はじめに

武蔵野大学データサイエンス学部(MUDS)2年の高橋快成です． scikit-learnをスクラッチで実装してみるというプロジェクトで，ロジスティック回帰を実装しました． わかりやすく一歩ずつまとめたつもりなので実際に手を動かしながら理解してみてください．

目次

ロジスティック回帰とは

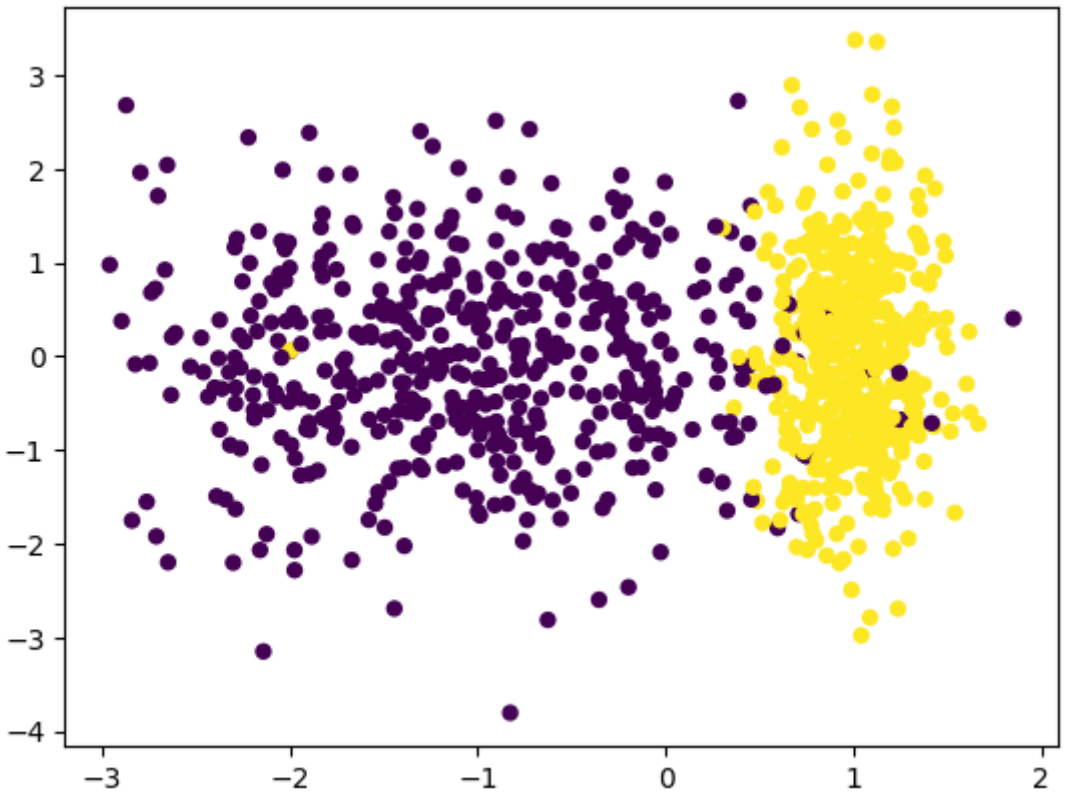
ベルヌーイ分布に従う変数の統計的回帰モデルの一種．主に二値分類で使われる． 今回は二値分類のロジスティック回帰を実装する，

scikit-learnの出力

以下のようなデータを作ります．特徴量は2つです．

```
import time
import random
import pandas as pd
import numpy as np
from sklearn.datasets import make_classification
from sklearn.datasets import make_regression
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.animation import ArtistAnimation
from IPython.display import HTML, Image, Video

N = 1000 #データの数
d = 2 # 次元数
K = 2 # クラス数

X, y = make_classification(
    n_samples=N,
    n_features=d,
```

```
    n_informative=1,
    n_redundant=0,
    n_clusters_per_class=1,
    n_classes=K,
    random_state = 20
)

plt.scatter(X[:, 0], X[:, 1], marker="o", c=y, s=25) #c=yで色指定できる
```



scikit-learnでは以下のような出力がされると思います.

```
model = LogisticRegression(
    penalty=None,
    dual=False,
    tol=0.0001,
    C = 1.0,
    fit_intercept= False,
    intercept_scaling= 1.0,
    class_weight=None,
    random_state=None,
    solver='newton-cg',
    max_iter=100,
    multi_class='auto',
    verbose=0,
    warm_start=False,
    n_jobs=None,
    l1_ratio=None
    )
```

```
start = time.time()
model.fit(X,y)
end = time.time()
time_diff = end - start
print('実行時間',time_diff)
print('クラス数',model.classes_)
print('特徴量の数',model.n_features_in_)
print('coef',model.coef_)
print('intercept',model.intercept_)
print('反復数',model.n_iter_)
```

