Independent Component Analysis (ICA)

The following is a step-by-step guide that shows a basic Independent Component Analysis (ICA) workflow:

1. A synthetic blind source separation (BSS) demo (classic toy example)
2. Evaluation (matching recovered sources to true sources)
3. A simple real-data example using `sklearn`'s `load_digits` to visualise learned spatial components

0. Setup

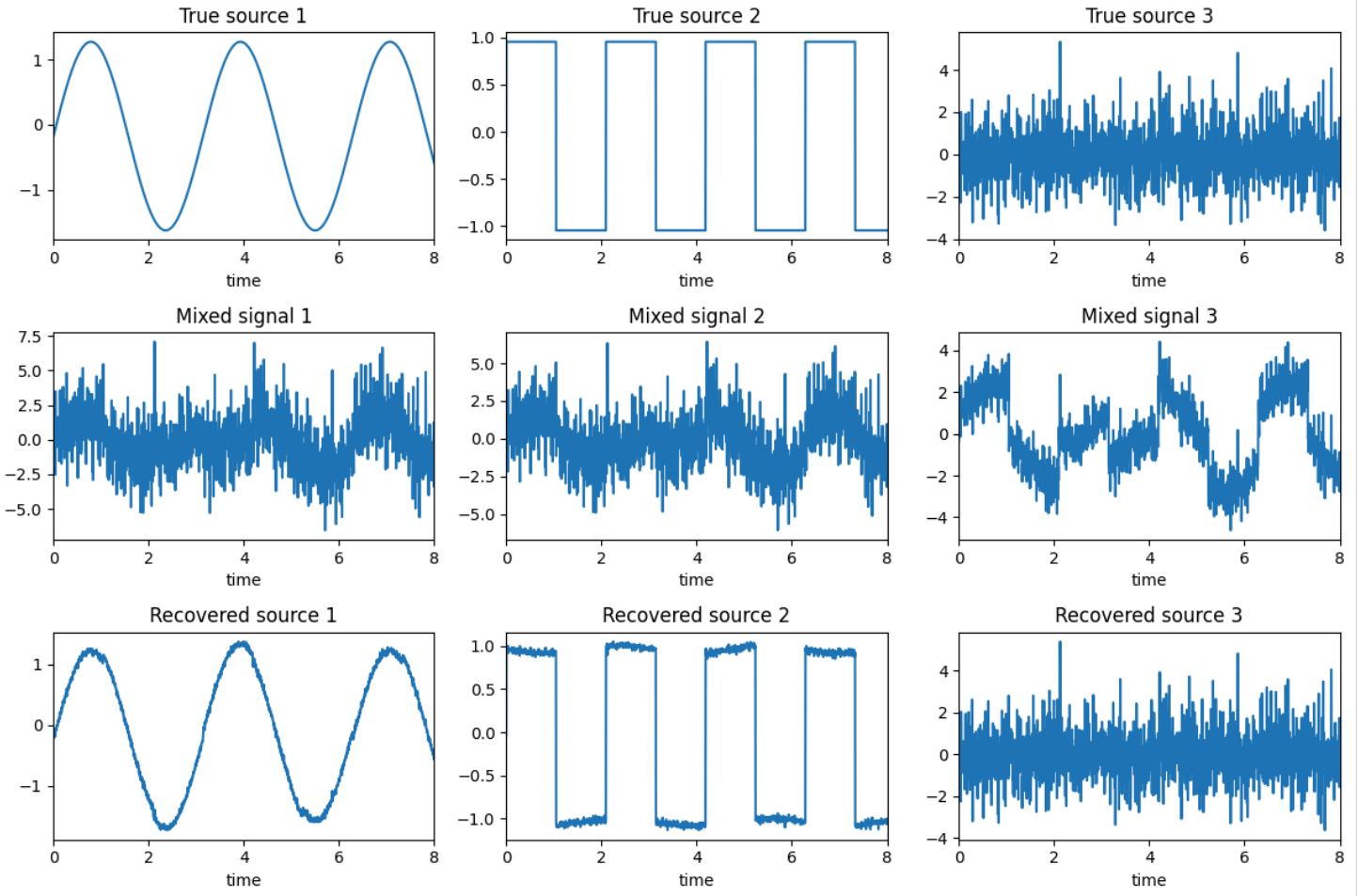
We begin by importing `numpy`, `matplotlib`, and the `FastICA` implementation from `scikit-learn`. `StandardScaler` is used for simple preprocessing. `PCA` is included for a quick comparison.

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.decomposition import FastICA, PCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_digits

# Plotting default in notebooks
%matplotlib inline
```

1. Synthetic ICA demo (classic BSS example)

We create three independent signals, mix them, run ICA, and plot original/mixed/recovered signals.

