Hands on Machine Learning with Scikit-Learn &TensolFlow

Chapter 3

Classification

Created by Yusuke FUJIMOTO

はじめに

- この資料は「Hands-On Machine Learning with Scikit-Learn and TensorFlow - O'Reilly Media」を読んだ際の（主にソースコードに関する）簡単な解説を残したものです。
- 全部を解説したわけではないので注意
- 余裕があればソースコード周りの背景知識もまとめたい
- 何かあったら yukkyo12221222@gmail.com まで

Chapter 3

Classification

MNIST

- 0 から 9 の手描き文字が集められているデータ。すごくメジャー
- 各画像は 28 x 28 のピクセル
- imshow 関数 Matplotlib 2.0.2 documentation

```
plt.imshow(
    some_digit_image,
    cmap = matplotlib.cm.binary, # カラーマップ
    interpolation="nearest"      # なめらかにするなどの設定
    )
# 'none', 'nearest', 'bilinear', 'bicubic', 'spline16',
# 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser',
# 'quadric', 'catrom', 'gaussian', 'bessel', etc...
```

画像を一気にたくさん表示したい場合の例

```
# EXTRA
def plot_digits(instances, images_per_row=10, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = \\
        [inst.reshape(size,size) for inst in instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        x = images_per_row
        rimages = images[row * x : (row + 1) * x]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image, cmap = matplotlib.cm.binary,
               **options)
    plt.axis("off")
```

- 引数についている `**` や `*` について
  - 可変長引数を扱える
  - `*` が付いた引数 : タプルとして渡される
  - `**` が付いた引数 : 辞書として渡される
  - 参考記事 : Python 可変長引数について - Qiita
- 例として以下のような関数を定義する

```
def func1(a, *args):
    print(a, args)

def func2(a, **args):
    print(a, args)

def func3(a, *args1, **args2):
    print(a, args1, args2)
```

- 実行結果は以下のようになる

```
func1("a", "b", "c")
# => a ('b', 'c')

# 辞書型なのにキー未指定なのでエラー
func2("a", "b", "c")
# => TypeError: func2() takes 1 positional argument but 3

# タプルなのにキー指定でエラー
func1("a", b = "b", c = "c")
# => TypeError: func1() got an unexpected keyword argument

func2("a", b = "b", c = "c")
# => a {'c': 'c', 'b': 'b'}
```

- 以下のように混ぜて使用することもできる

```
func3("a", "b1", "b2", c1 = "C1", c2 = "C2")
# => a ('b1', 'b2') {'c2': 'C2', 'c1': 'C1'}
# 変数1, タプル, 辞書
```

- ただしタプルや辞書に入れる変数は連続していれないといけないし、順番も守る

```
func3("b", "a", c1="C1", "d", c2="C2") #  連続していない
# => SyntaxError: positional argument follows keyword argu
func3("b", c1 = "C1", c2 = "C2", "a")  #  順番が違う
# => SyntaxError: positional argument follows keyword argu

#  成功例
func3("b", "a" , c1 = "C1", c2 = "C2")
# => b ('a',) {'c2': 'C2', 'c1': 'C1'}
func3("b", c1 = "C1", c2 = "C2")
# => b () {'c2': 'C2', 'c1': 'C1'}
```

Binary classifier

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3, random_state=42)

for train_index, test_index in skfolds.split(X_train, y_tr
    clone_clf = clone(sgd_clf)  # 同じパラメータの学習器をコピー
    X_train_folds = X_train[train_index]
    y_train_folds = (y_train_5[train_index])
    X_test_fold = X_train[test_index]
    y_test_fold = (y_train_5[test_index])

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    # TRUE は 1 として扱われる
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))
```

- confusion matrix は大事な概念なので調べる

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_train_5, y_train_pred)
```

