

学修番号・氏名

- 学修番号: 23745118
- 氏名: 高嶋隆一

課題1 日本語テキストのクラスタリング、分類

各自が収集した日本語のRSS/RDFニュースフィードに対し、クラスタリングおよび分類を行い、その結果について考察しなさい。

- 1.1 クラスタリング
 - 考察対象としたクラスタリングに用いたクラスタ数を決定した理由を示すこと
- 1.2 分類
 - 3つの分類アルゴリズムで分類を行うこと
 - 3つの分類アルゴリズムのそれぞれについて、教師データに対する分類精度（あてはまり）とテストデータに対する分類精度の比較を行うこと
 - テストデータの分類精度の計測では、グリッドサーチを用いて決定した最適なパラメータを用いること

以下のセルは自由に追加して構わない。また、以下に注意すること。

- コメントやMarkdown形式での説明を適宜入れ、プログラムや結果をわかりやすく説明すること
- 【特に重要】JupyterLabのタブの下のバー（メニュー）にある  (Restart the kernel, then re-run the whole notebook) をクリックし、先頭セルから順番に全てのセルを実行、セルの番号が [1] から順に付き、実行結果を得た状態で提出すること

準備：日本語テキストデータの読み込み、前処理

フィードデータの情報源 (feedlist_jp.txt)

www.nhk.or.jp から7カテゴリ、www.asahi.com から7カテゴリ、news.yahoo.co.jpから9カテゴリを購読した。

<https://www.nhk.or.jp/rss/news/cat1.xml>
<https://www.asahi.com/rss/asahi/national.rdf>
<https://www.nhk.or.jp/rss/news/cat4.xml>
<https://www.asahi.com/rss/asahi/politics.rdf>
<https://www.nhk.or.jp/rss/news/cat5.xml>

```
https://www.asahi.com/rss/asahi/business.rdf
https://www.nhk.or.jp/rss/news/cat6.xml
https://www.asahi.com/rss/asahi/international.rdf
https://www.nhk.or.jp/rss/news/cat7.xml
https://www.asahi.com/rss/asahi/sports.rdf
https://www.nhk.or.jp/rss/news/cat2.xml
https://www.asahi.com/rss/asahi/culture.rdf
https://www.nhk.or.jp/rss/news/cat3.xml
https://www.asahi.com/rss/asahi/science.rdf
https://news.yahoo.co.jp/rss/categories/domestic.xml
https://news.yahoo.co.jp/rss/categories/world.xml
https://news.yahoo.co.jp/rss/categories/business.xml
https://news.yahoo.co.jp/rss/categories/entertainment.xml
https://news.yahoo.co.jp/rss/categories/sports.xml
https://news.yahoo.co.jp/rss/categories/it.xml
https://news.yahoo.co.jp/rss/categories/science.xml
https://news.yahoo.co.jp/rss/categories/life.xml
https://news.yahoo.co.jp/rss/categories/local.xml
```

フィードデータの読み込み

- spaCy の日本語モデルの読み込み
- 日本語フィードデータの読み込み

```
In [1]: import pandas as pd
import re
import spacy

# FutureWarningを無視する
import warnings
warnings.filterwarnings('ignore', category=FutureWarning)

# 日本語モデル
nlp = spacy.load('ja_core_news_lg')

# フィードデータの読み込み、確認
feeds = pd.read_csv('data/output_jp.csv')

# title と summary を結合
# str.cat() により複数列の文字列を結合
# - sep=' ': 間に挟む文字列
# - na_rep='': NaN は空文字列に変換（指定しないと結合結果が NaN になる）
feeds['text'] = feeds['title'].str.cat(feeds['summary'], sep='。', na_rep='')

# 不要になった列を削除した処理用の DataFrame
df = feeds.drop(['title', 'summary'], axis=1)

# 確認
df.head()
```

Out[1]:

	url	text
0	https://www.nhk.or.jp/rss/news/cat1.xml	オスプレイ墜落事故 海兵隊オスプレイ2機が飛来 捜索活動など。アメリカ軍の輸送機オスプレイが...
1	https://www.nhk.or.jp/rss/news/cat1.xml	9人死亡 笹子トンネル事故から11年 現場近くで追悼慰靈式。山梨県の中央自動車道の笹子トンネ...
2	https://www.nhk.or.jp/rss/news/cat1.xml	二階派もパーティー収入のノルマ超え 不記載の疑い。自民党の派閥の政治資金パーティーをめぐる問...
3	https://www.nhk.or.jp/rss/news/cat1.xml	18歳女性 遺体遺棄事件 6月7日に自宅を出た当日に死亡か。東京都内に住む18歳の女性が山梨...
4	https://www.nhk.or.jp/rss/news/cat1.xml	忘年会は業務ですか？残業代、出ますか？調べてみると....。「忘年会は業務ですか？」 「残業代、出...

日本語テキストに対する前処理

preprocess(text) として定義

- 表記の正規化
- トークン化（形態素解析）
- ストップワードの除去
- 見出し語化

In [2]:

```
%time

# 不要な単語を除去
# - ストップワード (is_stop)
# - いくつかの品詞
#   AUX: 助動詞
#   PUNCT: 句読点
#   SPACE: 空白文字
#   SYM: 記号
#   X: その他
# - うまく取り除けない単語や文字
stop_pos = ['AUX', 'PUNCT', 'SPACE', 'SYM', 'X']
#stop_words = ['.']
# 処理結果に多く含まれる全角数字、括弧、年、月、日、時等を stop_words に含める
stop_words = ['.', '年', '月', '日', '時', '分',
              '1', '2', '3', '4', '5', '6', '7', '8', '9', '0',
              '(', ')', '?', '%', '(', ')', '%', '?', '/', '!', '!', '...']

def token_to_add(w):
    t = w.text      # 単語
    p = w.pos_      # 品詞
    l = w.lemma_    # 原型

    # ストップワードは None を返す
    if w.is_stop:
        return None
```

```

if p in stop_pos:
    return None
if l in stop_words:
    return None
if w.is_digit: # 数値をチェックして除去
    return None

if len(l) == 0:
    return t
return l

def preprocess(text):
    tokens = []

    for w in nlp(text):
        t = token_to_add(w)
        if t is not None:
            tokens.append(t)

    # トークンのリストを返す
    return tokens

```

CPU times: user 1 µs, sys: 0 ns, total: 1 µs
Wall time: 2.86 µs

テキストのベクトル化

```

In [3]: %%time

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

# BoW でベクトル化
vectorizer_bow = CountVectorizer(tokenizer=preprocess)
vector_bow = vectorizer_bow.fit_transform(df.text)

# TF-IDF でベクトル化
vectorizer_tfidf = TfidfVectorizer(tokenizer=preprocess)
vector_tfidf = vectorizer_tfidf.fit_transform(df.text)

```

/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/feature_extraction/text.py:528: UserWarning: The parameter 'token_pattern' will not be used since 'tokenizer' is not None'
warnings.warn(