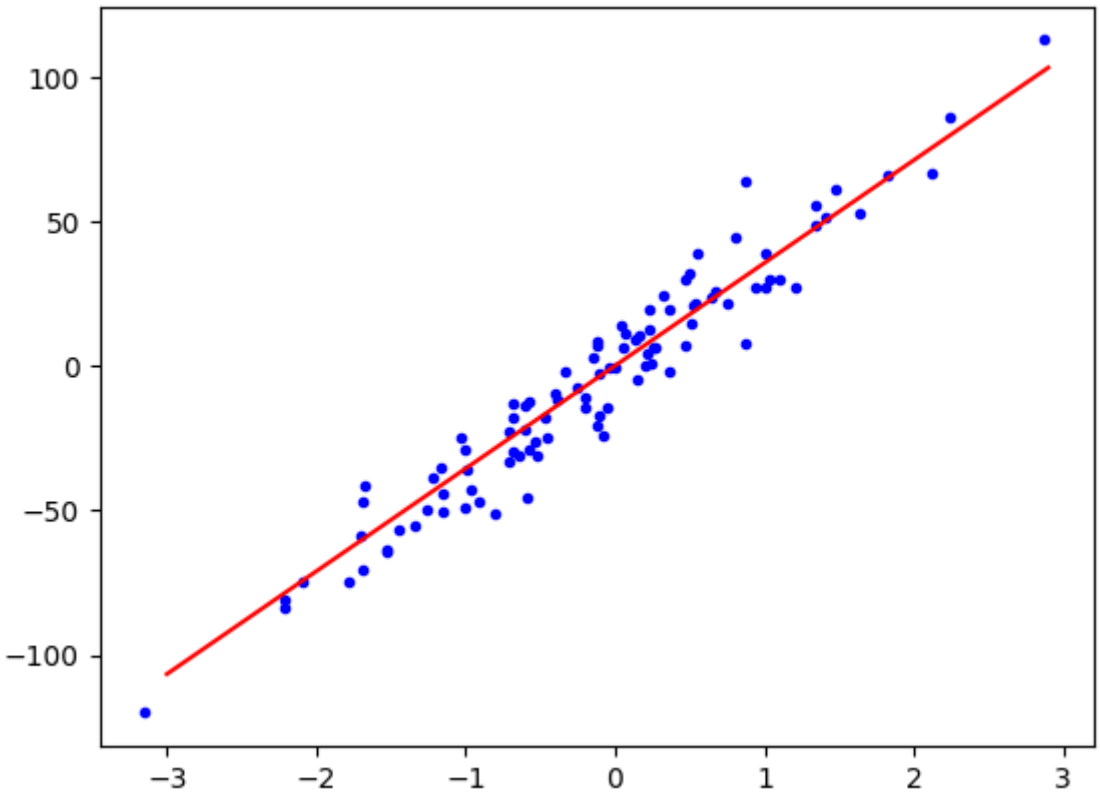
```
実行時間 0.004872798919677734
クラス数 [0 1]
特徴量の数 2
coef_ [[3.62605703 0.07726773]]
intercept_ [0.]
反復数 [5]
```

スクラッチで実装するので`coef_ [[3.62605703 0.07726773]]`と`intercept_ [0.]`を再現することができればクリアです． なお今回はバイアス項なしで実装するので`intercept_`は0で問題ありません．

ロジスティック回帰の気持ち

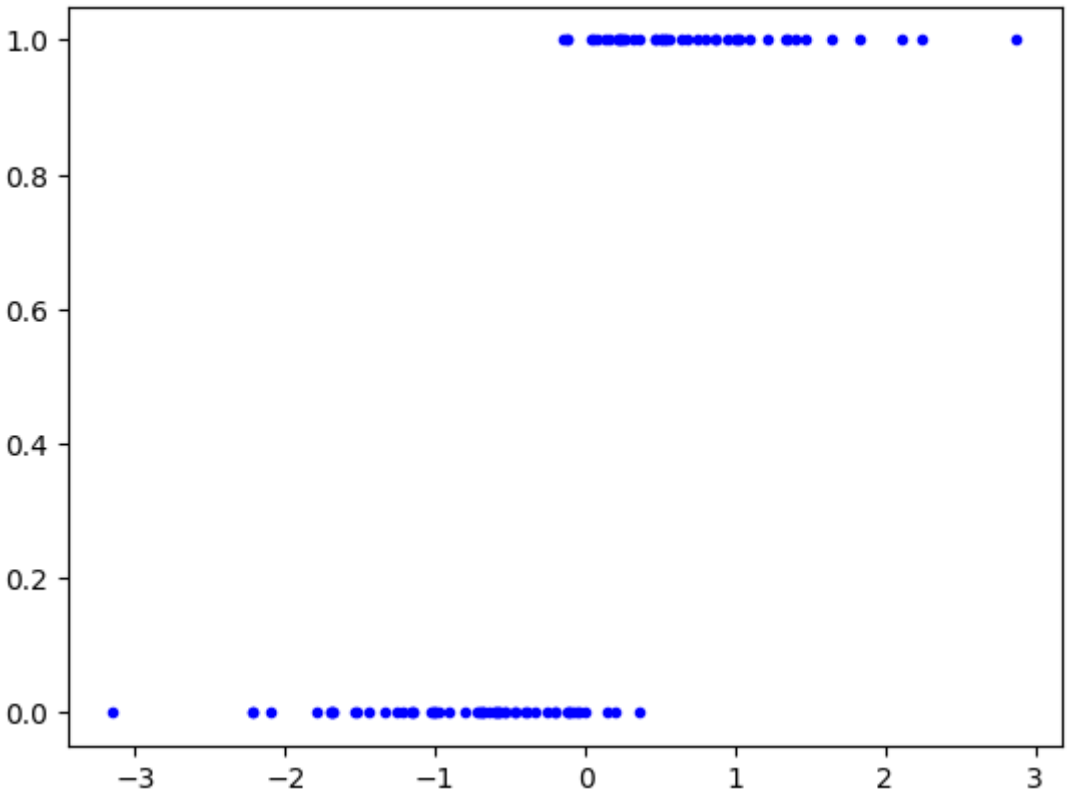
ざっくりイメージを掴むためにグラフやらなんやらでロジスティック回帰は何をしているのかを説明していきます．
そのために最初は**線形回帰**から考えていきます．こいつがベースです．

```
x_reg, y_reg, coef_reg = make_regression(random_state=12,
                        n_samples=100,
                        n_features=1,
                        n_informative=1,
                        noise=10.0,
                        bias=-0.0,
                        coef=True)

x0 = np.arange(-3,3,0.1)

plt.figure
plt.subplot(1, 1, 1)
plt.scatter(x_reg, y_reg, marker='.',color='blue')
plt.plot(x0,x0*coef_reg, color='red')
plt.plot()
```



