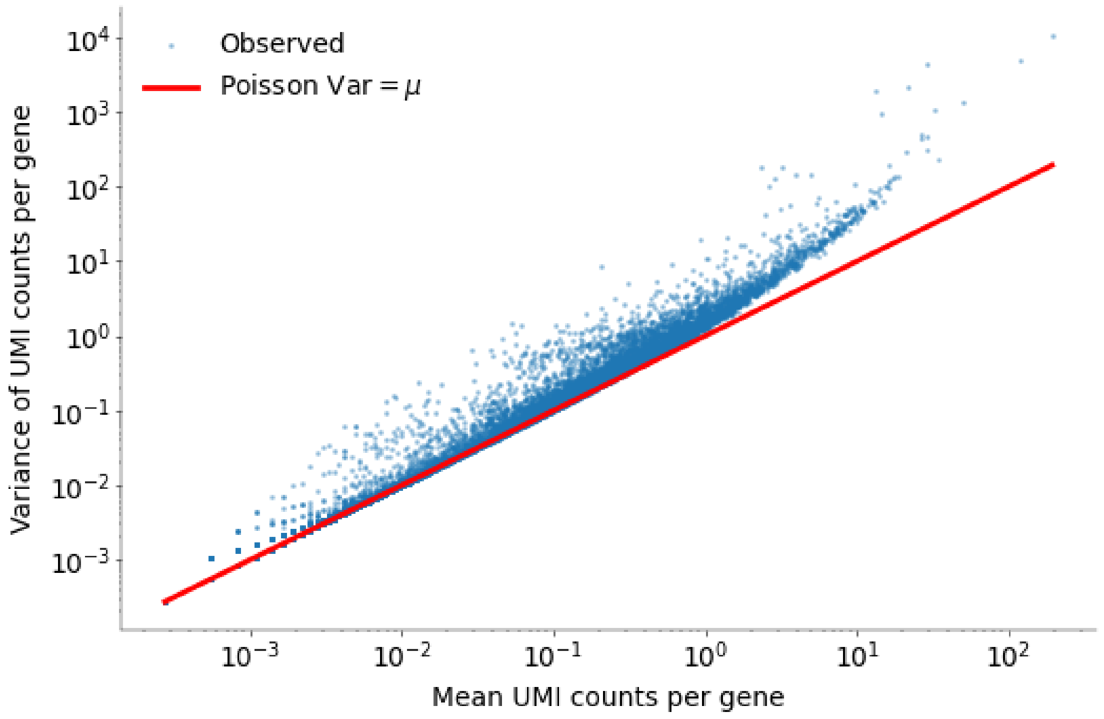
(2.5 pts)

[8]:
```
# ------------------------------------------------------------------
# Compute the variance of the expression counts of each gene (0.5 pt)
# ------------------------------------------------------------------
gene_vars = counts.var(axis=0)  # population variance, ddof=0
gene_vars
```

[8]:
```
array([1.49899881e-01, 5.45702429e-04, 2.72925744e-04, …,
       1.24532529e+00, 8.20630026e-02, 1.90734998e-03])
```

