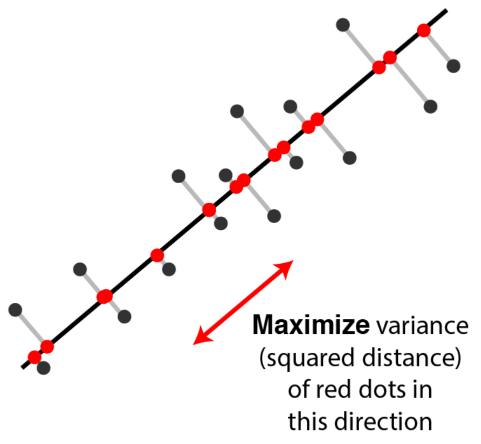
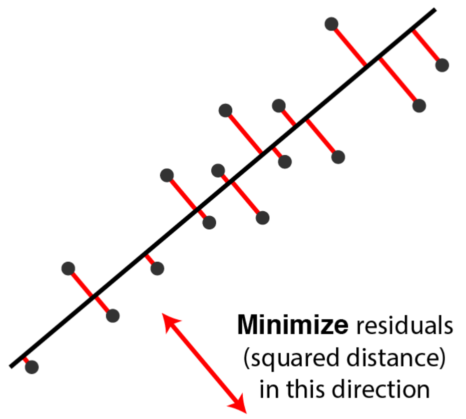
Lecture 11: Dimensionality Reduction with MDS and TSNE

Last class: PCA

- **Type**: Unsupervised learning (dimensionality reduction)
- **Model family**: Linear projection ($f_\theta(x) = W^\top x$)
- **Objective function**: Reconstruction error or variance maximization
- **Optimizer**: Matrix eigendecomposition

PCA losses: max variance = min reconstruction





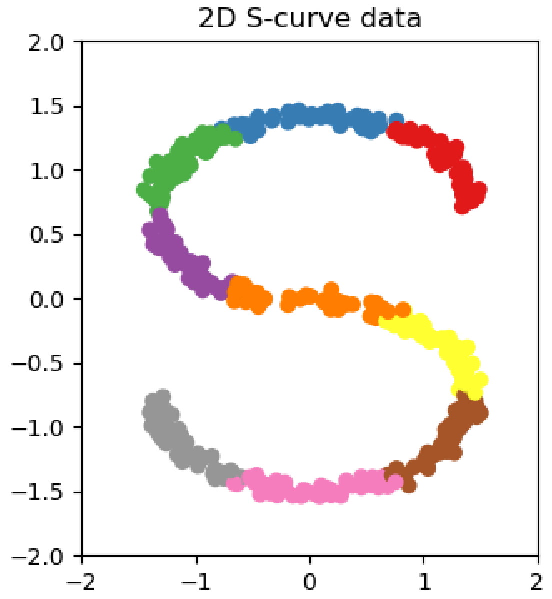
$$\arg\max_{\theta} \sum_{i=1}^{n} \left[\| f_\theta(x^{(i)}) - \sum_{j=1}^{n} [f_\theta(x^{(j)})] \|^2 \right] = \arg\min_{\theta} \sum_{i=1}^{n} \| x^{(i)} - f_\theta^{-1}(f_\theta(x^{(i)})) \|_2^2$$

When PCA fails

PCA is intuitive and easy to implement, but fails on non-linear structures

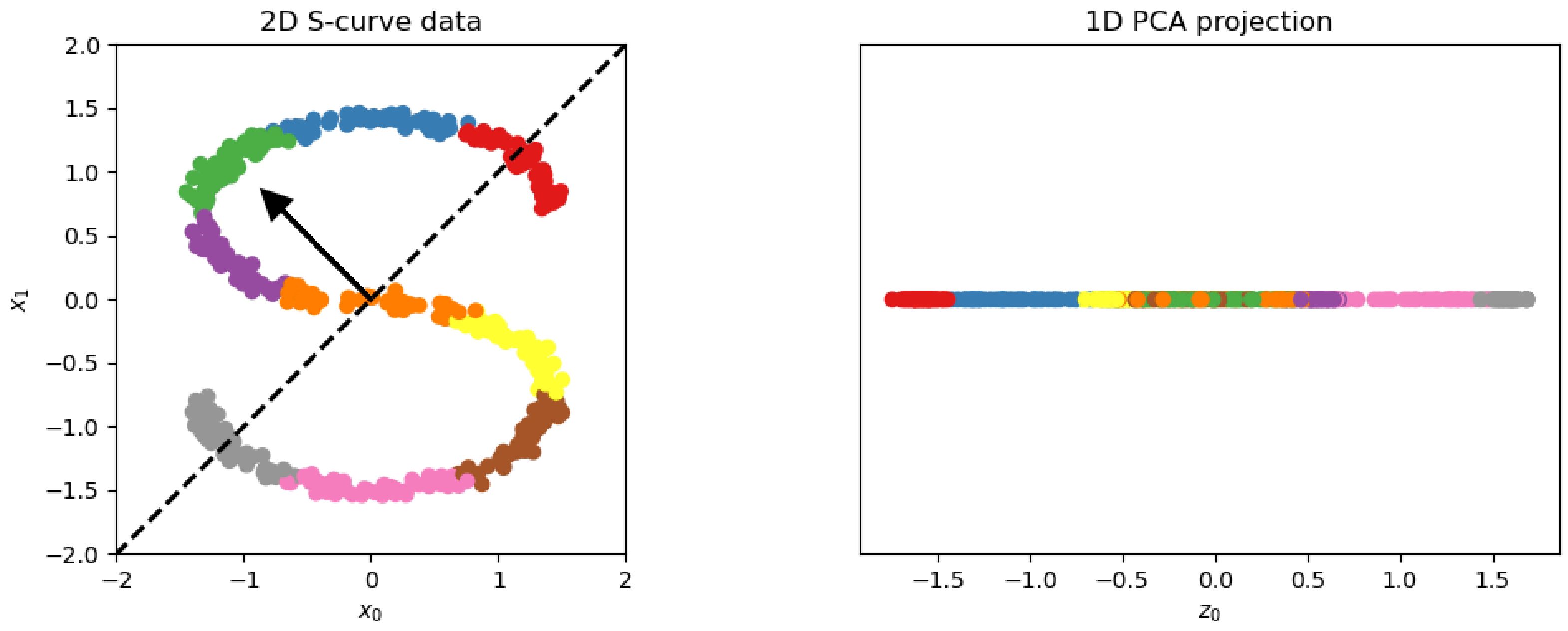
Let's consider this 2D S-curve. **What will the PCA look like?**

```
X, y = make_s_curve_2d(n_samples=500, noise=0.05)
fig, ax = fig_ax_with_padding()
ax.set_title('2D S-curve data')
ax.scatter(X[:,0], X[:,1], c=y, cmap='Set1');
```



Running our PCA implementation on it gives:

```
def pca_project(X, p=2):
    np.random.seed(0)
    Sigma = X.T.dot(X) / X.shape[0] # form covariance matrix
    L, Q = np.linalg.eig(Sigma) # perform eigendecomposition
    W = Q[:,:p] # get top p eigenvectors
    Z = X.dot(W) # project on these eigenvectors
    return np.real(Z), W # return projected points and the projection matrix

Z, W = pca_project(X, p=2)
plot_pca_1d(X, Z, W)
```

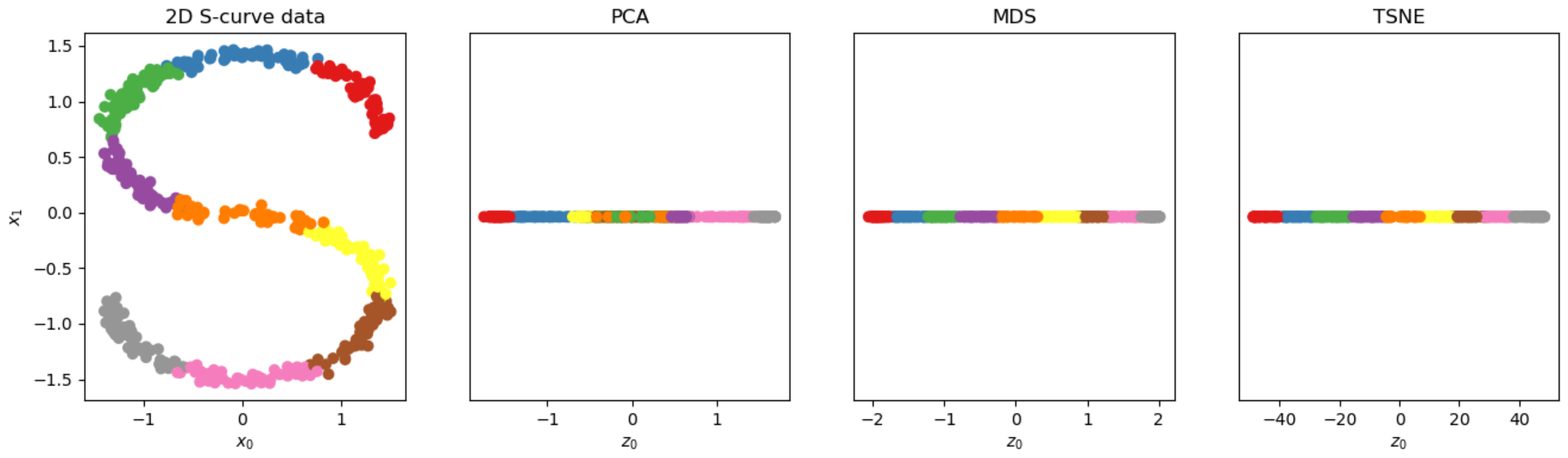


So what happened? PCA's linear projection fails on the non-linear structure

This class: Two non-linear visualization and embedding methods:

