PYTHON機械学習プログラミング

補足資料

すうがくぶんか

　　　　　　　　　　　　　　　　　　　【目次】

[１　コスト関数の最小化 3](#_Toc468799048)

[（１）確率的降下法 3](#_Toc468799049)

[（２）最小二乗法 3](#_Toc468799050)

[（３）シミュレーション例 11](#_Toc468799051)

[２　ロジスティック回帰 15](#_Toc468799052)

[（１）ロジスティック関数 15](#_Toc468799053)

[（２）ロジスティック関数の微分 16](#_Toc468799054)

[（３）ベルヌーイ試行 16](#_Toc468799055)

[（４）尤度 17](#_Toc468799056)

[（５）まとめ 17](#_Toc468799057)

[３　ロジスティック関数 19](#_Toc468799058)

[４　交差エントロピー 20](#_Toc468799059)

# １　コスト関数の最小化

## （１）確率的降下法

ADALINEでは、全てのトレーニングサンプルからコスト関数の勾配を求め、コスト関数が小さくなる方向に向かって重みを更新していました。その為エポック毎に重みが更新されることになります。

これに対し、1トレーニングサンプル毎にコスト関数の勾配を求め、重みを更新してやる方法を[確率的勾配降下法](http://d.hatena.ne.jp/keyword/%B3%CE%CE%A8%C5%AA%B8%FB%C7%DB%B9%DF%B2%BC%CB%A1)と言います。

この方法を用いると1つのトレーニングサンプルが得られた時に、リアルタイムで重みを更新することができます。

最新の結果をリアルタイムで反映することができるので、オンライン勾配降下法とも言います。

また、全トレーニングサンプルからコスト関数の重みを更新する従来の方法はバッチ勾配降下法と呼ばれています。

※以下の式は全て行列で表記しています。大文字で太字は行列、小文字で太字は縦ベクトルを表します。

**バッチ勾配降下法** (全トレーニングサンプルを使って重みを更新)

Δ**w**=−η(−α**X**T**y**+α2 **X**T **Xw**)

[**確率的勾配降下法**](http://d.hatena.ne.jp/keyword/%B3%CE%CE%A8%C5%AA%B8%FB%C7%DB%B9%DF%B2%BC%CB%A1) (1トレーニングサンプルを使って重みを更新)

Δ**w**=−η(−α**y**i+α 2 **x**i**w**)

xi はi 番目のトレーニングサンプルを表しており、

xi =[1  x i1  xi2  ⋯ x in  ]

です。

## （２）最小二乗法

ADALINEのコスト関数は、**w** に対して単純な2次関数になっており、コスト関数の最小値を与える**w** が1組だけ存在します。

最小値におけるコスト関数の勾配が0になることから、勾配降下法のような逐次計算を行わなくても、実は一発でコスト関数の最小値を与える**w** を求める事が出来ます。

この方法を最小二乗法と言います。

=−α**X**T**y**+α 2**X**T **Xw**=0

∴   **w**= (**X**T**X**) −1 **X** T**y**

最小二乗法では一発でコスト関数の最小値を与える**w** が求まるので一番いい方法なのですが、もっと複雑な関数の場合は使えません。限られた場合にしか使えないことに注意してください。

前回のADALINEのコードを少し変更して、[確率的勾配降下法](http://d.hatena.ne.jp/keyword/%B3%CE%CE%A8%C5%AA%B8%FB%C7%DB%B9%DF%B2%BC%CB%A1)がどのような振る舞いをするのか見ていきたいと思います。

[確率的勾配降下法](http://d.hatena.ne.jp/keyword/%B3%CE%CE%A8%C5%AA%B8%FB%C7%DB%B9%DF%B2%BC%CB%A1)では1トレーニングサンプル毎に重みが更新されるので、トレーニングサンプルの順番をランダムにシャッフルしてやった方が学習効率は上がります。この機能もつけようと思います。

また、最小二乗法で一発で求めた解とも比較してみます。

# -\*- coding: utf-8 -\*-

import sys

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

def main():

# ---アヤメデータの取得

df = pd.read\_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data", header=None)

y = df.iloc[0:100, 4].values

y = np.where(y == "Iris-setosa", -1, 1)

X = df.iloc[0:100, [0, 2]].values

# ---学習実施(学習率:0.01 最大エポック数:20 (特徴量を標準化,シャッフルオフ,バッチ勾配降下法))

Xstd = np.copy(X)

Xstd[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()

Xstd[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()

ada = Adaline(eta=0.01, numEpoch=20, shuffle=False)

ada.fit(Xstd, y)

plotResult(ada,Xstd,y,"Standardization","Off","GD")

# ---学習実施(学習率:0.01 最大エポック数:20 (特徴量を標準化,シャッフルオフ,確率的勾配降下法))

Xstd = np.copy(X)

Xstd[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()

Xstd[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()

ada = Adaline(eta=0.01, numEpoch=20, shuffle=False)

ada.fitSGD(Xstd, y)

plotResult(ada, Xstd, y, "Standardization", "Off", "SGD")