[9]:
```
# ------------------------------------------------------------
# Plot the mean-variance relationship on a log-log plot (1 pt)
# Plot the Poisson prediction as a line
# ------------------------------------------------------------

fig, ax = plt.subplots(figsize=(6, 4))

# scatter observed mean vs variance
ax.scatter(gene_means, gene_vars, s=5, alpha=0.3, label="Observed")

# Poisson prediction: Var =
x_line = np.logspace(np.log10(gene_means.min()), np.log10(gene_means.max()),␣
  ↪100)
ax.plot(x_line, x_line, "r-", lw=2, label=r"Poisson $\mathrm{Var}=\mu$")

ax.set_xscale("log")
ax.set_yscale("log")
ax.set_xlabel("Mean UMI counts per gene")
ax.set_ylabel("Variance of UMI counts per gene")
ax.legend()
plt.tight_layout()
plt.show()
```

```
/var/folders/76/g6ys7mkj75zg7zyn8m093rk40000gn/T/ipykernel_82853/1986976209.py:2
0: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



Explanation (1 pt) …

Under a Poisson model, the mean and variance are equal:

$$Var(X) = \mu.$$

In the log–log plot, the red line marks Var $= \mu$. For lowly expressed genes ($\mu \ll 1$), observed variances lie close to the Poisson expectation, indicating sampling noise dominates. As mean expression increases ($\mu \gtrsim 1$), the **observed** variances exceed ( ), revealing **overdispersion** from technical noise (e.g. dropout, amplification bias) and genuine cell-to-cell heterogeneity. Hence, single-cell UMI counts are better modeled by a negative-binomial (or zero-inflated) distribution, which allows \$ Var(X) = + , ^2 \$ for some overdispersion parameter $\alpha > 0$.

3.0.3 1.3. Relationship between the mean and the Fano factor

Compute the Fano factor for each gene and make a scatter plot of expression mean vs. Fano factor in log-log coordinates, and interpret what you see in the plot. If the expression follows the Poisson distribution, then the Fano factor (variance/mean) should be equal to 1 for all genes.

(2.5 pts)

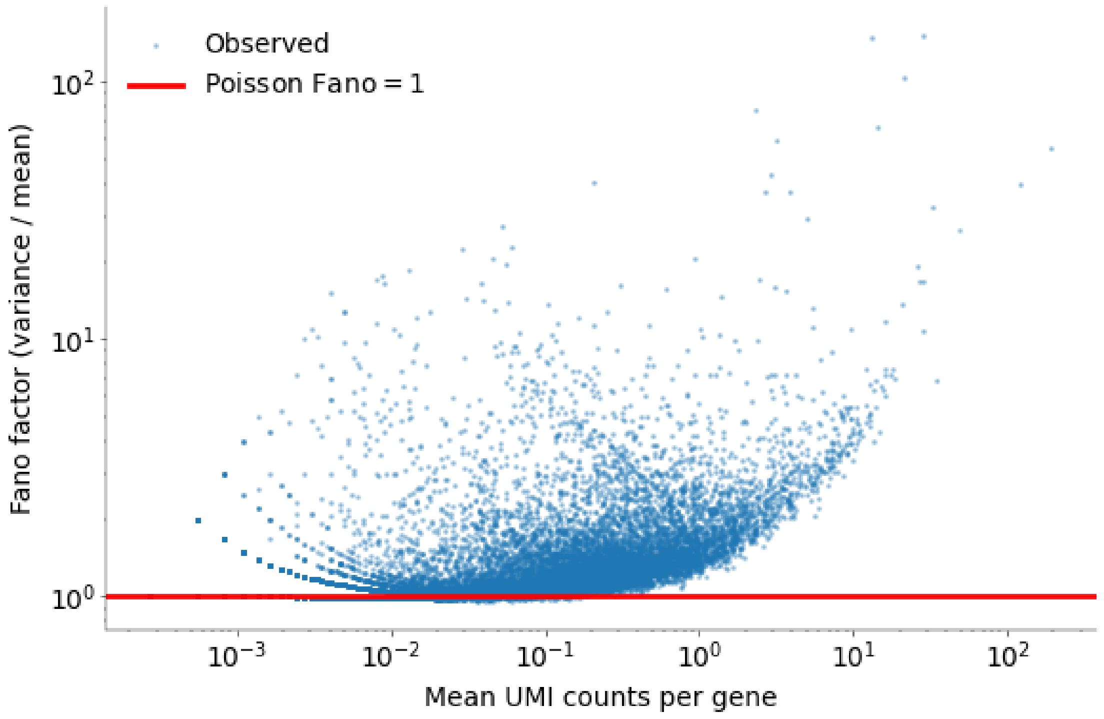
[10]:
```
# ---------------------------------------------
# Compute the Fano factor for each gene (0.5 pt)
# ---------------------------------------------

# (Fano factor = variance / mean)
```

```
fano = gene_vars / gene_means
fano
```

```
[10]: array([1.74867282, 0.999454  , 0.999727  , …, 1.41445784, 1.07355992,
             0.998089  ])
```

```
[11]: # -------------------------------
      # plot fano-factor vs mean (1 pt)
      # incl. fano factor
      # -------------------------------
      # Plot a Poisson prediction as line
      # Use the same style of plot as above.

      fig, ax = plt.subplots(figsize=(6, 4))

      # scatter observed mean vs Fano
      ax.scatter(gene_means, fano, s=5, alpha=0.3, label="Observed")

      # Poisson prediction: Fano = 1
      ax.axhline(1, color="r", lw=2, label="Poisson $\\mathrm{Fano}=1$")

      ax.set_xscale("log")
      ax.set_yscale("log")
      ax.set_xlabel("Mean UMI counts per gene")
      ax.set_ylabel("Fano factor (variance / mean)")
      ax.legend()
      plt.tight_layout()
      plt.show()
```

```
/var/folders/76/g6ys7mkj75zg7zyn8m093rk40000gn/T/ipykernel_82853/1615664090.py:2
1: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



Explanation (1 pt)

Under a Poisson model, the Fano factor is constant:

$$Fano = \frac{Var(X)}{\mu} = 1$$

Single-cell UMI counts are better modeled with a negative-binomial (or zero-inflated) distribution, which allows

$$Var(X) = \mu + \alpha\,\mu^2$$

for some overdispersion.

3.0.4 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 expreimental factors. Make a histogram of sequencing depths.

