Machine Learning Workshop

Marwadi University, Rajkot, Dec 2019

Faculty: Santosh Chapaneri

Dimensionality Reduction: t-SNE (t-Distributed Stochastic Neighbor Embedding)

t-SNE is a tool for data visualization. It reduces the dimensionality of data to 2 or 3 dimensions so that it can be plotted easily. Local similarities are preserved by this embedding.

t-SNE converts distances between data in the original space to probabilities. First, we compute conditional probabilites

$$p_{j|i} = \frac{\exp\left(-d(\boldsymbol{x}_i, \boldsymbol{x}_j)/(2\sigma_i^2)\right)}{\sum_{i \neq k} \exp\left(-d(\boldsymbol{x}_i, \boldsymbol{x}_k)/(2\sigma_i^2)\right)}, \quad p_{i|i} = 0,$$

which will be used to generate joint probabilities

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}.$$

The σ_i will be determined automatically. This procedure can be influenced by setting the `perplexity` of the algorithm.

A heavy-tailed distribution will be used to measure the similarities in the embedded space

$$q_{ij} = \frac{(1 + ||\boldsymbol{y}_i - \boldsymbol{y}_j)||^2)^{-1}}{\sum_{k \neq l} (1 + ||\boldsymbol{y}_k - \boldsymbol{y}_l)||^2)^{-1}},$$

and then we minimize the Kullback-Leibler divergence

$$KL(P|Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

between both distributions with gradient descent (and some tricks). Note that the cost function is not convex and multiple runs might yield different results.

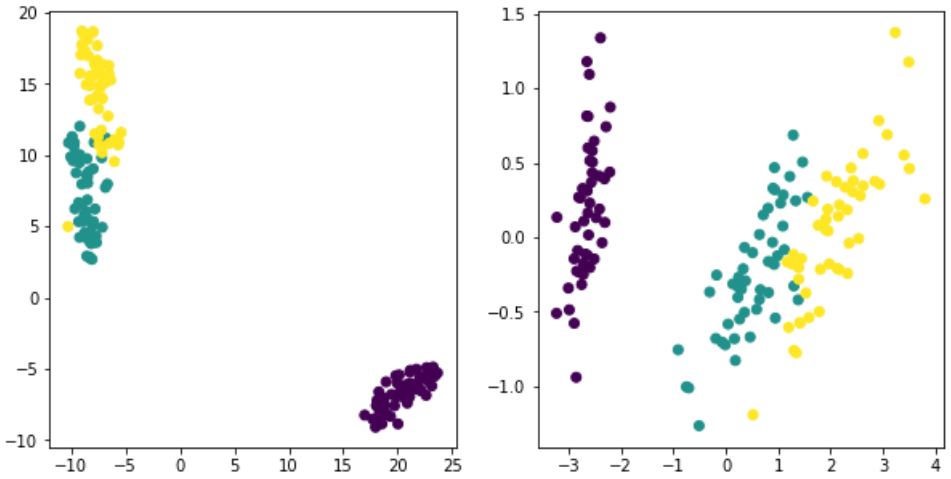
More information can be found here:

- Original paper: Visualizing High-Dimensional Data Using t-SNE (http://jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf)

In [0]:
```
from sklearn.manifold import TSNE
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA

iris = load_iris()
X_tsne = TSNE(learning_rate=100).fit_transform(iris.data)
X_pca = PCA().fit_transform(iris.data)
```

t-SNE can help us to decide whether classes are separable in some linear or nonlinear representation. Here we can see that the 3 classes of the Iris dataset can be separated quite easily. They can even be separated linearly which we can conclude from the low-dimensional embedding of the `PCA`.

In [0]:
```
plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=iris.target)
plt.subplot(122)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=iris.target)
plt.show()
```



High-dimensional data: Digits dataset

In high-dimensional and nonlinear domains, PCA is not applicable any more and many other manifold learning algorithms do not yield good visualizations either because they try to preserve the global data structure.