CPU times: user 3min 40s, sys: 1.47 s, total: 3min 41s
Wall time: 3min 42s

課題 1.1 クラスタリング

1.1.1 プログラムと実行結果

クラスタ数の推定

- 非階層的クラスタ分析ではクラスタ数を決めてデータをグループに分割
- エルボー法
- シルエット分析

In [4]:

```
%%time

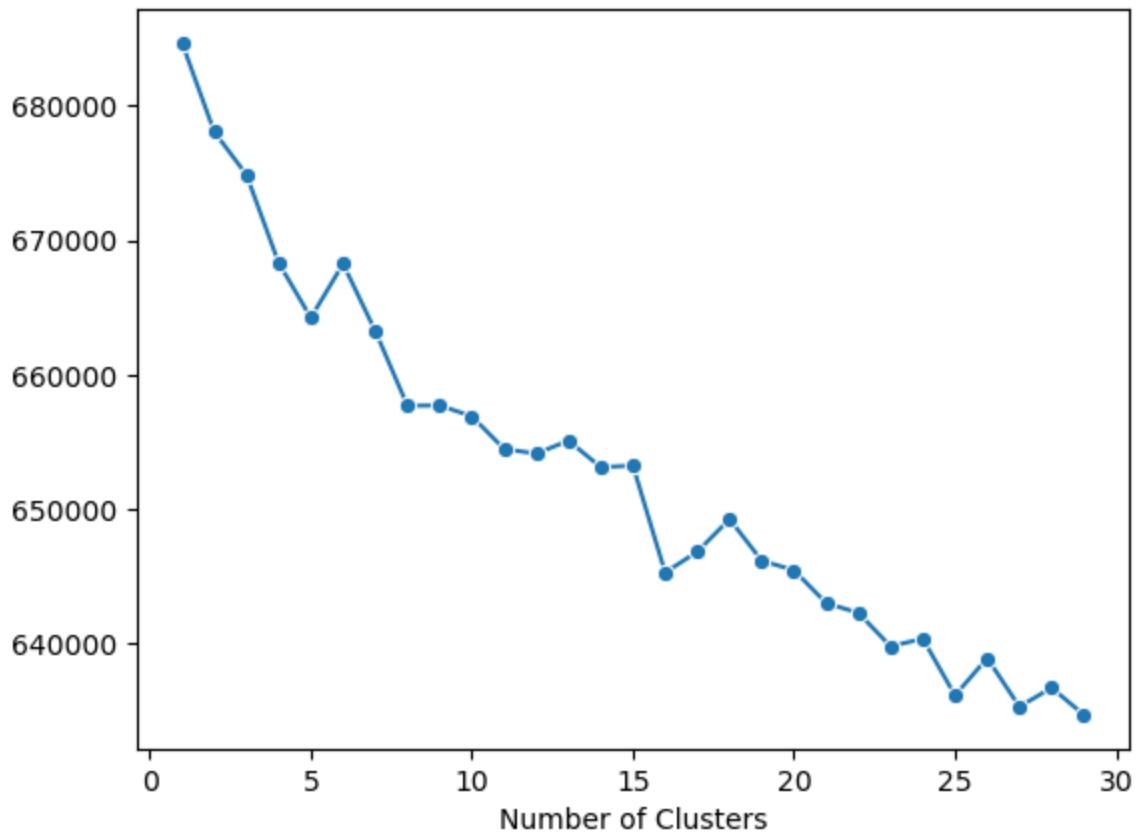
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# エルボー法
# https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/
def elbow(vector, title):
    x_range = range(1, 30)
    distortions = []
    for n in x_range:
        model = KMeans(n_clusters=n, random_state=0, n_init=10)
        model.fit(vector)
        distortions.append(model.inertia_)

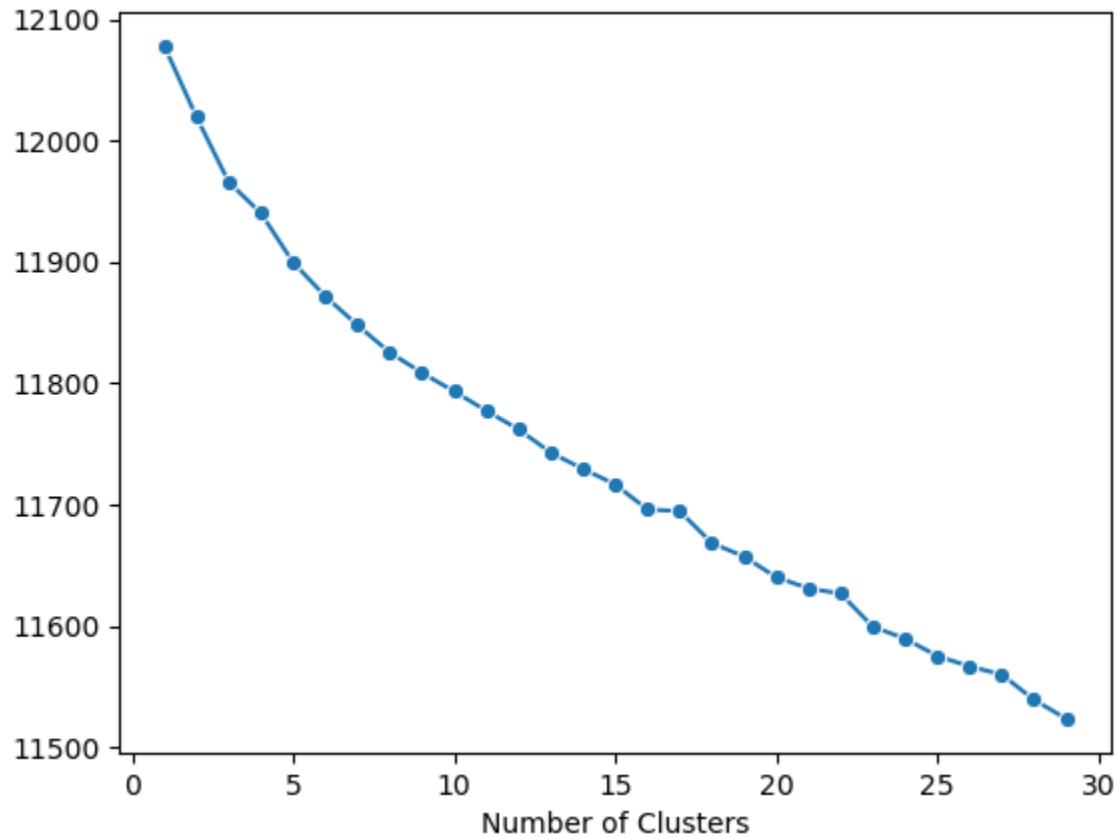
    sns.lineplot(x=x_range, y=distortions, marker='o')
    plt.xlabel('Number of Clusters')
    plt.title(title)
    plt.show()

elbow(vector_bow, "Elbow: BoW")
elbow(vector_tfidf, "Elbow: TF-IDF")
```

Elbow: BoW



Elbow: TF-IDF



CPU times: user 11min 35s, sys: 4min 6s, total: 15min 41s
Wall time: 2min 2s

In [5]:

```
%%time

# シルエット分析
# https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/
import numpy as np
from sklearn.metrics import silhouette_samples
from matplotlib import cm

def silhouette(X, n, plot=True):
    model = KMeans(n_clusters=n, random_state=0, n_init=10)
    model.fit(X)
    cluster_labels = set(model.labels_)
    n_clusters = len(cluster_labels)
    silhouette_vals = silhouette_samples(X, model.labels_, metric='euclidean')
    if plot:
        y_lower, y_upper = 0, 0
        yticks = []
        for i, c in enumerate(cluster_labels):
            c_silhouette_vals = silhouette_vals[model.labels_ == c]
            c_silhouette_vals.sort()
            y_upper += len(c_silhouette_vals)
            color = cm.jet(float(i) / n_clusters)
            plt.barh(range(y_lower, y_upper), c_silhouette_vals, height=1.0,
                    edgecolor='none', color=color)
            yticks.append((y_lower + y_upper) / 2.)
            y_lower += len(c_silhouette_vals)

        silhouette_avg = np.mean(silhouette_vals)
        if plot:
            plt.axvline(silhouette_avg, color="red", linestyle="--")
            plt.show()
        print('{} Clusters: Average silhouette coefficient: {:.3f}'.format(n, si

print("silhouette: BoW")
for n in range(2, 30):
    silhouette(vector_bow, n, plot=False)

print("silhouette: TF-IDF")
for n in range(2, 30):
    silhouette(vector_tfidf, n, plot=False)
```

silhouette: BoW

2 Clusters: Average silhouette coefficient: 0.113
3 Clusters: Average silhouette coefficient: 0.081
4 Clusters: Average silhouette coefficient: 0.083
5 Clusters: Average silhouette coefficient: 0.082
6 Clusters: Average silhouette coefficient: 0.029
7 Clusters: Average silhouette coefficient: 0.045
8 Clusters: Average silhouette coefficient: 0.048
9 Clusters: Average silhouette coefficient: 0.041
10 Clusters: Average silhouette coefficient: 0.044
11 Clusters: Average silhouette coefficient: 0.035
12 Clusters: Average silhouette coefficient: 0.029
13 Clusters: Average silhouette coefficient: -0.019
14 Clusters: Average silhouette coefficient: -0.048
15 Clusters: Average silhouette coefficient: -0.002
16 Clusters: Average silhouette coefficient: 0.016
17 Clusters: Average silhouette coefficient: 0.018
18 Clusters: Average silhouette coefficient: 0.017
19 Clusters: Average silhouette coefficient: 0.020
20 Clusters: Average silhouette coefficient: 0.030
21 Clusters: Average silhouette coefficient: 0.047
22 Clusters: Average silhouette coefficient: 0.028
23 Clusters: Average silhouette coefficient: 0.042
24 Clusters: Average silhouette coefficient: 0.044
25 Clusters: Average silhouette coefficient: 0.050
26 Clusters: Average silhouette coefficient: 0.043
27 Clusters: Average silhouette coefficient: 0.015
28 Clusters: Average silhouette coefficient: -0.072
29 Clusters: Average silhouette coefficient: 0.025

silhouette: TF-IDF

2 Clusters: Average silhouette coefficient: 0.002
3 Clusters: Average silhouette coefficient: 0.003
4 Clusters: Average silhouette coefficient: 0.004
5 Clusters: Average silhouette coefficient: 0.005
6 Clusters: Average silhouette coefficient: 0.004
7 Clusters: Average silhouette coefficient: 0.005
8 Clusters: Average silhouette coefficient: 0.005
9 Clusters: Average silhouette coefficient: 0.006
10 Clusters: Average silhouette coefficient: 0.006
11 Clusters: Average silhouette coefficient: 0.006
12 Clusters: Average silhouette coefficient: 0.008
13 Clusters: Average silhouette coefficient: 0.007
14 Clusters: Average silhouette coefficient: 0.007
15 Clusters: Average silhouette coefficient: 0.007
16 Clusters: Average silhouette coefficient: 0.008
17 Clusters: Average silhouette coefficient: 0.008
18 Clusters: Average silhouette coefficient: 0.009
19 Clusters: Average silhouette coefficient: 0.008
20 Clusters: Average silhouette coefficient: 0.009
21 Clusters: Average silhouette coefficient: 0.009
22 Clusters: Average silhouette coefficient: 0.010
23 Clusters: Average silhouette coefficient: 0.010
24 Clusters: Average silhouette coefficient: 0.010
25 Clusters: Average silhouette coefficient: 0.009
26 Clusters: Average silhouette coefficient: 0.010
27 Clusters: Average silhouette coefficient: 0.010

```
28 Clusters: Average silhouette coefficient: 0.012
29 Clusters: Average silhouette coefficient: 0.013
CPU times: user 13min 1s, sys: 4min 40s, total: 17min 41s
Wall time: 3min 19s
```

非階層的クラスタリング

- TF-IDF の結果をクラスタリング

In [6]:

```
%%time

# クラスタ数
N_clusters = 5

# KMeans の初期化
clusters = KMeans(n_clusters=N_clusters, n_init = 10).fit_predict(vector_tfidf)

# 結果を DataFrame にまとめる
df_cluster = pd.DataFrame(clusters, columns=['cluster'])
```

```
CPU times: user 4.45 s, sys: 1.27 s, total: 5.72 s
Wall time: 764 ms
```

クラスタごとの棒グラフ

- クラスタごとに、頻度とTF-IDFの棒グラフを描画

In [7]:

```
%%time

# matplotlib: 日本語フォントの設定
from matplotlib import rcParams
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Hiragino Maru Gothic Pro', 'Yu Gothic', 'Mei
                                'Takao', 'IPAexGothic', 'IPAPGothic', 'Noto S

# 単語の DataFrame (Bow)
df_words_bow = pd.DataFrame(vectorizer_bow.get_feature_names_out())

# テキストを Bow ベクトル化した結果を DataFrame
df_vector_bow = pd.DataFrame.sparse.from_spmatrix(vector_bow)

# 単語の DataFrame (TF-IDF)
df_words_tfidf = pd.DataFrame(vectorizer_tfidf.get_feature_names_out())

# テキストをベクトル化した結果の DataFrame
df_vector_tfidf = pd.DataFrame.sparse.from_spmatrix(vector_tfidf)

for i in range(0, N_clusters):
    print('Cluster', i)
    _cdf = df_cluster[df_cluster['cluster'] == i]

    # 頻度の棒グラフ
    df_counts = pd.concat([df_words_bow, df_vector_bow.iloc[_cdf.index].sum(
        df_counts.columns=['word', 'counts']])
```