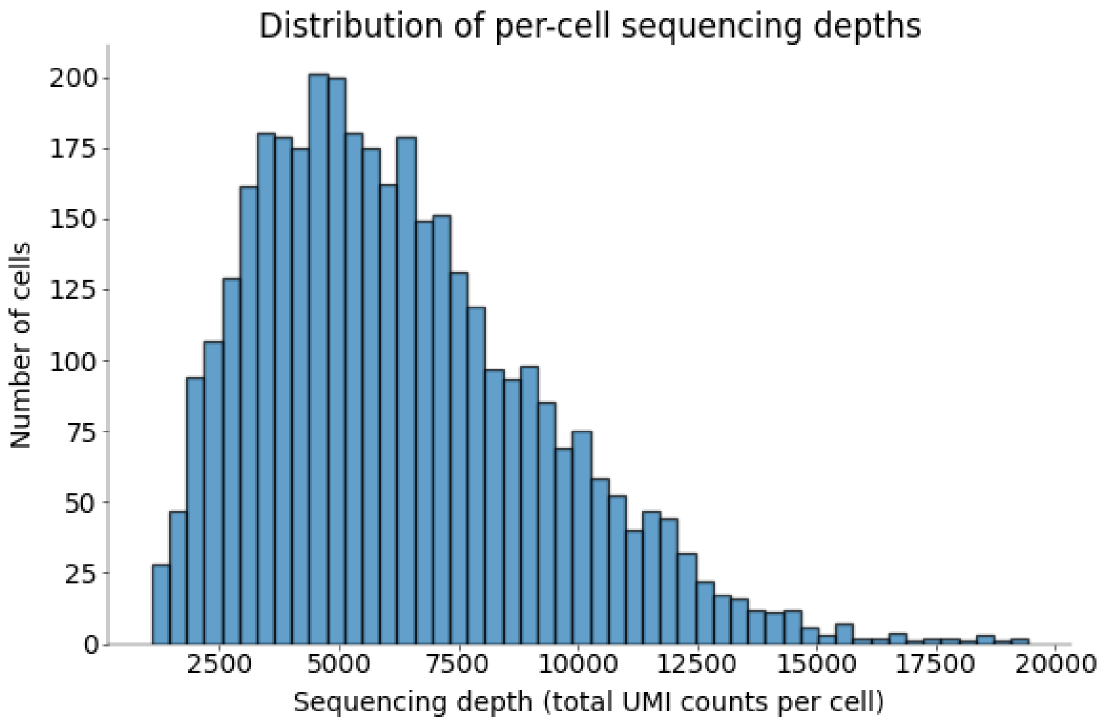
(1.5 pts)

[12]:
```
# --------------------------------
# Compute sequencing depth (0.5 pt)
# --------------------------------
# Sum UMI counts across all genes for each cell
depths = counts.sum(axis=1)
depths
```

```
[12]: array([ 1662,  2087,  2524, …,  9924, 10558,  9492])
```

```
[13]: # ------------------------------------------
      # Plot histogram of sequencing depths (1 pt)
      # ------------------------------------------

      fig, ax = plt.subplots(figsize=(6, 4))

      # choose a reasonable number of bins, e.g. 50
      ax.hist(depths, bins=50, edgecolor="black", alpha=0.7)

      ax.set_xlabel("Sequencing depth (total UMI counts per cell)")
      ax.set_ylabel("Number of cells")
      ax.set_title("Distribution of per-cell sequencing depths")
      plt.tight_layout()
      plt.show()
```

```
/var/folders/76/g6ys7mkj75zg7zyn8m093rk40000gn/T/ipykernel_82853/2341488430.py:1
3: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



3.0.5 1.5. Fano factors after normalization

Normalize counts by the sequencing depth of each cell and multiply by the median sequencing depth. Then make the same expression vs Fano factor plot as above. 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.

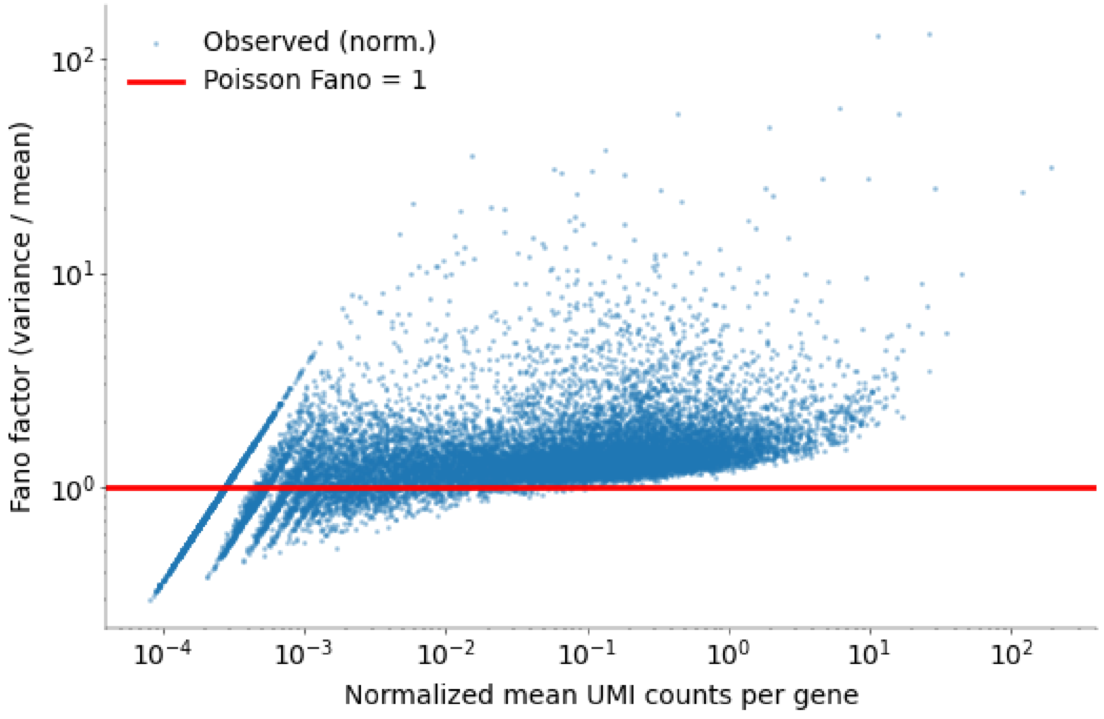
(2.5 pts)

[14]:
```
# --------------------------------------------------
# compute normalized counts and fano factor (1 pt)
# --------------------------------------------------
# depths has shape (n_cells,)
median_depth = np.median(depths)

