クレジット:
UTokyo Online Educationデータマイニング入門 2018 森　純一郎

東京大学
THE UNIVERSITY OF TOKYO

CC BY NC ND

課題6 クラスタリング

In [ ]:
```
# モジュールのインポート
import csv
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# matplotlibでの日本語表示用
from matplotlib import font_manager
fp = font_manager.FontProperties(fname="ipag.ttf")
```

Q1 階層化クラスタリング

以下では、課題4で扱った講義データについて講義名間の類似度（距離）に基づき講義を階層化クラスタリングすることを考えます。

Q1.1 特徴量ベクトル

まず、以下の `tfidf()` 関数を準備します。 `tfidf()` 関数は、課題4と同様に講義名のテキストデータ（"course_list.csv"ファイル）と講義名に使われるキーワードのデータ（"keyword_list.csv"ファイル）から、各講義のTFIDFベクトルを行とする講義-単語行列を作成する関数です。

`tfidf()` 関数は、講義名を要素とするリスト `courses` 、キーワードを要素とするリスト `vocab` 、講義-単語行列（要素はTFIDF値） `tfidf_matrix` 、をそれぞれ返します。 `tfidf()` 関数を完成させてください。

In [ ]:
```
def tfidf():

```

以下の `similar_courses()` 関数は、課題4で作成した `find_similar_course()` 関数と同様に入力の講義に対してcos類似度に基づいて類似する講義名を探す関数ですが、以下の2つのリストを返します。

`ind` :入力の講義および類似する講義のインデックスのリスト

`labels` :入力の講義名および類似する講義名のリスト

`similar_courses()` 関数を完成させてください。

In [ ]:
```
def cos_sim(vec1, vec2):


def similar_courses(target, tfidf_matrix, courses):
    ### 引数：
    # target: 入力講義のインデックス
    # tfidf_matrix: TFIDF値が要素の講義-単語行列
    # courses: 講義名のリスト
```

Q1.2 デンドログラム

以下では、「生態統計学」を入力の講義として `similar_courses()` 関数を用いて類似講義を求めた結果を示しています。入力の講義自身も含めて100の類似講義があります。

In [ ]:
```
title="生態統計学"
ind, labels=similar_courses(courses.index(title),
                            tfidf_matrix, courses)
print(len(ind))
# print(ind)
# print(labels)
```

以下では、入力の講義について `similar_courses()` 関数で取得した類似講義（入力の講義も含む）をデータセットとして、講義名間の類似度（距離）に基づきデータセットを階層化クラスタリングします。

クラスタリングには `SciPy` の階層化クラスタリングモジュール（ `scipy.cluster.hierarchy` ）を用います。

`linkage` 関数は、特徴量ベクトルを行とする行列を入力とし、 `metric` 引数で指定した距離尺度と `method` 引数で指定した距離計算方法で階層化クラスタリングを行い、特徴量ベクトル間のクラスタリングの過程を行列として返します。

- metric引数に指定できる距離尺度: cosine, euclidean, jaccard, hamming, correlation, mahalanobis, など [linkage関数]
- method引数に指定できる距離計算方法: single, complete, average, ward, など [距離計算方法]

以下では、 `linkage` 関数の入力を `tfidf_matrix[ind,:]` として、先に抽出した類似講義のTFIDFベクトルを行とする講義-単語行列を指定しています。

`dendrogram` 関数は、 `linkage` 関数が出力したクラスタリング過程を表す行列を入力として、クラスタリングの過程をデンドログラムとして可視化します。 `labels` 引数には各特徴量ベクトルに付与するラベルのリストを指定します。

In [ ]:
```
from scipy.cluster.hierarchy import linkage
from scipy.cluster.hierarchy import dendrogram

courses, vocab, tfidf_matrix=tfidf()
title="生態統計学"
ind, labels=similar_courses(courses.index(title), tfidf_matrix,
                            courses)

# 階層化クラスタリング
# 距離尺度を'euclidean'、距離計算は'single'
clusters = linkage(tfidf_matrix[ind,:],
                   metric = 'euclidean', method = 'single')

# デンドログラム
plt.figure(figsize=(25,10))
dend = dendrogram(clusters,  labels=labels)
# ラベルを日本語表示
for l in plt.gca().get_xticklabels(): l.set_fontproperties(fp)

plt.ylabel('Distance');
```

