



この課題の目的

- コーディング力を向上させる
- スクラッチを通してロジスティック回帰を理解する
- 分類問題について基礎を学ぶ

```
In [1]: #データの取り込み
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

%matplotlib inline

In [2]: # Irisデータセットの用意
iris = pd.read_csv("/Users/tsuneo/kaggle/IrisSpecies/Iris.csv")

In [3]: # 型の確認
iris.shape

Out[3]: (150, 6)

In [4]: iris.head(3)

Out[4]:
   Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species
0  1         5.1          3.5         1.4          0.2    Iris-setosa
1  2         4.9          3.0         1.4          0.2    Iris-setosa
2  3         4.7          3.2         1.3          0.2    Iris-setosa

In [5]: # 目的変数Speciesの中から2つに限定する
iris = iris.query("Species in ['Iris-versicolor', 'Iris-virginica']")

In [6]: iris.shape

Out[6]: (100, 6)

In [7]: # Irisのデータを説明変数、目的変数に分割する
X = iris[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']].values
y = iris['Species']
print(type(y))
y["Species"] = y["Species"].map({'Iris-versicolor':0, 'Iris-virginica':1})
print(type(y))
X.shape, y.shape

<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.frame.DataFrame'>

/Users/tsuneo/.pyenv/versions/anaconda3-4.3.0/lib/python3.6/site-packages/ipykernel_launcher.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy
"""

Out[7]: ((100, 4), (100, 1))

In [8]: # バイアス項対応のため、値が1の列を、最後の列に追加
X = np.hstack((X, np.ones((X.shape[0], 1))))
X.shape

Out[8]: (100, 5)

In [9]: # Iris-versicolorとIris-virginicaの割合が偏らないようにする
# すいません、sklearnに頼りました
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y["Species"])
X_train.shape, X_test.shape, y_train.shape, y_test.shape

Out[9]: ((80, 5), (20, 5), (80, 1), (20, 1))

In [10]: # pandasのDataFrameからNumPy配列に変換
# X_train = X_train.values
# X_test = X_test.values
y_train = y_train.values
y_test = y_test.values
```

ロジスティック回帰のクラスを作成する

シグモイド関数を実装する。

```
In [11]: def sigmoid_func(z):
    return 1/(1+np.exp(-z))

In [12]: x = np.arange(-10, 11, 0.1)
y = sigmoid_func(x)
plt.plot(x, y, "r", label="sigmoid-function")
plt.xlabel("number", fontsize=15)
plt.ylabel("sigmoid", fontsize=15)
plt.title("sigmoid-function", fontsize=20)
ax = plt.gca()
ax.yaxis.grid(True)
plt.legend()
plt.show()
```

ロジスティック回帰class

```
In [13]: import matplotlib.pyplot as plt
class ScratchLogisticRegression:

    # コンストラクタ
    def __init__(self, lamda, lr, iteration, bias):
        self.lamda = lamda
        self.lr = lr
        self.iteration = iteration
        # 重みの初期化はコンストラクタの中では行わない。これでは、任意の特徴量に対応できないため
        self.theta = np.random.rand(4, 1) # 重みパラメータ
        self.bias = bias
        # バイアス項を入れない場合はTrue

    # シグモイド関数
    def _sigmoid_func(self, z):
        return 1/(1+np.exp(-z))

    # 仮定関数(X:N行x4列, theta:4行x1列 = N行x1列)
    def _assumed_func(self, X_train):
        return np.dot(X_train, self.theta)

    # 目的関数(損失関数)を実装する(戻り値はスカラー値)
    def _logistic_reg_func(self, X_train, y_train):
        m = len(X_train) # 入力されるデータ数
        z = self._assumed_func(X_train) # 入力総和(ベクトル)
        h = self._sigmoid_func(z) # 仮定関数(ベクトル)
        return (1/m)*np.sum(y_train*np.log(h) + (1-y_train)*np.log(1-h)) + (self.lamda/2*m)*np.sum(self.theta**2)

    # 勾配下降法で重みを更新する
    def _gradient_descent(self, X_train, y_train):
        m = len(X_train)
        self.theta = np.random.rand(4, 1)
        for i in range(self.iteration):
            # 仮定関数(予測値)と教師データとの差分を取得し、転置する
            A = (self._assumed_func(X_train) - y_train).T
            # 重みを更新する
            self.theta = self.theta - self.lr*((1/m)*np.sum(np.dot(A, X_train))) + (self.lamda/m)*self.theta

            # 損失値を取得する
            cost = self._logistic_reg_func(X_train, y_train)

            # 格納していく
            self.theta_list.append(self.theta)
            self.cost_list.append(cost)

    # 学習
    def fit(self, X_train, y_train):
        cost = []
        self.cost_list = []
        self.theta_list = []

        # コンストラクタに記載していた重みの初期化をfitの中で実装した
        # まだ、バイアス項を入れない判断ロジックを追加
        if self.bias:
            self.theta = np.random.rand(X_train.shape[1], 1)
            X_train = np.delete(X_train, obj=1, axis=1)
        else:
            self.theta = np.random.rand(X_train.shape[1], 1)

        self._gradient_descent(X_train, y_train)

        # 確率を予測・求めた重みを元に、予測するだけ
        def predict_proba(self, X_test):
            # バイアス項を入れないロジックを追加
            if not self.bias:
                X_test = np.delete(X_test, obj=1, axis=1)
            return self._assumed_func(X_test)

        # 推定したラベルを出す
        def predict_label(self, X_test):
            proba = self.predict_proba(X_test)

            # for文を回避する
            proba[proba[:,-1] >= 0.5, -1] = 1
            proba[proba[:,-1] < 0.5, -1] = 0

            # for文で実施していたが、コメントアウトした
            #for i, proba_to_flag in enumerate(proba):
            #    if proba_to_flag > 0.5:
            #        proba[i, -1] = 1
            #    else:
            #        proba[i, -1] = 0
            #    proba = proba.astype(np.int)

            return proba

        # 描画
        def visualize_graph(self):
            plt.plot(self.cost_list, "r", label="data")
            plt.xlabel("iteration", fontsize=15)
            plt.ylabel("loss", fontsize=15)
            plt.title("Learning curve", fontsize=20)
            plt.legend()
            plt.show()

    # 評価
    def evaluate(self, X_train, y_train):
        cost = []
        self.cost_list = []
        self.theta_list = []

        # コンストラクタに記載していた重みの初期化をfitの中で実装した
        # まだ、バイアス項を入れない判断ロジックを追加
        if self.bias:
            self.theta = np.random.rand(X_train.shape[1], 1)
            X_train = np.delete(X_train, obj=1, axis=1)
        else:
            self.theta = np.random.rand(X_train.shape[1], 1)

        self._gradient_descent(X_train, y_train)

        # 確率を予測・求めた重みを元に、予測するだけ
        def predict_proba(self, X_test):
            # バイアス項を入れないロジックを追加
            if not self.bias:
                X_test = np.delete(X_test, obj=1, axis=1)
            return self._assumed_func(X_test)

        # 推定したラベルを出す
        def predict_label(self, X_test):
            proba = self.predict_proba(X_test)

            # for文を回避する
            proba[proba[:,-1] >= 0.5, -1] = 1
            proba[proba[:,-1] < 0.5, -1] = 0

            # for文で実施していたが、コメントアウトした
            #for i, proba_to_flag in enumerate(proba):
            #    if proba_to_flag > 0.5:
            #        proba[i, -1] = 1
            #    else:
            #        proba[i, -1] = 0
            #    proba = proba.astype(np.int)

            return proba

        # 描画
        def visualize_graph(self):
            plt.plot(self.cost_list, "r", label="data")
            plt.xlabel("iteration", fontsize=15)
            plt.ylabel("loss", fontsize=15)
            plt.title("Learning curve", fontsize=20)
            plt.legend()
            plt.show()
```

```
In [14]: # 学習してみる
sir = ScratchLogisticRegression(lamda=0.00001, lr=0.000001, iteration=30000, bias=False)
sir.fit(X_train, y_train)

In [15]: # 描画してみる
sir.visualize_graph()
```