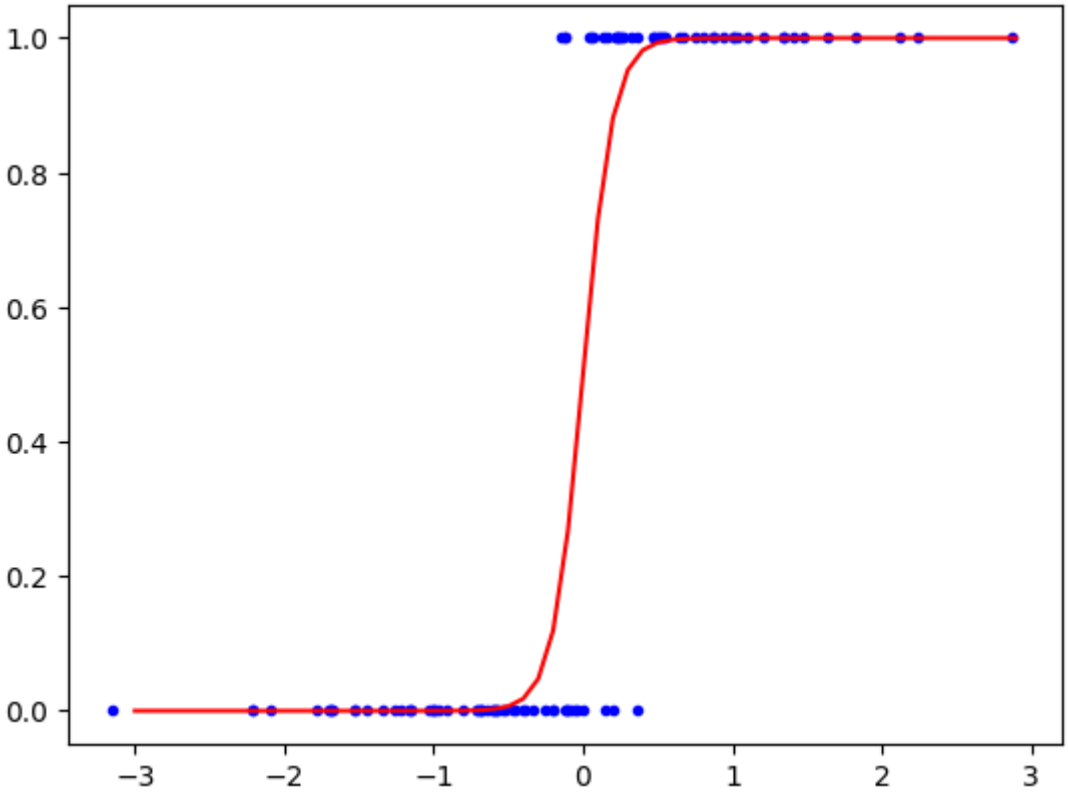
ここから二値分類を考えていきます． 試しに．上のデータを縦軸の値で0以上を取ったら1，0以下を取れば0としてみます． 要するに横に半分で割って二値にします．

```
y_reg_0or1 = y_reg[:]

for i in range(len(y_reg_0or1)):
    if y_reg_0or1[i] > 0:
        y_reg_0or1[i] = 1
    else:
        y_reg_0or1[i] = 0

plt.figure
plt.subplot(1, 1, 1)
plt.scatter(x_reg, y_reg_0or1, marker='.',color='blue')
plt.plot()
```