距離計算方法を complete や average に変更して階層化クラスタリングを行い、得られるクラスタがどのように変化するか、それぞれのデンドログラムを可視化して観察してください。

In [ ]:
```
```

参考

linkage 関数には、以下のように特徴量ベクトル間の距離を表す行列を入力することもできます。 pdist 関数は metric 引数で指定した距離尺度で特徴量ベクトル間の距離を計算し、それらの距離を要素とする行列（正確には行列の上半分をベクトルにしたもの）を返します。距離行列を linkage 関数の入力とした場合は、 linkage 関数で距離尺度を指定する必要はありません。

In [ ]:
```
from scipy.spatial.distance import pdist

clusters = linkage(pdist(tfidf_matrix[ind,:], metric='euclidean'),
                   method='single')

plt.figure(figsize=(25,10))
dend = dendrogram(clusters,  labels=labels)
for l in plt.gca().get_xticklabels(): l.set_fontproperties(fp)
plt.ylabel('Distance');
```

In [ ]:
```
from scipy.spatial.distance import squareform
# 距離行列
dist_matrix = \
 pd.DataFrame(squareform(pdist(tfidf_matrix[ind,:], metric='euclidean'
              columns=labels, index=labels)
dist_matrix
```

scikit-learnライブラリ

`scikit-learn` ライブラリには分類、回帰、クラスタリング、次元削減、前処理、モデル選択などの機械学習の処理を行うためのモジュールが含まれています。以下では、 `scikit-learn` ライブラリの `KMeans` クラスを使ったk-means法によるクラスタリングについて説明します。

機械学習では、観察されたデータをよく表すようにモデルのパラメータの調整を行います。パラメータを調整することでモデルをデータに適合させるので、「学習」と呼ばれます。特に、観測されたデータの特徴のみからそのデータセットの構造やパターンをよく表すようなモデルを学習することを**教師なし学習**と呼びます。クラスタリングは教師なし学習の例です。クラスタリングでは、観測されたデータをクラスタと呼ばれる集合にグループ分けします。

`scikit-learn` ライブラリには機械学習に用いる代表的なデータセットが含まれています。以下では、 `load_iris` 関数により `iris` データセットをロードしています。 `iris` データセットにはアヤメの花を4つの特徴量（'sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)'）で表した150個の特徴量ベクトルが含まれています（ `iris['data']`)。また、各花には3種類（0:'setosa', 1:'versicolor', 2:'virginica'）のいずれかがラベルとして付与されています（ `iris['target']`)

`iris['data']` 行列の各行は1つの花の特徴量ベクトルに対応しており、行数はデータの総数（この場合は150）を表します。1列目から4列目までの各列は上記の特徴量に対応しています。 `scikit-learn` では、この特徴量ベクトルからなる行列を `NumPy` 配列または `pandas` のデータフレームに格納し、入力データとして処理します。

In [ ]:
```
from sklearn.datasets import load_iris
iris = load_iris() # データセットのロード
print(len(iris['data'])) # データの総数
print(iris['feature_names']) # 特徴量名
print(iris['data'][0:5]) # データ（先頭5行を表示）
print(iris['target_names']) # ラベル名
print(iris['target'][0:5], iris['target'][50:55],
      iris['target'][100:105]) # ラベル
```

`iris['data']` の3列目と4列目の2つの特徴量（'petal length (cm)', 'petal width (cm)'）を用いて散布図でデータを可視化してみます。

In [ ]:
```
X_iris=iris['data'][:,2:4]
plt.figure(figsize=(7,5))
plt.xlabel(iris["feature_names"][2])
plt.ylabel(iris["feature_names"][3])
plt.scatter(X_iris[:,0],X_iris[:,1],c='black');
```

アヤメの種類（0:'setosa', 1:'versicolor', 2:'virginica'）ごとに色を変えて同様にデータを可視化してみると、種類ごとにグループになっており、アヤメの種類ごとに似た特徴量を持っていることがわかります。以下では、クラスタリングによりこれらのグループをクラスタとして自動的に抽出することを考えます。

