L4 – Unsupervised Learning: Preprocessing and Transformation

- In unsupervised learning, the learning algorithm is just shown the input data and asked to extract knowledge
- **Type I:** transformations of the dataset
  - Create a new representation of the data which might be easier for humans or other machine learning algorithms to understand
  - E.g., converting a high-dimensional representation of the data into a new way to represent this data that summarizes the essential characteristics with fewer features.
- **Type II:** clustering
  - Partition data into distinct groups of similar items
  - e.g., divide all faces into groups of faces that look similar

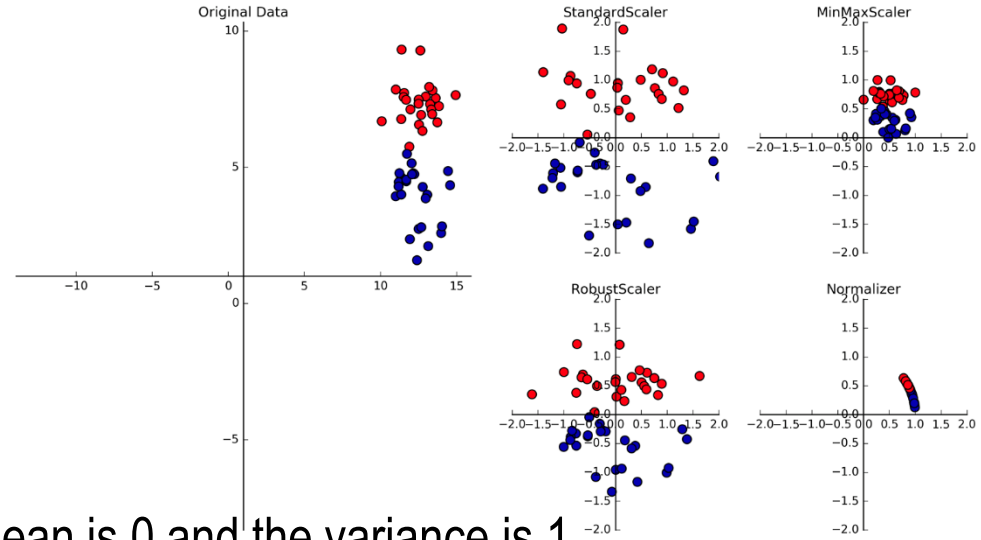
Challenges in Unsupervised Learning

- A major challenge: evaluating whether the algorithm learned something useful
  - Unsupervised algorithms are used often in an exploratory setting when a data scientist wants to understand the data better
  - Another common application for unsupervised algorithms is as a preprocessing step for supervised algorithms
    - To improve the accuracy of supervised algorithms
    - Can lead to reduced memory and time consumption
- We start from discussing some simple preprocessing methods that often come in handy

Preprocessing and Scaling

- Neural networks and SVMs are very sensitive to the scaling of the data

mglearn.plots.plot_scaling()



- Different types of scaling
  - MinMaxScaler

    Shift data into [0,1] for all features
  - StandardScaler

    Ensure that for each feature the mean is 0 and the variance is 1
  - RobustScaler

    Using the median & quartiles rather than mean & variance – can ignore data points are very different from the rest (i.e., outliers)
  - Normalizer: make the feature vector has a Euclidean length of 1

- Use MinMaxScaler for preprocessing data for kernel SVM
  - Step 1) Constructing the scaler
  - Step 2) Fitting the scaler
  - Step 3) Transform the dataset

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=1)
print(X_train.shape)                    print(X_test.shape)
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler() scaler.fit(X_train)

# transform data
X_train_scaled = scaler.transform(X_train)
# print dataset properties before and after scaling
print("transformed shape: {}".format(X_train_scaled.shape))
print("per-feature minimum before scaling:\n {}".format(X_train.min(axis=0)))
print("per-feature maximum before scaling:\n {}".format(X_train.max(axis=0)))
print("per-feature minimum after scaling:\n {}".format(X_train_scaled.min(axis=0)))
print("per-feature maximum after scaling:\n {}".format(X_train_scaled.max(axis=0)))
```

- When applying the same transform to the test dataset
  - The method always subtracts the training set minimum and divides by the training set range, which might be different from the minimum and range for the test set
  - Consequence: the minimum and the maximum are not 0 and 1