こんな感じになりました． ではこの図に残差が少ない直線をうまく引けるでしょうか？ 一個前の図のように直線を引いたとしても回帰直線は0より下の値，もしくは1より上の値も取ってしまうので適切とは言い難いです． なんとか点をうまくなぞるようにして0と1の間に線を収めたいですね． そしたら以下のようにしてみましょう．

```
def sigmoid_method(x, gain):
    return 1 / (1 + np.exp(-x * gain))

sigmoid_x = np.arange(-3, 3, 0.1)

plt.figure
plt.subplot(1, 1, 1)
plt.scatter(x_reg, y_reg_0or1, marker='.',color='blue')
plt.plot(sigmoid_x,sigmoid_method(sigmoid_x, gain=10), color='red')
plt.plot()
```



こんな感じになります． やったことはシグモイド関数に線形回帰の式$\boldsymbol{\hat{y}}=X\boldsymbol{w}$を当てはめているだけです． これはシグモイド関数(厳密にはロジスティック関数)を使用しているため，ロジット変換とも言われたりします． やることは**これだけ**です．あとはこれを数式で表現しましょう．

シグモイド関数は後で説明します．

理論と数式

以降で使用する記号を以下とする． 列ベクトルは(⚪ ×1行列)のように表現したりもする． n:データ数
m:次元数
$X\in \mathbb{R}^{n\times m}$:説明変数
$\boldsymbol{y}\in \mathbb{R}^{n\times 1}$:目的変数
$\boldsymbol{w}\in \mathbb{R}^{m\times 1}$:パラメータ，重み

さっきやったことを数式で確認

まず線形回帰の式を用意する． $$ X\boldsymbol{w} $$ 次にシグモイド関数を用意する．(この中のtは特に意味はない．) $$ \sigma(t) = \frac{1}{1-e^{-t}} $$ そしてシグモイド関数に線形回帰の式を代入する． $$ \boldsymbol{\hat{y}} = \frac{1}{1-e^{-X\boldsymbol{w}}} \ $$ これで出力値が**ほぼ**二値(0,1)のモデルができた．
ほぼ二値なのはさっきシグモイド関数のグラフをみたのでわかっている．
ほぼなので最終的にはシグモイド関数の出力を0.5を基準に二値に分ける操作が必要である．

最尤法

尤度をLとする．
二値の実測値yが$y=1$となる確率を$P=p(y=1)$，$y=0$となる確率を$1-P=p(y=0)$とする．

ここでPの具体的な形はわかっていないので注意． まだ$y=1$のときの確率pはPで，$y=0$のときの確率pは$1-P$であるとしただけ．

$y(y_1,y_2,...,y_n)$が独立であり，$p(y_i=1)=P_i$であるとき尤度Lは以下になる．

```
L = \prod_{i=1}^n P_i^{y_i}(1-P_i)^{1-y_i}
```

$y_i=1$のとき，$P_i^{y_i}$が残って，$y_i=0$のとき$(1-P_i)^{1-y_i}$が残るって感じ．

計算しやすくするため対数尤度をとる．

```
\begin{align}
log(L)
&= log(\prod_{i=1}^n P_i^{y_i}(1-P_i)^{1-y_i}) \\
&= \sum_{i=1}^{n}(y_i logP_i+(1-y_i)log(1-P_i))\\
\end{align}
```

この対数尤度を最大化する． そして具体的な形が不明だった確率P_iは以下である．$\hat{y}_i=P_i$としたのは，$\hat{y}_i$はそのまま予測確率だから．

```
P_i = \hat{y}_i = \sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)
```

そして，負の対数尤度を$E(\boldsymbol{w})$として$E(\boldsymbol{w})$を最小化することを考える．

```
E(\boldsymbol{w})=-\log (L) = \sum_{i=1}^{N}(Y_i \log
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)+(1-Y_i)\log(1-
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)))
```

したがって，この最適値をニュートン法によって求める．

```
\nabla E(\boldsymbol{w})=0
```

を求めていく．(勾配が0になる点を求めていくということ．)

ニュートン法

ネタバレすると，最終的に使用する更新式は以下．

```
\boldsymbol{w}^{new}=\boldsymbol{w}^{old}-H^{-1}\nabla
E(\boldsymbol{w}^{old})
```

これがパラメータ$\boldsymbol{w}$の更新式である．これを繰り返すことで自ずと適切なパラメータが求められる． これはニュートン法と呼ばれる手法を使用している． 数式からは少し逸れてしまうが，ニュートン法のイメージをざっくり掴んでおく．