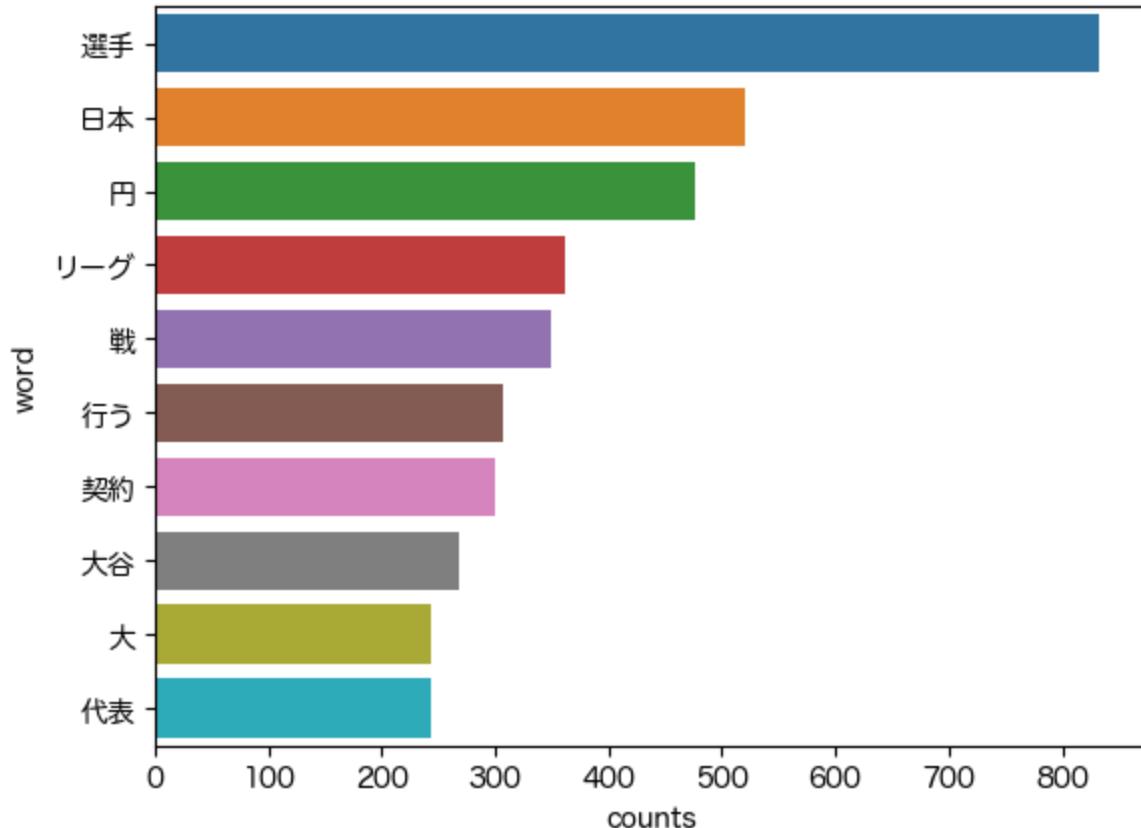
```

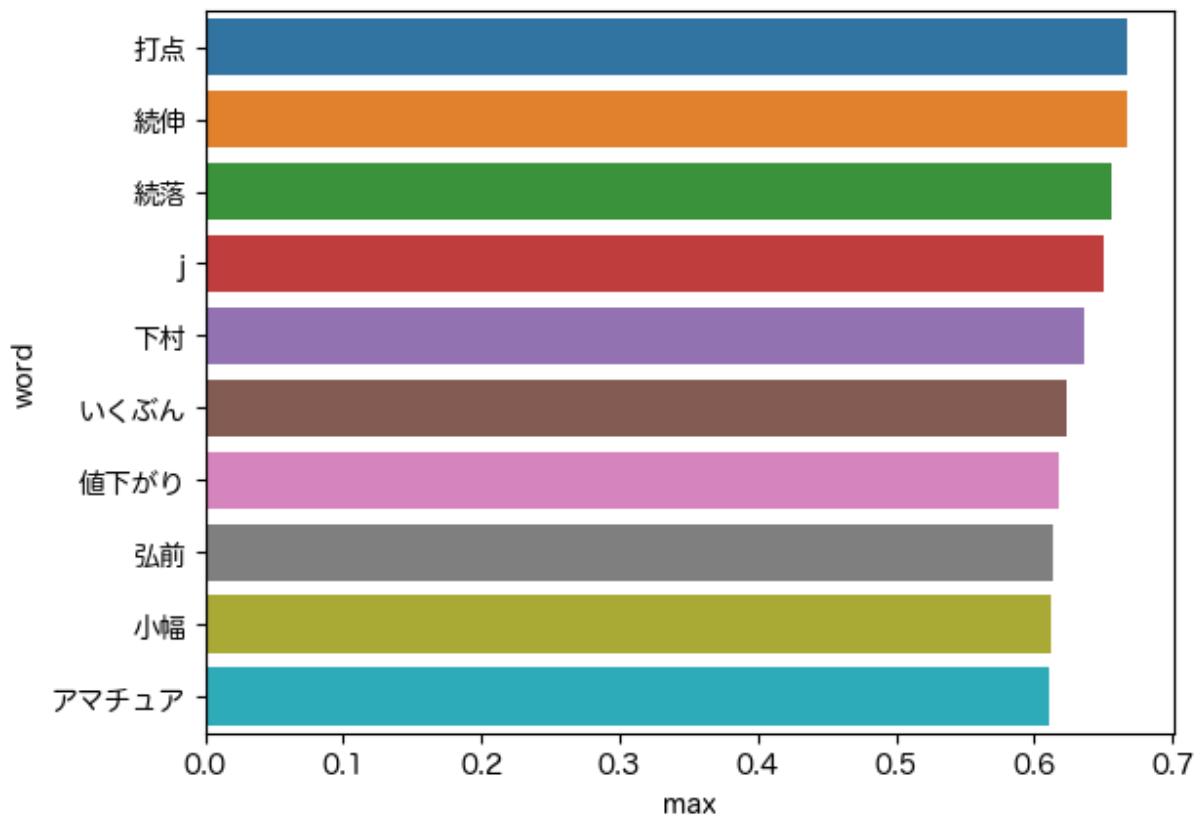
df_bar_bow = df_counts.sort_values('counts', ascending=False).head(10)
sns.barplot(x=df_bar_bow.counts, y=df_bar_bow.word, orient='h')
plt.show()

# TF-IDFの棒グラフ
_cdf_max = df_vector_tfidf.iloc[_cdf.index].max()
df_max = pd.concat([df_words_tfidf, _cdf_max], axis=1)
df_max.columns=['word', 'max']
df_bar_tfidf = df_max.sort_values('max', ascending=False).head(10)
sns.barplot(x=df_bar_tfidf['max'], y=df_bar_tfidf.word, orient='h')
plt.show()

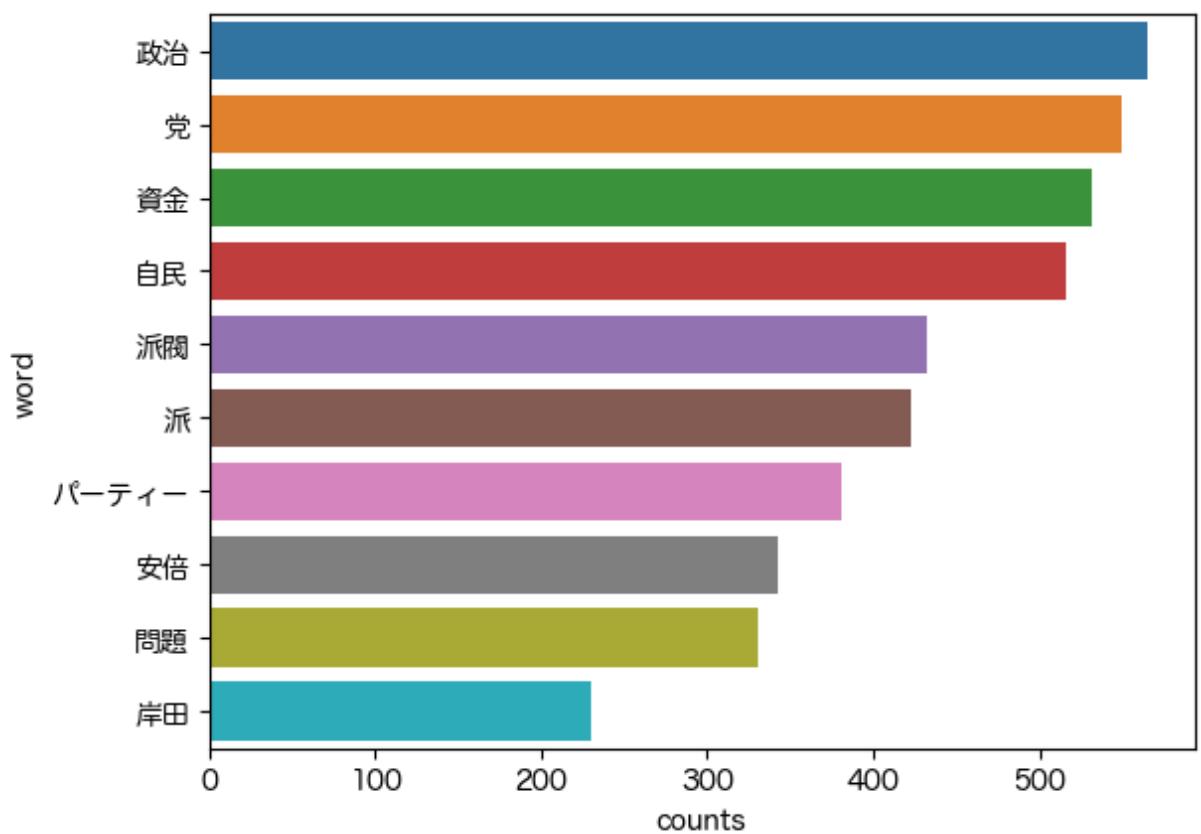
```