# ---学習実施(学習率:0.01 最大エポック数:20 (特徴量を標準化,シャッフルオン,確率的勾配降下法))

Xstd = np.copy(X)

Xstd[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()

Xstd[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()

ada = Adaline(eta=0.01, numEpoch=20, shuffle=True)

ada.fitSGD(Xstd, y)

plotResult(ada, Xstd, y, "Standardization", "On", "SGD")

# ---学習実施(学習率:0.01 最大エポック数:20 (特徴量を標準化,シャッフルオン,最小二乗法))

Xstd = np.copy(X)

Xstd[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()

Xstd[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()

ada = Adaline(eta=0.01, numEpoch=20, shuffle=True)

ada.fitLS(Xstd, y)

plotResult(ada, Xstd, y, "Standardization", "On", "LS")

def plotResult(ada,X,y,trainSampleState,shuffeled,fitAlgorism):

#---プロットのプロパティ

markers = ('s', 'o', 'x', '^', 'v')

colors = ('green', 'yellow','red', 'blue', 'lightgreen', 'gray', 'cyan')

cmap = ListedColormap(colors[:len(np.unique(y))])

labels = ('setosa', 'versicolor')

fig = plt.figure(figsize=(12, 10))

plt.clf()

ax1 = fig.add\_subplot(221)

ax2 = fig.add\_subplot(222)

ax3 = fig.add\_subplot(223)

ax4 = fig.add\_subplot(224)

# ---エポック数と重みのプロット

epochTimes = np.array(range(0, ada.numEpoch + 1))

ada.W\_ = np.array(ada.W\_)

ax1.plot(epochTimes, ada.W\_[:, 0].T[0], color="blue", label="w0")

ax1.plot(epochTimes, ada.W\_[:, 1].T[0], color="red", label="w2")

ax1.plot(epochTimes, ada.W\_[:, 2].T[0], color="green", label="w1")

ax1.legend(loc="upper left")

ax1.set\_xlabel('Epochs')

ax1.set\_ylabel('Weight')

ax1.set\_title("Epochs and weight")

ax1.set\_xlim(0, ada.numEpoch)

# ---ax1.set\_xticks(range(0,numEpoch+1,1))

ax1.grid()

# ---エポック数と正答率のプロット

epochTimes = range(1, ada.numEpoch + 1)

ax2.plot(epochTimes, np.array(ada.correctAnswerRateEachEpoch\_) \* 100, color="blue", marker="o",

label="Correct answer rate")

# ---ax2.legend(loc="upper left")

ax2.set\_title("Epochs and correct answer rate")

ax2.set\_xlabel('Epochs')

ax2.set\_ylabel('Correct answer rate[%]')

ax2.set\_xlim(0, ada.numEpoch)

ax2.grid()

plt.xlim(0, ada.numEpoch)

# ---エポック数とコスト関数のプロット

epochTimes = range(1, ada.numEpoch + 1)

ax3.plot(epochTimes, np.array(ada.J\_)[:, 0][:, 0], color="blue", marker="o",

label="Cost function values")

# ---ax3.legend(loc="upper left")

ax3.set\_title("Epochs and cost function values")

ax3.set\_xlabel('Epochs')

ax3.set\_ylabel('Cost function values')

ax3.set\_xlim(0, ada.numEpoch)

ax3.grid()

# ---ax3.set\_xticks(range(0,numEpoch+1,1))

# ---決定領域のプロット

x1\_min, x1\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

x2\_min, x2\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

dx = 0.02

X1 = np.arange(x1\_min, x1\_max, dx)

X2 = np.arange(x2\_min, x2\_max, dx)

X1, X2 = np.meshgrid(X1, X2)

Z = ada.predict(np.array([X1.ravel(), X2.ravel()]).T)

Z = Z.reshape(X1.shape)

# ---決定領域

ax4.contourf(X1, X2, Z, alpha=0.5, cmap=cmap)

ax4.set\_xlim(X1.min(), X1.max())

ax4.set\_ylim(X2.min(), X2.max())

# ---アヤメデータ

for idx, cl in enumerate(np.unique(y)):

ax4.scatter(x=X[y == cl, 0], y=X[y == cl, 1],

alpha=1.0, c=cmap(idx),

marker=markers[idx], label=labels[idx])

ax4.set\_title("Decision regions")

ax4.set\_xlabel("Sepal length [cm]") # がく片の長さ

ax4.set\_ylabel("Petal length [cm]") # 花びらの長さ

ax4.legend(loc="upper left")

ax4.grid()

pngFileName=u"ADALINE学習結果\_トレーニングサンプル" +trainSampleState +u"\_シャッフル"+shuffeled+u"\_Fitアルゴリズム"+fitAlgorism+ u"\_学習率"+str(ada.eta)+u"\_最大エポック数"+str(ada.numEpoch)+ ".png"

plt.savefig(pngFileName, dpi=300)

class Adaline(object):

def \_\_init\_\_(self, alpha=1.0, eta=0.01, numEpoch=10, shuffle=True, random\_state=None):

self.alpha = alpha # 活性化関数の定数

self.eta = eta #学習率

self.numEpoch = numEpoch #最大エポック数

self.shuffle = shuffle #トレーニングサンプルをシャッフルする？

if random\_state:

np.random.seed(random\_state)

self.W\_=[] #各エポックごとに重みを保存するリスト

self.J\_=[] #各エポックごとにコスト関数を保存するリスト

self.correctAnswerRateEachEpoch\_=[] #各エポックごとに正答率を保存するリスト

def \_\_actFunc(self, z):