ニュートン法のイメージ

ニュートン法によって$(x - 1)^2$の解を近似的に求めてみる． ニュートン法の式は以下

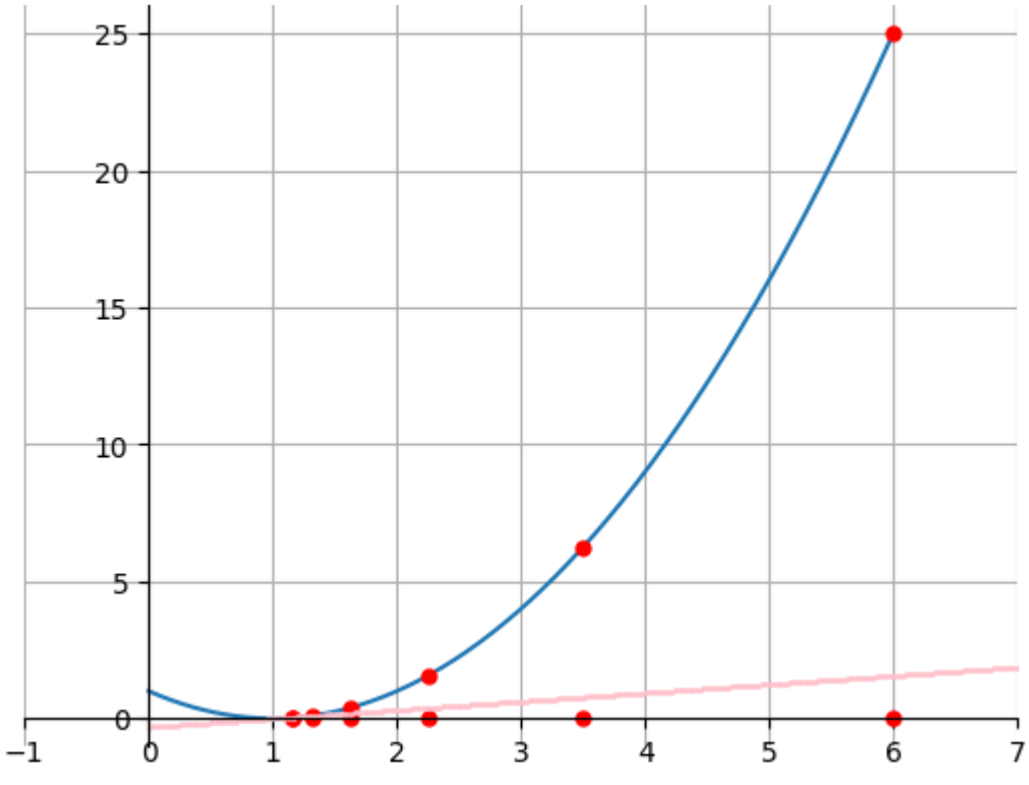
```
x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}
```

実際にニュートン法で$(x-1)^2$を近似してみる．
代入すると以下のようになっている．

```
x^{new} = x^{old} - \frac{(x-1)^2}{2x}
```

```
# 近似したい関数
def here_function(x):
    y = (x - 1) ** 2
    return y

# 0を返すだけの関数，xの位置がわかりやすくするため．
def zero_function(x):
    return 0

def diff(diff ,x ,x0 ,intercept):
    return 2 * diff * (x-x0) + intercept

def here_function_derivative(x):
    return 2 * (x - 1)

start = 0 #定義域の左端
last = 6 # 定義域の右端

x0 = np.arange(start,last, 0.01)
y0 = here_function(x0)
x1 = np.arange(0,8, 0.01)

fig = plt.figure() # Figureオブジェクトを作成
ax = fig.add_subplot(1,1,1) # figに属するAxesオブジェクトを作成
#ax.set_aspect('equal')

ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.spines['bottom'].set_position('zero')
ax.spines['left'].set_position('zero')
ax.set_xlim(-1,7)
```

```
ax.set_ylim(-1,26)
ax.grid() # グラフにグリッドを追加


dx = 0.1
ax.plot(x0, y0)

artist_list = []
point_list_x = []
point_list_y = []
point_list_0 = []

x_ = 6.0
x_new = np.inf
for i in range(150):
    y_ = here_function(x_)
    zero = zero_function(x)

    # ニュートン法
    x_new = x_ - y_ / here_function_derivative(x_)
    if abs(x_new - x_) < 0.001:
        break
    point_list_x.append(x_)
    point_list_y.append(y_)
    point_list_0.append(zero)

    art = plt.plot(point_list_x, point_list_y, marker='.', markersize=10,
color='red', linestyle='None')
    art += plt.plot(point_list_x, point_list_0, marker='.', markersize=10,
color='red',linestyle='None')
    art += plt.plot(x1, diff(here_function_derivative(x_new),x1,x_, y_),
marker='.',markersize=1 ,color='pink',linestyle='None')

    artist_list.append(art)
    x_ = x_new

ani = ArtistAnimation(fig, artist_list, interval = 200)

plt.close()
HTML(ani.to_jshtml())
```



このように解を近似的に求める．(コードは動画)