1. Multidimensional scaling (MDS)
2. t-distributed stochastic neighbor embedding (t-SNE)

```
visualize_pca_mds_tsne(X, y, p=1);
```



Part 1: Multidimensional scaling (MDS)

Distinguishing projections and embeddings

PCA is based on **projections**

$$f_\theta(x) = W^\top x \in \mathbb{R}^d \text{ where } \theta = W$$

- PCA parameterizes a (linear) mapping of the data

This mapping gives **embeddings**, defined as $z^{(i)} = f_\theta\left(x^{(i)}\right)$

- The embeddings are obtained by applying the mapping to every data point

Embedding-based visualizations

In contrast to PCA, MDS and TSNE directly parameterize the **embeddings**

$$f_\theta\left(x^{(i)}\right) = z^{(i)} \in \mathbb{R}^d \text{ where } \theta = \left\{z^{(i)}\right\}_{i=1}^{n}$$

- Extremely connected to the training dataset, cannot apply to new points
- There is no explicit map or projection
- Gives much more flexibility, points can be associated anywhere

Optimizing embeddings

We can't use the same objective as PCA. Let's see why.

Optimizing variance doesn't work ❌

$$\arg\max_{\theta} \sum_{i=1}^{n} \left[\|f_\theta(x^{(i)}) - \sum_{j=1}^{n} [f_\theta(x^{(j)})]\|^2 \right]$$

In PCA, this worked because the embeddings were constrained to the hyperplane of the projection.

With just embeddings, nothing connects $z^{(i)}$ back to $x^{(i)}$ to constrain them. So, the variance will go to infinity

Reconstruction error also doesn’t work ❌

$$\arg\min_{\theta} \sum_{i=1}^{n} \|x^{(i)} - f_\theta^{-1}(f_\theta(x^{(i)}))\|_2^2$$

In PCA, this again worked because of the constraints of the map.

With just embeddings, arguably the inverse of the embedding is already just the original data point, i.e., $f_\theta^{-1}(f_\theta(x^{(i)})) = x^{(i)}$. So, the objective is zero everywhere

Reconstructing pairwise distances

Consider a pair of points $x^{(i)}$ and $x^{(j)}$. Ideally we want the distance of the corresponding embeddings to match their distance:

$$\|x^{(i)} - x^{(j)}\| \approx \|z^{(i)} - z^{(j)}\|$$

Since the embeddings are in a lower-dimensional space, the distances won't be perfectly matched.

But, they'll get close and hopefully give us a good visualization.

The stress loss function

We define the **stress** loss function on every pair of points as

$$J_{\text{stress}}(\theta) = \sum_{i \neq j} (\|x^{(i)} - x^{(j)}\| - \|z^{(i)} - z^{(j)}\|)^2$$

Question: Does PCA also optimize stress under the linear mapping constraint?

Optimizing MDS

There is not a closed-form solution to optimizing the stress of the embeddings.

MDS optimization is usually done with an algorithm called SMACOF (scaling by majorization a complicated function), which iteratively bounds the objective with a convex function.

Optimizing MDS

Here, we will simply do MDS with gradient descent
(noting it is not-standard and not optimal)

1. Initialize the embeddings $\theta = \{z_i\}$.
2. Repeat until converged
    - $\theta \leftarrow \theta - \alpha \nabla_\theta J_{\text{stress}}(\theta)$

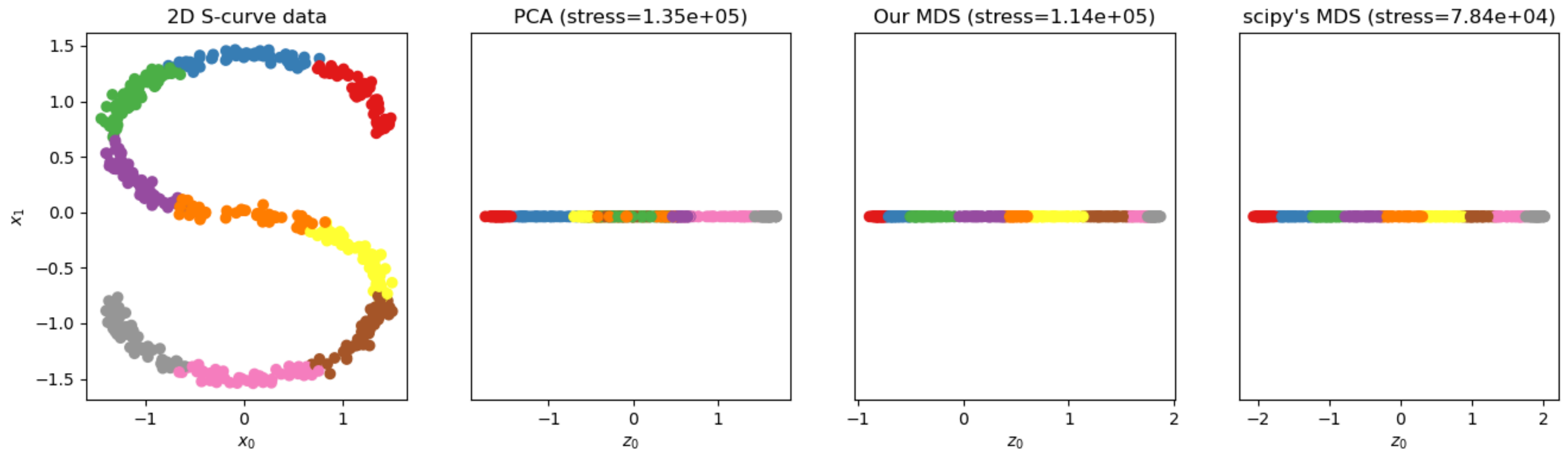
Let's implement it:

```python
import jax # use jax for speed
import jax.numpy as jnp

def euclidean_distances(X):
    return jnp.sqrt(jnp.sum((X[:,None] - X[None])**2, axis=-1))

def stress_fn(Z, D):
    Z_dists = euclidean_distances(Z)
    val = jnp.sum((D - Z_dists)**2) / 2.
    return val

def stress_grad_fn(Z, D):
    Z_dists = euclidean_distances(Z)
    grad = (Z[:,None] - Z[None]) * (Z_dists - D)[:,:,None]
    grad = jnp.sum(grad, axis=1)
    return grad

stress_fn = jax.jit(stress_fn)
stress_grad_fn = jax.jit(stress_grad_fn)

def mds(X, p=2, num_iterations=2000, lr=1e-2, verbose=False):
    np.random.seed(0)
    D = euclidean_distances(X) # only need to compute once
    Z = np.random.uniform(size=(X.shape[0], p))
    for i in range(num_iterations):
        if i % 5000 == 0 and verbose:
            loss = stress_fn(Z, D)
            print(f'Iteration {i}, loss: {loss:.2e}')
        grad_Z = stress_grad_fn(Z, D)
        Z -= lr * grad_Z / Z.shape[0]**2
    return Z
```