"""活性化関数(\_\_でプライベート関数)"""

return self.alpha\*z

def \_\_quantizer(self, phi):

"""量子化器(\_\_でプライベート関数)"""

return np.where(np.array(phi) >= 0.0, 1, -1)

def \_\_shuffle(self, X, y):

"""トレーニングサンプルのシャッフル(\_\_でプライベート関数)"""

r = np.random.permutation(len(y))

return X[r], y[r]

def predict(self, X):

"""予測関数"""

X = np.matrix(X)

m, n = X.shape

X = np.c\_[np.matrix(np.ones((m, 1))), X]

z=X\*self.w\_

phi=self.\_\_actFunc(z)

return self.\_\_quantizer(phi)

def fit(self, X, y):

"""バッチ勾配降下法による学習の実施"""

# ---トレーニングサンプルのシャッフル

if self.shuffle:

X, y = self.\_\_shuffle(X, y)

# ---特徴行列

X = np.matrix(X)

m, n = X.shape

# ---しきい値用に一番左側の列に1を追加

X = np.c\_[np.matrix(np.ones((m, 1))), X]

# ---教師データベクトル

y = np.matrix(y).T

# ---重みベクトル(初期化)

self.w\_ =np.matrix(np.zeros((n+1,1)))

# 重みを学習回数ごとに保存

self.W\_.append(self.w\_.tolist())

for indexEpoch in range(1,self.numEpoch+1): #エポックループ

correctAnswer=[] #各トレーニングセットに対する教師データと予測値の正誤表をイニシャライズ

# ---特徴行列と重みベクトルの掛け合わせ

z = X \* self.w\_

# ---活性化関数出力ベクトル

phi =self.\_\_actFunc(z)

# ---教師データと活性化関数出力の残差ベクトル

e = y - self.alpha \* X \* self.w\_

# ---コスト関数

J = e.T \* e / 2

# ---コスト関数の勾配

gradJ = -self.alpha \* X.T \* y + self.alpha \*\* 2 \* X.T \* X \* self.w\_

# ---重み更新

dw = - self.eta \* gradJ

self.w\_ = self.w\_ + dw

# 重みをエポックごとに保存

self.W\_.append(self.w\_.tolist())

# コスト関数をエポックごとに保存

self.J\_.append(J.tolist())

#--正答表

correctAnswer = np.where(np.array(y == self.\_\_quantizer(phi)) == True, 1, 0)

#各エポックごとに正誤表から正答率算出し保存

self.correctAnswerRateEachEpoch\_.append(float(sum(correctAnswer))/len(correctAnswer))

def fitSGD(self, X, y):

"""確率的勾配降下法による学習の実施"""

# ---トレーニングサンプルのシャッフル

if self.shuffle:

X, y = self.\_\_shuffle(X, y)

# ---特徴行列

X = np.matrix(X)

m, n = X.shape

# ---しきい値用に一番左側の列に1を追加

X = np.c\_[np.matrix(np.ones((m, 1))), X]

# ---教師データベクトル

y = np.matrix(y).T

# ---重みベクトル(初期化)

self.w\_ = np.matrix(np.zeros((n + 1, 1)))

# 重みを学習回数ごとに保存

self.W\_.append(self.w\_.tolist())

for indexEpoch in range(1, self.numEpoch + 1): # エポックループ

correctAnswer = [] # 各トレーニングセットに対する教師データと予測値の正誤表をイニシャライズ

for xi, yi in zip(X, y): # トレーニングセットループ

# ---コスト関数の勾配

gradJi = -self.alpha \* xi.T \* yi + self.alpha \*\* 2 \* xi.T \* xi \* self.w\_

# ---重み更新

dwi = - self.eta \* gradJi

self.w\_ = self.w\_ + dwi

# ---特徴行列と重みベクトルの掛け合わせ

z = X \* self.w\_

# ---活性化関数

phi = self.\_\_actFunc(z)

# ---教師データと活性化関数出力の残差ベクトル

e = y - self.alpha \* X \* self.w\_

# ---コスト関数

J = e.T \* e / 2

# ---コスト関数の勾配

gradJ = -self.alpha \* X.T \* y + self.alpha \*\* 2 \* X.T \* X \* self.w\_

# # ---重み更新

# dw = - self.eta \* gradJ

# self.w\_ = self.w\_ + dw

# 重みをエポックごとに保存

self.W\_.append(self.w\_.tolist())