```
# transform test data
X_test_scaled = scaler.transform(X_test)
# print test data properties after scaling
print("per-feature minimum after scaling:\n{}".format(X_test_scaled.min(axis=0)))
print("per-feature maximum after scaling:\n{}".format(X_test_scaled.max(axis=0)))
```

  - It is important to apply exactly the same transformation to the training set and the test set for the supervised model to work on the test set
  - What if the scaling is given in an incorrect way? See the example below

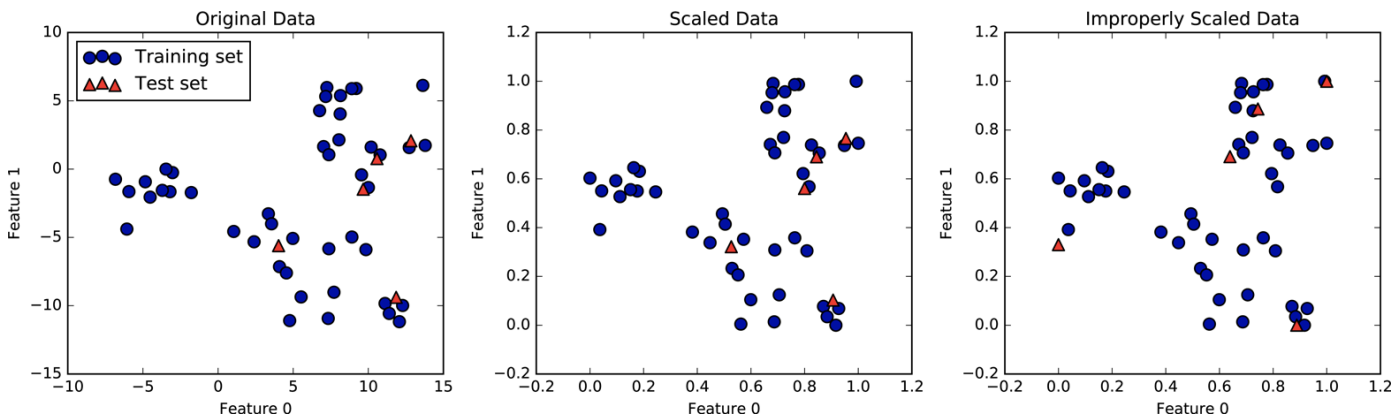
```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

# make synthetic data
X, _ = make_blobs(n_samples=50, centers=5, random_state=4, cluster_std=2)
# split it into training and test sets
X_train, X_test = train_test_split(X, random_state=5, test_size=.1)
# plot the training and test sets
fig, axes = plt.subplots(1, 3, figsize=(13, 4))
axes[0].scatter(X_train[:, 0], X_train[:, 1], c=mglearn.cm2(0), label="Training set", s=60)
axes[0].scatter(X_test[:, 0], X_test[:, 1], marker='^', c=mglearn.cm2(1), label="Test set", s=60)
axes[0].legend(loc='upper left')          axes[0].set_title("Original Data")

# scale the data using MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)          X_test_scaled = scaler.transform(X_test)
# visualize the properly scaled data
axes[1].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1], c=mglearn.cm2(0), label="Training set", s=60)
axes[1].scatter(X_test_scaled[:, 0], X_test_scaled[:, 1], marker='^', c=mglearn.cm2(1), label="Test set", s=60)
axes[1].set_title("Scaled Data")
```

```
# rescale the test set separately, so test set min is 0 and test set max is 1
# DO NOT DO THIS! For illustration purposes only.
test_scaler = MinMaxScaler()
test_scaler.fit(X_test)
X_test_scaled_badly = test_scaler.transform(X_test)