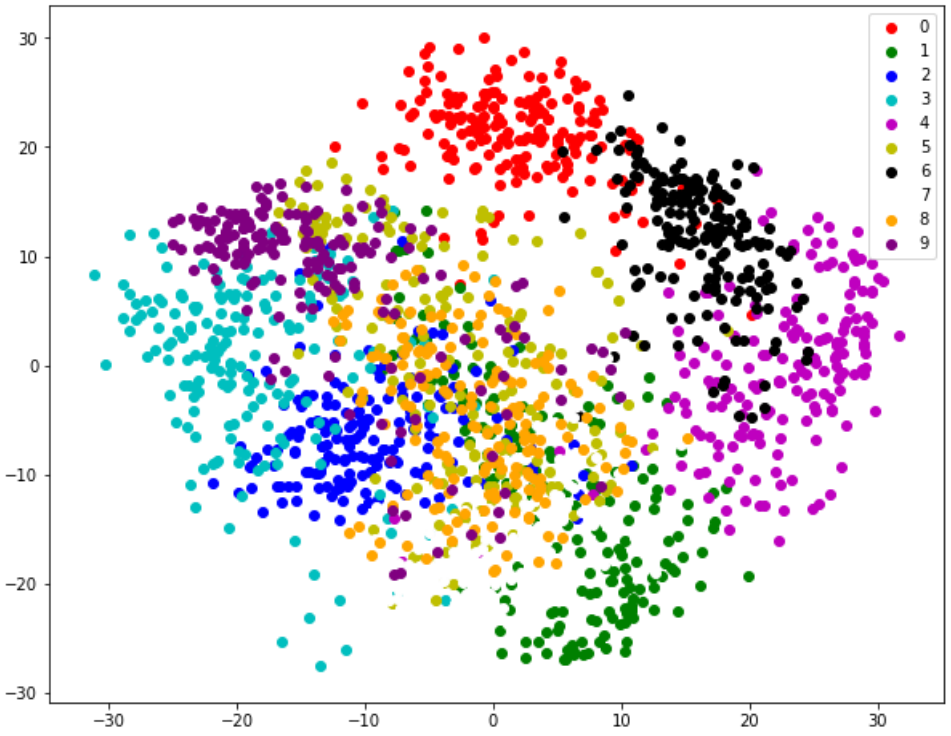
In [0]:
```
from sklearn import datasets
digits = datasets.load_digits()
X = digits.data
y = digits.target
```

In [0]:
```
from sklearn.manifold import TSNE
tsne = TSNE(n_components=2, random_state=0)
```

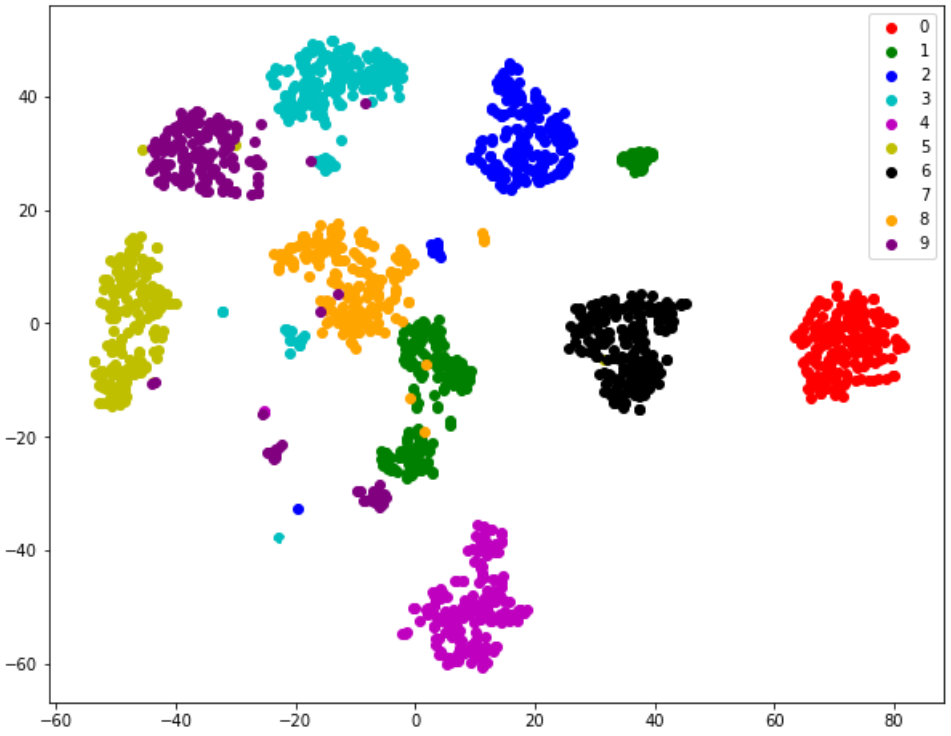
In [0]:
```
# Project data to 2D
X_2d = tsne.fit_transform(X)
```