- precision : 精度 = TP / (TP + FP)
- recall : 復元率 = TP / (TP + FN)
- f(1) value : precision と recall の調和平均
  - $\frac{2 \times P \times R}{P + R}$

```
from sklearn.metrics import precision_score, recall_score
from sklearn.metrics import f1_score
```

precision recall curve (導入）

- 最小二乗法の欠点 : 出力を確率と解釈できない、外れ値に弱い、複数のクラスが一直線に並んでいるような分布をしている場合にうまくいかない
- フィッシャーの線形判別
  - $y = \mathbf{w}^T \mathbf{x}$ 、y は1次元の値
  - $y \geq w_0$ ならクラス1、否ならクラス2
  - 重みベクトル $\mathbf{w}$ を調節し分離を最大にする射影を選択する。

Precision Recall curves

- とりあえず `sgd_clf` もスコアが出力できる
  - そのスコアの大小でクラスを分けている
  - 閾値を変えれば precision と recall も変わる
  - 縦軸を precision、横軸を recall とした散布図（カーブ）が描ける → prcision recall curves

ROC curves

- ROC (Receiver Operating Characteristic : 受信者動作特性) 曲線
- この曲線の下の部分の面積(AUC : Area Under the Curve) も指標の一つ

Multiclass classification

- 多クラス分類 : 2クラス分類をたくさん用意して行うことができる。その場合は大きく2種類（One vs One, One vs Others)
  - One vs One : $K \times (K-1)$ 個の分類器を用意
  - One vs Others : K 個の分類器を用意

Multilabel classification

- 出力自体が複数個ある分類（他の種類のラベルも付いている、等）

Multioutput classification

- Multilabel classification とだいたい同じ
- こちらは $28 \times 28 = 784$ 次元の値を出力
  => 画像自体を出力しているようなもの
- サンプルコードではノイズ除去のような動作ができている

Extra material

Dummy classifier とは

調べる

KNN classifier

- サンプルコードでは
- 少しずらした画像も用意してともに学習させている
  => 実際にやや精度が上がった
- K-近傍法 について調べてもらう

Extra material（カプセル化のメリット）

カプセル化は自分の機能を「管理」するためのもの

- インスタンス変数の変更を管理できる
  - 使う側が勝手に変えられちゃうとエラーが起きた場所が分かりづらかったりする
  - 変数の範囲や型などのチェックを行う場所がなくなる
  - 特にほかの人が変な値を入れた場合、どこでその変数をいじったのか探さなければならない
  - 加えてその人が呼び出した付近まで修正しなきゃいけない可能性が出てくる

Extra material（カプセル化のメリット）

- 勝手にすべてのメソッドが呼び出されなくなる
  - 人が作ったクラスを動かす際に欲しいのは「動かし方」のみ
  - それ以外のサブタスクを行う関数は別に見たくない
  - むしろサブタスク関数を単独で使われると意図しない挙動やインスタンス変数の変更がされる可能性がある

- カプセル化に関する例（カプセル化してない）

```
class SampleUser:
    def __init__(self, age=0):
        self.age = age

user = SampleUser(10)
print(user.age) # => 10
user.age = -1
print(user.age) # => -1
```

- 上記の例の問題点
  - age の範囲を制限できない
  - age の値をメソッドを使わずに変えられる

- カプセル化に関する例（カプセル化してる）

```
class SampleUser:
    def __init__(self, age=0):
        self.set_age(age)

    def set_age(self, age):
        if age < 0 or age > 100:
            print("age が大きすぎます")
        else:
            self.__age = age

    def get_age(self):
        return self.__age
```

- 変数の先頭に `__` をつけるとプライベート変数（疑似）になる

- カプセル化に関する例（カプセル化してる）

```
user = SampleUser(10)
print(user.get_age())  # => 10

# 勝手に値をセットできない例
user.__age = -1
print(user.get_age())  # => 10

# set を使えば値をセットできる例
user.set_age(50)
print(user.get_age())  # => 50

# 変な値も設定できない例
user.set_age(-1)  # => "age が大きすぎます"
print(user.get_age())  # => 50
```

- SampleUser クラスが値のチェックとかまでやってくれている！