In [2]:

```
%%HTML
<link rel="stylesheet" type="text/css" href="static/style.css">
```

Jubaclassifier Hands-on

- 主に書籍の内容について、コードを実行しながら説明
  - データは書籍とは違うものを使います。

Agenda

1. Jubatusとは
2. 分類とは
3. 実際に動かしてみる

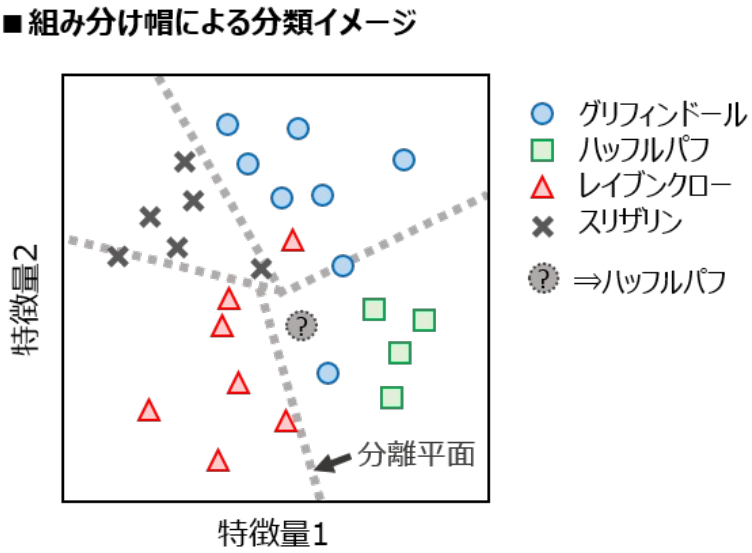
分類とは

分類(Classifier)とは、与えられたデータに対して**適切なクラスを推定する処理**を指します。

- スパムメール判定
- 金融における顧客のデフォルト(不払い)予測
- etc ...

分類のイメージ

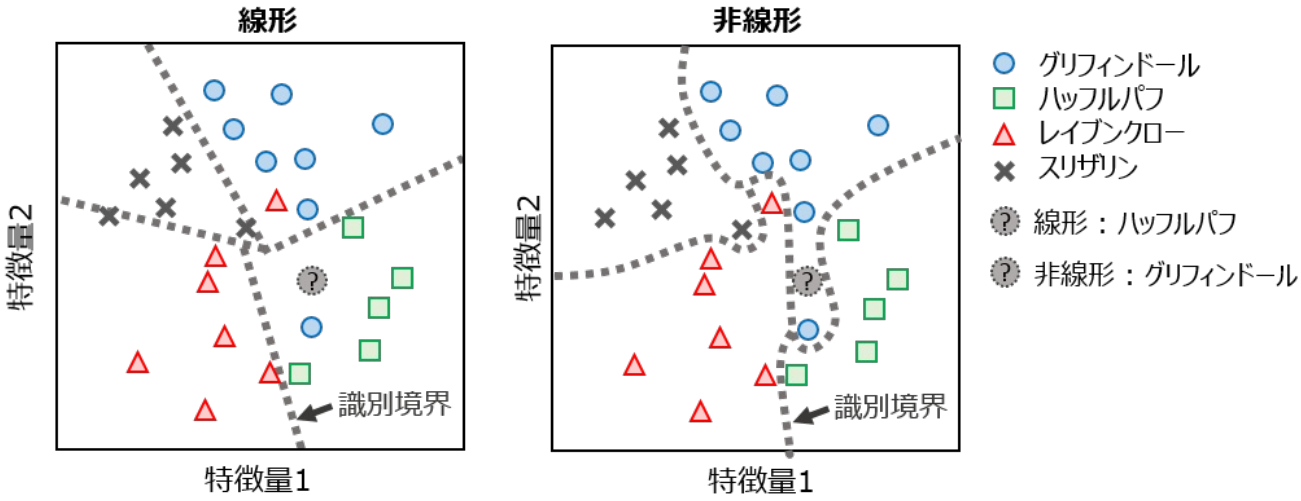
Harry Potterの組み分け帽を例にとる



線形分類器 VS 非線形分類器

- 線形分類器：空間を**直線/平面**で区切っていく
  - シンプルで早いが、どうしても分けられない場合も出てくる
- 非線形分類器：空間を**曲線/曲面**で区切っていく
  - 精度はいいが、**過学習**の恐れあり