Let's run the MDS code on the S-curve data

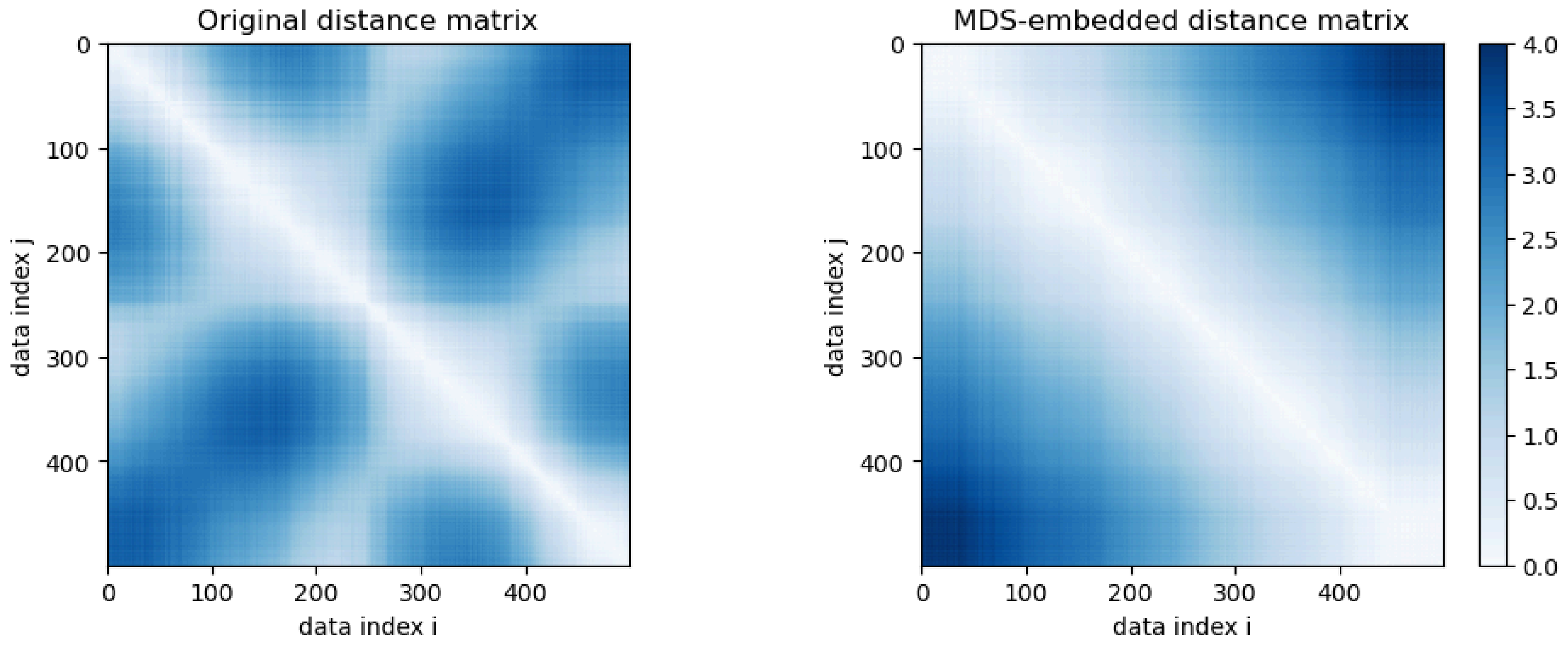
```
Z = mds(X, p=1, num_iterations=50000, verbose=True)
Z_scipy = visualize_our_mds(X, y, Z)
```

```
Iteration 0, loss: 3.32e+05
Iteration 5000, loss: 3.05e+05
Iteration 10000, loss: 2.76e+05
Iteration 15000, loss: 2.48e+05
Iteration 20000, loss: 2.20e+05
Iteration 25000, loss: 1.95e+05
Iteration 30000, loss: 1.72e+05
Iteration 35000, loss: 1.53e+05
Iteration 40000, loss: 1.37e+05
Iteration 45000, loss: 1.24e+05
```



It's also useful to look at the distance matrices. This shows the difficulty of reproducing the full fidelity of the original distances.

```
I = np.argsort(y); y = y[I]; X = X[I]; Z = Z[I]; Z_scipy = Z_scipy[I] # sort by location

D_original = euclidean_distances(X)
D_embedded = euclidean_distances(Z_scipy)

fig, axs = plt.subplots(1, 2, figsize=(12, 4))
axs[0].imshow(D_original, cmap='Blues', vmin=0., vmax=4.)
img = axs[1].imshow(D_embedded, cmap='Blues', vmin=0., vmax=4.)
axs[0].set_title('Original distance matrix'); axs[1].set_title('MDS-embedded distance matrix'); fig.colorbar(img, cmap='Blues');
for ax in axs: ax.set_xlabel('data index i'); ax.set_ylabel('data index j')
```



Multidimensional scaling (MDS)

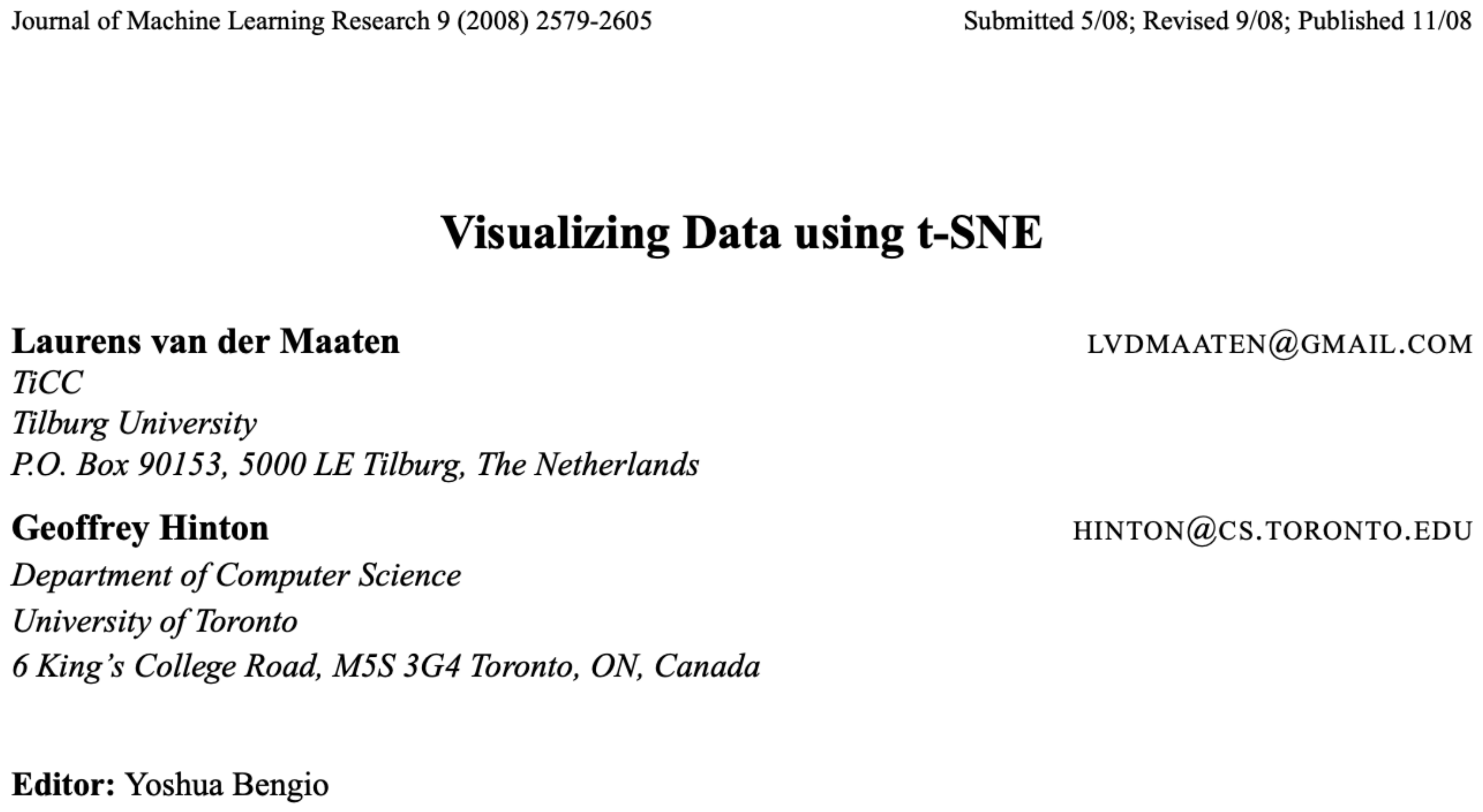
- **Type**: Unsupervised learning (dimensionality reduction)
- **Model family**: Embeddings ($f_\theta(x^{(i)}) = z^{(i)}$)
- **Objective function**: Stress $J_{\text{stress}}(\theta) = \sum_{i \neq j}(\|x^{(i)} - x^{(j)}\| - \|z^{(i)} - z^{(j)}\|)^2$
- **Optimizer**: Iterative (ideally SMACOF)

The MDS variant we have covered is called "*metric MDS*", and there are others.

On the homework, we will see one called "*classical MDS*" (or principal coordinates analysis) that optimizes a different loss (called strain) and has a closed-form solution.

Part 2: t-distributed stochastic neighbor embedding (TSNE)

47.8k citations (Oct 9, 2024)



Journal of Machine Learning Research 9 (2008) 2579-2605 Submitted 5/08; Revised 9/08; Published 11/08

Visualizing Data using t-SNE

Laurens van der Maaten LVDMAATEN@GMAIL.COM
TiCC
Tilburg University
P.O. Box 90153, 5000 LE Tilburg, The Netherlands