ニュートン法の式

ニュートン法のイメージが掴めたので今回の更新式についても理解したい．
ニュートン法の式と今回使う式の対応を考えよう．
今回の更新式は以下

```
\boldsymbol{w}^{new}=\boldsymbol{w}^{old}-H^{-1}\nabla
E(\boldsymbol{w}^{old})
```

ニュートン法の式は以下

```
x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}
```

一見何が対応しているのかわからないが，整理していく．
H^{-1}はヘッセ行列(二階微分)の逆行列であり．$\frac{1}{f'(x_1)}$に対応する．$f'(x)^{-1}$と書くとわかりやすい．
$\nabla E(\boldsymbol{w}^{old})$はただの一階微分であり，$f(x_1)$に対応する．

今回は最終的にニュートン法で$\nabla E(\boldsymbol{w})\simeq 0$になるよう目指している．

次は,この更新式($\boldsymbol{w}^{new}=\boldsymbol{w}^{old}-H^{-1}\nabla E(\boldsymbol{w}^{old})$)に当てはまる具体的な数値を見ていく．

パラメータの更新式

パラメータのベクトル$\boldsymbol{w}$はすでにあるとする．(ニュートン法で求めるために初期値としてランダムに$\boldsymbol{w}$の値を設定するため．)

では，ここから以下のようなステップで進んでいく．

1. シグモイド関数を微分する．
2. $\nabla E(\boldsymbol{w})$を求める．($E(\boldsymbol{w})$の一階微分)
3. ヘッセ行列Hを求める．($E(\boldsymbol{w})$の二階微分)
4. 更新式に代入する．

1. シグモイド関数の微分

シグモイド関数の微分

$t=w^{T}x_i$とおく．

```
\frac{d}{dt}\sigma(t) = (1+e^{-t})^{-1}
```

ここで，$u=(1+e^{-t})$とおく．
合成関数の微分より

```
\begin{align}
\frac{d}{dt}\sigma(t)&=\frac{d}{du}u^{-1}\frac{du}{dt}u\\
&=-u^{-2}\frac{du}{dt}(1+e^{-t})\\
&=-(1+e^{-t})^{-2}\cdot-e^{-t}\\
\end{align}
```

補足としてe^{-t}の微分(わかっている人は次に進もう)

T=-tとおく
合成関数の微分より

```
\begin{align}
\frac{d}{dt}(e^{-t})&=\frac{d}{dT}(e^{T})\frac{dT}{dt}T\\
&=e^{T}\frac{d}{dt}(-t)\\
&=e^{-t}\cdot-1\\
&=-e^{-t}
\end{align}
```

さっきの途中から

```
\begin{align}
\frac{d}{dt}\sigma(t)&=-(1+e^{-t})^{-2}\cdot-e^{-t}\\
&=\frac{e^{-t}}{(1+e^{-t})^{2}}\\
&=\frac{e^{-t}}{1+e^{-t}}\frac{1}{1+e^{-t}}\\
```

```
&=(\frac{1+e^{-t}}{1+e^{-t}}-\frac{1}{1+e^{-t}})\frac{1}{1+e^{-t}}\\
\frac{d}{dt}\sigma(t)&=(1-\sigma(t))\sigma(t)
\end{align}
```

これでシグモイド関数の微分を求めることができた．(これは後々何回か出てくる．)

2. ∇E

$\nabla E(\boldsymbol{w}$)を求める．$(E(\boldsymbol{w})$の一階微分)

```
\nabla E(\boldsymbol w) = -\sum_{i=1}^n[y_i\nabla\log
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)+(1-y_i)\nabla\log (1-
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i))]
```

$\frac{d}{dt}\log t=\frac1t$より，$\log\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)$を微分すると以下

```
= -\sum_{i=1}^n[y_i\frac{1}
{\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)}\nabla\sigma(\boldsymbol{w}^{T
}\boldsymbol{x}_i)
+
(1-y_i)\frac{1}{1-\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)}\nabla (1-
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i))]
```

シグモイド関数の微分より

```
\begin{align}
&= -\sum_{i=1}^n[y_i\frac{1}
{\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)}\sigma(\boldsymbol{w}^{T}\bold
symbol{x}_i)(1-
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i))\boldsymbol{x}_i^{T}
+
(1-y_i)\frac{1}{1-\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)}(-
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)(1-
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)))\boldsymbol{x}_i^{T}]\\

&= -\sum_{i=1}^n[y_i(1-
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i))\boldsymbol{x}_i^{T}
+
(1-y_i)(-\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)\boldsymbol{x}_i^{T}]\\

&= -\sum_{i=1}^n[y_i-y_i\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)
-
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)+y_i\sigma(\boldsymbol{w}^{T}\bo
ldsymbol{x}_i)]\boldsymbol{x}_i^{T}\\

&= -\sum_{i=1}^n[y_i-
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)]\boldsymbol{x}_i^{T}\\
```

```
&= \sum_{i=1}^n(\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)-
y_i)\boldsymbol{x}_i^{T}\\
\end{align}
```