# コスト関数をエポックごとに保存

self.J\_.append(J.tolist())

# --正答表

correctAnswer = np.where(np.array(y == self.\_\_quantizer(phi)) == True, 1, 0)

# 各エポックごとに正誤表から正答率算出し保存

self.correctAnswerRateEachEpoch\_.append(float(sum(correctAnswer)) / len(correctAnswer))

def fitLS(self, X, y):

"""最小二乗法による学習の実施"""

# ---トレーニングサンプルのシャッフル

if self.shuffle:

X, y = self.\_\_shuffle(X, y)

# ---特徴行列

X = np.matrix(X)

m, n = X.shape

# ---しきい値用に一番左側の列に1を追加

X = np.c\_[np.matrix(np.ones((m, 1))), X]

# ---教師データベクトル

y = np.matrix(y).T

# ---重みベクトル(初期化)

self.w\_ = np.matrix(np.zeros((n + 1, 1)))

# 重みを学習回数ごとに保存

self.W\_.append(self.w\_.tolist())

for indexEpoch in range(1, self.numEpoch + 1): # エポックループ

correctAnswer = [] # 各トレーニングセットに対する教師データと予測値の正誤表をイニシャライズ

# ---コスト関数を最小にする重みを一発で算出

self.w\_ = np.linalg.inv(X.T\*X)\*X.T\*y /self.alpha

# ---特徴行列と重みベクトルの掛け合わせ

z = X \* self.w\_

# ---活性化関数

phi = self.\_\_actFunc(z)

# ---教師データと活性化関数出力の残差ベクトル

e = y - self.alpha \* X \* self.w\_

# ---コスト関数

J = e.T \* e / 2

# 重みをエポックごとに保存

self.W\_.append(self.w\_.tolist())

# コスト関数をエポックごとに保存

self.J\_.append(J.tolist())

# --正答表

correctAnswer = np.where(np.array(y == self.\_\_quantizer(phi)) == True, 1, 0)

# 各エポックごとに正誤表から正答率算出し保存

self.correctAnswerRateEachEpoch\_.append(float(sum(correctAnswer)) / len(correctAnswer))

if \_\_name\_\_ == "\_\_main\_\_":

main()

[view raw](https://gist.github.com/darden1/19427cf418bbf8c1fe8572a4627f7528/raw/9f3f90c0e22e379761023e67f2980100402ca5ba/StudyMachineLearning_AdalineSGD.py) [StudyMachineLearning\_AdalineSGD.py](https://gist.github.com/darden1/19427cf418bbf8c1fe8572a4627f7528#file-studymachinelearning_adalinesgd-py) hosted with ❤ by [GitHub](https://github.com/)

Adalineクラスにトレーニングサンプルをシャッフルするプライベートメソッド\_\_shuffleを追加してあります。

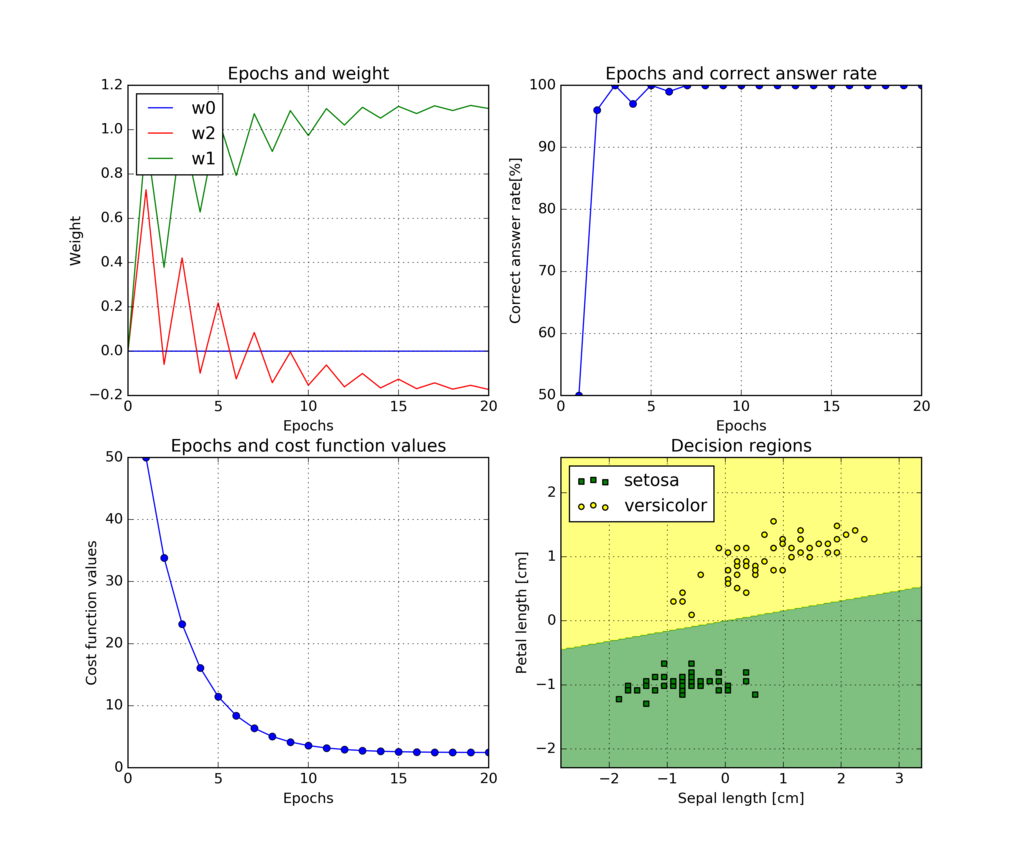
また、[確率的勾配降下法](http://d.hatena.ne.jp/keyword/%B3%CE%CE%A8%C5%AA%B8%FB%C7%DB%B9%DF%B2%BC%CB%A1)による学習実施メソッドfitSGDと、最小二乗法による学習メソッドfitLSを追加しました。