# visualize wrongly scaled data
axes[2].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
c=mglearn.cm2(0), label="training set", s=60)
axes[2].scatter(X_test_scaled_badly[:, 0], X_test_scaled_badly[:, 1], marker='^', c=mglearn.cm2(1),
label="test set", s=60)
axes[2].set_title("Improperly Scaled Data")
for ax in axes:
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
fig.tight_layout()
```



Shortcut and Efficient Alternatives

- Often, you want to fit a model on some dataset, and then transform it
  - There is an alternative as fit_transform, which is more efficient in some models (although may not be the case for all models)

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# calling fit and transform in sequence (using method chaining)
X_scaled = scaler.fit(X_train).transform(X_train)
# same result, but more efficient computation
X_scaled_d = scaler.fit_transform(X_train)
```

  - After this, it is time to study the effectiveness of preprocessing on supervised learning

- See the effect of using the MinMaxScaler on learning SVC

```
from sklearn.svm import SVC
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)
svm = SVC(C=100)
svm.fit(X_train, y_train)
print("Test set accuracy: {:.2f}".format(svm.score(X_test, y_test)))
```

  - After fitting on the original data, see the result on scaled dataset

```
# preprocessing using 0-1 scaling
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# learning an SVM on the scaled training data
svm.fit(X_train_scaled, y_train)
# scoring on the scaled test set
print("Scaled test set accuracy: {:.2f}".format(svm.score(X_test_scaled, y_test)))
```

  - As we can see, the effect of scaling the data is quite significant
  - Try different other preprocessing method (e.g., RobustScaler)

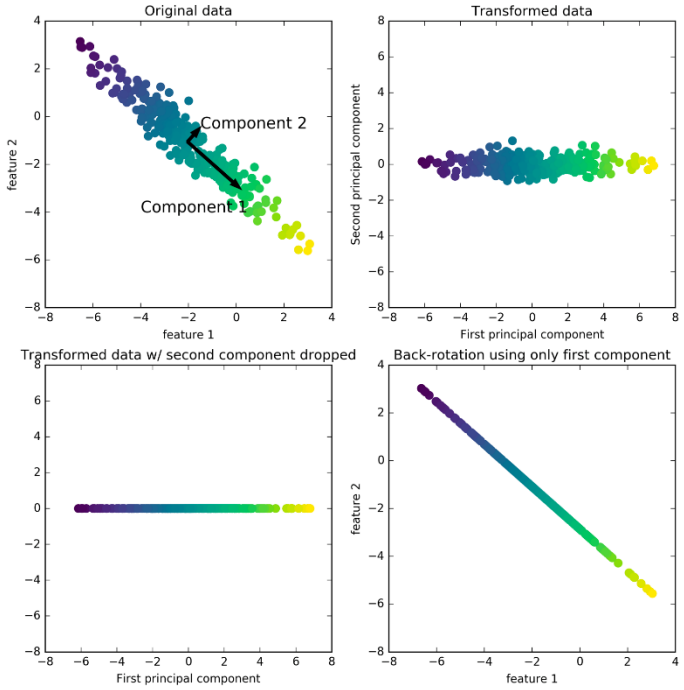
Dimensionality Reduction, Feature Extraction, and Manifold Learning

- Motivations for unsupervised learning in the mode of transformation
  - Visualization
  - Compressing the data
  - Finding a representation that is more informative for further processing
- Algorithms to be learned here
  - Principal Component Analysis (PCA)
  - Non-Negative Matrix Factorization (NMF)
  - Manifold Learning with t-SNE

Principal Component Analysis (PCA)

- A method that rotates the dataset in a way such that
  - The rotated features are statistically uncorrelated
  - Followed by selecting only a subset of the new features (according to how important they are for explaining the data)

mglearn.plots.plot_pca_illustration()



- Principal components
  - Main direction of variance
  - Usually sorted by the importance
  - Head or tail of an arrow

  is less important

- Applying PCA to the cancer dataset for visualization
  - For a high-dimensional dataset, per-class feature histogram is often used for visualization

```
fig, axes = plt.subplots(15, 2, figsize=(10, 20))
malignant = cancer.data[cancer.target == 0]
benign = cancer.data[cancer.target == 1]
ax = axes.ravel()
for i in range(30):
        _, bins = np.histogram(cancer.data[:, i], bins=50)
        ax[i].hist(malignant[:, i], bins=bins, color=mglearn.cm3(0), alpha=.5)
        ax[i].hist(benign[:, i], bins=bins, color=mglearn.cm3(2), alpha=.5)
        ax[i].set_title(cancer.feature_names[i])
        ax[i].set_yticks(())
ax[0].set_xlabel("Feature magnitude")
ax[0].set_ylabel("Frequency")
ax[0].legend(["malignant", "benign"], loc="best")
fig.tight_layout()
```