■線形分類と非線形分類



実際に動かしてみる

環境

VMで配った環境に、以下のものが入っているか確認してください。

- Jubatus
- Python(3.x)
  - Jubatus
  - sklearn
  - numpy
  - pandas
  - matplotlib

配布物

下記のものがフォルダ内にあるか確認してください。

- スクリプト(Python Notebookで配布します)
  - `classifier.ipynb`
- データ(data/配下)
  - `default_train.csv`
  - `default_test.csv`
- コンフィグファイル(config/配下)
  - `linear.json` (AROW)
  - `nonlinear.json` (NN/Euclid LSH)

今回扱うデータ

default of credit card clients Data Set

- 台湾の顧客に対し支払いの**不払い(デフォルト)があったかどうか**を集めたデータ
- 年齢や性別など、**23個の特徴量**で構成されている

※本とは違うデータを用いています。前処理等で、コードが少し異なる箇所もありますが、流れは本のままです。

- DEFAULTに支払いが履行されたかどうかの情報が含まれている
- DEFAULTを分類器の答え合わせ（正解ラベル)に用いる

列番	特徴量	概要	特徴量の型
X1	Amount of the given credit	信用貸付額	整数
X2	Gender	性別	カテゴリ変数
X3	Education	学歴	カテゴリ変数
X4	Martial status	結婚歴	カテゴリ変数
X5	Age	年齢	整数
X6-X11	History of past payment	過去の支払い。きちんと支払ったかどうか(※1)	整数
X12-X17	Amount of bill statememt	過去の請求額(※1)	整数
X18-X23	Amount of previous payment	過去の支払額(※1)	整数
Y	DEFAULT	デフォルトしたかどうか	カテゴリ変数（正解ラベル）

※1 2005年4月から2005年9月までの6ヶ月分のデータ

Pandasを使ってデータを読み込んでみる

In [ ]:

```
import pandas as pd
df = pd.read_csv("data/default_train.csv") #データの読み込み
print(df.head())
```