In [ ]:
```
labels=iris['target']
plt.figure(figsize=(7,5))
plt.xlabel(iris["feature_names"][2])
plt.ylabel(iris["feature_names"][3])
plt.scatter(X_iris[labels==0,0],X_iris[labels==0,1],
            c='green', alpha=0.2)
plt.scatter(X_iris[labels==1,0],X_iris[labels==1,1],
            c='yellow', alpha=0.2)
plt.scatter(X_iris[labels==2,0],X_iris[labels==2,1],
            c='blue', alpha=0.2);
```

`scikit-learn` では、以下の手順でデータからモデルの学習を行います。

- 使用するモデルのクラスの選択
- モデルのハイパーパラメータの選択とインスタンス化
- データの準備
  - 教師なし学習では、特徴量データを準備
- モデルをデータに適合（ `fit()` メソッド）
- モデルの評価
  - 教師なし学習では、 `transform()` または `predict()` メソッドを用いて特徴量データのクラスタリングや次元削減などを行う

以下では、 `iris` データセットの2つの特徴量（'petal length (cm)', 'petal width (cm)'）を元にアヤメのデータをk-means法によりクラスタリングする手続きを示しています。

モデルのクラスとして `KMeans` を選択し、モデルのハイパーパラメータの選択とインスタンス化では、引数 `n_clusters` にハイパーパラメータとしてクラスタ数、ここでは3、を指定して `KMeans` クラスのインスタンスを作成しています。そして、 `fit()` メソッドによりモデルを入力データ `X_iris` に適合させ、 `predict()` メソッドを用いて各データが所属するクラスタ（0,1,2で表現）の情報をリストとして取得しています。最後に、クラスタリングの結果を元に、クラスタごとに異なる色でデータを可視化しています。

In [ ]:
```
from sklearn.cluster import KMeans

model = KMeans(n_clusters=3)
model.fit(X_iris)
clusters=model.predict(X_iris)

# クラスタの可視化（irisデータセット、3クラスタ、特徴量2つを仮定）
plt.figure(figsize=(7,5))
plt.xlabel(iris["feature_names"][2])
plt.ylabel(iris["feature_names"][3])

plt.scatter(X_iris[clusters==0,0],X_iris[clusters==0,1],
            c='blue',alpha=0.2)
plt.scatter(X_iris[clusters==1,0],X_iris[clusters==1,1],
            c='green', alpha=0.2)
plt.scatter(X_iris[clusters==2,0],X_iris[clusters==2,1],
            c='yellow', alpha=0.2)

plt.scatter(model.cluster_centers_[:,0], model.cluster_centers_[:,1],
            c='red'); # クラスタ中心
# print(model.inertia_) # コスト関数の値
```

Q2 k-means法

Q2.1 平方ユークリッド距離

n-次元ベクトル空間における、任意の2つのベクトル、$\vec{x} = (x_1, x_2, \ldots, x_n)$、$\vec{y} = (y_1, y_2, \ldots, y_n)$、の間の平方ユークリッド距離$\|\vec{x} - \vec{y}\|^2$は以下のように定義されます。

$$\Sigma_{i=1}^{n}(x_i - y_i)^2 = \vec{x} \cdot \vec{x} + \vec{y} \cdot \vec{y} - 2\vec{x} \cdot \vec{y}$$

入力ベクトル $\vec{x}, \vec{y}$をそれぞれ `NumPy` の配列として引数で受け取り、それらのベクトル間の平方ユークリッド距離を計算して返す関数 `squared_euclid` を完成させてください。

In [ ]:
```
def squared_euclid(x, y):

```

Q2.2 k-means法

以下では、k-means法によるクラスタリングを行う `kmeans` 関数を実装します。 `kmeans` 関数では第1引数に入力 `X` （データ数(m)×特徴量数(n)の行列）、第2引数にクラスタ数を受け取り、以下の処理を行います。（引数には第3引数に中心への割り当てと中心点の更新の繰り返し数（既定値10）、第4引数に中心をランダムに選ぶ際のシード（既定値0）、を指定できますが省略可能です。）