In [0]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2, random_state=0)
X_2d_pca = pca.fit_transform(X)
```

In [0]:
```
# Visualize the data with PCA
target_ids = range(len(digits.target_names))

from matplotlib import pyplot as plt
plt.figure(figsize=(10, 8))
colors = 'r', 'g', 'b', 'c', 'm', 'y', 'k', 'w', 'orange', 'purple'
for i, c, label in zip(target_ids, colors, digits.target_names):
    plt.scatter(X_2d_pca[y == i, 0], X_2d_pca[y == i, 1], c=c, label=label)
plt.legend()
plt.show()
```



In [0]:
```
# Visualize the data with TSNE
target_ids = range(len(digits.target_names))

from matplotlib import pyplot as plt
plt.figure(figsize=(10, 8))
colors = 'r', 'g', 'b', 'c', 'm', 'y', 'k', 'w', 'orange', 'purple'
for i, c, label in zip(target_ids, colors, digits.target_names):
    plt.scatter(X_2d[y == i, 0], X_2d[y == i, 1], c=c, label=label)
plt.legend()
plt.show()
```



Density Estimation: Gaussian Mixture Models

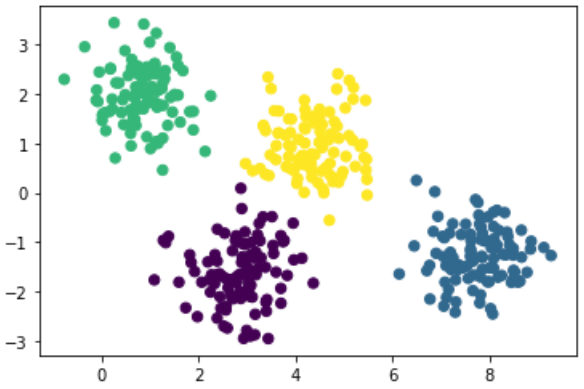
- We now explore **Gaussian Mixture Models**, which is an unsupervised clustering & density estimation technique.
- We previously saw an example of K-Means, which is a clustering algorithm which is most often fit using an expectation-maximization approach.
- Here we'll consider an extension to this which is suitable for both **clustering** and **density estimation**.

Motivating GMM: Weaknesses of K-Means

In [0]:
```
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.patches import Ellipse


def plot_kmeans(kmeans, X, n_clusters=4, rseed=0, ax=None):
    labels = kmeans.fit_predict(X)

    # plot the input data
    ax = ax or plt.gca()
    ax.axis('equal')
    ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2)

    # plot the representation of the KMeans model
    centers = kmeans.cluster_centers_
    radii = [cdist(X[labels == i], [center]).max()
             for i, center in enumerate(centers)]
    for c, r in zip(centers, radii):
        ax.add_patch(plt.Circle(c, r, fc='#CCCCCC', lw=3, alpha=0.5, zorder=1))

def draw_ellipse(position, covariance, ax=None, **kwargs):
    """Draw an ellipse with a given position and covariance"""
    ax = ax or plt.gca()

    # Convert covariance to principal axes
    if covariance.shape == (2, 2):
        U, s, Vt = np.linalg.svd(covariance)
        angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
        width, height = 2 * np.sqrt(s)
    else:
        angle = 0
        width, height = 2 * np.sqrt(covariance)