この[スクリプト](http://d.hatena.ne.jp/keyword/%A5%B9%A5%AF%A5%EA%A5%D7%A5%C8)を実行すると以下4つの学習結果をグラフで出力します。

## （３）シミュレーション例

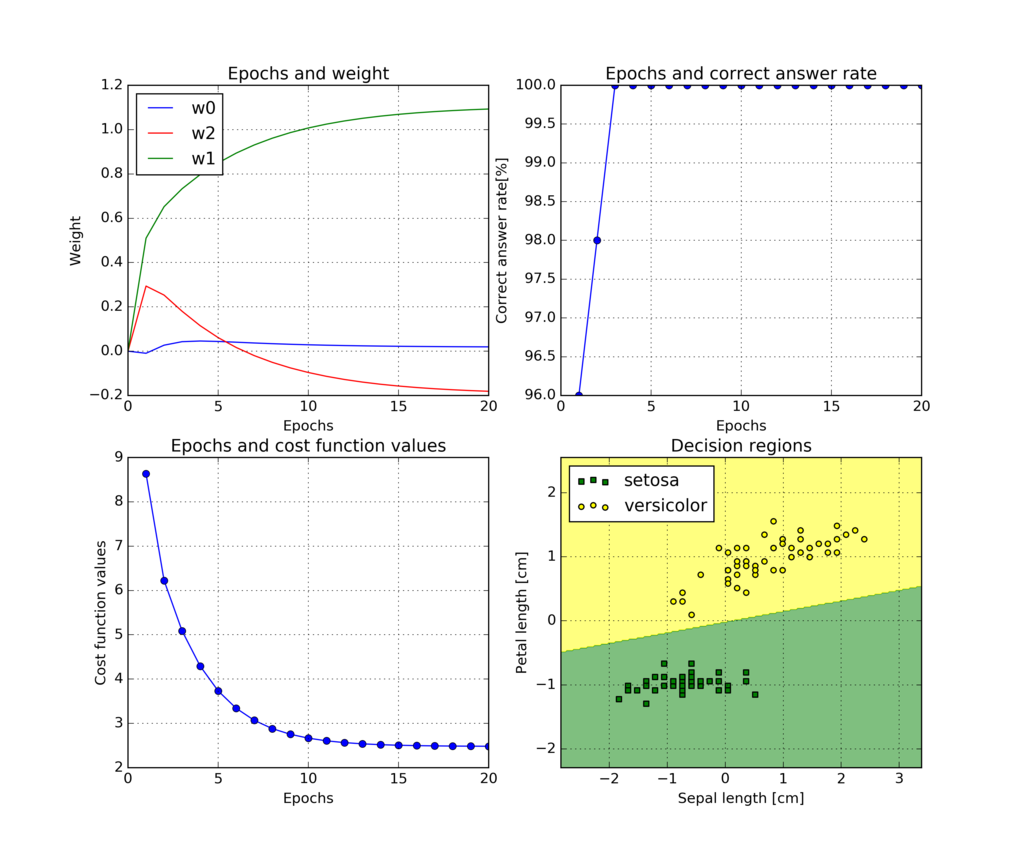
**バッチ勾配効果法の結果(学習率:0.01 最大エポック数:20 特徴量を標準化:オン トレーニングサンプルシャッフル:オフ)**