- ***初期化***
  - まず、クラスタの数(K)だけ中心を入力データからランダムに選び、それらを初期の中心$\mu^{(k)}(k = 0, \ldots, K-1)$とします。
- ***1. 中心への割り当て***
  - 各データ$x^{(i)}(i = 0, \ldots, m-1)$と各中心$\mu^{(k)}$の平方ユークリッド距離を計算し、その結果を距離行列 `d[i,k]` に代入します。距離行列 `d` はデータを行、中心を列として、各データと各中心の距離を要素とする行列です。
  - 距離行列 `d[i,k]` の各行について、その行に対応するデータに最も近い中心をそのデータが属するクラスタの中心として選びます。これにより、各データ$x^{(i)}$がどのクラスタ中心$\mu^{(k)}$に属するかを表す以下の配列 `clusters` を更新します。
  - データ$x^{(i)}$の最近接の中心が$\mu^{(k)}$であれば `clusters[i]=k`
  - この時、以下のように距離行列 `d` の各行ごとにその行の最小値を持つ列のインデックスを `np.argmin` で取得することで配列 `clusters` を作成できます。

```
clusters = np.argmin(d,axis=1)
```

- ***2. 中心の更新***
  - そして、各中心$\mu^{(k)}$に属するデータ$x^{(k,i)}$を用いて各中心を更新します
  - $\mu^{(k)} = \Sigma_i x^{(k,i)}/(\mu^{(k)}$**に属するデータ数**$)$
  - この時、 `clusters` 配列とインデックス参照を用いて、中心$\mu^{(k)}$に属するデータの特徴ベクトル集合を入力 `X` から以下のように取得できます。

```
X[clusters==k,:]
```

  - これらの特徴ベクトルの平均のベクトル（ヒント: `np.mean`)を新たな中心$\mu^{(k)}$のベクトルとします。

k-means法では、上記のデータの中心への割り当てと中心の更新を繰り返すことで、クラスタリングを行います。繰り返し処理が終わったら、 `kmeans` 関数は、各データの所属するクラスタを表す配列 `clusters` 、各クラスタ中心のベクトルを行とする行列 `centers` 、最終的なコスト関数の値 `cost` を返します。上記に従って、 `kmeans` 関数を完成させてください。

In [ ]:
```
def kmeans(X, n_clusters, max_iter=10, rand_seed=0):
    ### 引数：
    # X: 入力（データ数x特徴量数の行列）
    # n_clusters: クラスタ数
    # max_iter: 中心への割り当てと中心の更新の繰り返し数
    # rand_seed: 中心をランダムに選ぶ際のシード

    # クラスタ数のだけ中心を入力データからランダムに選ぶ
        # 補1: k-means++の場合は、この中心の選び方を変更
        # 補2: クラスタ数が比較的小さい時は異なる初期中心を試し
        # 最終的にコストが小さくなる初期中心を採用するのがよい
    np.random.seed(rand_seed)
    centers=X[np.random.choice(X.shape[0],n_clusters),:]
```

Q2.3 エルボー法

クラスタリングにおいて、クラスタ数を変化させた時のクラスタリングのコストの変化に基づいて、最適なクラスタ数を決定する方法をエルボー法と呼びます。エルボー法では、クラスタ数を増やしていった時にコストが最も大きく減少する時のクラスタ数を採用します。以下の `elbow` 関数は、Q2.2で作成した `kmeans` 関数を用いて、クラスタ数を1から1つずつ増やしていった時のクラスタリングのコストの変化を可視化します。 `elbow` 関数を完成させてください。

irisデータセットの2つの特徴量（'petal length (cm)', 'petal width (cm)'）から花の特徴量ベクトルを作成し、k-means法でクラスタリングするとコストが最も大きく減少するのはクラスタ数を1から2にした時であることがわかる。実際の花の種類は3種類であるが、'versicolor'と'virginica'の種類の花は特徴空間上では近傍に分布しているため、全体を'setosa'とそれ以外（'versicolor', 'virginica'）の2つクラスタとみなした時、全体のデータを大きく2つに識別でき、クラスタリングのコストが大きく減少する。

In [ ]:
```
def elbow(X, K):
    ### 引数：
    # X: 入力データ
    # K: 最大のクラスタ数

iris = load_iris()
X_iris=iris['data'][:,2:4]
elbow(X_iris, 5)
```