Jubatusを起動

- ターミナルに戻り、下記コマンドを入力します
- 線形分類器(AROW)を使ってみます

`$ jubaclassifier -f config/linear.json&`

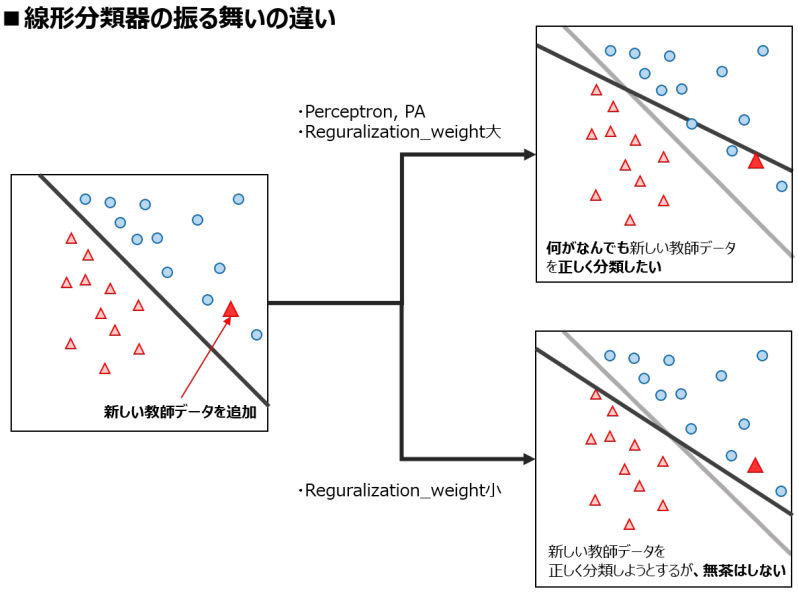
Jubatus分類器に入っているアルゴリズム

Jubatusでは、線形分類器と非線形分類器にそれぞれ下表のアルゴリズムを用意しています。

<table>
<tr><th></th><th colspan="2">アルゴリズム</th></tr>
<tr><td rowspan="6">線形</td><td colspan="2">Perceptron</td></tr>
<tr><td colspan="2">PA</td></tr>
<tr><td colspan="2">PA-I, PA-II</td></tr>
<tr><td colspan="2">CW</td></tr>
<tr><td colspan="2">AROW</td></tr>
<tr><td colspan="2">NHERD</td></tr>
<tr><td rowspan="5">非線形</td><td rowspan="5">k-NN（k近傍法）
それぞれ近傍点の距離の
算出方法が異なる。</td><td>Euclidean</td></tr>
<tr><td>Cosine</td></tr>
<tr><td>euclidLSH</td></tr>
<tr><td>LSH</td></tr>
<tr><td>Minhash</td></tr>
</table>

Regularization Weghtとは

`Regularization_weight`が大⇒新しく入ってきたデータを何が何でも分類する



データの読み込み

- **pandas**を用いてデータを読み込む
- 特徴量ベクトル, ラベル, 特徴量の名前をcsvから出力

In [ ]:

```
def read_dataset(path):
    df = pd.read_csv(path)
    labels = df['Y'].tolist()
    df = df.drop('Y', axis=1)
    features = df.as_matrix().astype(float)
    columns = df.columns.tolist()
    return features, labels, columns

features_train, labels_train, columns = read_dataset("data/default_train.csv")
```

In [ ]:

```
print("columns : {}".format(columns))
print("features : {}".format(features_train))
```

学習用データをDatum形式に変換

- Jubatusにデータを投げるために、pandasで読み込んだデータを**Datum形式**にする必要がある
- データと正解ラベルを1セットにしてデータを保持

[(正解ラベル, Datum(key, value))]

****

In [ ]:

```
from jubatus.common import Datum
features_train, labels_train, columns = read_dataset('data/default_train.csv')
train_data = []
for x, y in zip(features_train, labels_train):
    d = Datum({key: float(value) for key, value in zip(columns, x)})
    train_data.append([str(y), d])
```

作成したデータをJubatusサーバに投入

In [ ]:

```
from jubatus.classifier.client import Classifier
client = Classifier('127.0.0.1', 9199, '') # Jubatusサーバのホストとポートを指定する
client.clear() #過去の学習結果を一度初期化する(任意)
client.train(train_data) # 学習を実行
```

モデルを用いて分類/評価をする

- 作った学習モデルを使って、実際に分類をしてみる
- data/default_test.csvを用いる

※実際の評価の際は、Cross Validationなどを用いて評価を行うと良いでしょう。

In [ ]:

```
# テスト用Datumリストを作る
features_test, labels_test, columns = read_dataset('data/default_test.csv')
test_data = []
for x, y in zip(features_test, labels_test):
    d = Datum({key: float(value) for key, value in zip(columns, x)})
    test_data.append(d)
```

テストをする

実際にJubaclassifierにデータを投入し、返ってくるラベルを見る

In [ ]:

```
# テストをする
results = client.classify(test_data)
```

JubatusがdefaultにたいしてYes/No どちらが可能性が高いかをスコアリングしてくれている

In [ ]:

```
print(results[0])
```

結果の分析

- 先ほどのスコアの大きい方を分類結果として返す`get_most_likely`関数を作る

In [ ]:

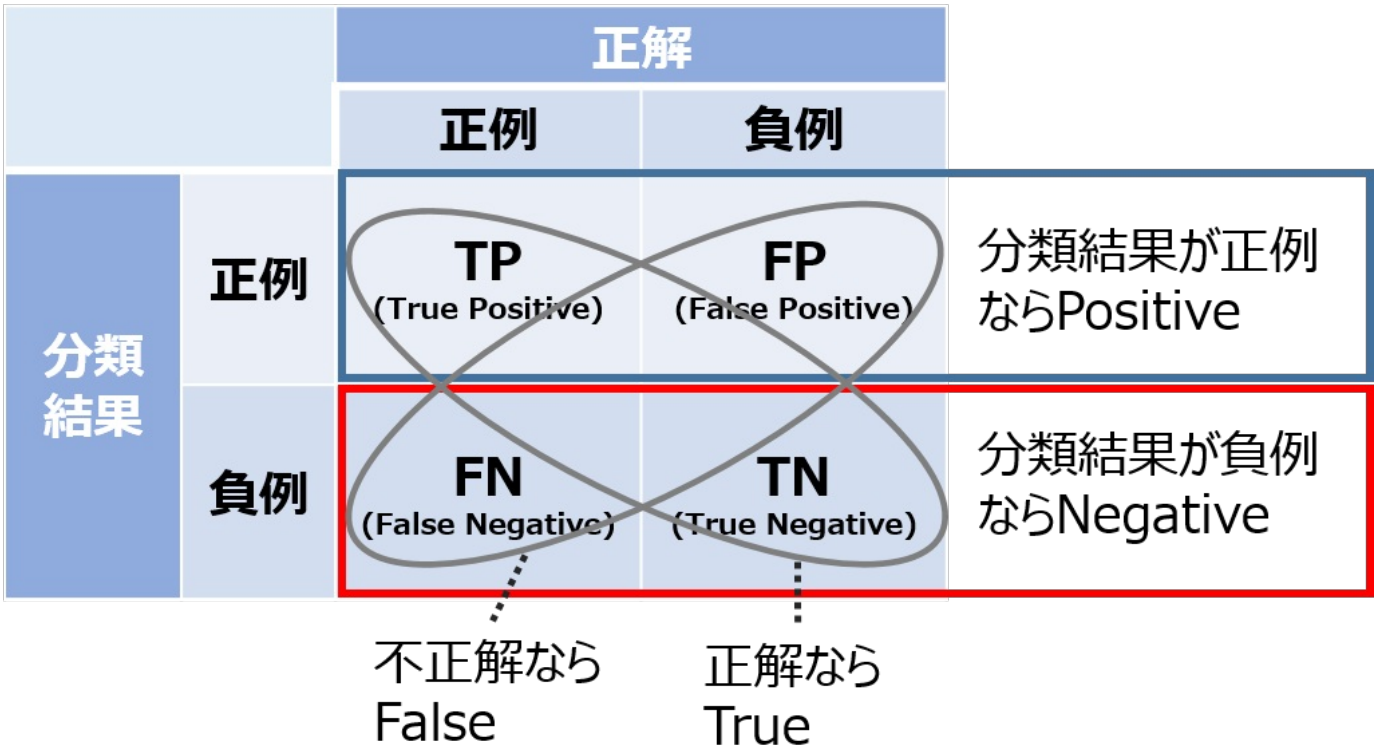
```
# 結果を分析する(スコアの大きい方のラベルを選ぶだけ)
def get_most_likely(result):
    return max(result, key = lambda x: x.score).label
```

分類結果の評価指標

- 分類結果は、
  正例か負例かという観点と、
  答えが合ってたかどうかという観点で4種類に分けられる
- 実際のデータが...
  - Defaultしている：正例(Positive)
  - Defaultしていない：負例(Negative)
- 推定結果が...
  - 正しい：True
  - 間違ってる：False

混合行列(Confusion Matrix)