補足(あっているはず...)

```
\begin{align}
\frac{d}{d\boldsymbol{w}}\boldsymbol{w}^{T}\boldsymbol{x}
=\frac{d}{d\boldsymbol{w}}\boldsymbol{x}^{T}\boldsymbol{w}
&=
\frac{d}{d\boldsymbol{w}}
\begin{bmatrix}
x_1,&x_2,&\cdots&,x_m
\end{bmatrix}
\begin{bmatrix}
w_1\\ w_2 \\ \vdots\\ w_m
\end{bmatrix}\\
&=
\frac{d}{d\boldsymbol{w}}\begin{bmatrix}
x_1w_1+&x_2w_2+&\cdots&+x_mw_m
\end{bmatrix}\\
&=
\begin{bmatrix}
\frac{\partial}{\partial{w_1}}(x_1w_1+&x_2w_2+&\cdots&+x_mw_m)
,\frac{\partial}{\partial{w_2}}(x_1w_1+&x_2w_2+&\cdots&+x_mw_m)
,\cdots
,\frac{\partial}{\partial{w_m}}(x_1w_1+&x_2w_2+&\cdots&+x_mw_m)\\
\end{bmatrix}\\
&=
\begin{bmatrix}
x_1,&x_2,&\cdots&,x_m
\end{bmatrix}\\
&=
\boldsymbol{x}^{T}
\end{align}
```

$\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)-Y_i$はスカラーで $\boldsymbol{x}_i^{T}$はm次元の行ベクトルである．したがって

$\sum_{i=1}^n[\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)-y_i]\boldsymbol{x}_i^{T}$の意味は$\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)-y_i$倍された$\boldsymbol{x}_i^{T}$ベクトルを$n$について足し合わせたものである．

行列を使用することで$\sum_{i=1}^n[\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)-y_i]\boldsymbol{x}_i^{T}$の$\sum$を消すことができる．そうすることでもっとシンプルに表すことができる．$\hat{y}_i=\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)$とおいて．

```
\sum_{i=1}^n(\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)-
y_i)\boldsymbol{x}_i^{T
}=\sum_{i=1}^n(\hat{y}_i-y_i)\boldsymbol{x}_i^{T}\\
=\begin{bmatrix}
(\hat{y}_1-y_1)x_{11} & (\hat{y}_1-y_1)x_{12} & \cdots & (\hat{y}_1-
y_1)x_{1m}\\
&+&&\\
(\hat{y}_2-y_2)x_{21} & (\hat{y}_2-y_2)x_{22} & \cdots & (\hat{y}_2-
y_2)x_{2m}\\
&+&&\\
& \vdots &&\\
&+&&\\
(\hat{y}_n-y_n)x_{n1} & (\hat{y}_n-y_n)x_{n2} & \cdots & (\hat{y}_n-
y_n)x_{nm}\\
\end{bmatrix}
```

これを行列Xで表すと

```
\nabla E(\boldsymbol{w})=X^{T}(\boldsymbol{\hat{y}}-\boldsymbol{y})
```

これで一階微分を求めることができた.

3. H

ヘッセ行列Hを求める．($E(\boldsymbol{w})$の二階微分)

ヘッセ行列H(二階微分)

ヘッセ行列には厳密に説明があるが，今回は単純に「二階微分すればヘッセ行列ができる」とだけ意識して式変形していく． 以下の式をたてる.

```
H_{jk}=\frac{d}{dw_k}[\sum_{i=1}^{n}
((\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_{i})-y_i)x_{ij})]
```

- H_{jk}：ヘッセ行列のj行k列目の要素は右辺であることを表す.
- w_k：スカラーで$m×1$次元の$\boldsymbol{w}$ベクトルの要素を指す．(したがって$\frac{d}{dw_k}$は$\boldsymbol{w}$ベクトルのk番目の要素で微分するということ．)
- $\sum_{i=1}^{n}$：各行について計算しn行目まで足し合わせる．Xについて行方向について作用している.
- $(予測値-実測値)$×Xのi行j列目の要素

```
(\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_{i})-y_i)x_{ij}
```

これは行列で整理する前の一階微分の式と一緒．前は$\boldsymbol{x}^{T}$がついていたけど，今は$\boldsymbol{x}$のj番目を取り出しているので転置が外れているし，そもそもベクトルではなくスカラーになっている．

- 突然j,kが出てきたがこれはXのi行j,k列目を考えたいがために出てきた文字である．特別j,kであることには意味はないし，どちらも$\boldsymbol{x}$の「〜番目の要素」を指定したいだけなのでj,kを入れ替えても答えは一致する．j,kは区別されているだけのほぼ同じものと捉えて問題ない．

整理できたのでいよいよ微分していく．

```
\hat{y}=\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_{i})
```

```
g=\boldsymbol{w}^{T}\boldsymbol{x}_{i}\\とする．
```