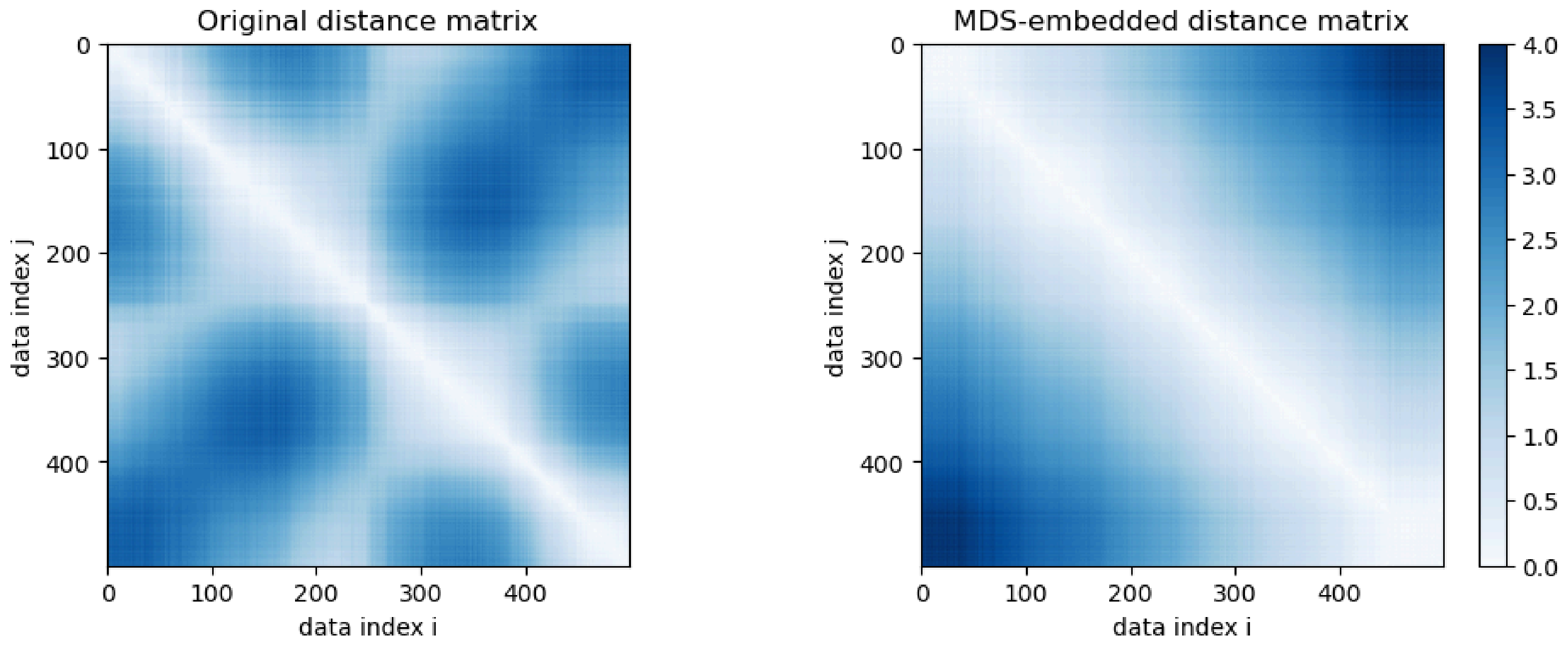
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Department of Computer Science
University of Toronto
6 King's College Road, M5S 3G4 Toronto, ON, Canada

Editor: Yoshua Bengio

Difficulties of reproducing pairwise distances

Reproducing all of the pairwise distances in a lower-dimensional space is hard, even when we have full control over the embedding locations.

```
fig # display the MDS distances from before
```



How to improve poor pairwise distance reconstructions?

1. If we just care about reproducing the distances ad not visualization, we could make the dimension of the embedding space larger
2. If we want to stay in a low-dimensional space, we can change the notion of distance (this is what TSNE does)

Probabilistic (stochastic) pairwise distances

Given a point $x^{(i)}$, measure the "distance" to another point by the probability the distance to that point was sampled from a Gaussian, and normalize it.

Formally for $j \neq i$, this is $p_{j|i} \propto \exp\{-\|x^{(i)} - x^{(j)}\|^2 / 2\sigma_i^2\}$ normalized so that $\sum_j p_{j|i} = 1$. Then, the probabilities are symmetrized as $p_{ij} \propto p_{j|i} + p_{i|j}$.

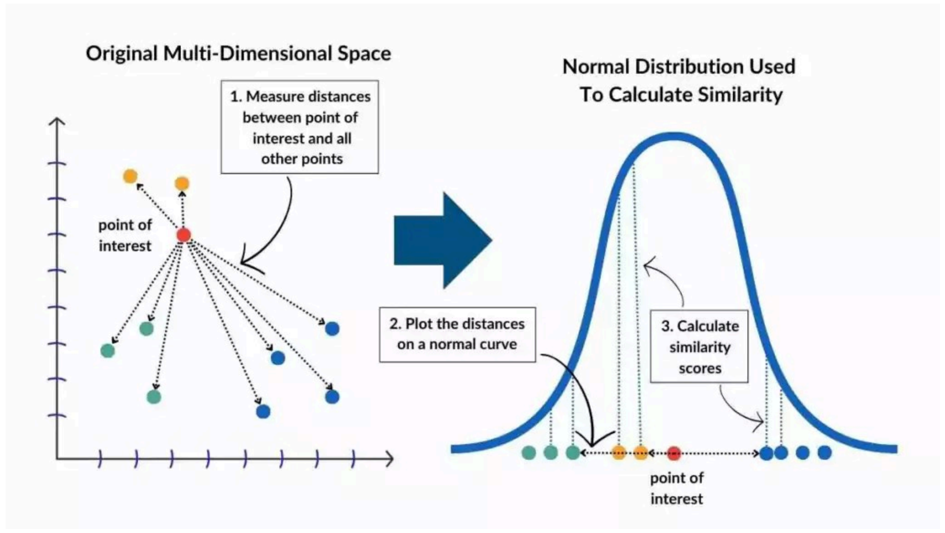
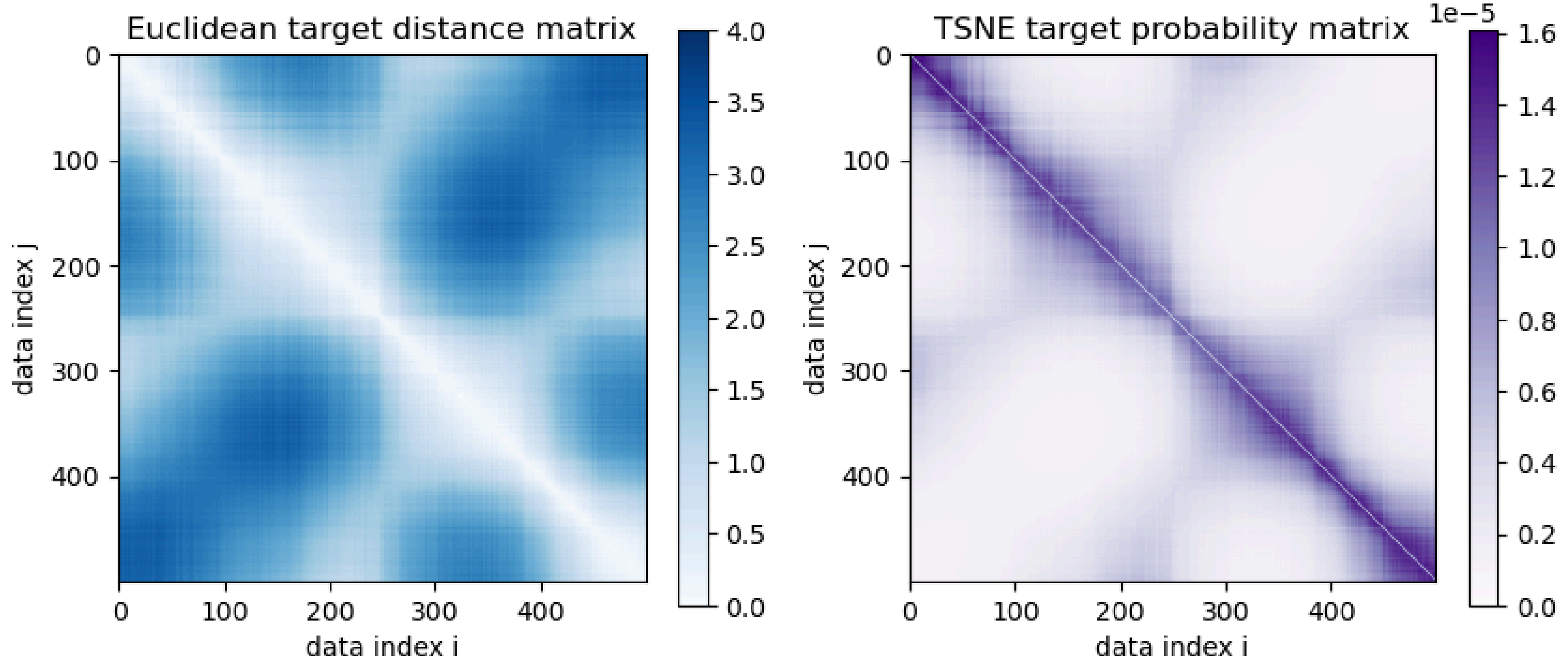

Original Multi-Dimensional Space
1. Measure distances between point of interest and all other points
point of interest
Normal Distribution Used To Calculate Similarity
2. Plot the distances on a normal curve
3. Calculate similarity scores
point of interest

Image source: https://spotintelligence.com/2023/12/22/practical-guide-t-sne/

```python
def gaussian_distances(X, bandwidth):
    return np.exp(-euclidean_distances(X) / (2.*bandwidth))

def get_tsne_target_probability(X, bandwidth=0.6):
    D = gaussian_distances(X, bandwidth=bandwidth) # in the full TSNE, the bandwidth searched over for each i.
    p_j_given_i = D / (D.sum(axis=1) - 1.) # normalize the distances so \sum_j p_{j|i}=1 for every i.
    np.fill_diagonal(p_j_given_i, 0.) # p_{i|i} = 0 for every i.
    p_ij = (p_j_given_i + p_j_given_i.T) / (2.*X.shape[0]) # symmetrize and normalize so \sum_{i,j} p_{i,j} = 1.
    return p_ij

p_ij = get_tsne_target_probability(X)
fig, axs = plt.subplots(1, 2, figsize=(10, 4))
img = axs[0].imshow(D_original, cmap='Blues', vmin=0., vmax=4.); fig.colorbar(img)
img = axs[1].imshow(p_ij, cmap='Purples'); fig.colorbar(img)
axs[0].set_title('Euclidean target distance matrix'); axs[1].set_title('TSNE target probability matrix')
for ax in axs: ax.set_xlabel('data index i'); ax.set_ylabel('data index j')
```