# normalize each cell to the median depth
counts_norm = counts / depths[:, None] * median_depth

# recompute mean & variance per gene on normalized data
gene_means_norm = counts_norm.mean(axis=0)
gene_vars_norm = counts_norm.var(axis=0)

# fano factor after normalization
fano_norm = gene_vars_norm / gene_means_norm

fano_norm
```

[14]:
```
array([1.4743044 , 1.64107479, 1.01368967, …, 1.81740881, 1.18732466,
       0.78672188])
```

[15]:
```
# ----------------------------------------------------------
# plot normalized counts and find the top 10 genes (1 pt)
# hint: keep appropriate axis scaling in mind
# ----------------------------------------------------------

fig, ax = plt.subplots(figsize=(6, 4))

ax.scatter(gene_means_norm, fano_norm, s=5, alpha=0.3, label="Observed (norm.)")
ax.axhline(1, color="r", lw=2, label="Poisson Fano = 1")

ax.set_xscale("log")
ax.set_yscale("log")
ax.set_xlabel("Normalized mean UMI counts per gene")
ax.set_ylabel("Fano factor (variance / mean)")
ax.legend()
plt.tight_layout()
plt.show()
```

```
/var/folders/76/g6ys7mkj75zg7zyn8m093rk40000gn/T/ipykernel_82853/2063836131.py:1
6: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



[16]:
```
# ----------------------------------------------------------------------
# Find top-10 genes with the highest normalized Fano factor (0.5 pts)
# Print them sorted by the Fano factor starting from the highest
# Gene names are stored in the `genes` array
# ----------------------------------------------------------------------

# get indices of genes sorted by descending fano_norm
top10_idx = np.argsort(fano_norm)[::-1][:10]

# print gene names and their normalized Fano factors
for i in top10_idx:
    print(f"{genes[i]:<15}  Fano={fano_norm[i]:.2f}")
```

```
Sst              Fano=131.14
Npy              Fano=128.40
Vip              Fano=59.11
Cck              Fano=55.65
Cpne2            Fano=55.22
```

```
Pcp4                Fano=47.65
Ptpn23              Fano=37.25
Pdzd9               Fano=35.06
Malat1              Fano=31.42
Armc2               Fano=30.59
```

4 Task 2: Low dimensional visualization

In this task we will construct a two dimensional visualization of the data. First we will normalize the data with some variance stabilizing transformation and study the effect that different approaches have on the data. Second, we will reduce the dimensionality of the data to a more feasible number of dimensions (e.g. $d = 50$) using PCA. And last, we will project the PCA-reduced data to two dimensions using t-SNE.

4.0.1 2.1. PCA with and without transformations

Here we look at the influence of variance-stabilizing transformations on PCA. We will focus on the following transformations: - Square root (`sqrt(X)`): it is a variance-stabilizing transformation for the Poisson data. - Log-transform (`log2(X+1)`): it is also often used in the transcriptomic community.