説明

ロジスティック回帰とは

線形分類問題と二値分類問題に使われるアルゴリズムです。名前には「回帰」とついていますが、分類問題を扱います。ラベルの値が2種類しかないような教師あり訓練データに適用されます。ラベルの値は0または1だとします。与えられた特徴量のサンプルxに対して、ラベルyが1になる確率をPで表し、yが0になる確率を1-Pで表します。

$$P(Y=1|X=x) = \sigma(w_0 + \sum_{j=1}^d w_j x_j) = \sigma(w^T x)$$

ここで、 σ はシグモイド関数と呼ばれるもので $\sigma(z) = 1/(1 + e^{-z})$ で定義されます。

式1でxの中を見ると、xについての線形関数となっています。ここでのアイディアは、サンプルxがラベル1に属するかどうかを確からしさは線形関数の値が大きいほど大きくなるという仮定ですが、単純に線形関数の値を考えるとその値は-∞から+∞までの値となります。

交差エントロピー誤差関数とは

損失関数に分類される関数です。ロジスティック(logistic)関数と呼ばれることもあります。機械学習では活性化関数として利用されます。yの値が0 < y < 1の範囲となるため、確率を表すことができる。xが0の時に0.5となる。

正則化とは

機械学習で過学習を防ぐ。正則化はモデル(仮定関数)のパラメータの学習に使われます。汎化能力を高めるために使われます。L1正則化とL2正則化があるが、今回のSPRINTはL2正則化の方で行った。

過学習が発生しているモデルは、「バリアンスが高い」(high variance)とも表現される。その原因はパラメータの数が多すぎるため。データに対してモデルが複雑すぎることが考えられる。同様に、モデルは学習不足(underfitting)に陥ることもある。つまり、トレーニングデータセットのパターンをうまく捕捉するにはモデルの複雑さが十分ではなく、未知のデータに対する性能が低いことを意味する。このようなモデルは「バイアスが高い」(high bias)とも呼ばれます。

平均二乗誤差と交差エントロピー誤差

ロジスティック回帰では、平均二乗誤差ではなく交差エントロピー誤差を使う理由を説明せよ。

損失関数を $L(t, f(w, x))$ とする

線形回帰の平均二乗誤差

$$L(t, f(w, x)) = \sum_{i=1}^n (t_i - f(w, x_i))^2$$

正解tと関数の出力f(w, x)が近くなるようする

ロジスティック回帰の交差エントロピー誤差

$$L(t, f(w, x)) = -\sum_{i=1}^n t_i \log(f(w, x_i))$$

正解tと出力f(w, x)をそれぞれ確率分布とみなして、確率分布に近づける。

線形回帰は、正解ラベルと予測値との差分を引き算で計算できた。しかし、ロジスティック回帰は確率を求めるモデルのため、単純に引き算で誤差を計算してはいけない。そのため、線形とロジスティックでは行っていることが異なるので、棲み分けしている。