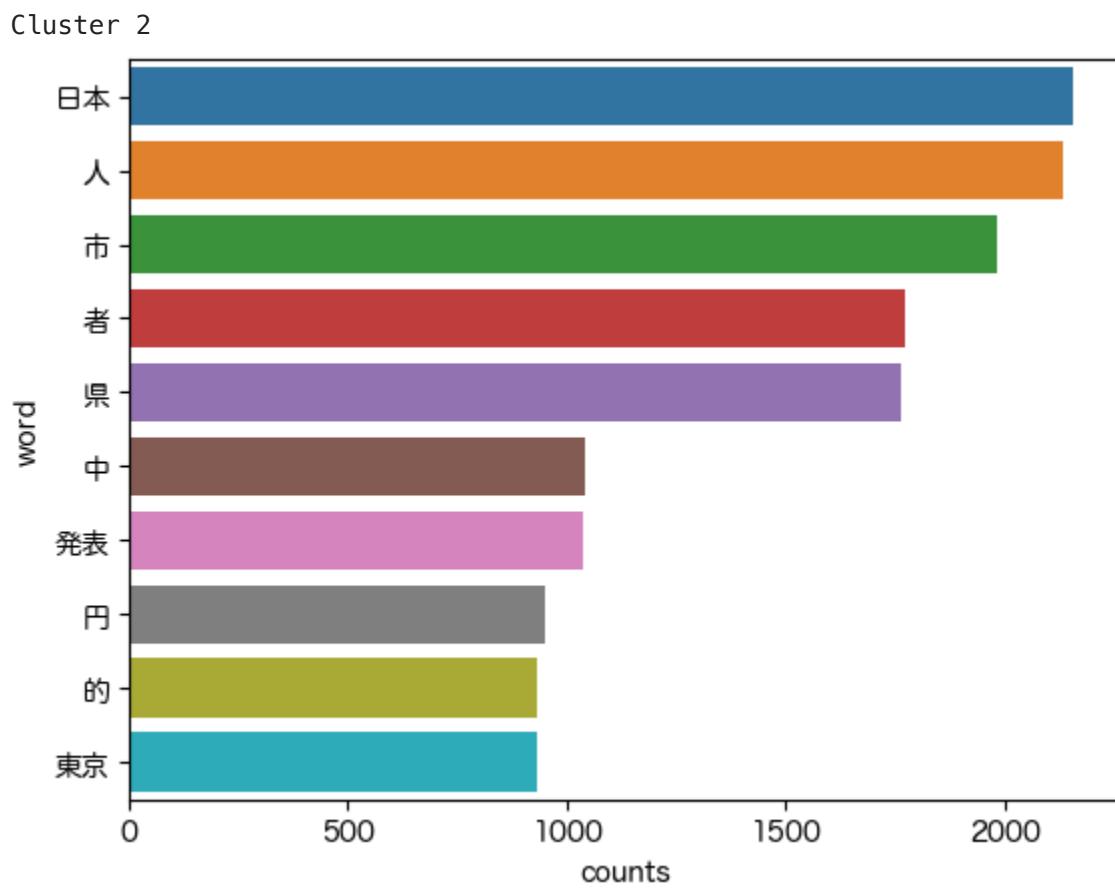
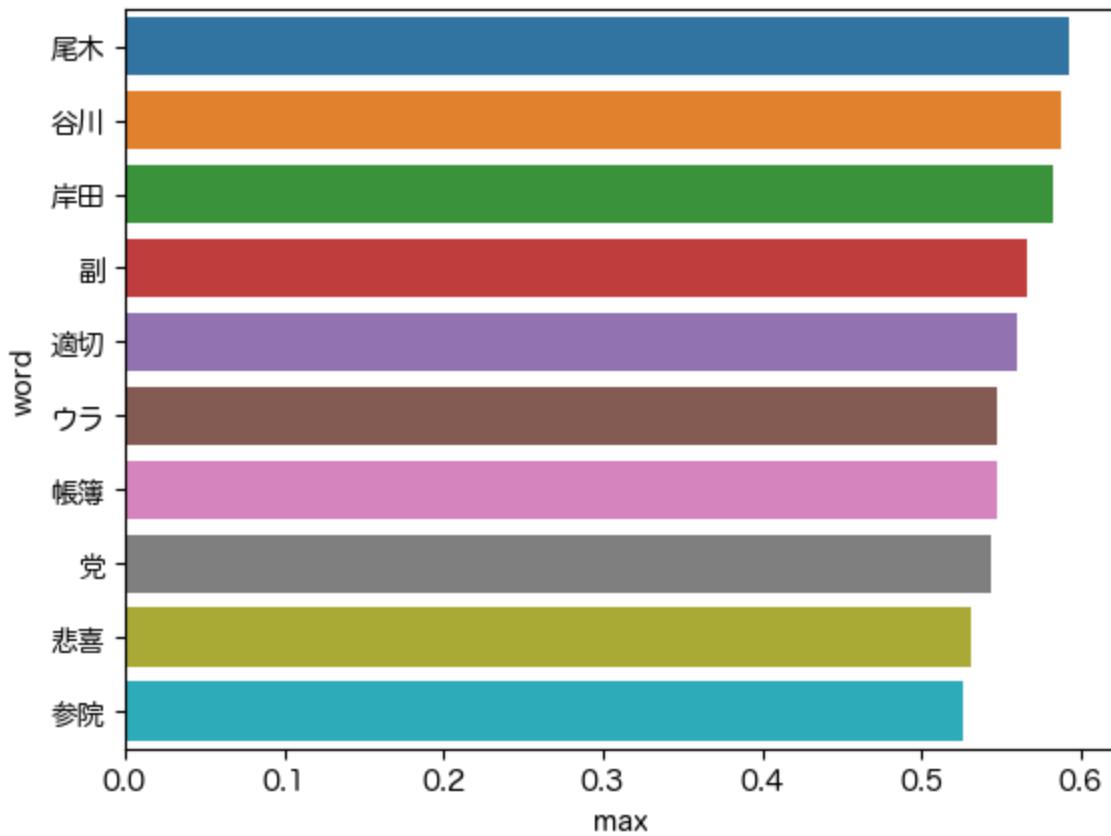
Cluster 0