We will only work with the most important genes. For that, transform the counts into normalized counts (as above) and select all genes with normalized Fano factor above 3 and remove the rest. We will look at the effect that both transformations have in the PCA-projected data by visualizing the first two components. Interpret qualitatively what you see in the plot and compare the different embeddings making use of the ground truth clusters.

(3.5 pts)

[46]:
```python
# --------------------------------
# Select important genes (0.5 pts)
# --------------------------------

# keep genes with normalized Fano factor > 3
depths = counts.sum(axis=1)
counts_norm = counts / depths[:, None] * np.median(depths)

fano_norm = counts_norm.var(axis=0) / counts_norm.mean(axis=0)
hv_mask = fano_norm > 3
counts_sel = counts_norm[:, hv_mask]
genes_sel = genes[hv_mask]
print("Selected", counts_sel.shape[1], "highly variable genes")
```

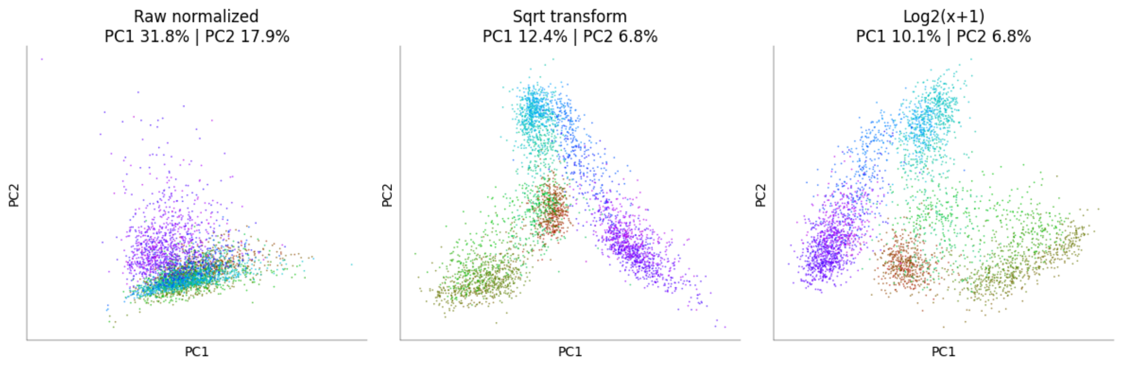
```
Selected 707 highly variable genes
```

[47]:
```python
# --------------------------------------
# transform data and apply PCA (1 pt)
# --------------------------------------
```

```
from sklearn.decomposition import PCA

raw_data = counts_sel
sqrt_data = np.sqrt(counts_sel)
log_data = np.log2(counts_sel + 1)

pc_raw = PCA(n_components=2).fit_transform(raw_data)
pc_sqrt = PCA(n_components=2).fit_transform(sqrt_data)
pc_log = PCA(n_components=2).fit_transform(log_data)
```