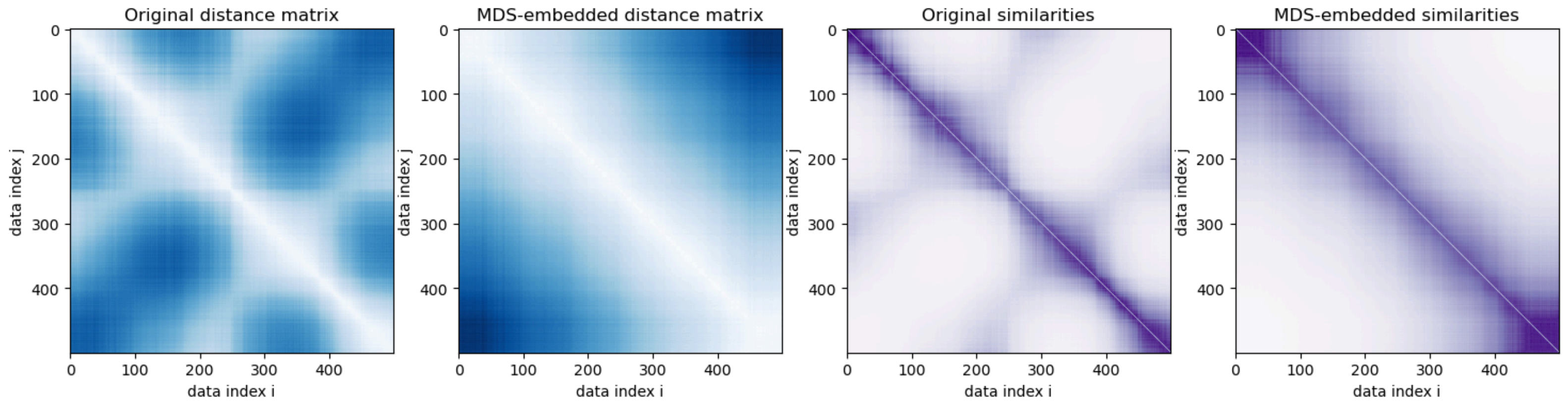
We can also see that the new pairwise similarity matrix is a valid probability distribution and has entries that sum to 1:

```python
print(f'sum_ij p_ij: {p_ij.sum():.2f}')
```

```
sum_ij p_ij: 1.00
```

We can look at our MDS solution under this new probabilistic similarity:

```
p_ij_embedded = get_tsne_target_probability(Z_scipy)
fig, axs = plt.subplots(1, 4, figsize=(18, 4))
axs[0].imshow(D_original, cmap='Blues', vmin=0., vmax=4.)
axs[1].imshow(D_embedded, cmap='Blues', vmin=0., vmax=4.)
axs[2].imshow(p_ij, cmap='Purples')
axs[3].imshow(p_ij_embedded, cmap='Purples')
axs[0].set_title('Original distance matrix'); axs[1].set_title('MDS-embedded distance matrix')
axs[2].set_title('Original similarities'); axs[3].set_title('MDS-embedded similarities')
for ax in axs: ax.set_xlabel('data index i'); ax.set_ylabel('data index j')
```



The TSNE loss: the Kullback-Leibler

Let's call P_X the similarity matrix of the true data and Q_θ the similarity matrix of the embeddings. Since these are valid probability distributions, we can define the loss to be the distance between them. TSNE does this with the Kullback-Leibler divergence:

$$J_{\text{TSNE}} = \text{KL}(P_X || Q_\theta) := \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

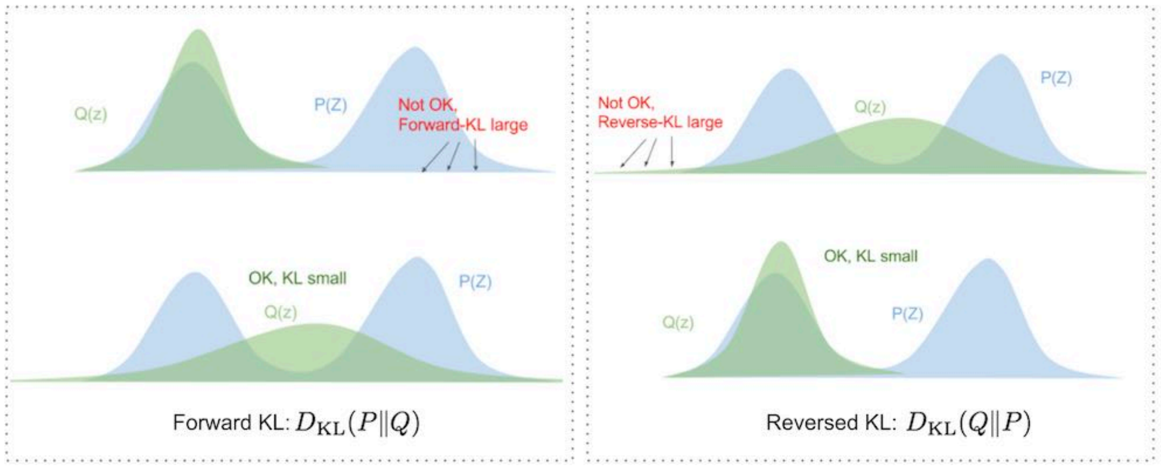


Image source: https://blog.evjang.com/2016/08/variational-bayes.html

TSNE's perplexity parameter

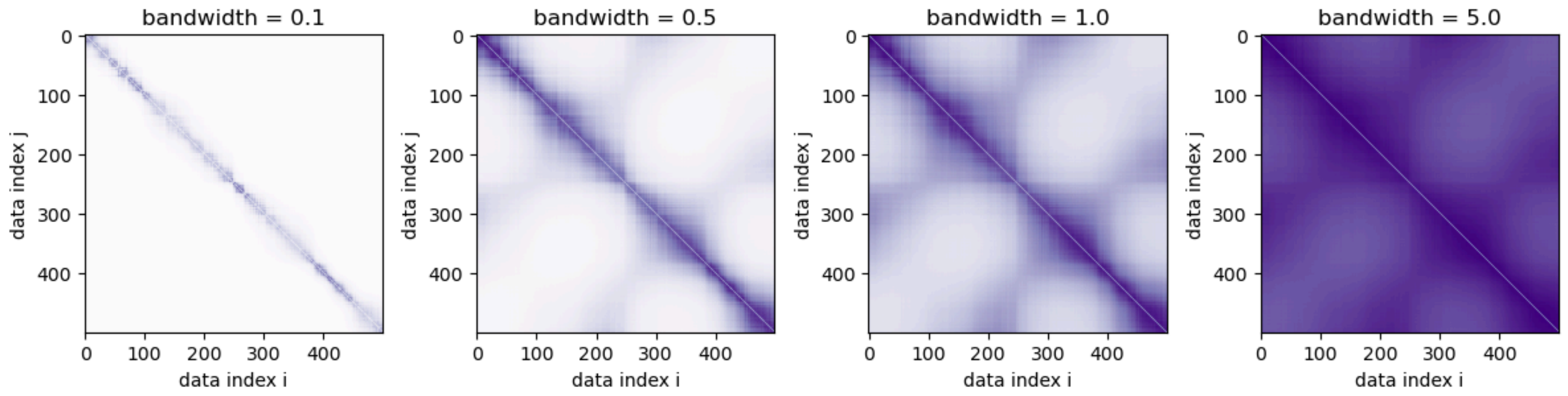
In creating $p_{j|i} \propto \exp\{-\|x^{(i)} - x^{(j)}\|^2/2\sigma_i^2\}$, there is the choice of what bandwidth σ_i to use.

TSNE advocates to adaptively set the bandwidth different for every i so every $p_{j|i}$ has the same **entropy** and **perplexity**.

The **entropy** of a categorical distribution is defined by $H(p) = -\sum_j p_j \log p_j$, and the **perplexity** is defined by $\mathrm{Perp}(p) = 2^{H(p)}$

We can think of the perplexity parameter as controlling how spread out the mass of the reference distribution is — how many neighbors are used. One way to see this is just to ablate across constant bandwidths. We'll ablate it more later.

```python
bandwidths = [0.1, 0.5, 1.0, 5.0]

fig, axs = plt.subplots(1, len(bandwidths), figsize=(12,4))
for ax, bandwidth in zip(axs, bandwidths):
    ax.set_xlabel('data index i'); ax.set_ylabel('data index j')
    p = get_tsne_target_probability(X, bandwidth=bandwidth)
    ax.imshow(p, cmap='Purples')
    ax.set_title(f'bandwidth = {bandwidth:.1f}')
fig.tight_layout()
```



Constructing Q_θ with a student’s t

TSNE uses a Gaussian to construct the reference similarity matrix P_X, but advocates for another choices for the embedded similarity matrix Q_θ.