  - Which does not show anything about the interactions between variables and how these relate to the classes

- Before applying PCA, need to scaling dataset

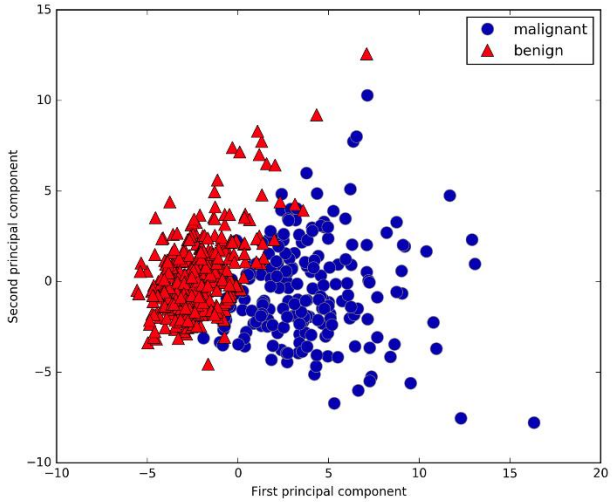
```
cancer = load_breast_cancer()                    # from sklearn.datasets import load_breast_cancer
scaler = StandardScaler()
scaler.fit(cancer.data)
X_scaled = scaler.transform(cancer.data)
```

- Need to specify how many components we want to keep

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)                        # keep the first two principal components of the data
pca.fit(X_scaled)                                # fit PCA model to breast cancer data
X_pca = pca.transform(X_scaled)                  # transform data onto the first two principal components
print("Original shape: {}".format(str(X_scaled.shape)))
print("Reduced shape: {}".format(str(X_pca.shape)))
# plot first vs. second principal component, colored by class
plt.figure(figsize=(8, 8))
mglearn.discrete_scatter(X_pca[:, 0], X_pca[:, 1], cancer.target)
plt.legend(cancer.target_names, loc="best")
plt.gca().set_aspect("equal")
plt.xlabel("First principal component”)          plt.ylabel("Second principal component")
```

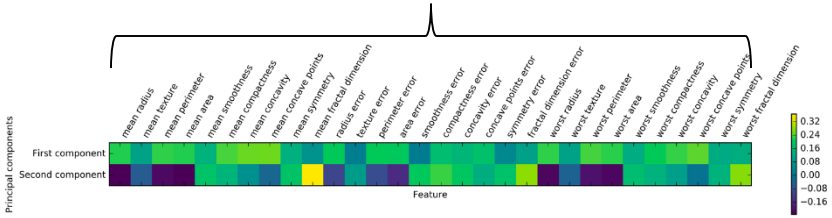
- As an unsupervised method, it simply looks at the correlations

- Visualization in 2D is very helpful
  - Two classes separate quite well
  - Even a linear classifier can distinguish
- Downside of PCA
  - The meaning of axes is hard to interpret
  - PCs are the linear combination of the original features
  - Each row in components_ corresponds to one PC (and sorted by their importance)




```
print("PCA component shape: {}".format(pca.components_.shape))
print("PCA components:\n{}".format(pca.components_))
```

All positive in PC1 means that there is a general correlation between all features.