バッチ勾配降下法の結果です。比較用に出力しています。



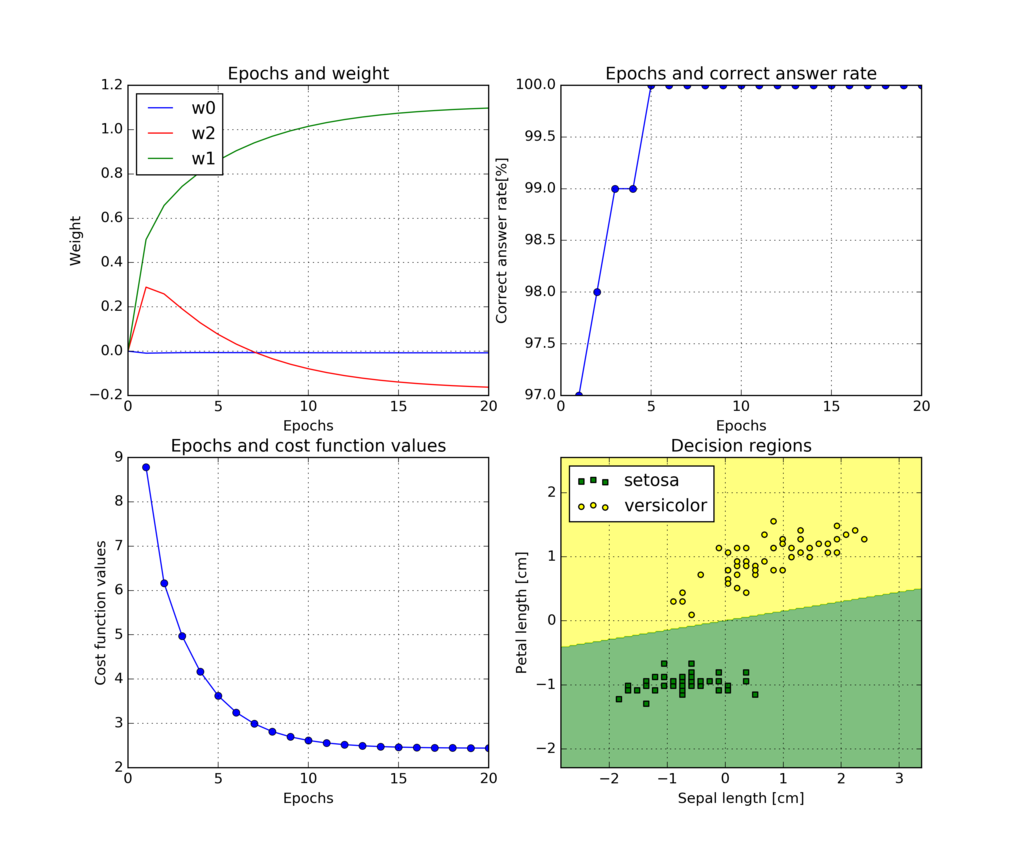
**確率的勾配効果法の結果(学習率:0.01 最大エポック数:20 特徴量を標準化:オン トレーニングサンプルシャッフル:オフ)**

バッチ勾配効果法に比べ、1エポック目からコスト関数がずいぶん小さくなっているのがわかります。1エポック目の正答率も96%と高くなっています。



**確率的勾配効果法の結果(学習率:0.01 最大エポック数:20 特徴量を標準化:オン トレーニングサンプルシャッフル:オン)**

トレーニングサンプルをシャッフルする事でもっと学習効率が上がるかと思いましたが、そんなに大したことなかったですね。でも1エポック目の正答率が97%とシャッフルしなかった時より高くなっているので少しは効果があります。