They argue for a student's t distribution to help ignore dissimilar points.

Optimizing TSNE

The formulation for TSNE is thus

$$\arg\min_{\theta}\{J_{\text{TSNE}}(\theta) := \text{KL}(P_X, Q_\theta)\}$$

where P_X is the data similarity matrix and Q_θ is the similarity matrix induced by the embeddings.

Optimizing TSNE

Like in metric MDS, the solution is not known in closed form. It is often approximated with gradient-based optimization, which in the simplest form is:

1. Initialize the embeddings $\theta = \{z_i\}$.
2. Repeat until converged
    - $\theta \leftarrow \theta - \alpha \nabla_\theta J_{\text{TSNE}}(\theta)$

TSNE

- **Type**: Unsupervised learning (dimensionality reduction)
- **Model family**: Embeddings ($f_\theta(x^{(i)}) = z^{(i)}$)
- **Objective function**: KL between probabilistic similarities, $J_{\text{TSNE}} = \text{KL}(P_X || Q_\theta)$
- **Optimizer**: Gradient descent

Part 3: Visualizing high-dimensional data

We now apply PCA, MDS, and TSNE to some high-dimensional data.

Newsgroups

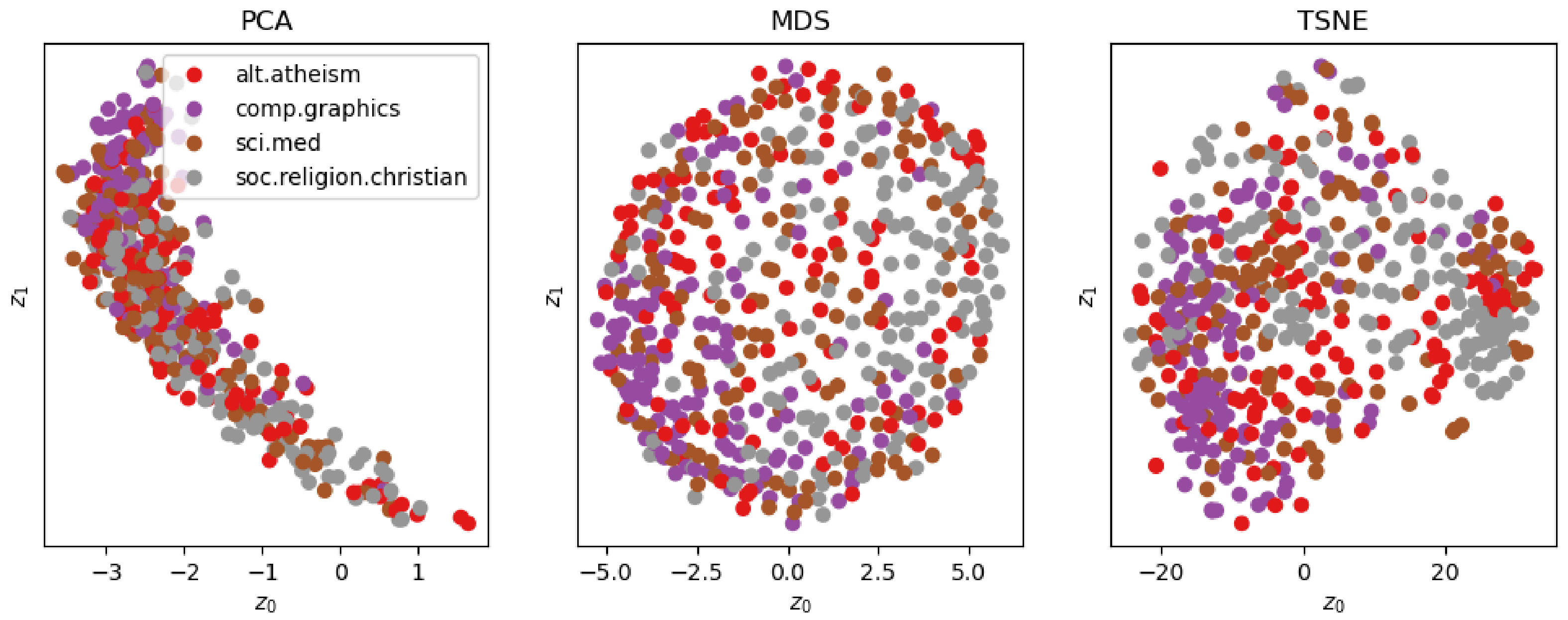
We'll start with the newsgroups data from the Naïve Bayes lecture.

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer

categories = ['alt.atheism', 'soc.religion.christian', 'comp.graphics', 'sci.med']
twenty_train = fetch_20newsgroups(subset='train', categories=categories, shuffle=True, random_state=42)
count_vect = CountVectorizer(binary=True, max_features=100)
X = count_vect.fit_transform(twenty_train.data).toarray().astype(np.float32)
y = twenty_train.target
np.random.seed(0)
idx = np.random.choice(X.shape[0], size=500, replace=False)
X, y = X[idx], y[idx]
print('a single example: ', X[0])
```

```
a single example:  [0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.
 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 0. 0.
 1. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.
 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1.
 1. 0. 0. 0.]
```

The visualizations for the newsgroup data is:

```
visualize_pca_mds_tsne(X, y, p=2, legend=twenty_train['target_names']);
```

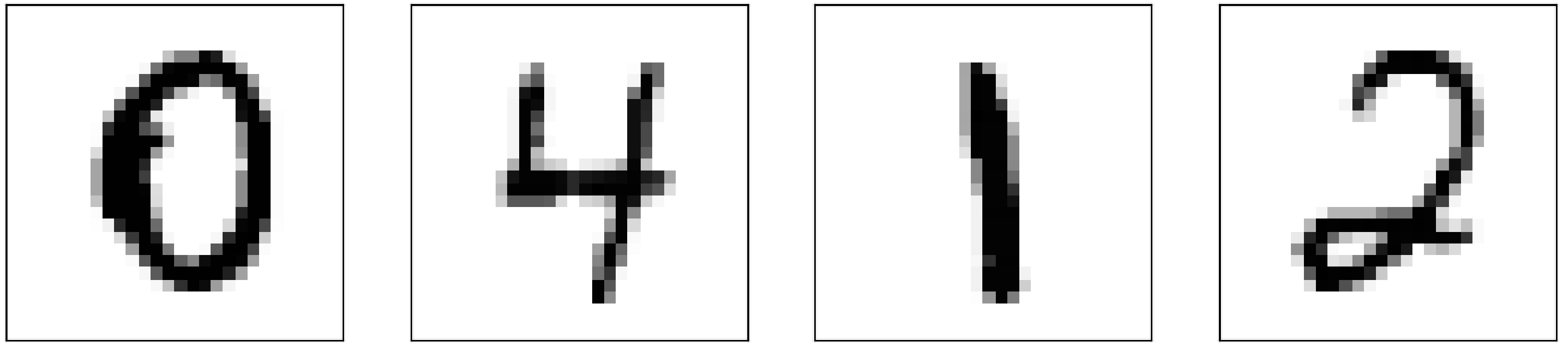


Visualizing image data: MNIST

This classic dataset consists of grayscale images representing handwritten digits

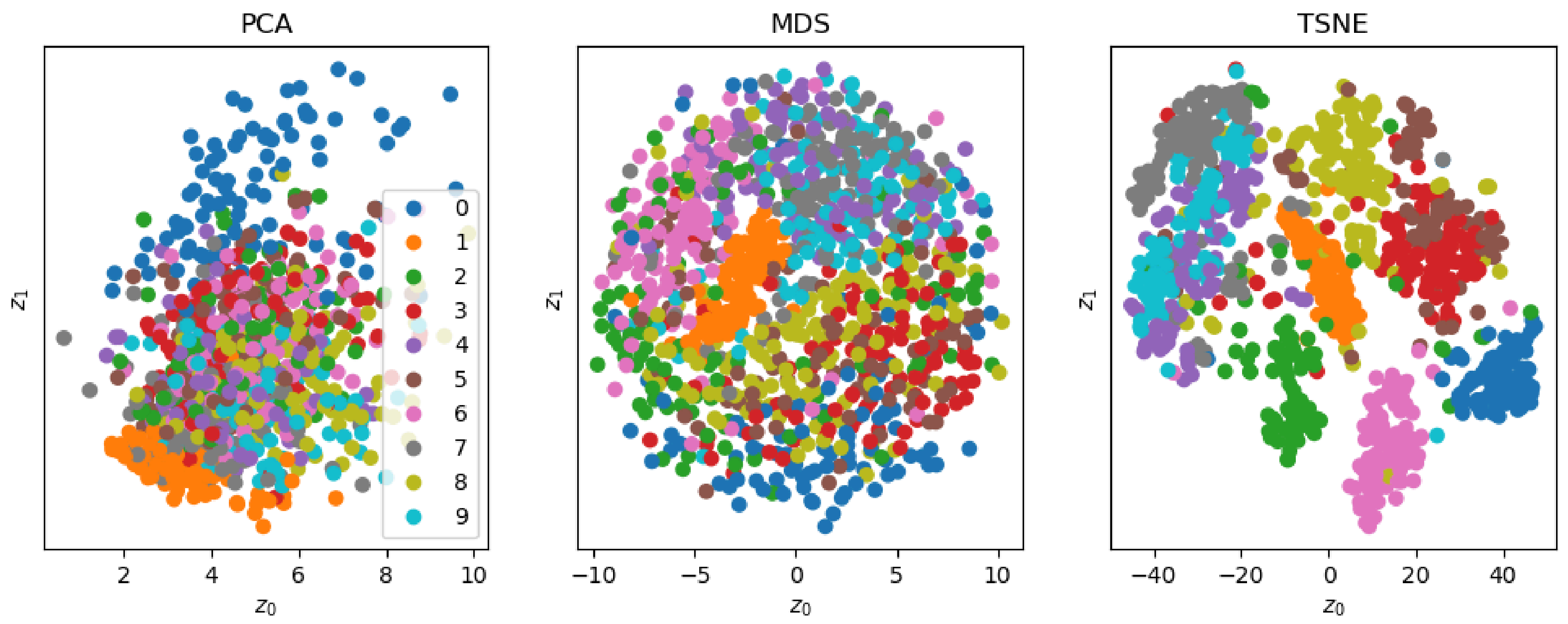
```python
from sklearn.datasets import fetch_openml
X, y = fetch_openml("mnist_784", version=1, return_X_y=True, as_frame=False, parser='auto')
np.random.seed(0)
idx = np.random.choice(X.shape[0], size=1000, replace=False)
X, y = X[idx], y[idx]
X = X.astype(np.float64) / 255.
y = y.astype(int)
print('image pixel values: ', X[0][120:131])
fig, axs = plt.subplots(ncols=4)
for i, ax in enumerate(axs):
    ax.imshow(X[i].reshape((28, 28)), cmap='Greys')
    ax.set_xticks([]); ax.set_yticks([])
```