    # Draw the Ellipse
    for nsig in range(1, 4):
        ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                             angle, **kwargs))

def plot_gmm(gmm, X, label=True, ax=None):
    ax = ax or plt.gca()
    labels = gmm.fit(X).predict(X)
    if label:
        ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2)
    else:
        ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
    ax.axis('equal')

    w_factor = 0.2 / gmm.weights_.max()
    for pos, covar, w in zip(gmm.means_, gmm.covariances_, gmm.weights_):
        draw_ellipse(pos, covar, alpha=w * w_factor)
```

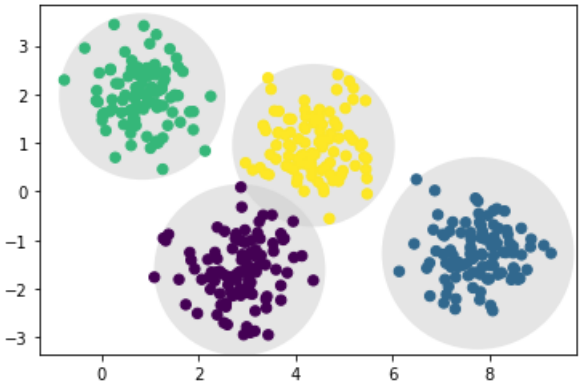
In [0]:
```
# Generate some data
from sklearn.datasets.samples_generator import make_blobs

X, y_true = make_blobs(n_samples=400, centers=4,
                       cluster_std=0.60, random_state=0)

X = X[:, ::-1] # flip axes for better plotting
```

In [0]:
```
# Plot the data with K Means Labels
from sklearn.cluster import KMeans

kmeans = KMeans(4, random_state=0)

labels = kmeans.fit(X).predict(X)

plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
```



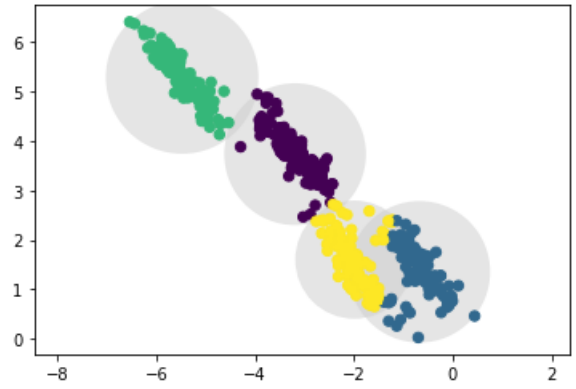
- From an intuitive standpoint, we might expect that the clustering assignment for some points is more certain than others: for example, there appears to be a very slight overlap between the two middle clusters, such that we might not have complete confidence in the cluster assigment of points between them.
- Unfortunately, the *K*-means model has no intrinsic measure of probability or uncertainty of cluster assignments.
- For this, we must think about generalizing the model.
- One way to think about the *K*-means model is that it places a circle (or, in higher dimensions, a hyper-sphere) at the center of each cluster, with a radius defined by the most distant point in the cluster.
- This radius acts as a **hard cutoff for cluster assignment** within the training set: any point outside this circle is not considered a member of the cluster.
- We can visualize this cluster model with the following function:

In [0]:
```
kmeans = KMeans(n_clusters=4, random_state=0)
plot_kmeans(kmeans, X)
```



- An important observation for *K*-means is that these cluster models **must be circular**: *K*-means has no built-in way of accounting for oblong or elliptical clusters.
- So, for example, if we take the same data and transform it, the cluster assignments end up becoming confused:

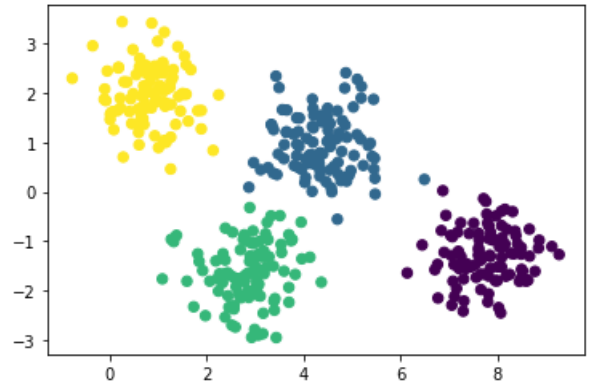
In [0]:
```
rng = np.random.RandomState(13)
X_stretched = np.dot(X, rng.randn(2, 2)) # data multipled with Gaussian noise