```
# For reproducible randomness
RANDOM_STATE = 42
rng = np.random.RandomState(RANDOM_STATE)

# Step 1: Creating synthetic independent sources
n_samples = 2000
time = np.linspace(0, 8, n_samples)

s1 = np.sin(2 * time)               # sinusoidal
s2 = np.sign(np.sin(3 * time))      # square wave
s3 = rng.laplace(size=n_samples)    # non-Gaussian noise

# Stacking column-wise and standardising
S = np.c_[s1, s2, s3]
S -= S.mean(axis=0)
S /= S.std(axis=0)

# Step 2: Mixing sources using a random mixing matrix
A = rng.uniform(low=0.5, high=1.5, size=(3,3))
X = S.dot(A.T)

# Step 3: Applying FastICA
ica = FastICA(
    n_components = 3,
    random_state = RANDOM_STATE,
    whiten = 'unit-variance',
    max_iter = 1000,
    tol = 1e-4
)
S_est = ica.fit_transform(X)
A_est = ica.mixing_

# Step 4: Plotting original, mixed, and recovered signals
fig, axes = plt.subplots(3, 3, figsize=(12,8))
axes = axes.flatten()

titles = [
    "True source 1", "True source 2", "True source 3",
    "Mixed signal 1", "Mixed signal 2", "Mixed signal 3",
    "Recovered source 1", "Recovered source 2", "Recovered source 3"
]
signals = [S[:,0], S[:,1], S[:,2],
           X[:,0], X[:,1], X[:,2],
           S_est[:,0], S_est[:,1], S_est[:,2]]

for ax, data, title in zip(axes, signals, titles):
    ax.plot(time, data)
    ax.set_title(title)
    ax.set_xlim(time[0], time[-1])
    ax.set_xlabel("time")
plt.tight_layout()
plt.show()
```



What this shows:

1. Top row: The true independent sources that we made.
2. Middle row: The observed mixtures that an ICA algorithm would realistically receive.
3. Bottom row: The components received by FastICA.

Note: ICA recovers sources up to permutation and sign/scale, so order and sign may differ in some cases.

2. Evaluating how well ICA recovered the true sources

As mentioned earlier, ICA recovery has two ambiguities: permutation and scaling (including sign). We will compute the absolute correlation matrix between true and estimated sources, and then match them (Hungarian or greedy fallback).

```
def match_components(S_true, S_est):
  """
  Return best matching of columns of S_est to S_true using absolute correlation.
  Uses SciPy's linear_sum_assignment if available, otherwise greedy matching.
  """
  n_true = S_true.shape[1]
  n_est = S_est.shape[1]
  assert n_true==n_est, "Number of true and estimated components must match for matching helper."

  # Correlation matrix between true components and estimated components
  corr = np.zeros((n_true, n_est))
  for i in range(n_true):
    for j in range(n_est):
      corr[i,j] = np.corrcoef(S_true[:,i], S_est[:,j])[0,1]
  abs_corr = np.abs(corr)

  # Trying Hungarian (optimal) assignment if available
  try:
    from scipy.optimize import linear_sum_assignment
    cost = -abs_corr
    row_ind, col_ind = linear_sum_assignment(cost)
  except Exception:
    # Greedy matching fallback
    col_ind = []
    used = set()
    for i in range(n_true):
      j = np.argmax(abs_corr[i,:])
      k = 0
      while j in used and k < n_true:
        abs_corr[i,j] = -1.0
        j = np.argmax(abs_corr[i,:])
        k += 1
```

```
        col_ind.append(j)
        used.add(j)
      col_ind = np.array(col_ind)

    # For reporting: matches and signed correlations
    matches = []
    for i, j in enumerate(col_ind):
      signed_corr = corr[i,j]
      matches.append((i, j, signed_corr))
    return matches, corr

matches, corr_matrix = match_components(S, S_est)

print("Matches (true_index, est_index, signed_correlation):")
for m in matches:
    print(m)

# Computing average absolute correlation of matched pairs
avg_abs_corr = np.mean([abs(m[2]) for m in matches])
print(f"\nAverage |correlation| of matched pairs: {avg_abs_corr:.4f}")

# Computing reconstruction error
X_rec = S_est.dot(A_est.T) + X.mean(axis=0)    # Reconstructing observed mixtures
recon_err = np.mean((X - X_rec) ** 2)
print(f"Mean squared reconstruction error (observations): {recon_err:.6f}")
```

```
Matches (true_index, est_index, signed_correlation):
(0, np.int64(0), np.float64(0.9980694101435091))
(1, np.int64(1), np.float64(0.9993773837866247))
(2, np.int64(2), np.float64(0.9995725069177209))