```
image pixel values:  [0.   0.   0.   0.   0.   0.35 0.58 0.58 0.98 0.9  0.23]
```



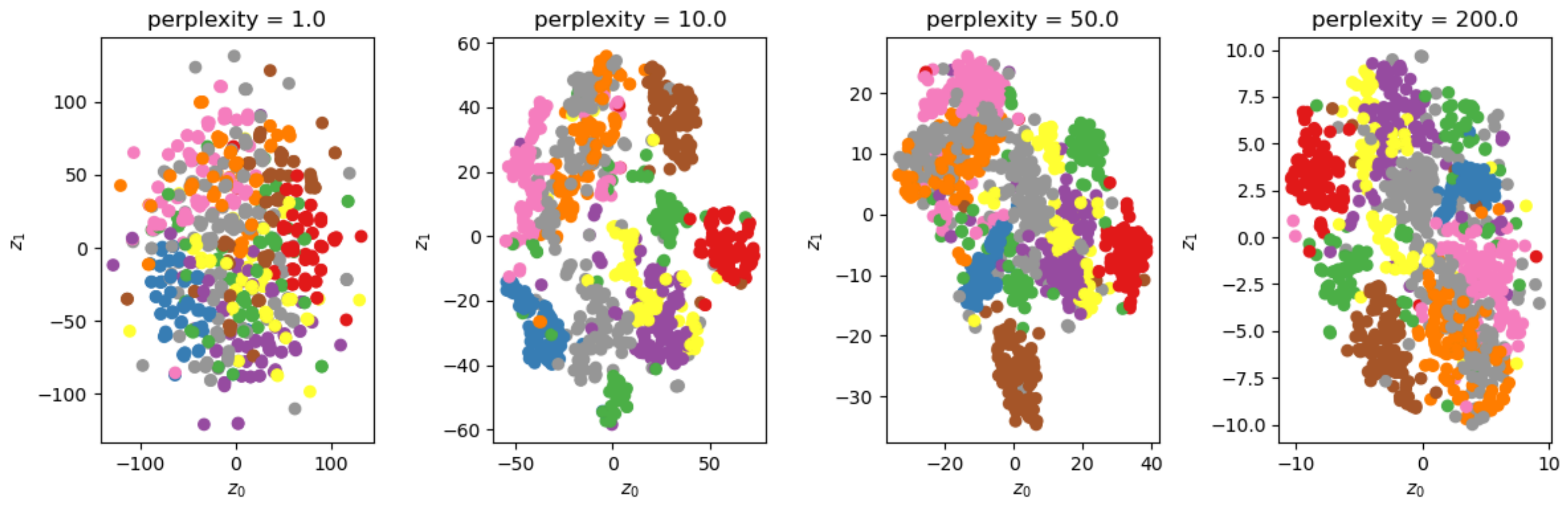
TSNE clearly visualizes the clusters in the MNIST data better than PCA and MDS. This was a key component of their original paper.

```
legend = [str(i) for i in range(10)]
visualize_pca_mds_tsne(X, y, p=2, extra_plot_kwargs={'cmap': 'tab10'}, legend=legend);
```



We can also use the MNIST data to ablate TSNE's perplexity parameter, controlling how many neighbors are taken into consideration:

```
perplexities = [1.0, 10., 50., 200.]

fig, axs = plt.subplots(1, len(bandwidths), figsize=(12,4))
for ax, perplexity in zip(axs, perplexities):
    ax.set_xlabel('$z_0$'); ax.set_ylabel('$z_1$')
    Z = TSNE(n_components=2, perplexity=perplexity).fit_transform(X)
    ax.scatter(Z[:,0], Z[:,1], c=y, cmap='Set1')
    ax.set_title(f'perplexity = {perplexity:.1f}')
fig.tight_layout()
```



Voting data

We'll next look at some congressional voting data from congress members in 1984.

```
house.votes.84: United States Congressional Voting Records 1984

This data set includes votes for each of the U.S. House of Representatives Congressmen on the 16 key
votes identified by the CQA. The CQA lists nine different types of votes: voted for, paired for, and
announced for (these three simplified to yea), voted against, paired against, and announced against
(these three simplified to nay), voted present, voted present to avoid conflict of interest, and did
not vote or otherwise make a position known (these three simplified to an unknown disposition).
```

Here's what the data looks like.

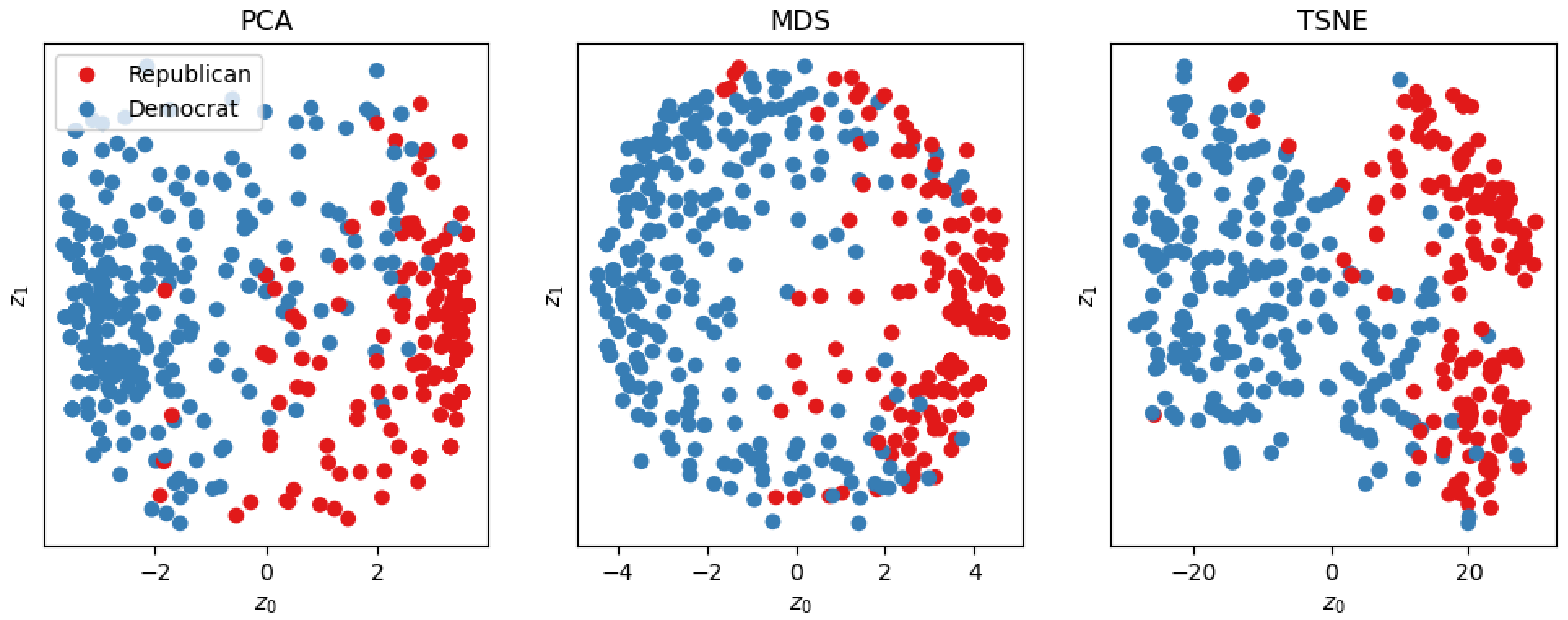
```
df, X, y = get_voting_data()
print(f'label={y[0]}, features={X[0]}')
df.head()
```