[55]:
```
# ------------------------------------------------
# plot first 2 PCs for each transformation (1 pt)
# ------------------------------------------------

fig, axs = plt.subplots(1, 3, figsize=(12, 4), facecolor="white")
for ax, pca, pc, title in zip(
    axs,
    [pca_raw, pca_sqrt, pca_log],
    [pc_raw, pc_sqrt, pc_log],
    ["Raw normalized", "Sqrt transform", "Log2(x+1)"],
):
    v1, v2 = pca.explained_variance_ratio_[:2]
    ax.scatter(
        pc[:, 0], pc[:, 1], s=6, alpha=0.7, c=clusterColors[clusters],␣
  ↪edgecolors="none"
    )
    ax.set_title(f"{title}\nPC1 {v1*100:.1f}% | PC2 {v2*100:.1f}%", fontsize=12)
    ax.set_xlabel("PC1")
    ax.set_ylabel("PC2")
    ax.set_xticks([])
    ax.set_yticks([])
    for side in ["top", "right"]:
        ax.spines[side].set_visible(False)

plt.tight_layout()
plt.show()
```

```
/var/folders/76/g6ys7mkj75zg7zyn8m093rk40000gn/T/ipykernel_82853/3816277384.py:2
6: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



Explanation (1 pt) …

The three panels show PCA on the top Fano > 3 genes after different variance-stabilizing transforms.

- **Raw normalized**: $\text{PC1} = 31.8\%$ and $\text{PC2} = 17.9\%$ of total variance. The embedding is stretched along PC1, reflecting a few highly expressed genes dominating the variance. Clusters are partially separable but elongated.

- **Sqrt transform**: $\text{PC1} = 12.4\%$ and $\text{PC2} = 6.8\%$. The square-root stabilizes Poisson noise, producing three "arms" that better resolve major cell-type groups (e.g. Sst, Pvalb, Vip).

- **Log (x+1) transform**: $\text{PC1} = 10.1\%$ and $\text{PC2} = 6.8\%$. This also compresses extreme values and separates clusters, though with slightly less dynamic range than the sqrt.

Overall, variance stabilization ($\surd$ or log) before PCA yields cleaner, more balanced embeddings by down-weighting outlier genes and highlighting biological heterogeneity.

4.0.2 2.2. tSNE with and without transformations

Now, we will reduce the dimensionality of the PCA-reduced data further to two dimensions using t-SNE. We will use only $n = 50$ components of the PCA-projected data. Plot the t-SNE embedding for the three versions of the data and interpret the plots. Do the different transformations have any effect on t-SNE?

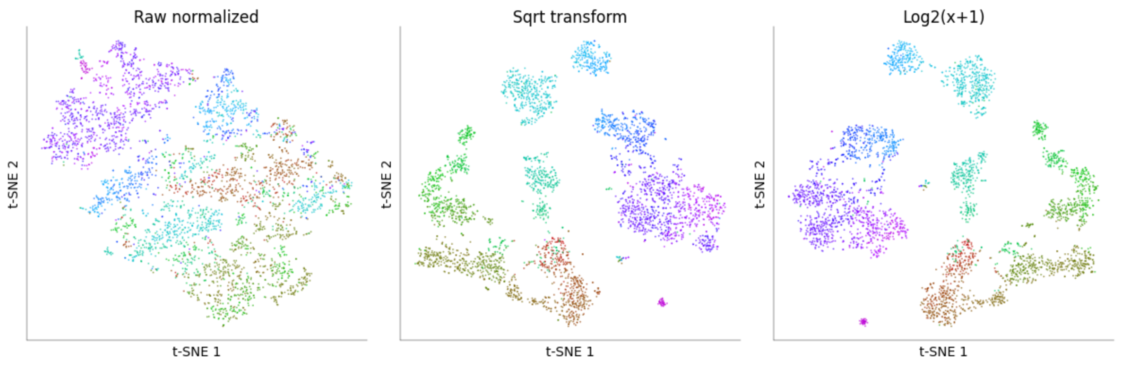
(1.5 pts)

[49]:
```python
# ------------------------
# Perform tSNE (0.5 pts)
# ------------------------

from openTSNE import TSNE

pca50_raw = PCA(n_components=50).fit_transform(raw_data)
pca50_sqrt = PCA(n_components=50).fit_transform(sqrt_data)
pca50_log = PCA(n_components=50).fit_transform(log_data)
```

```
tsne_raw = TSNE(n_components=2, perplexity=30, random_state=0).fit(pca50_raw)
tsne_sqrt = TSNE(n_components=2, perplexity=30, random_state=0).fit(pca50_sqrt)
tsne_log = TSNE(n_components=2, perplexity=30, random_state=0).fit(pca50_log)
```

[50]:
```
# ----------------------------------------------
# plot t-SNE embedding for each dataset (1 pt)
# ----------------------------------------------

fig, axs = plt.subplots(1, 3, figsize=(12, 4), facecolor="white")
for ax, emb, title in zip(
    axs,
    [tsne_raw, tsne_sqrt, tsne_log],
    ["Raw normalized", "Sqrt transform", "Log2(x+1)"],
):
    ax.scatter(
        emb[:, 0],
        emb[:, 1],
        s=6,
        alpha=0.7,
        c=clusterColors[clusters],
        edgecolors="none",
    )
    ax.set_title(title)
    ax.set_xlabel("t-SNE 1")
    ax.set_ylabel("t-SNE 2")
    ax.set_xticks([])
    ax.set_yticks([])
    for side in ["top", "right"]:
        ax.spines[side].set_visible(False)
plt.tight_layout()
plt.show()
```

```
/var/folders/76/g6ys7mkj75zg7zyn8m093rk40000gn/T/ipykernel_82853/3616080685.py:1
7: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



4.0.3 2.3. Leiden clustering

Now we will play around with some clustering and see whether the clustering methods can produce similar results to the original clusters from the publication. 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.

Choose one representation of the data (best transformation based in your results from the previous task) to use further in this task and justify your choice. Think about which level of dimensionality would be sensible to use to perform clustering. Visualize in the two-dimensional embedding the resulting clusters and compare to the original clusters.