Average |correlation| of matched pairs: 0.9990
Mean squared reconstruction error (observations): 0.000000
```

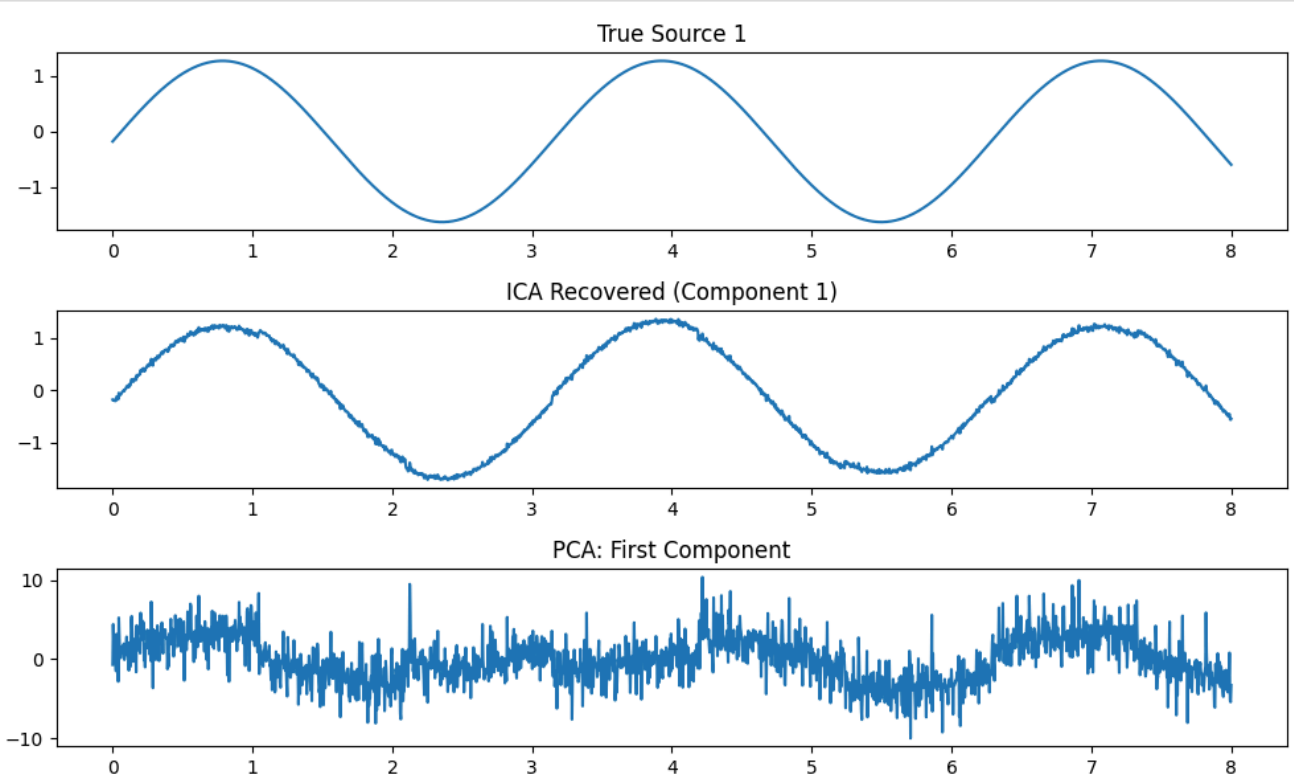
Explanation:

1. The `match_components` function computes pairwise correlations and finds a best assignment, so we can compare the correct true vs recovered signals.
2. The average absolute correlation gives an idea of how well the independent components were recovered.
3. We also reconstruct the mixed observations from estimated sources and compute MSE (small error indicates good fit).

3. Quick PCA comparison

It is often useful to compare ICA with Principal Component Analysis (PCA) on the same mixed data, since PCA components are orthogonal and aim to explain variance, not independence.

```
pca = PCA(n_components=3, random_state=RANDOM_STATE)
S_pca = pca.fit_transform(X)

fig, ax = plt.subplots(3, 1, figsize=(10, 6))
ax[0].plot(time, S[:,0]); ax[0].set(title="True Source 1")
ax[1].plot(time, S_est[:,0]); ax[1].set(title="ICA Recovered (Component 1)")
ax[2].plot(time, S_pca[:,0]); ax[2].set(title="PCA: First Component")
plt.tight_layout()
plt.show()
```



Note: PCA's first component maximises variance but will typically not separate independent non-Gaussian sources as ICA does.

4. Applying ICA to a real dataset (visualising learned components)

The `digits` dataset (comprising 8×8 images) is an easy real example: we treat each image as a vector and look for independent spatial basis components learned by ICA.

What to expect: Each independent component (IC) can look like a stroke, edge or simple stroke-combination. ICA often finds localised features (i.e., non-Gaussian, independent spatial patterns) in small image patches.

```
digits = load_digits()
X_digits = digits.data
n_components = 25       # number of independent components to extract

# Standardising features (centering + scaling)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_digits)

ica = FastICA(n_components=n_components, random_state=RANDOM_STATE, max_iter=1000)
S_digits = ica.fit_transform(X_scaled)
components = ica.components_

# Visualising components as 8x8 images
fig, axes = plt.subplots(5, 5, figsize=(8, 8))
for i, ax in enumerate(axes.flat):
    if i < n_components:
        comp_img = components[i].reshape(8, 8)
        ax.imshow(comp_img)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_title(f"IC {i+1}")
plt.tight_layout()
plt.show()
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/decomposition/_fastica.py:127: ConvergenceWarning: FastICA did not converge. Consider increasir
  warnings.warn(
```