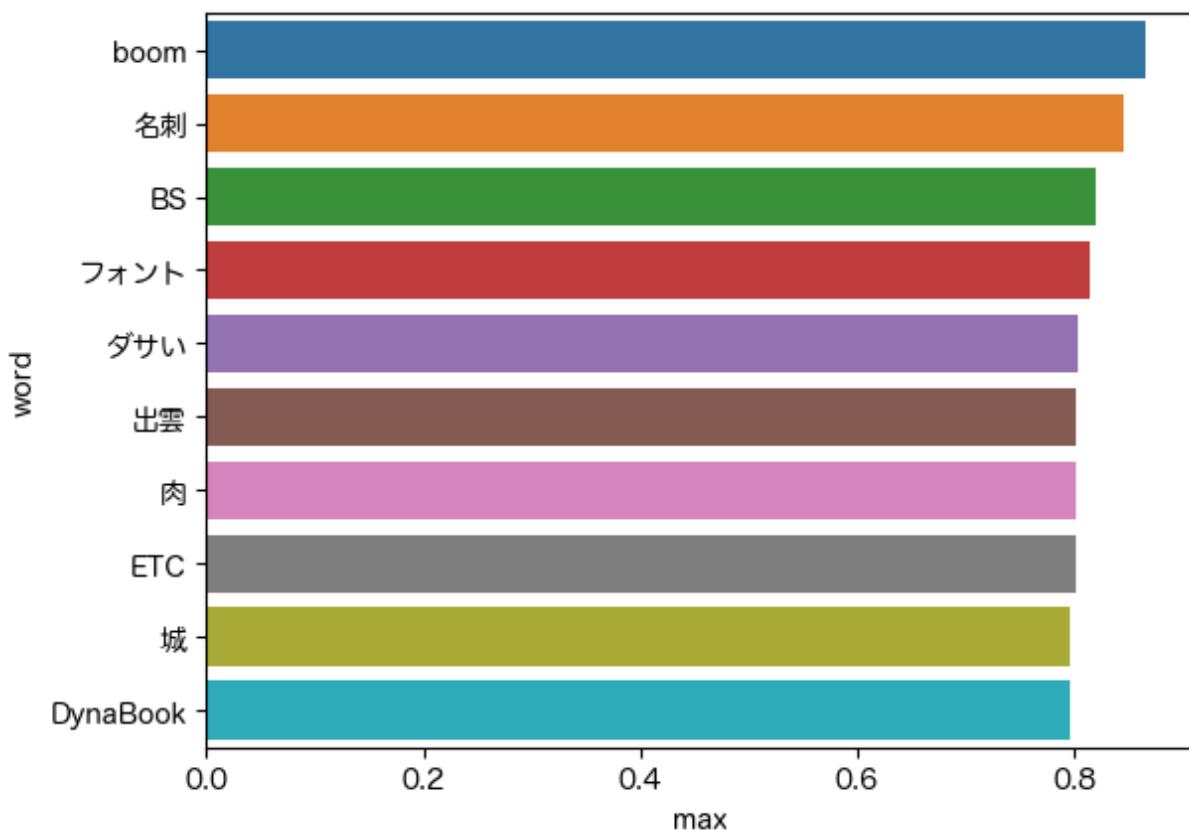




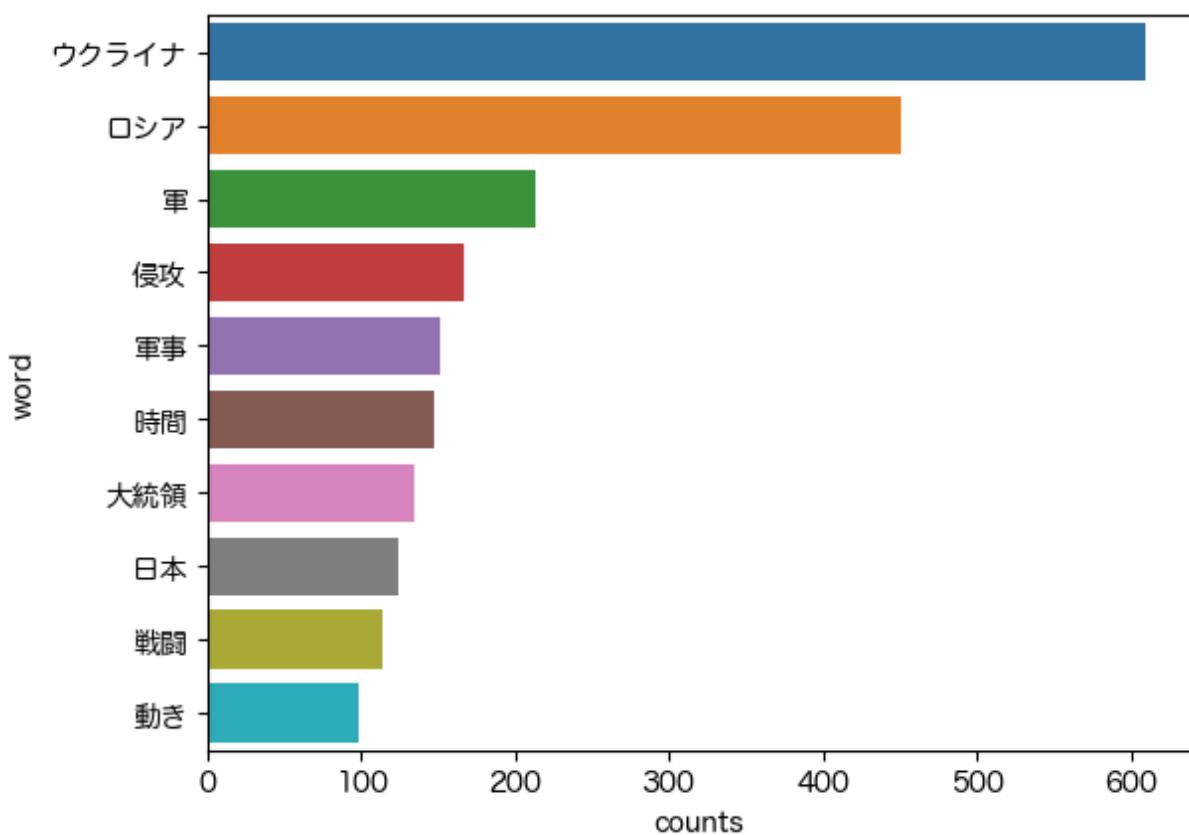
Cluster 1

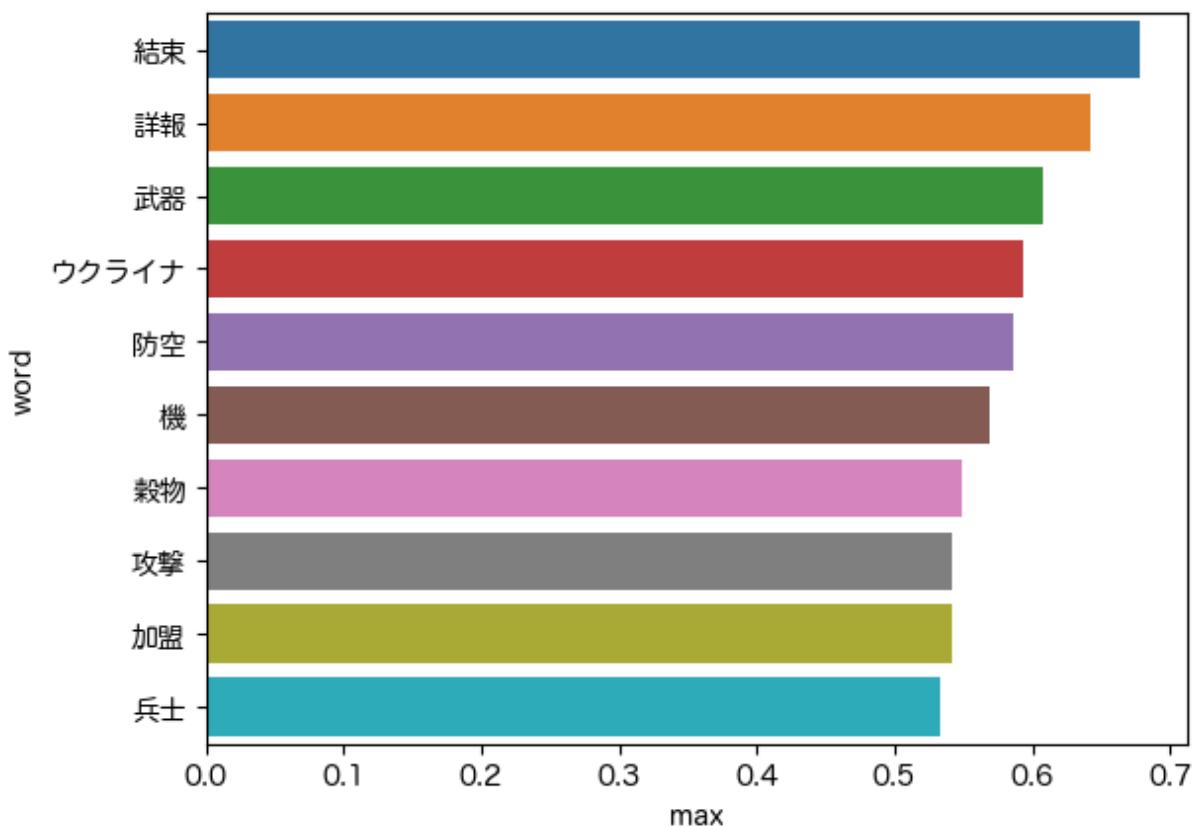




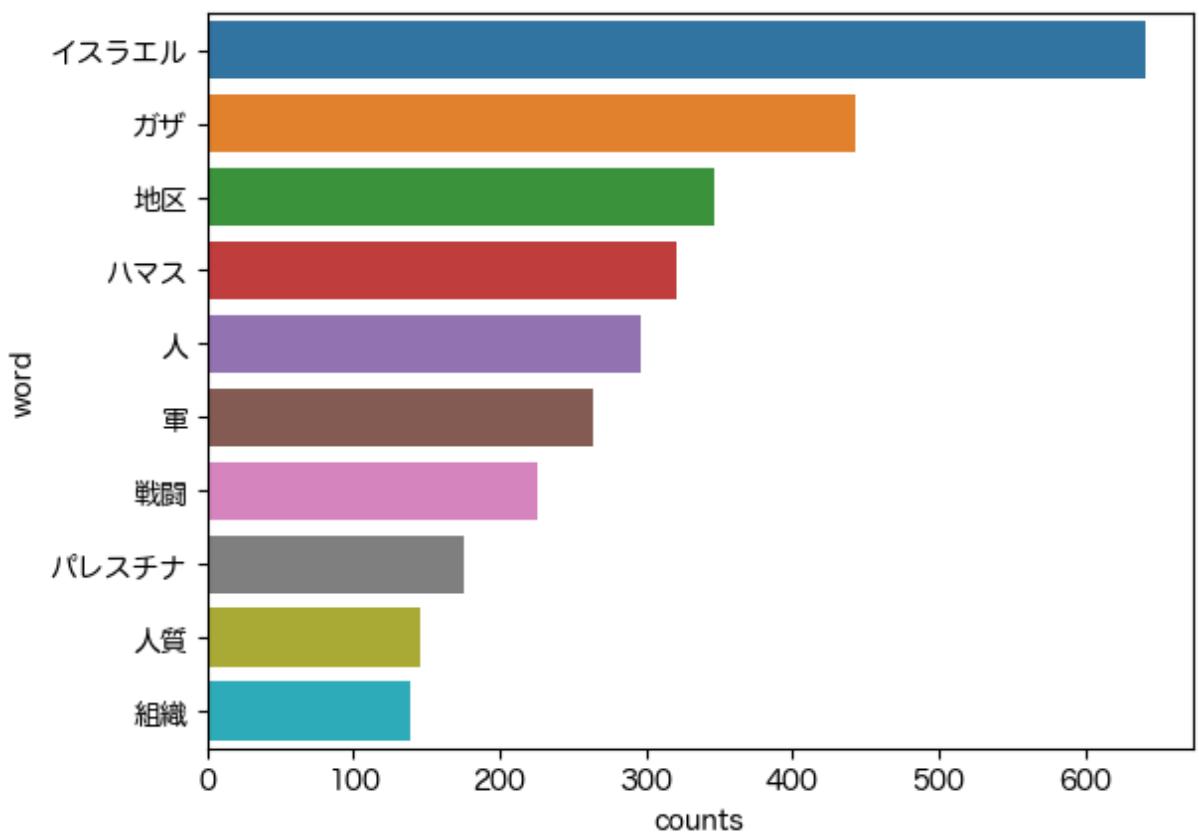


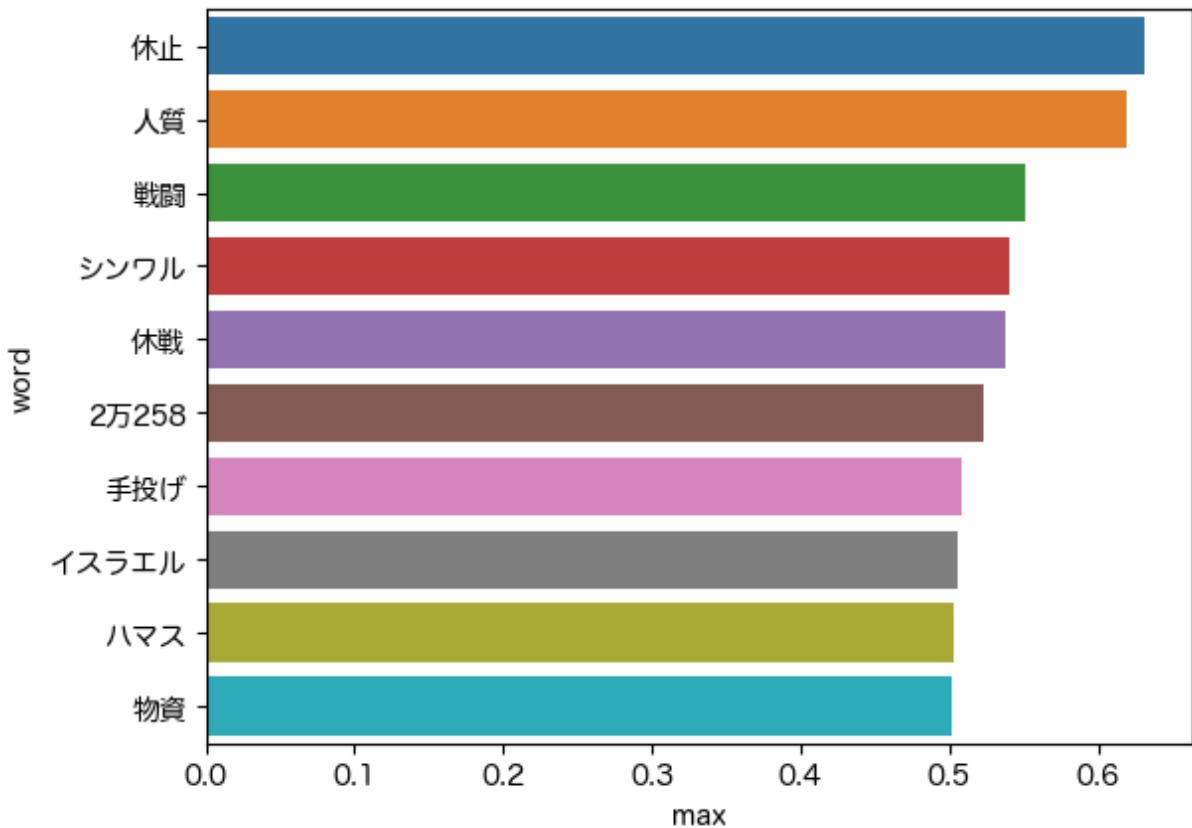
Cluster 3





Cluster 4





CPU times: user 39.4 s, sys: 2.57 s, total: 41.9 s
Wall time: 35.6 s

1.1.2 実行結果の考察

前処理

当初、ストップワードをピリオドのみとしていたが、単語の出現頻度を見ると全角の数字や記号の他、年月日時分といった時間に関する単語が共通して見られた為、これらの除去を行う様に変更した。

```
# 処理結果に多く含まれる全角数字、括弧、年、月、日、時等を stop_words
に含める
stop_words = ['.', '年', '月', '日', '時', '分',
              '1', '2', '3', '4', '5', '6', '7', '8',
              '9', '0',
              '（', '）', '?', '%', '（', '）', '%', '?', '/',
              '!', '！', '…', '=']
```