kmeans = KMeans(n_clusters=4)
plot_kmeans(kmeans, X_stretched)
```



- By eye, we recognize that these transformed clusters are non-circular, and thus circular clusters would be a poor fit.
- K-means is not flexible enough to account for this, and tries to force-fit the data into four circular clusters.
- This results in a mixing of cluster assignments where the resulting circles overlap: see especially the bottom-right of this plot.
- The disadvantages of K-means: its lack of flexibility in cluster shape and lack of probabilistic cluster assignment, means that for many datasets (especially low-dimensional datasets) it may not perform well.
- Time for Gaussian Mixture Models:

Gaussian Mixture Models

In [0]:
```
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=4).fit(X)
labels = gmm.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
```



- Since GMM is a probabilistic model under the hood, it is also possible to find probabilistic cluster assignments—in Scikit-Learn this is done using the `predict_proba` method.
- This returns a matrix of size `[n_samples, n_clusters]` which measures the probability that any point belongs to the given cluster:

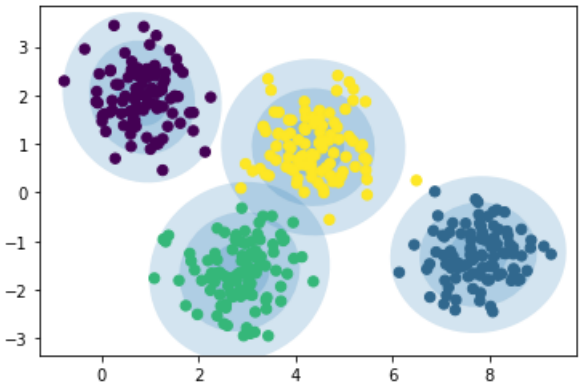
In [0]:
```
probs = gmm.predict_proba(X)
print(probs[:5].round(3))
```

```
[[0.469 0.531 0.    0.   ]
 [0.    0.    1.    0.   ]
 [0.    0.    1.    0.   ]
 [0.    1.    0.    0.   ]
 [0.    0.    1.    0.   ]]
```

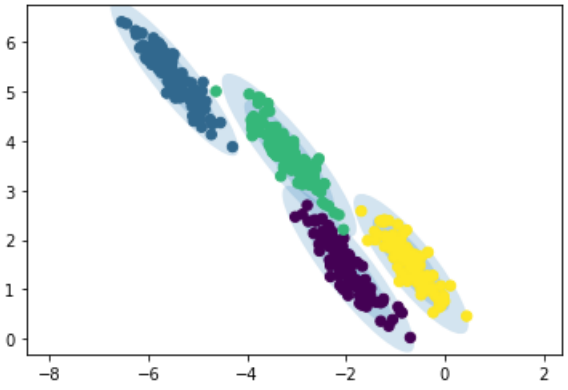
Under the hood, a Gaussian mixture model is very similar to K-means: it uses an **Expectation–Maximization** approach which qualitatively does the following:

1. Choose starting guesses for the location and shape
2. Repeat until converged:
   A. **E-step**: for each point, find weights encoding the probability of membership in each cluster
   B. **M-step**: for each cluster, update its location, normalization, and shape based on *all* data points, making use of the weights

In [0]:
```
gmm = GaussianMixture(n_components=4)
plot_gmm(gmm, X)
```



In [0]:
```
gmm = GaussianMixture(n_components=4, covariance_type='full')
plot_gmm(gmm, X_stretched)
```



In [0]:
```
print(gmm.weights_)
print(gmm.means_)
print(gmm.covariances_)
```

```
[0.24741389 0.24857307 0.25439375 0.24961929]
[[-1.95431213  1.40968809]
 [-5.47365516  5.27446354]
 [-3.15134442  3.71183859]
 [-0.68376734  1.51593736]]
[[[ 0.19785219 -0.2242275 ]
  [-0.2242275   0.31470063]]

 [[ 0.18634173 -0.20655237]
  [-0.20655237  0.27557375]]

 [[ 0.206731   -0.22384322]
  [-0.22384322  0.30246175]]

 [[ 0.15015313 -0.15799757]
  [-0.15799757  0.22222067]]]
```