```
plt.matshow(pca.components_, cmap='viridis')
plt.yticks([0, 1], ["First component", "Second component"])
plt.colorbar()
plt.xticks(range(len(cancer.feature_names)), cancer.feature_names, rotation=60, ha='left')
plt.xlabel("Feature")
plt.ylabel("Principal components")
```

Eigenfaces for Feature Extraction (PCA)

- Another application of PCA is feature extraction
  - Idea behind: finding a representation of your data that is better suited to analysis than the raw representation
  - Example: feature extraction on face images

```
from sklearn.datasets import fetch_lfw_people
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
image_shape = people.images[0].shape
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
subplot_kw={'xticks': (), 'yticks': ()})
for target, image, ax in zip(people.target, people.images, axes.ravel()):
        ax.imshow(image)
        ax.set_title(people.target_names[target])
print("people.images.shape: {}".format(people.images.shape))
print("Number of classes: {}".format(len(people.target_names)))
```

- Study the samples in the dataset of face images

```
counts = np.bincount(people.target)   # count how often each target appears
# print counts next to target names
for i, (count, name) in enumerate(zip(counts, people.target_names)):
        print("{0:25} {1:3}".format(name, count), end='     ')
        if (i + 1) % 3 == 0:
                print()
```

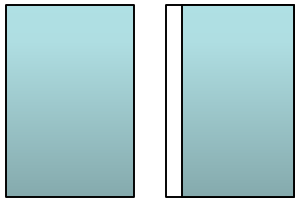
  - A bit skewed as containing a lot of images of Bush and Powell
  - To make the data less skewed, we will only take up to 50 images of each person (otherwise, the feature extraction would be overwhelmed by the likelihood of Bush)

```
mask = np.zeros(people.target.shape, dtype=np.bool)
for target in np.unique(people.target):
        mask[np.where(people.target == target)[0][:50]] = 1
X_people = people.data[mask]
y_people = people.target[mask]
# scale the grayscale values to be between 0 and 1
# instead of 0 and 255 for better numeric stability
X_people = X_people / 255
```

- A common task: face recognition
  - One way: to build a classifier for each person
    - Problem - too many classifiers and too few images for each classifier
  - A solution: to use a one-nearest-neighbor classifier in pixel space

```
from sklearn.neighbors import KNeighborsClassifier
# split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_people, y_people, stratify=y_people, random_state=0)
# build a KNeighborsClassifier using one neighbor
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
print("Test set score of 1-nn: {:.2f}".format(knn.score(X_test, y_test)))
```

- The accuracy of random draw: 1/62 = 1.6%
  - kNN is only slightly better than random draw
  - Reasons:
    - Computing distances in the pixel space is very bad choice
    - Shifting one pixel will make two images have a dramatic distance but they are actually similar to each other



- Principal Component Analysis (PCA) with whitening option
  - The same as using StandardScaler after the transformation

```
mglearn.plots.plot_pca_whitening()
```

  - Fit the PCA object to training data and extract the first 100 PCs

```
pca = PCA(n_components=100, whiten=True, random_state=0).fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
print("X_train_pca.shape: {}".format(X_train_pca.shape))
```

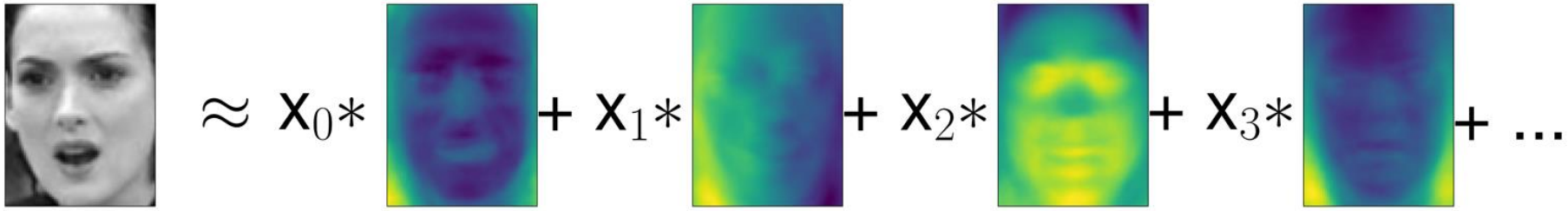
  - Using kNN classifier again

```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train_pca, y_train)
print("Test set score of 1-nn: {:.2f}".format(knn.score(X_test_pca, y_test)))
```

  - For image data, we can also visualize the PCs that are found

```
print("pca.components_.shape: {}".format(pca.components_.shape))
fix, axes = plt.subplots(3, 5, figsize=(15, 12), subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(pca.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape), cmap='viridis')
    ax.set_title("{}. component".format((i + 1)))
```

- Schematic view of PCA as decomposing an image into a weighted sum of components



  - x_0, x_1, and so on are the coefficients of PCs
  - They are the representation of the image in the rotated space
  - A few are used, a compressed image (with coarser features) is obtained

```
mglearn.plots.plot_pca_faces(X_train, X_test, image_shape)
```

  - From the scatter plot of the first two PCs, not too much info.