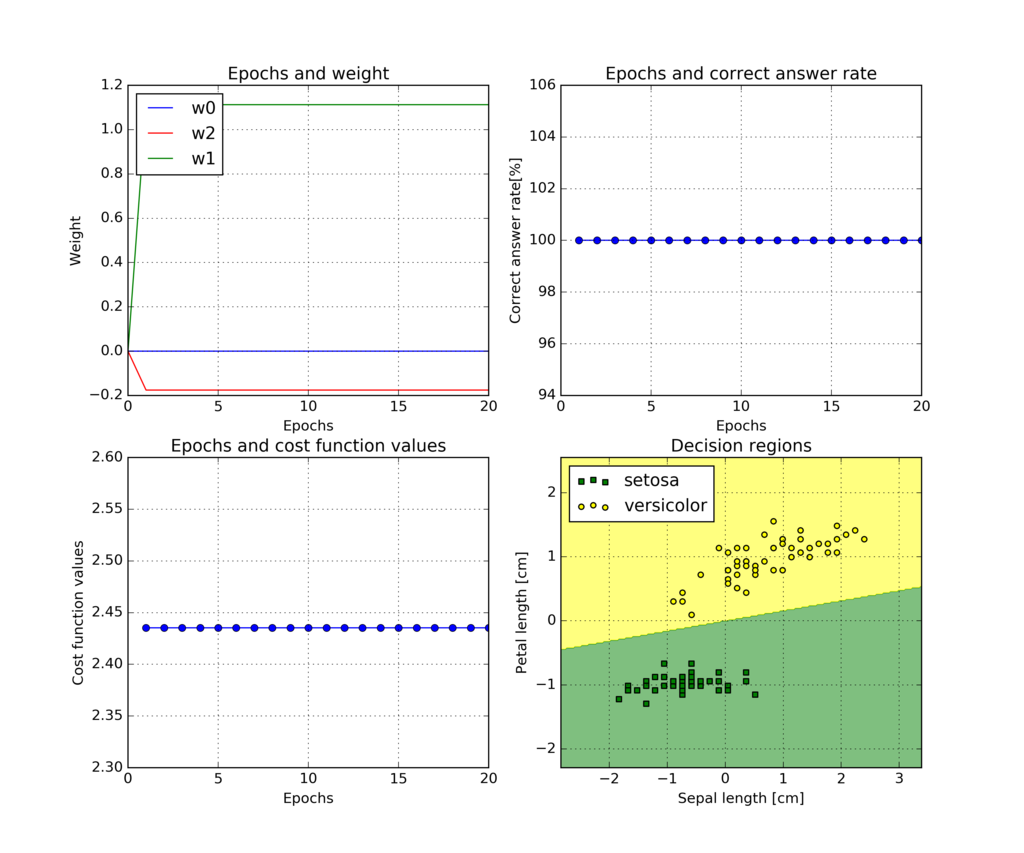


**最小二乗法の結果(学習率:0.01 最大エポック数:20 特徴量を標準化:オン トレーニングサンプルシャッフル:オン)**

さすが最小二乗法ですね。

一発で最小となるコスト関数が求まっています。最小二乗法では、学習率、エポック数、特微量の標準化、トレーニングサンプルのシャッフルに関係なく、一発でコスト関数の最小値が求まります。

ちなみにコスト関数の最小値は2.435です。



コスト関数が重み**w**で[微分](http://d.hatena.ne.jp/keyword/%C8%F9%CA%AC)可能で、極小値がただ一つだけ存在する場合は最小二乗法を用いるのがいいです。

しかしながら、特微量行列の転置と特微量行列の積の[逆行列](http://d.hatena.ne.jp/keyword/%B5%D5%B9%D4%CE%F3)(**X** T**X**) −1  を求める必要があるのでトレーニングサンプルが非常に多い場合は、計算コストが増大しメモリ不足で[逆行列](http://d.hatena.ne.jp/keyword/%B5%D5%B9%D4%CE%F3)が計算できなくなってしまう恐れがあります。

実際はトレーニングサンプルが非常に多くなる場合がほとんどで、勾配降下法で逐次的にコスト関数の最小値を求める方がはるかに計算コストが少なくなる為、一般的にはあまり使われていないみたいです。

# ２　ロジスティック回帰

ロジスティック回帰はADALINEに確率的な解釈を与えたアルゴリズムです。

活性化関数とコスト関数に対し、ロジスティック関数と尤度を用いています。ADALINEとの違いは、この活性化関数とコスト関数だけです。

ここでは、ロジスティック関数と尤度の概念を学んでいきたいと思います。

## （１）ロジスティック関数

トレーニングサンプルと重みをかけ合わせた値( **z**=**Xw** )を、ロジスティック関数に入力し活性化関数の値として出力します。

ϕ(z)=

ロジスティック関数ϕ(z) は、－∞～∞までの全実数入力z に対し、0～1までの値を出力します。

トレーニングサンプル**X** が与えられた場合に、教師データ**y**を与える条件付き確率として解釈されます。

確かに、ロジスティック関数の出力は0～1までの値になるので確率として解釈できそうですが、その概念は少々難解です。この概念について少し堀り下げてみようと思います。

ロジスティック関数はその形からシグモイド関数とも呼ばれています。－∞～＋∞までの実数入力を受けて、0～1までの実数を出力する関数です。

今1組のトレーニングサンプルx i が与えられた時、クラスラベルy i が1となる条件付き確率をp とします。

p=p(y i =1|x i )

この時オッズ比r を以下のように定義します。オッズ比は[0,1]の定義域をとり、[0,∞)の値域を取ります。

r=（）

オッズ比の対数を取ったものはロジット関数と呼ばれています。対数を取ることで、[0,1]の定義域に対し、値域が(-∞,+∞)に拡張されます。

z=log

このロジット関数の逆関数がロジスティック関数となります。ロジスティック関数の定義域は(-∞,+∞)で値域は[0,1]になります。

p=

ロジスティック関数の出力値が確率を表すことが、なんとなくわかりました。

ではなぜオッズ比とロジット関数を考える必要があるかというと、定義域が(-∞,+∞)であるロジスティック関数の入力z を、トレーニングサンプルと重みの積の和**x** i **w** に対応させるためです。

少々強引な気もしますが、**w** は、トレーニングサンプル**x** i に対し、クラスラベル**y** i が1となる条件付き確率分布関数を決定するパラメータになっていると解釈できるようになるわけです。