クラスタ数

Bag of Words (以下BoW)、TF-IDF それぞれの手法を用いてテキストのベクトル化を行い、それに対してエルボー法、シルエット解析によるクラスタ数の推定を行った。

エルボー解析の結果、BoWではクラスタ数 $N = 5, 6, 16$ の時に大きな傾きの変化が見られたが、TF-IDFでは顕著な変化は見られなかった。

シルエット解析の結果、BoWでは $N = 2, 3, 4, 5$ の時にシルエット係数 >0.08 となり、相対的に大

きな値となったが、小数点以下2桁の差であり極端に大きな差は表われなかった。TF-IDFのシルエット係数は更に小数点3桁での差となり、大きな差は見られなかった。

上記の結果から、クラスタ数 $N = 5$ を今回のクラスタリングに用いる事とした。

実行結果

実行結果の単語出現頻度とTF-IDFの上位に出てくる単語を見ると、各クラスタ毎に下記の様な特徴がある。

クラスタ番号	出現頻度上位の特徴	TF-IDF上位の特徴	推定される分類
0	国内地名、円等	統一性無し	国内全般
1	政治関連用語	政治家氏名、帳簿、ウラ等	国内政治
2	イスラエル、ガザ、ウクライナ、ロシア等	ウクライナ、人質、休戦等	国際全般
3	スポーツ関連用語	スポーツ選手氏名、スポーツ名	スポーツ
4	犯罪、警察、死亡等	地名、側溝、溺れる等	国内犯罪・事件

考察

フィードデータの情報源としては、3サイト22カテゴリを設定したがその中には国内、ローカル、国際の様にカバーする範囲の大きなフィードが含まれている。その為、クラスタ数の推定をする場合にも、情報源毎に明確に特徴が別れる様な形とはならず、複数の情報源に跨ってクラスタとして認識される結果になった。

クラスタ毎に特に特徴として表われたのは連日報道があった国内政治関連(クラスタ1)、国際関連(クラスタ2)であり、前者は「帳簿」「派閥」等、後者は「ウクライナ」「ロシア」「ガザ」「イスラエル」等特徴的な一般名詞、固有名詞が頻出した事からクラスタとして認識された。

また、スポーツ(クラスタ3)についても、出現頻度ではスポーツの一般名詞が、TF-IDFはスポーツ選手名等が上位にあり、クラスタとして認識された理由が明確である。国内犯罪・事件(クラスタ4)についても同様の傾向にある。

一方、国内全般と推定されるクラスタ0は出現頻度としては「日本」「県」「市」「円」等が上位にある一方で、TF-IDFの上位にある単語には統一性がない。国内政治関連(クラスタ1)、スポーツ(クラスタ3)、国内犯罪・事件(クラスタ4)に入らない、国内のニュースがクラスタとして認識されていると考えられる。

課題 1.2 分類

1.2.1 プログラムと実行結果

教師データの作成

URLに基づいた教師ラベルの設定を下記とする：

label 0 (国内政治、経済)

- cat4
- cat5
- politics
- business

label 1 (国際全般)

- cat6
- international
- world

label 2 (スポーツ)

- cat7
- sports

label 3 (エンターテインメント)

- cat2
- entertainment

label 4 (その他国内)

- 上記以外

```
In [8]: label_0 = ['cat4', 'cat5', 'politics', 'business']
label_1 = ['cat6', 'international', 'world']
label_2 = ['cat7', 'sports']
label_3 = ['cat2', 'entertainment']

# label_0 の単語が url の中に含まれている : 0
# label_1 の単語が url の中に含まれている : 1
# label_2 の単語が url の中に含まれている : 2
# label_3 の単語が url の中に含まれている : 3
# それ以外 : 4

def get_label(url):
    url = url.lower()
    # map: label_0 の単語それぞれについて、url の中に含まれているか真偽を返す
    # any: mapの結果について論理和をとる
    if any(map(lambda x: x in url, label_0)):
        return 0
    if any(map(lambda x: x in url, label_1)):
        return 1
```

```

if any(map(lambda x: x in url, label_2)):
    return 2
if any(map(lambda x: x in url, label_3)):
    return 3
else:
    return 4

# df['url'] について get_label を適用した結果を df['label'] として追加
df['label'] = df['url'].map(lambda x: get_label(x))
# 各ラベルの数を確認
df['label'].value_counts()

```

Out[8]:

label	count
4	5568
0	2225
1	1951
2	1378
3	1030

Name: count, dtype: int64

分類モデルの作成とテスト

注意事項:

下記のコードでは scikit-learn の FutureWarning が多数存在するが、モジュール内でのエラーであり JupyterNotebook のプログラム内で

```
warnings.filterwarnings('ignore', category=FutureWarning)
```

を記述しても抑制できなかった為、PDF上エラーが連続するページは削除している。

共通部

In [9]:

```

%%time
from sklearn.metrics import accuracy_score, precision_score, recall_score, c
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt

# 説明変数、目的変数
vector = vector_tfidf
X = vector.toarray()
Y = df.label

# データセットを訓練セットとテストセットに分割
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.5, ran

```

CPU times: user 655 ms, sys: 628 ms, total: 1.28 s
Wall time: 1.04 s

ナイーブベイズ分類器

In [10]:

```
%%time
```

```

# ナイーブベイズのパラメータグリッドの設定

param_range_nb = [0.01, 0.1, 1.0, 10.0]
grid_nb = {'alpha': param_range_nb}

# グリッドサーチの作成、実行
gs = GridSearchCV(MultinomialNB(), param_grid=grid_nb, scoring='roc_auc_ovr')
gs.fit(X_train, Y_train)

# 最適なパラメータとスコアの表示
print("グリッドサーチ結果")
print("\t最適なパラメータ (alpha):", gs.best_params_['alpha'])
print("\t最適なパラメータでのROC AUCスコア: {:.3f}\n".format(gs.best_score_))

print("\tROC AUCスコア一覧")
cv_results = gs.cv_results_
for i in range(len(cv_results['params'])):
    alpha = cv_results['params'][i]['alpha']
    score = cv_results['mean_test_score'][i]
    print("\t\talpha = {}: ROC AUC = {:.3f}".format(alpha, score))

# 最適なパラメータでモデルを再学習
model_nb = MultinomialNB(alpha=gs.best_params_['alpha'])
model_nb.fit(X_train, Y_train)

# 訓練セットとテストセットでの精度を比較
accuracy_train = accuracy_score(Y_train, model_nb.predict(X_train))
accuracy_test = accuracy_score(Y_test, model_nb.predict(X_test))
print('訓練データに対する分類精度: {:.3f}'.format(accuracy_train))
print('テストデータに対する分類精度: {:.3f}'.format(accuracy_test))

# テストデータに対する予測確率
Y_proba = model_nb.predict_proba(X_test)

# マルチクラスのためのROC曲線の計算
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(Y.unique())):
    fpr[i], tpr[i], _ = roc_curve(Y_test, Y_proba[:, i], pos_label=i)
    roc_auc[i] = auc(fpr[i], tpr[i])

# 各クラスに対するROC曲線をプロット
for i in range(len(Y.unique())):
    plt.plot(fpr[i], tpr[i], label='Class {} (AUC={:.3f})'.format(i, roc_auc[i]))

plt.xlabel('偽陽性率 (FP率)')
plt.ylabel('真陽性率 (TP率)')
plt.title('テストデータのROC曲線')
plt.legend()
plt.show()

```

```
/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/utils/validation.py:605: FutureWarning: is_sparse is deprecated and will be removed in a future version. Check `isinstance(dtype, pd.SparseDtype)` instead.  
    if is_sparse(pd_dtype):  
/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/utils/validation.py:614: FutureWarning: is_sparse is deprecated and will be removed in a future version. Check `isinstance(dtype, pd.SparseDtype)` instead.  
    if is_sparse(pd_dtype) or not is_extension_array_dtype(pd_dtype):
```