```
mglearn.discrete_scatter(X_train_pca[:, 0], X_train_pca[:, 1], y_train)
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

  - Conclusion: PCA only captures very rough characteristics

Non-Negative Matrix Factorization (NMF)

- Similar to PCA but different unsupervised learning
  - Both approximate each data as a weighted sum of components
  - PCA: want components to be orthogonal
    - To catch as much variance of the data as possible
  - NMF: want components and coefficients to be non-negative
    - To lead to more interpretable components than PCA as negative components and coefficients can lead to hard-to-interpret cancellation effects
- In contrast to PCA, we need to ensure that our data is positive for NMF to be able to operate on the data

mglearn.plots.plot_nmf_illustration()

- All components in NMF play at an equal importance

- Applying NMF to face images
  - NMF uses a random initialization

```
mglearn.plots.plot_nmf_faces(X_train, X_test, image_shape)
```

  - Quality of the back-transformed data is slightly worse than PCA
  - But let's look at the components

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=10, random_state=0)
nmf.fit(X_train)
X_train_nmf = nmf.transform(X_train)
X_test_nmf = nmf.transform(X_test)
fix, axes = plt.subplots(2, 5, figsize=(15, 12), subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(nmf.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape))
    ax.set_title("{}. component".format(i))
```

  - It is interesting to see some component (e.g., 1 & 7) with faces looking at left / right
  - Let's have a look at the faces have large coefficients for these

```
compn = 1
# sort by 1st component, plot first 10 images
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8), subplot_kw={'xticks': (), 'yticks': ()})
fig.suptitle("Large component 1")
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
        ax.imshow(X_train[ind].reshape(image_shape))

compn = 7
# sort by 7th component, plot first 10 images
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig.suptitle("Large component 7")
fig, axes = plt.subplots(2, 5, figsize=(15, 8), subplot_kw={'xticks': (), 'yticks': ()})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
        ax.imshow(X_train[ind].reshape(image_shape))
```

- Non-negative coefficients are important for applications
  - Such as Audio track of multiple people speaking
  - Or music with many instruments

- Extracting patterns by NMF works best for data with additive structure, including audio, gene expression & text
  - Let's say that we are interested in signal that is a combination of three different sources

```
S = mglearn.datasets.make_signals()
plt.figure(figsize=(6, 1))
plt.plot(S, '-')
plt.xlabel("Time")          plt.ylabel("Signal")
```

  - Unfortunately, we cannot observe the original signal but only an additive mixture of all three of them

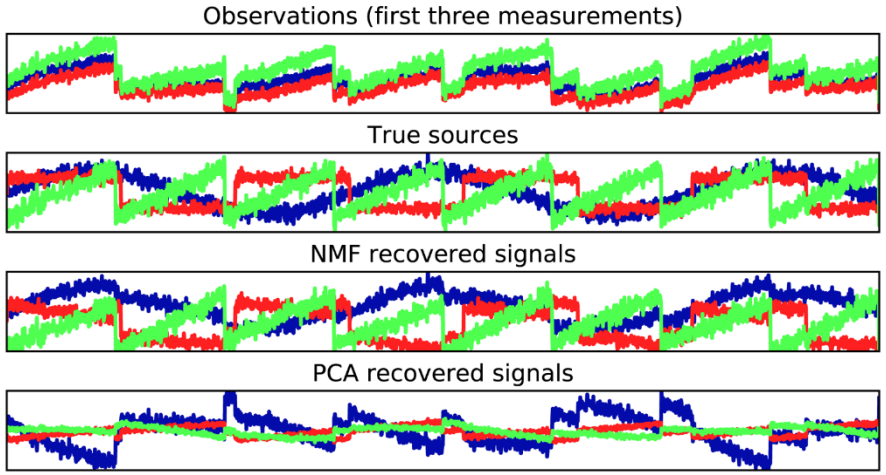
```
# mix data into a 100-dimensional state
A = np.random.RandomState(0).uniform(size=(100, 3))
X = np.dot(S, A.T)
print("Shape of measurements: {}".format(X.shape))
```

  - We can use NMF to recover the three signals