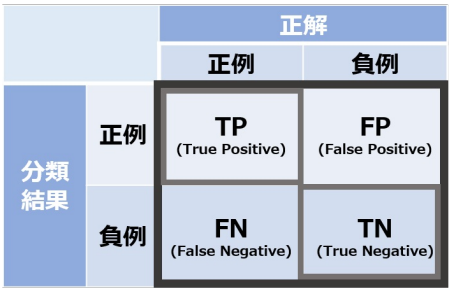
この4種類を一つの表にまとめたもの



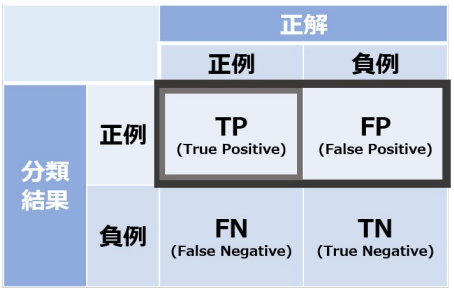
精度の評価指標

一口に『精度』と言っても、実はいろいろある

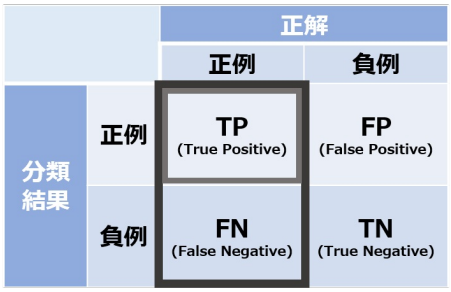
Accuracy



Precision



Recall



目的に応じて、見るべき指標は変わる

In [ ]:

```
def analyze_results(labels, results, pos_label="1", neg_label="0"):
    tp, fp, tn, fn = 0, 0, 0, 0
    for label, result in zip(labels, results):
        estimated = get_most_likely(result)
        label = str(label)
        estimated = str(estimated)
        if label == pos_label and label == estimated:
            tp += 1
        elif label == pos_label and label != estimated:
            fn += 1
        elif labels != pos_label and label == estimated:
            tn += 1
        else:
            fp += 1
    accuracy = float(tp + tn) / float(tp + tn + fp + fn)
    precision = float(tp) / float(tp + fp)
    recall = float(tp) / float(tp + fn)
    f_value = 2.0 * recall * precision / (recall + precision)
    # confusion matrix
    confusion = pd.DataFrame([[tp, fp], [fn, tn]], index=[pos_label, neg_label], columns=[pos_label, neg_la
bel])
    return confusion, accuracy, precision, recall, f_value
```

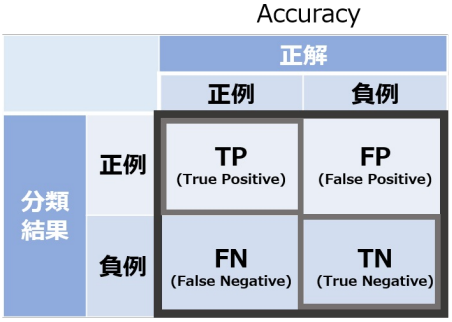
In [ ]:

```
confusion, accuracy, precision, recall, f_value = analyze_results(labels_test, results)
print('confusion matrix\n{0}\n'.format(confusion))
print('metric    : score')
print('accuracy  : {0:.3f}'.format(accuracy))
print('precision : {0:.3f}'.format(precision))
print('recall    : {0:.3f}'.format(recall))
print('f_value   : {0:.3f}'.format(f_value))
```

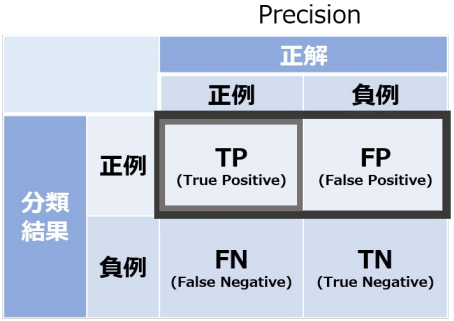
Recallが低い問題

- Accuracyが高いにも関わらず、Recallが低い
- Recall：Defaultする人に対し、Defaultしたと予測できているか
  - これが低い => Defaultするはずの人を捉えられていない => リスク管理できない！