グリッドサーチ結果

最適なパラメータ (alpha): 0.01
最適なパラメータでのROC AUCスコア: 0.919

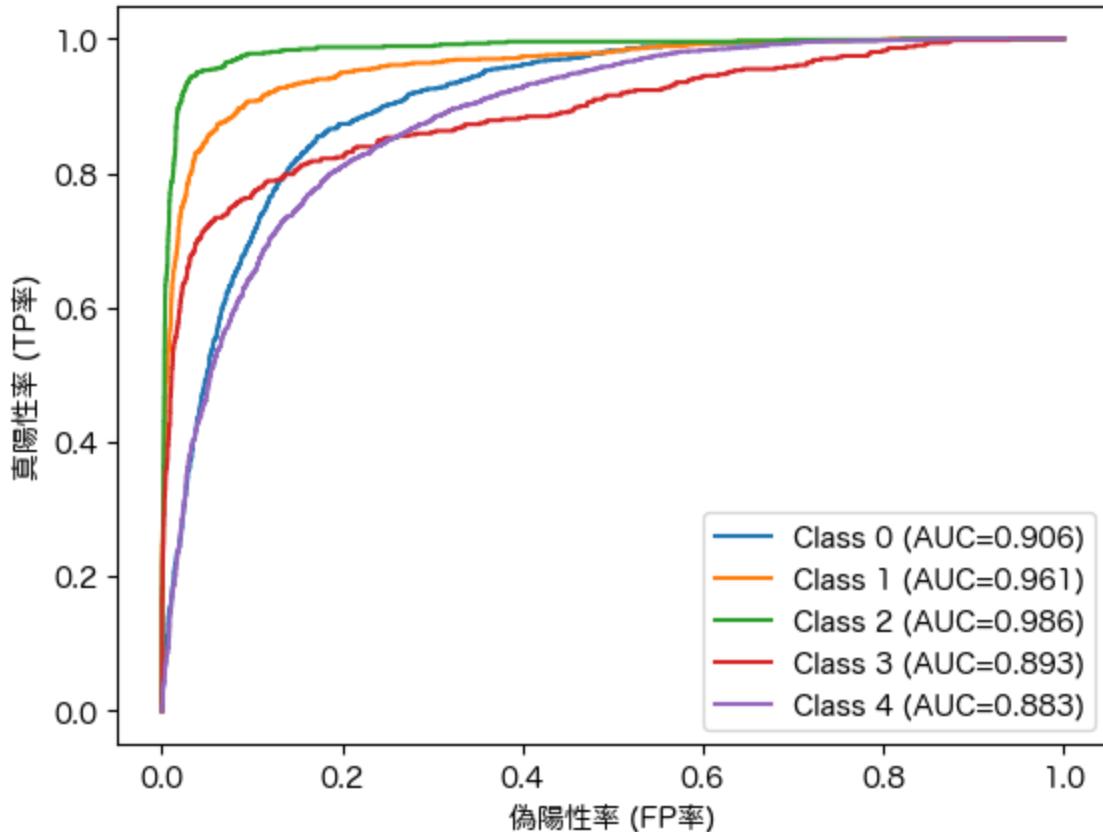
ROC AUCスコア一覧

alpha = 0.01: ROC AUC = 0.919
alpha = 0.1: ROC AUC = 0.919
alpha = 1.0: ROC AUC = 0.896
alpha = 10.0: ROC AUC = 0.872

訓練データに対する分類精度: 0.936

テストデータに対する分類精度: 0.762

テストデータのROC曲線



CPU times: user 6.48 s, sys: 2.36 s, total: 8.84 s
Wall time: 11.7 s

決定木分類器

In [11]: `%time`

```
# param_range_dt = list(range(1, 16, 1))  
param_range_dt = list(range(1, 32, 2))  
grid_dt = [{  
    'max_depth': param_range_dt}]
```

```

# グリッドサーチの作成、実行
gs = GridSearchCV(DecisionTreeClassifier(), param_grid=grid_dt, scoring='roc_auc')
gs.fit(X_train, Y_train)

# 最適なパラメータとスコアの表示
print("グリッドサーチ結果")
print("\t最適なパラメータ:", gs.best_params_)
print("\t最適なパラメータでのROC AUCスコア: {:.3f}".format(gs.best_score_))

print("\t\tROC AUCスコア一覧")
cv_results = gs.cv_results_
for i in range(len(cv_results['params'])):
    depth = cv_results['params'][i]['max_depth']
    score = cv_results['mean_test_score'][i]
    print("\t\tmax_depth = {}: ROC AUC = {:.3f}".format(depth, score))

# 最適なパラメータでモデルを再学習
model_dt = DecisionTreeClassifier(**gs.best_params_)
model_dt.fit(X_train, Y_train)

# 訓練セットとテストセットでの精度を比較
accuracy_train = accuracy_score(Y_train, model_dt.predict(X_train))
accuracy_test = accuracy_score(Y_test, model_dt.predict(X_test))
print('訓練データに対する分類精度: {:.3f}'.format(accuracy_train))
print('テストデータに対する分類精度: {:.3f}'.format(accuracy_test))

# テストデータに対する予測確率
Y_proba = model_dt.predict_proba(X_test)

# マルチクラスのためのROC曲線の計算
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(Y.unique())):
    fpr[i], tpr[i], _ = roc_curve(Y_test, Y_proba[:, i], pos_label=i)
    roc_auc[i] = auc(fpr[i], tpr[i])

# 各クラスに対するROC曲線をプロット
for i in range(len(Y.unique())):
    plt.plot(fpr[i], tpr[i], label='Class {} (AUC={:.3f})'.format(i, roc_auc[i]))

plt.xlabel('偽陽性率 (FP率)')
plt.ylabel('真陽性率 (TP率)')
plt.title('テストデータのROC曲線')
plt.legend()
plt.show()

```

```
/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/utils/validation.py:605: FutureWarning: is_sparse is deprecated and will be removed in a future version. Check `isinstance(dtype, pd.SparseDtype)` instead.  
    if is_sparse(pd_dtype):  
/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/utils/validation.py:614: FutureWarning: is_sparse is deprecated and will be removed in a future version. Check `isinstance(dtype, pd.SparseDtype)` instead.  
    if is_sparse(pd_dtype) or not is_extension_array_dtype(pd_dtype):  
/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/utils/validation.py:605: FutureWarning: is_sparse is deprecated and will be removed in a future version. Check `isinstance(dtype, pd.SparseDtype)` instead.  
    if is_sparse(pd_dtype):  
/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/utils/validation.py:614: FutureWarning: is_sparse is deprecated and will be removed in a future version. Check `isinstance(dtype, pd.SparseDtype)` instead.  
    if is_sparse(pd_dtype) or not is_extension_array_dtype(pd_dtype):  
/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/utils/validation.py:605: FutureWarning: is_sparse is deprecated and will be removed in a future version. Check `isinstance(dtype, pd.SparseDtype)` instead.  
    if is_sparse(pd_dtype):  
/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/utils/validation.py:614: FutureWarning: is_sparse is deprecated and will be removed in a future version. Check `isinstance(dtype, pd.SparseDtype)` instead.  
    if is_sparse(pd_dtype) or not is_extension_array_dtype(pd_dtype):  
/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/utils/validation.py:605: FutureWarning: is_sparse is deprecated and will be removed in a future version. Check `isinstance(dtype, pd.SparseDtype)` instead.  
    if is_sparse(pd_dtype):  
/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/utils/validation.py:614: FutureWarning: is_sparse is deprecated and will be removed in a future version. Check `isinstance(dtype, pd.SparseDtype)` instead.  
    if is_sparse(pd_dtype) or not is_extension_array_dtype(pd_dtype):
```

グリッドサーチ結果

最適なパラメータ: {'max_depth': 31}

最適なパラメータでのROC AUCスコア: 0.759

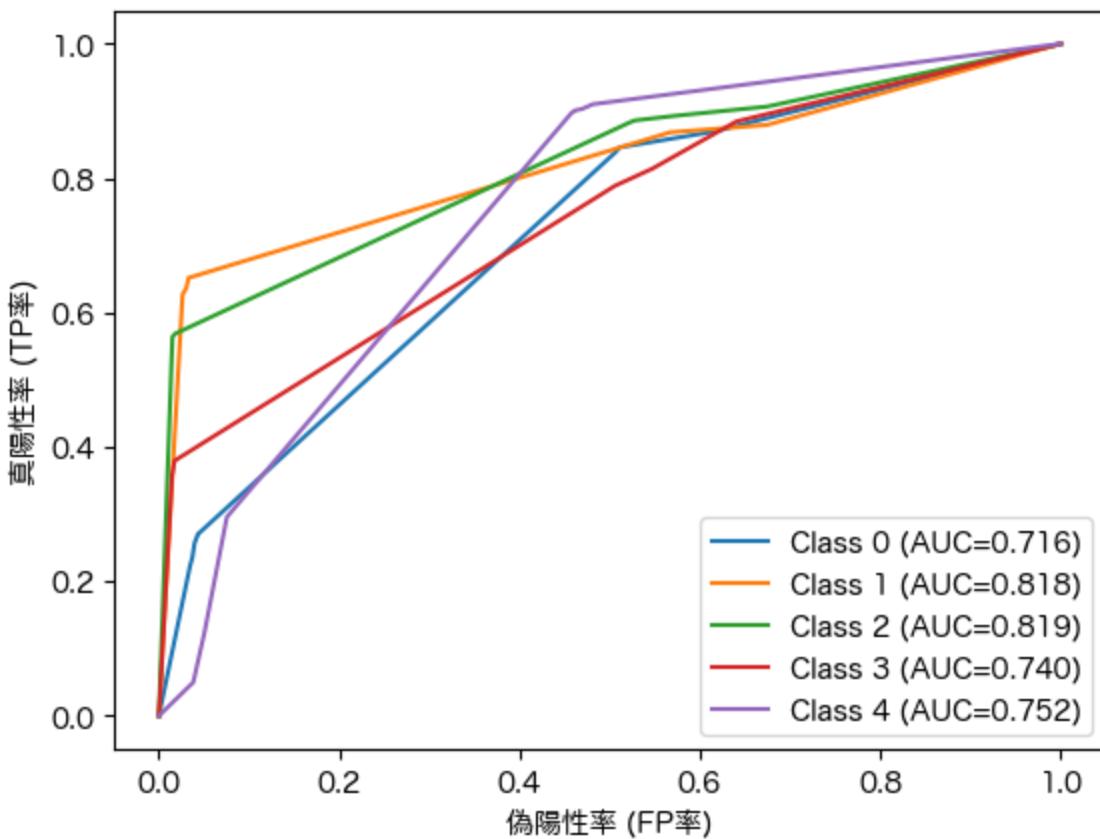
ROC AUCスコア一覧

```
max_depth = 1: ROC AUC = 0.545  
max_depth = 3: ROC AUC = 0.592  
max_depth = 5: ROC AUC = 0.630  
max_depth = 7: ROC AUC = 0.666  
max_depth = 9: ROC AUC = 0.691  
max_depth = 11: ROC AUC = 0.710  
max_depth = 13: ROC AUC = 0.728  
max_depth = 15: ROC AUC = 0.730  
max_depth = 17: ROC AUC = 0.735  
max_depth = 19: ROC AUC = 0.744  
max_depth = 21: ROC AUC = 0.748  
max_depth = 23: ROC AUC = 0.754  
max_depth = 25: ROC AUC = 0.751  
max_depth = 27: ROC AUC = 0.753  
max_depth = 29: ROC AUC = 0.757  
max_depth = 31: ROC AUC = 0.759
```