合成関数の微分より，

```
\begin{align}
H_{jk}
&=\frac{d}{dw_k}[\sum_{i=1}^{n}
(\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_{i})x_{ij}-y_ix_{ij})]\\
&=\sum_{i=1}^{n}\frac{d}{dw_k}
[(\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_{i})x_{ij}-y_ix_{ij})]\\
&=\sum_{i=1}^{n}\frac{d}{d\hat{y}}\frac{d\hat{y}}{dg}\frac{dg}{dw_k}\\
\end{align}
```

順番に解いていく．

- $\frac{d}{d\hat{y}}$

```
\frac{d}{d\hat{y}}=\frac{d}{d\hat{y}}\hat{y}x_{ij}-\frac{d}
{d\hat{y}}y_ix_{ij}\\
=x_{ij}\\
```

- $\frac{d\hat{y}}{dg}$

```
\frac{d\hat{y}}{dg}
=\frac{d\hat{y}}{dg}\hat{y}\\
=\frac{d\hat{y}}{dg}\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)\\
=\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)(1-
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i))\\
```

- $\frac{dg}{dw_k}$

```
\frac{dg}{dw_k}=\frac{dg}{dw_k}g\\
=\frac{dg}{dw_k}\boldsymbol{w}^{T}\boldsymbol{x}_i\\
=\frac{dg}{dw_k}(w_1x_{i1}+w_2x_{i2}+\cdots+w_kx_{ik}+\cdots+w_mx_{im})\\
=x_{ik}
```

3つ全て求められた．合体する．

```
\begin{align}
H_{jk}
&=\sum_{i=1}^{n}\frac{d}{d\hat{y}}\frac{d\hat{y}}{dg}\frac{dg}{dw_k}\\
&=\sum_{i=1}^{n}x_{ij}\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i)(1-
\sigma(\boldsymbol{w}^{T}\boldsymbol{x}_i))x_{ik}
\end{align}
```

できた，しかし$\hat{y}$でおいた方がわかりやすいので置き換える．

```
H_{jk}=\sum_{i=1}^{n}x_{ij}\hat{y}(1-\hat{y})x_{ik}
```

ヘッセ行列のjk成分が求められた．
jk成分が求められたので今度はHそのもの，ヘッセ行列全体を求める．
jk成分を並べてヘッセ行列を求めてもいいのだが，すでにヘッセ行列は以下のように求められるとわかっている．
対角行列Rを以下とする．

```
R=
\begin{pmatrix}
  \hat{y}_1(1-\hat{y}_1)                    \\
    &\hat{y}_2(1-\hat{y}_2)                 \\
    &&\ddots     \\
    &&&\hat{y}_n(1-\hat{y}_n)
\end{pmatrix}
```

すると，ヘッセ行列は以下

```
H=X^{T}RX
```

これでヘッセ行列が求められた．

4. 更新式に代入

材料は以下．

$\nabla E(\boldsymbol{w})$($E(\boldsymbol{w})$の一階微分)

- $\nabla E(\boldsymbol{w})=X^{T}(\boldsymbol{\hat{y}}-\boldsymbol{Y})$

ヘッセ行列H(二階微分)

- $H=X^{T}RX$

更新式

- $\boldsymbol{w}^{new}=\boldsymbol{w}^{old}-H^{-1}\nabla E(\boldsymbol{w}^{old})$

式に代入していく

以下の式変形は逆行列を作る,単位行列を作る,結合法則しか利用していないので補足は抜く，自分でどこをいじってるのか確認してみると良いかも.

```
\begin{align}
\boldsymbol{w}^{new}
&=\boldsymbol{w}^{old}-H^{-1}\nabla E(\boldsymbol{w}^{old})\\
&=\boldsymbol{w}^{old}-(X^{T}RX)^{-1}X^{T}(\boldsymbol{\hat{y}}-
\boldsymbol{y})\\
&=(X^{T}RX)^{-1}X^{T}RX\boldsymbol{w}^{old}-(XRX^{T})^{-1}X^{T}RR^{-1}
(\boldsymbol{\hat{y}}-\boldsymbol{y})\\
&=(X^TRX)^{-1}(X^TR)[X\boldsymbol{w}^{old}-R^{-1}(\hat{y}-\boldsymbol{y})]
\end{align}
```

更新式が作れた.

実装

具体的な更新式が得られたので実装できる．やってみよう.

```
class LogisticRegression():
    def __init__(self, tol: float=0.0001, max_iter: int=100):
        self.tol = tol
        self.max_iter = max_iter

    @staticmethod
    def _sigmoid(Xw):
        return ...

    def fit(self, x, y):
        self.w = ...
        self.r = ...
        tol_vec = ...
        tol_= ...
        while ...:

    def predict(self, x):
```

```
        ...
        return y_pred
```

```
model = LogisticRegression(
    tol=0.0001,
    max_iter=100
)

start = time.time()
model.fit(X,y)
end = time.time()
time_diff = end - start
print('実行時間',time_diff)

print("coef_",model.w)
pred = model.predict(X)
print("accuracy",accuracy_score(y, pred))
```

参考文献

- 「ロジスティック回帰分析」,統計学WEB
- 「機械学習のエッセンス」,加藤公一