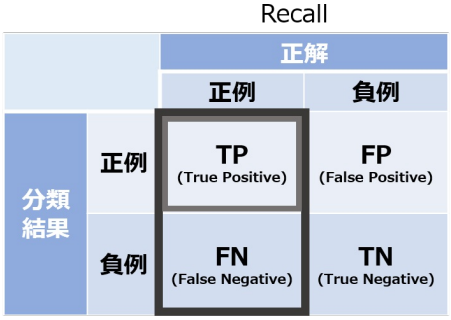
Accuracy



Precision



Recall



Recallを上げるために => アンダーサンプリング

In [ ]:

```
print(df[df["Y"]==0].shape)
print(df[df["Y"]==1].shape)
```

- 現状
  - 正例(defaultした) : 5577件
  - 負例(defaultしていない) : 19421件
- 負例に引きずられて、分類がうまくいっていない可能性あり

In [ ]:

```
import random
random.seed(42) # シードで乱数を固定(再現性を得たい場合に実行)
def under_sampling(features, labels, reduce_label, reduce_rate=0.2):
    sampled_features, sampled_labels = [], []
    for feature, label in zip(features, labels):
        label = str(label)
        if label != reduce_label or random.random() < reduce_rate:
            sampled_features.append(feature)
            sampled_labels.append(label)
    return sampled_features, sampled_labels
```

In [ ]:

```
# アンダーサンプリング
reduce_rate = 0.2
sampled_features_train, sampled_labels_train = under_sampling(features_train, labels_train, reduce_label="0
",reduce_rate=reduce_rate)
# 学習用Datumリストを作る
sampled_train_data = []
for x, y in zip(sampled_features_train, sampled_labels_train):
    d = Datum({key: float(value) for key, value in zip(columns, x)})
    sampled_train_data.append([str(y), d])
# 学習をする
client = Classifier('127.0.0.1', 9199, '')
client.clear()
client.train(sampled_train_data)
# テストをする
results = client.classify(test_data)
```

In [ ]:

```
# 結果を分析する
confusion, accuracy, precision, recall, f_value = analyze_results(labels_test, results)
print('confusion matrix\n{0}\n'.format(confusion))
print('metric    : score')
print('accuracy  : {0:.3f}'.format(accuracy))
print('precision : {0:.3f}'.format(precision))
print('recall    : {0:.3f}'.format(recall))
print('f_value   : {0:.3f}'.format(f_value))
```

PrecisionとRecallのトレードオフ

metric	score normal	score under sampling
Accuracy	0.803	0.542
Precision	0.568	0.276
Recall	0.291	0.716
f_value	0.385	0.398

In [ ]:

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

rs = np.linspace(0,1.0,11)
precisions = []
recalls = []
```

In [ ]:

```
for r in rs:
    sampled_features_train, sampled_labels_train = under_sampling(features_train, labels_train, reduce_label="0",reduce_rate=r)
    # 学習用Datumリストを作る
    sampled_train_data = []
    for x, y in zip(sampled_features_train, sampled_labels_train):
        d = Datum({key: float(value) for key, value in zip(columns, x)})
        sampled_train_data.append([str(y), d])
    # 学習をする
    client = Classifier('127.0.0.1', 9199, '')
    client.clear()
    client.train(sampled_train_data)
    # テストをする
    results = client.classify(test_data)
    confusion, accuracy, precision, recall, f_value = analyze_results(labels_test, results)
    precisions.append(precision)
    recalls.append(recall)
#     print(confusion)
    print("rate:{:.1f} precision:{:.2f} recall:{:.2f} accuracy:{:.2f} f-value:{:.2f}".format(r, precision, recall, accuracy, f_value))
```

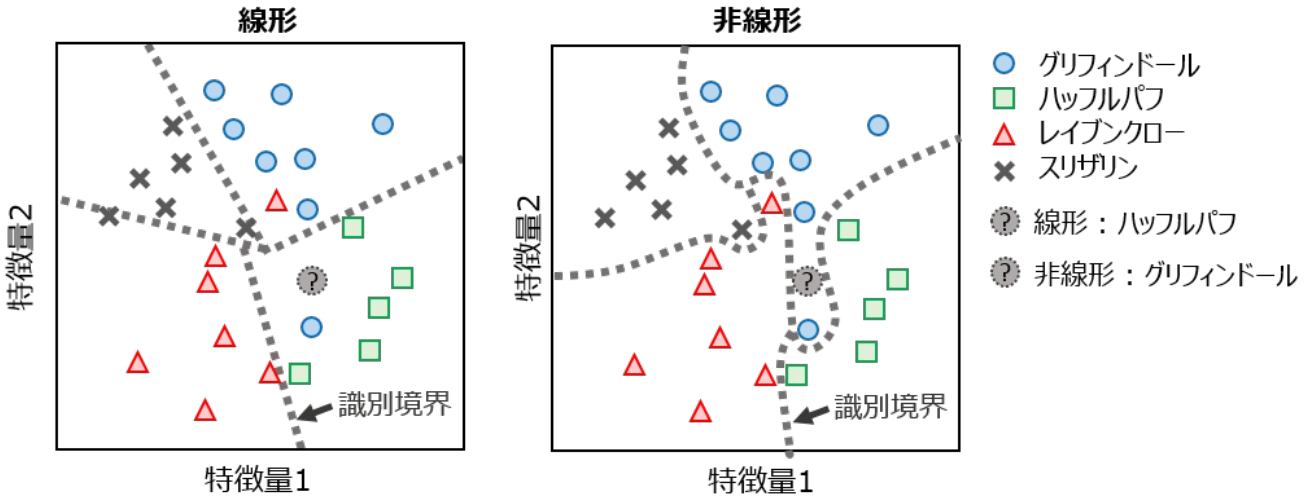
In [ ]:

```
plt.plot(rs, precisions,"o-",alpha=0.5,label="precision")
plt.plot(rs,recalls,"o-",alpha=0.5,label="recall")
plt.xlim(0,1)
plt.ylim(0,1)
plt.legend()
plt.show()
```

非線形分類器

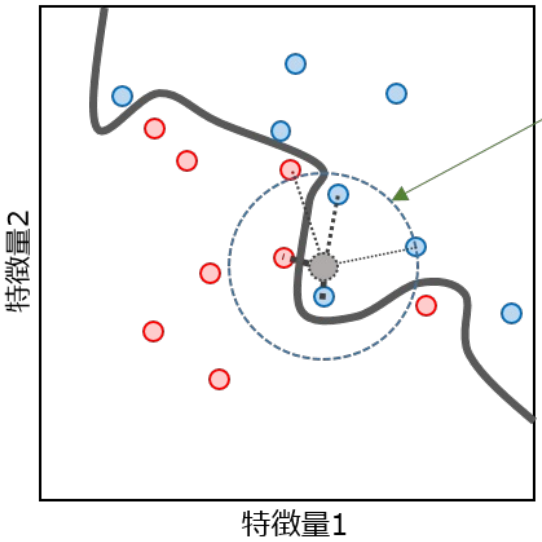
※今回、非線形分類器の実行は割愛

■線形分類と非線形分類



非線形分類器のパラメータ

■k近傍法(k Nearest Neighbor)



近傍点をk個(図は$k = 5$)を見て、
自分がどっちに行くかを決める
近いほど影響力大

クラス1　クラス2

新しい点がどっちのクラスなのかは、
スコアで決める

$Score(\text{クラス}) = \exp(-\text{distance} * S)$
Sは、距離に対する感度

設定可能な変数

近傍点の数：k　距離に対する感度：S　距離の測り方：**distance**

まとめ

- Jubatusを使って分類をやってみた
- 分類の評価指標はたくさんあるので、目的に合わせて選ぶ
- データをうまく処理することで、線形分類器でも精度を上げることが可能

NEXT : Jubakit

より簡単に分析が可能に!