訓練データに対する分類精度: 0.777

テストデータに対する分類精度: 0.659

テストデータのROC曲線



CPU times: user 28 s, sys: 1.05 s, total: 29 s
Wall time: 5min 16s

ロジスティック回帰分類器

In [12]:

```
%%time

# ロジスティック回帰のパラメータグリッドの設定
# Cは正則化の強さ、小さい値ほど正則化が強い
param_range_lr = [0.01, 0.1, 1.0, 10.0]
grid_lr = {'C': param_range_lr}

# グリッドサーチの作成、実行
gs = GridSearchCV(LogisticRegression(max_iter=1000), param_grid=grid_lr, scoring='roc_auc')
gs.fit(X_train, Y_train)

# 最適なパラメータとスコアの表示
print("グリッドサーチ結果")
print("\t最適なパラメータ (C):", gs.best_params_)
print("\t最適なパラメータでのROC AUCスコア: {:.3f}".format(gs.best_score_))

print("\tROC AUCスコア一覧")
cv_results = gs.cv_results_
for i in range(len(cv_results['params'])):
    C = cv_results['params'][i]['C']
    score = cv_results['mean_test_score'][i]
    print("\t\tC = {}: ROC AUC = {:.3f}".format(C, score))

# 最適なパラメータでモデルを再学習
```

```
model_lr = LogisticRegression(C=gs.best_params_['C'], max_iter=1000)
model_lr.fit(X_train, Y_train)

# 訓練セットとテストセットでの精度を比較
accuracy_train = accuracy_score(Y_train, model_lr.predict(X_train))
accuracy_test = accuracy_score(Y_test, model_lr.predict(X_test))
print('訓練データに対する分類精度: {:.3f}'.format(accuracy_train))
print('テストデータに対する分類精度: {:.3f}'.format(accuracy_test))

# テストデータに対する予測確率
Y_proba = model_lr.predict_proba(X_test)

# マルチクラスのためのROC曲線の計算
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(Y.unique())):
    fpr[i], tpr[i], _ = roc_curve(Y_test, Y_proba[:, i], pos_label=i)
    roc_auc[i] = auc(fpr[i], tpr[i])

# 各クラスに対するROC曲線をプロット
for i in range(len(Y.unique())):
    plt.plot(fpr[i], tpr[i], label='Class {} (AUC={:.3f})'.format(i, roc_auc[i]))

plt.xlabel('偽陽性率 (FP率)')
plt.ylabel('真陽性率 (TP率)')
plt.title('テストデータのROC曲線')
plt.legend()
plt.show()
```

```
/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/utils/validation.py:605: FutureWarning: is_sparse is deprecated and will be removed in a future version. Check `isinstance(dtype, pd.SparseDtype)` instead.  
    if is_sparse(pd_dtype):  
/opt/homebrew/anaconda3/lib/python3.11/site-packages/sklearn/utils/validation.py:614: FutureWarning: is_sparse is deprecated and will be removed in a future version. Check `isinstance(dtype, pd.SparseDtype)` instead.  
    if is_sparse(pd_dtype) or not is_extension_array_dtype(pd_dtype):
```

グリッドサーチ結果

最適なパラメータ (C): {'C': 1.0}
最適なパラメータでのROC AUCスコア: 0.924

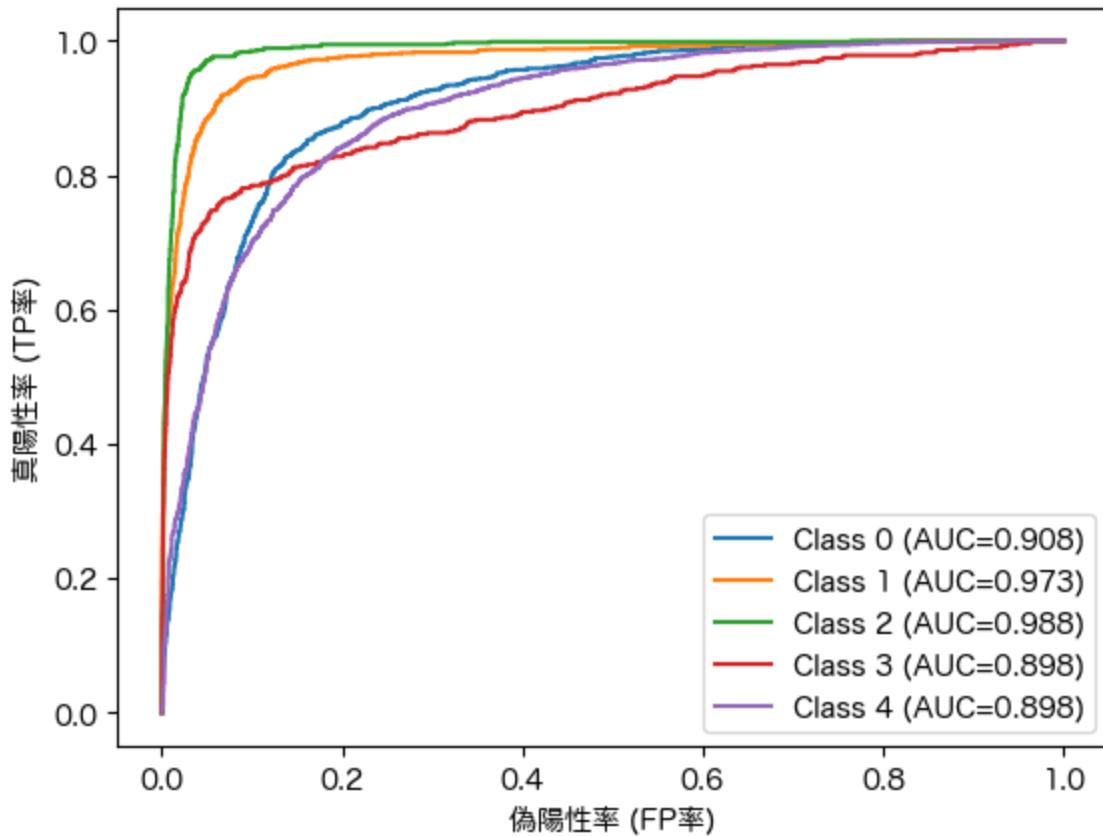
ROC AUCスコア一覧

C = 0.01: ROC AUC = 0.889
C = 0.1: ROC AUC = 0.905
C = 1.0: ROC AUC = 0.924
C = 10.0: ROC AUC = 0.922

訓練データに対する分類精度: 0.868

テストデータに対する分類精度: 0.734

テストデータのROC曲線



CPU times: user 5min 53s, sys: 12.5 s, total: 6min 6s

Wall time: 5min 54s

1.2.2 実行結果の考察

実行環境

下表の実行環境を利用した。

また、グリッドサーチ等並列度が指定可能な箇所については実メモリを越えて実行すると効率が低

下する為、Pythonインタプリタのメモリ占有量(*)を考慮して `n_jobs = 4` として実行時に実メモリ内に収まる様にした。

(*) 実行時、2.5GB～3.5GB程度を占有

項目	内容
ハードウェア	MacBook Air M2 2022 (メモリ16GB、CPU M2 8コア)
OS	macOS 14.2.1
ソフトウェア	<ul style="list-style-type: none">- Python: 3.11.7 (Anaconda3)- scikit-learn: 1.22 (Anaconda3)- seaborn: 0.12.2 (Anaconda3)- 他関連ライブラリ: Anaconda3 の 2023/12/28 時点の最新版

分類アルゴリズム

今回の3つの分類アルゴリズムとして、

- ナイーブベイズ
- 決定木
- ロジスティック回帰

を採用した。ランダムフォレスト、カーネルSVMについても検討したが本実行環境では数十分から数時間以上と実行時間が非常に長く、試行を繰り返す事が困難であった為除外した。

試験データ

元のフィードデータは課題1.1日本語テキストのクラスタリングと同じ、3サイト22カテゴリを設定した。また、実際に収集したフィードデータは12,153行となる。

今回の分類では、ホールドアウト法を用いて 1:1 の比率で教師データとテストデータを分割し、3つの分類器では全て同じ教師データ、テストデータを利用している。

実行結果比較

アルゴリズム	訓練データに対する分類精度	テストデータに対する分類精度	CPU時間/実時間(*)
ナイーブベイズ	0.936	0.762	0:06/0:12
決定木	0.777	0.659	0:28/5:16
ロジスティック回帰	0.868	0.734	5:53/5:54

(*) JupyterNotebook のマジックワード `%%time` で表示される CPU times, Wall timeを採用。実行時の環境に左右される為、厳密な比較ではなく参考とする。

ナイーブベイズ分類器に関する考察

訓練データに対する分類精度、テストデータに対する分類精度共に3つのアルゴリズム中最も良い値となった。一方、訓練データに対する分類精度が0.936に対し、テストデータに対する分類精度は0.762と過学習を起こしている事が分かる。

特筆すべきは実行時間であり、CPU時間、実時間共に秒のオーダーであり、他のアルゴリズムに較べて 10^{-1} から 10^{-2} 程度の差が存在する。これはナイーブベイズの特徴に対する統計(平均、標準偏差等)が他の特徴とは独立して計算でき、並列度の高い計算が出来る事に由来すると考えられる。

決定木分類器に関する考察

グリッドサーチでは深さが最大時(31)の際に最も良い数値となっているが、実行時間を考慮して探索を打ち切りとし、それより深い分割を行わない事とした。

訓練データに対する分類精度、テストデータに対する分類精度共に3つのアルゴリズム中最も悪い値となった。テストデータに対する各分類対象毎のROC曲線を見ても、凸部分が直線に近付いており、分類の精度が悪いことが分かる。更に、訓練データに対する分類精度が0.777に対し、テストデータに対する分類精度は0.659と過学習を起こしているのはナイーブベイズ分類器と同様である。

実行時間についてもCPU使用時間は0:28と比較的少ないが、実時間が5:16と非常に長くなっている。これは、決定木を作成する際に、各分割が次の分割の前提になるという逐次的な手順が問題となっていると考えられる。

ロジスティック回帰に関する考察

訓練データに対する分類精度、テストデータに対する分類精度共に3つのアルゴリズム中2番目に良い値となっており、テストデータに対する分類精度ではナイーブベイズに近い値となっている。

実行時間について見てみると、CPU使用時間は5:53に対し、実時間が5:54とほぼ等しい。CPU時間が長い理由は説明変数(プログラム中のX, TF-IDFでベクトル化したテキスト)の次元が非常に多い事に由来すると考えられる。

まとめ

今回用いた3つのアルゴリズムではナイーブベイズ分類器が訓練時間と分類精度のバランスが非常に良い事が分かり、PCやローカルサーバ上で実行する様な迷惑メールフィルタにも利用されてきた理由が理解できた。

一方、今回の実行環境では実行時間が長い為に比較できなかったが、訓練時間を十分に取れる場合や計算機資源が十分に用意できる様なケースでは、カーネルSVMやランダムフォレストについても視野に入ると考えられる。