(1.5 pts)

[36]:
```
# 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
```

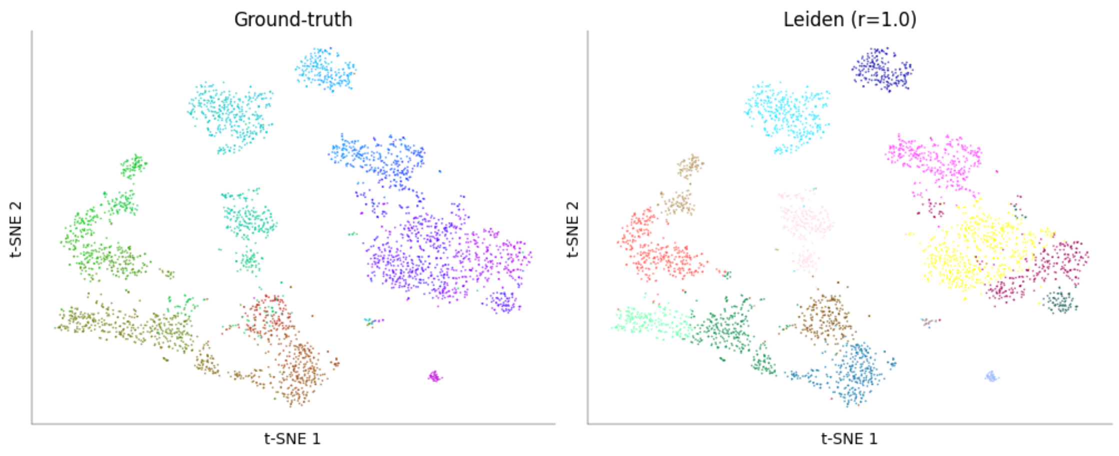
[37]:
```
# Define some contrast colors

clusterCols = [
    "#FFFF00",
    "#1CE6FF",
    "#FF34FF",
    "#FF4A46",
    "#008941",
    "#006FA6",
    "#A30059",
    "#FFDBE5",
    "#7A4900",
    "#0000A6",
    "#63FFAC",
    "#B79762",
    "#004D43",
    "#8FB0FF",
    "#997D87",
    "#5A0007",
    "#809693",
    "#FEFFE6",
    "#1B4400",
    "#4FC601",
    "#3B5DFF",
    "#4A3B53",
    "#FF2F80",
    "#61615A",
    "#BA0900",
    "#6B7900",
    "#00C2A0",
    "#FFAA92",
    "#FF90C9",
    "#B903AA",
    "#D16100",
    "#DDEFFF",
    "#000035",
    "#7B4F4B",
    "#A1C299",
    "#300018",
    "#0AA6D8",
    "#013349",
    "#00846F",
    "#372101",
    "#FFB500",
    "#C2FFED",
    "#A079BF",
    "#CC0744",
    "#C0B9B2",
    "#C2FF99",
    "#001E09",
    "#00489C",
    "#6F0062",
    "#0CBD66",
    "#EEC3FF",
    "#456D75",
    "#B77B68",
    "#7A87A1",
    "#788D66",
    "#885578",
    "#FAD09F",
    "#FF8A9A",
    "#D157A0",
    "#BEC459",
    "#456648",
    "#0086ED",
    "#886F4C",
    "#34362D",
    "#B4A8BD",
    "#00A6AA",
    "#452C2C",
    "#636375",
    "#A3C8C9",
    "#FF913F",
    "#938A81",
    "#575329",
    "#00FECF",
    "#B05B6F",
    "#8CD0FF",
    "#3B9700",
    "#04F757",
    "#C8A1A1",
    "#1E6E00",
    "#7900D7",
    "#A77500",
    "#6367A9",
    "#A05837",
    "#6B002C",
    "#772600",
    "#D790FF",
    "#9B9700",
    "#549E79",
    "#FFF69F",
    "#201625",
    "#72418F",
    "#BC23FF",
    "#99ADC0",
    "#3A2465",
    "#922329",
    "#5B4534",
    "#FDE8DC",
    "#404E55",
    "#0089A3",
    "#CB7E98",
    "#A4E804",
    "#324E72",
    "#6A3A4C",
    "#83AB58",
    "#001C1E",
    "#D1F7CE",
    "#004B28",
    "#C8D0F6",
    "#A3A489",
    "#806C66",
    "#222800",
    "#BF5650",
    "#E83000",
    "#66796D",
    "#DA007C",
    "#FF1A59",
    "#8ADBB4",
    "#1E0200",
    "#5B4E51",
    "#C895C5",
    "#320033",
    "#FF6832",
    "#66E1D3",
    "#CFCDAC",
    "#D0AC94",
    "#7ED379",
    "#012C58",
]

clusterCols = np.array(clusterCols)
```