```
nmf = NMF(n_components=3, random_state=42)
S_ = nmf.fit_transform(X)
print("Recovered signal shape: {}".format(S_.shape))
```

- For comparison, we also apply PCA and make a comparison

```
pca = PCA(n_components=3)
H = pca.fit_transform(X)
models = [X, S, S_, H]
names = ['Observations (first three measurements)', 'True sources',
         'NMF recovered signals',
         'PCA recovered signals']
fig, axes = plt.subplots(4, figsize=(8, 4),
         gridspec_kw={'hspace': .5},
         subplot_kw={'xticks': (), 'yticks': ()})
for model, name, ax in zip(models, names, axes):
    ax.set_title(name)
    ax.plot(model[:, :3], '-')
```



- There are many other algorithms can be used decompose each data point into a weighted sum as PCA and NMF do.
  - Independent component analysis (ICA)
  - Factor analysis (FA)
  - Sparse coding (dictionary learning)

Manifold Learning with t-SNE

- The nature of method such as PCA limits its usefulness with the scatter plot
  - Can be resolved by manifold learning algorithms (e.g., t-SNE)
  - Can only be applied to training set (rather than test set later)
  - Mainly used for visualization; Never for supervised learning later
- Idea behind t-SNE:
  - Find a two-dimensional representation of the data that preserves the distance between points as best as possible
  - Start with a random two-dimensional rep. for each data point
  - Then try to make points that are close in the original feature space closer, and points that are far apart farther apart

- We apply the t-SNE on dataset of handwritten
  - Each data point is an 8x8 gray-scale image

```
from sklearn.datasets import load_digits
digits = load_digits()                    #print(digits.images.shape)
fig, axes = plt.subplots(2, 5, figsize=(10, 5), subplot_kw={'xticks':(), 'yticks': ()})
for ax, img in zip(axes.ravel(), digits.images):
        ax.imshow(img)
```

  - Let's first use PCA to visualize the data reduced to 2D space

```
pca = PCA(n_components=2)          # build a PCA model
pca.fit(digits.data)               # transform the digits data onto the first two principal components
digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525", "#A83683", "#4E655E", "#853541",
      "#3A3120", "#535D8E"]
plt.figure(figsize=(10, 10))
plt.xlim(digits_pca[:, 0].min(), digits_pca[:, 0].max())       plt.ylim(digits_pca[:, 1].min(), digits_pca[:, 1].max())
for i in range(len(digits.data)):          # actually plot the digits as text instead of using scatter
    plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.target[i]),
      color = colors[digits.target[i]], fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("First principal component")        plt.ylabel("Second principal component")
```

- Let's apply t-SNE to the same data
  - As t-SNE does not support transforming new data, the TSNE class has no transform method
  - Instead, we call the fit_transform method

```
from sklearn.manifold import TSNE
tsne = TSNE(random_state=42)
# use fit_transform instead of fit, as TSNE has no transform method
digits_tsne = tsne.fit_transform(digits.data)

plt.figure(figsize=(10, 10))
plt.xlim(digits_tsne[:, 0].min(), digits_tsne[:, 0].max() + 1)
plt.ylim(digits_tsne[:, 1].min(), digits_tsne[:, 1].max() + 1)
for i in range(len(digits.data)):
    # actually plot the digits as text instead of using scatter
    plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.target[i]),
         color = colors[digits.target[i]], fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("t-SNE feature 0")
plt.xlabel("t-SNE feature 1")
```

- The result of t-SNE is quite remarkable
  - All the classes are quite clearly separated
  - Keep in mind that this method has no knowledge of the class labels: completely unsupervised
- t-SNE tries to preserve the information indicating which points are neighbors to each other