```
label=1, features=[-1  1 -1  1  1  1 -1 -1 -1  1  0  1  1  1 -1  1]
```

	class_name	handicapped_infants	water_project_cost_sharing	adoption_of_the_budget_resolution	physician_fee_freeze	el_salvador_aid
0	republican	n	y	n	y	y
1	republican	n	y	n	y	y
2	democrat	?	y	y	?	y
3	democrat	n	y	y	n	?
4	democrat	y	y	y	n	y

And embedding the voting records gives:

```
visualize_pca_mds_tsne(X, y, p=2, extra_plot_kwargs={'vmax': 8}, legend=['Republican', 'Democrat']);
```



Visualizing word embeddings

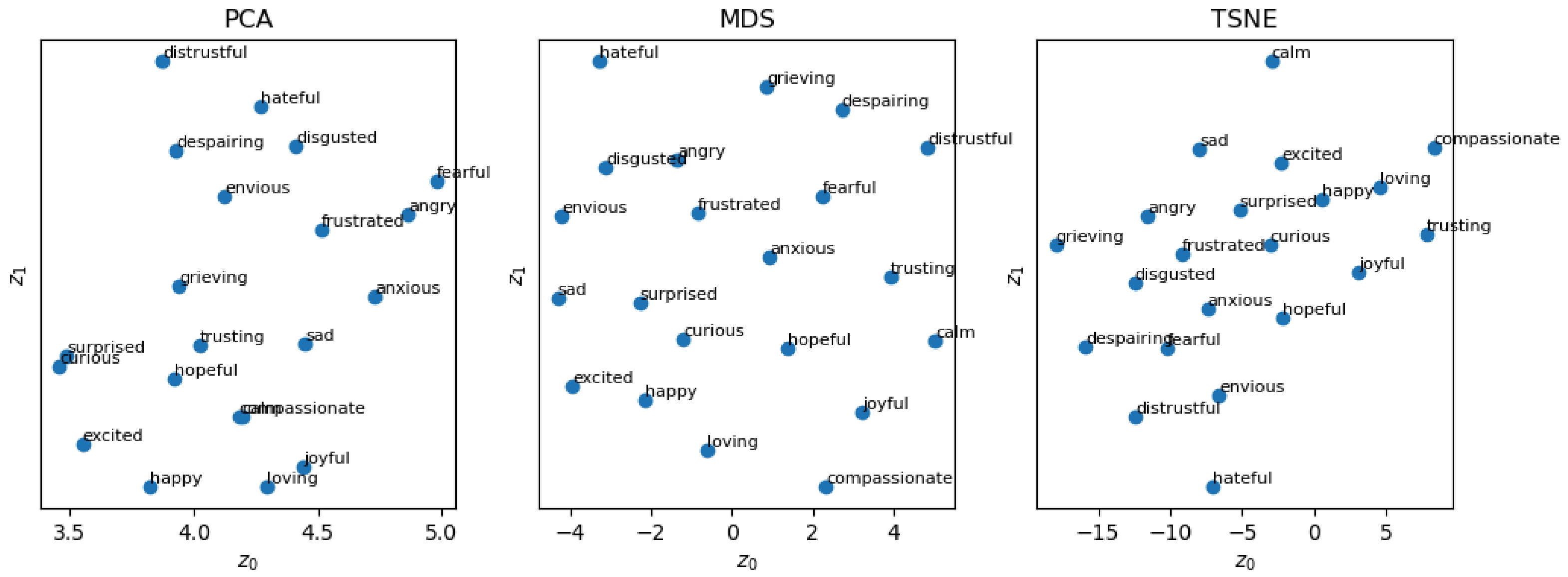
We'll last look at GloVe word embeddings (we could also use word2vec or fasttext embeddings). These are already semantically-rich vectors of words. Let's look at a few.

```
words = ["joyful", "happy", "sad", "angry", "fearful", "loving", "hateful", "surprised",
         "disgusted", "excited", "calm", "anxious", "frustrated", "hopeful",
         "despairing", "trusting", "distrustful", "envious", "compassionate", "grieving", "curious"]
embeddings = np.array([glove.emb(word) for word in words])
for i in range(5):
    print(f'word: {words[i]}\tembedding[:10]: {embeddings[i,:10]}')
```

```
word: joyful    embedding[:10]: [ 0.21  0.35 -0.36 -0.12  0.41  0.35  0.55  0.78  0.39  1.41]
word: happy     embedding[:10]: [ 0.04  0.41 -0.52 -0.07  0.09 -0.05  0.41 -0.43  0.19  2.39]
word: sad       embedding[:10]: [-0.01  0.28  0.07  0.14 -0.38 -0.18  0.29 -0.31  0.15  2.31]
word: angry     embedding[:10]: [-0.61 -0.12 -0.31 -0.19 -0.23  0.28  0.57  0.26  0.23  2.43]
word: fearful   embedding[:10]: [-0.06 -0.24  0.3   0.05 -0.08  0.12  0.19  0.86 -0.19  2.19]
```

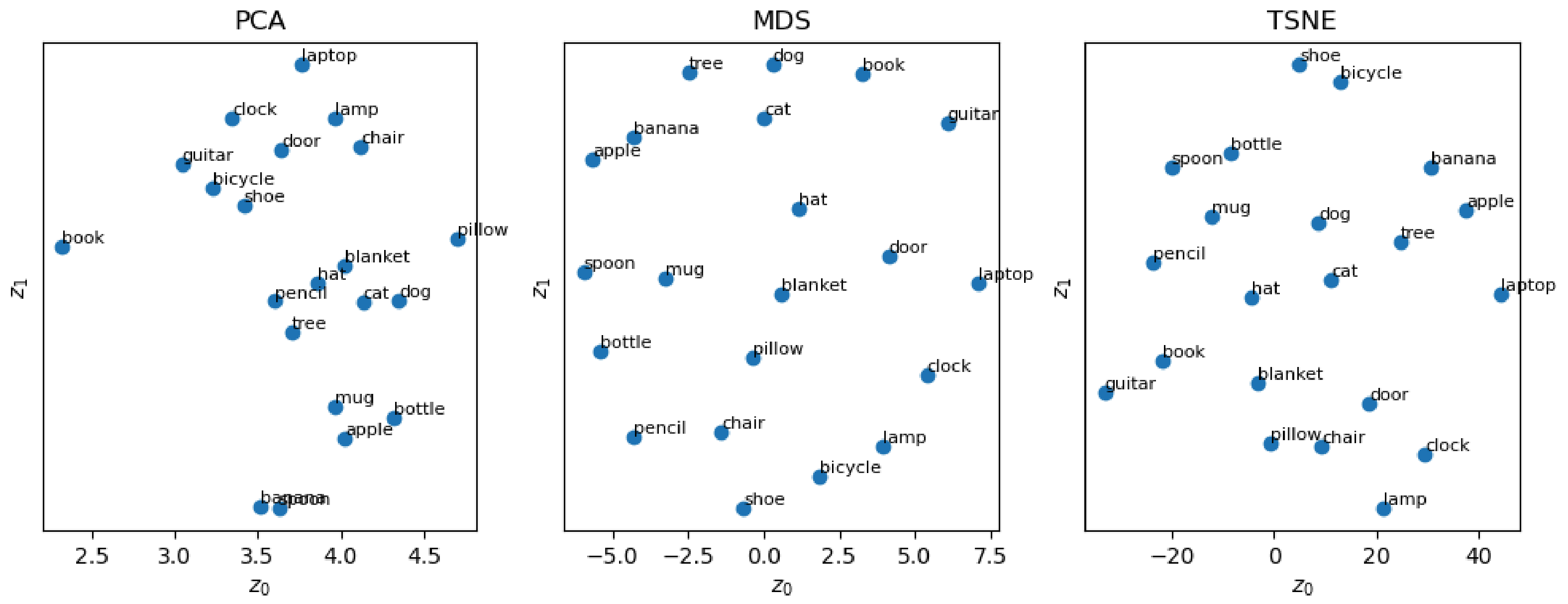
It's still hard to visualize how they relate to each other in this high-dimensional space, so let's embed them into 2 dimensions:

```
fig, axs = visualize_pca_mds_tsne(embeddings, p=2, tsne_perplexity=10, ax_cb=ax_cb)
```



Lastly let's look at another set of words:

```
words = ["apple", "cat", "dog", "banana", "chair", "pencil", "shoe", "lamp", "bicycle",
         "book", "pillow", "guitar", "mug", "tree", "clock",
         "blanket", "spoon", "laptop", "hat", "door", "bottle"]
embeddings = np.array([glove.emb(word) for word in words])
fig, axs = visualize_pca_mds_tsne(embeddings, p=2, tsne_perplexity=10, ax_cb=ax_cb)
```



Conclusions

The ill-defined nature of visualization leads to many options:

1. We covered PCA, MDS, TSNE. There's also LLE, UMAP, and others depending on the setting.
2. Can combine pieces to create new methods. For example the TSNE objective by parameterizing maps instead of embeddings.

Next class: Back to classification and introducing SVMs