[51]:
```
# -------------------------------------------------------
# create graph and run leiden clustering on it (0.5 pts)
# hint: use `la?`, `la.find_partition?` and `ig.Graph?`
# to find out more about the provided packages.
# -------------------------------------------------------

A = kneighbors_graph(
    pca50_sqrt, n_neighbors=15, mode="connectivity", include_self=False
)
src, tgt = A.nonzero()
g = ig.Graph(edges=list(zip(src, tgt)), directed=False)
g.simplify()
part = la.find_partition(g, la.RBConfigurationVertexPartition,␣
 ↪resolution_parameter=1.0)
labels1 = np.array(part.membership)
print("Leiden r=1.0 →", len(np.unique(labels1)), "clusters")
```

```
Leiden r=1.0 → 15 clusters
```

[52]:
```
# --------------------------
# Plot the results (1 pt)
# --------------------------

from sklearn.metrics import adjusted_rand_score

fig, axs = plt.subplots(1, 2, figsize=(10, 4), facecolor="white")
for ax, labs, title in zip(
    axs, [clusters, labels1], ["Ground-truth", "Leiden (r=1.0)"]
):
    ax.scatter(
        tsne_sqrt[:, 0],
        tsne_sqrt[:, 1],
        s=6,
        alpha=0.7,
        c=(clusterColors if title == "Ground-truth" else clusterCols)[labs],
        edgecolors="none",
    )
    ax.set_title(title)
    ax.set_xlabel("t-SNE 1")
    ax.set_ylabel("t-SNE 2")
    ax.set_xticks([])
    ax.set_yticks([])
    for side in ["top", "right"]:
        ax.spines[side].set_visible(False)

ari = adjusted_rand_score(clusters, labels1)
print(f"ARI = {ari:.2f}")
```

```
ARI = 0.40
```



4.0.4 2.4. Change the clustering resolution

The number of clusters can be changed by modifying the resolution parameter. How many clusters did we get with the default value? Change the resolution parameter to yield 2x more and 2x fewer clusters Plot all three results as t-SNE overlays (same as above).

(1.5 pts)

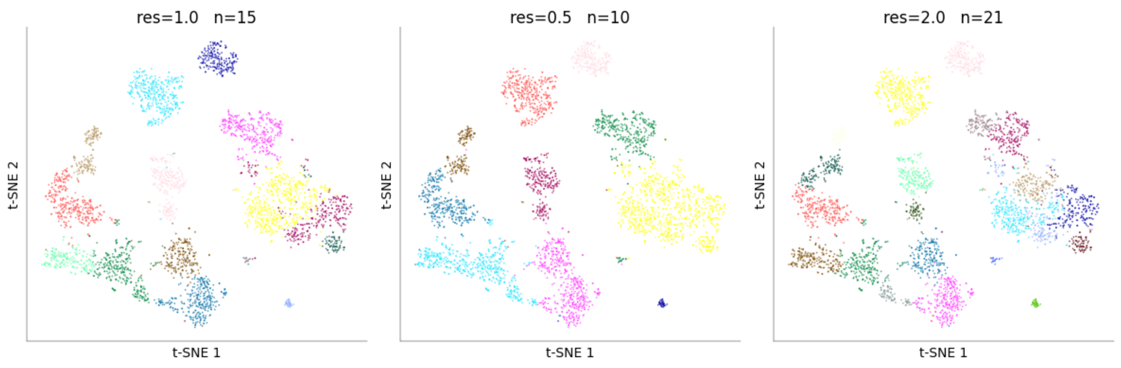
[53]:
```
# --------------------------------------------------------------------
# run the clustering for 3 different resolution parameters (0.5 pts)
# --------------------------------------------------------------------

res = [1.0, 0.5, 2.0]
labels = []
counts = []
```

```
for r in res:
    p = la.find_partition(g, la.RBConfigurationVertexPartition,␣
 ↪resolution_parameter=r)
    labs = np.array(p.membership)
    labels.append(labs)
    counts.append(len(np.unique(labs)))
print(dict(zip(res, counts)))
```

```
{1.0: 15, 0.5: 10, 2.0: 21}
```

[54]:
```
# --------------------------
# Plot the results (1 pt)
# --------------------------

fig, axs = plt.subplots(1, 3, figsize=(12, 4), facecolor="white")
for ax, r, labs, cnt in zip(axs, res, labels, counts):
    ax.scatter(
        tsne_sqrt[:, 0],
        tsne_sqrt[:, 1],
        s=6,
        alpha=0.7,
        c=clusterCols[labs],
        edgecolors="none",
    )
    ax.set_title(f"res={r}   n={cnt}")
    ax.set_xlabel("t-SNE 1")
    ax.set_ylabel("t-SNE 2")
    ax.set_xticks([])
    ax.set_yticks([])
    for side in ["top", "right"]:
        ax.spines[side].set_visible(False)
plt.tight_layout()
plt.show()
```

```
/var/folders/76/g6ys7mkj75zg7zyn8m093rk40000gn/T/ipykernel_82853/311304628.py:13
: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



```
[ ]:
```