## （２）ロジスティック関数の微分

コスト関数の勾配を求める際に、ロジスティック関数を微分してやる必要があるので、ここで確認しておきます。

ロジスティック関数ϕ(z) をz で微分してみます。



面白い性質を持っていますね。

## （３）ベルヌーイ試行

尤度を考える前にベルヌーイ試行というものを知っておいたほうがいいでしょう。

ベルヌーイ試行とはクラスラベルが1か0の試行で、コイン投げの試行等がそれに当たります。

クラスラベルがy i =1である確率がϕ i であるとき、その条件付き確率の確率分布関数は以下の式で表すことができます。

p(y i |ϕ i )= (1−ϕ i ) 1−y i

クラスラベルが1の時は、

p(y i =1|ϕ i )=ϕ i

クラスラベルが0の時は、

p(y i =0|ϕ i )=1−ϕ i

となり、一つの式で1と0のクラスラベルの確率分布を同時に表しています。

この式で定義される確率分布関数をベルヌーイ分布といいます。

ここで、ϕ i という確率は、クラスラベルがy i =1 である確率を表していますが、ϕ i は確定した量ではなく、パラメータ**w** をもつ確率分布関数に従っていることに注意してください。

例えばコイン投げにしても本当に表が出る確率は1/2でしょうか？(ひん曲がったコインを投げた場合、表が出る確率が1/2かどうかを考えるといいと思います。)

つまり、表が出る確率はあるパラメータ**w** を持つ、確率分布関数に従って変動すると考えます。

言い換えると、ここでは表が出る確率が1/2である確率。つまり確率の確率を考えています。

パラメータを考慮した場合、ベルヌーイ分布を表す式に以下のようにパラメータを考慮しているということを明示的に書いてやります。

p(y i |ϕ i ;**w**)= (1−ϕ i ) 1−y i

## （４）尤度

尤度とはある事象が観測された時、それぞれの事象の確率の確率分布関数のパラメータがどれだけ尤もらしいかを測る量で、事象の同時確率として定義されます。

l(w) =p(y 1 |ϕ 1 ;**w**)p(y 2 |ϕ 2 ;**w**)⋯ ⋯p(y m |ϕ m ;**w**)= p(y i |ϕ i ;**w**)

ひん曲がったコインを投げた時、表が出る確率が1/2になるかどうか分かりません。

表が出る確率を調べるためには、何回もコインを投げてどれくらいの頻度で表が出るかという事象を調べてやる必要があります。

表が出る確率がベルヌーイ分布に従っていると仮定し、観測された事象からパラメータ**w** を推定する事で、表が出る確率が分かるわけです。

尤度が最大値を与えるパラメータ**w** は、事象の確率が一番尤もらしい値となります。

このように尤度を最大化することで、一番尤もらしい事象の確率を求めてやることを最尤推定法と呼びます。

尤度の対数を取ったものは対数尤度と呼ばれており、尤度が最大になる時、対数尤度も最大値を取ります。

対数法則により掛け算が足し算になるので、対数尤度で評価した方が扱いが簡単になるという利点があります。

L(**w**)=- log[ p(y i |ϕ i ;**w**) ]

## （５）まとめ

ベルヌーイ分布を使ってコスト関数を表してやるので、教師データのクラスラベルは1と0とします。

i 番目のトレーニングサンプルと教師データのセットが得られた時、活性化関数を以下の様に表します。活性化関数の出力は教師データのクラスラベルが1である確率を表しています。

ϕ i (z i )= 

**z** i =**x** i **w**

コスト関数は対数尤度にマイナスを付けた値を用います。(コスト関数が最小値の時に、対数尤度は最大値となります。)

J(**w**)=−  [ y i logϕ i +(1−y i )log(1−ϕ i ) ]

コスト関数をj 番目の重みw j で微分して、j 番目のコスト関数の勾配を求めてみます。

=− [ y i logϕ i +(1−y i )log(1−ϕ i ) ]

=− [ y i +(1−y i ) ]

=− [ y i ϕ i (1−ϕ i )x ij +(1−y i )ϕ i (1−ϕ i )xij ]

=− (y i −ϕ i )x ij

重みの更新は、従来通りコスト関数の勾配に学習率η をかけ、マイナスをつけて足し込みます。

Δw j =−η(− (y i −ϕ i )x ij ) w j :=w j +Δw j

**まとめ(行列による表記)**

numpyでコーディングしやすいように行列表記しておきます。

**トレーニングサンプルと重みの積の和**

**z**=**Xw**

**活性化関数**

ϕ(z)= 

**コスト関数**

J(**w**)=−**y** T logϕ−(**1**−**y**) T log(**1**−ϕ)

**コスト関数の勾配**

∂J(**w**)/∂**w** =−**X** T (**y**−ϕ)

**重みの更新**

Δ**w**=−η(−**X** T (**y**−ϕ))

**w**:=**w**+Δ**w**

**コスト関数の最小値を与える重み**

コスト関数の勾配を0と置いて式をいじってみたところ、式の上ではコスト関数が最小となる重みが一発で求まりそうだったのでこちらも試してみようかと思います。

−**X** T (**y**−ϕ)=0 が常に成り立つためには、**y**=ϕ です。

ϕを与える**z** は、ロジット関数より求まるので、

z=log

式を展開します。

**Xw**＝logϕ−log(**1**−ϕ)

両辺に**X T**をかけ、ϕ=**y** より、ϕ を消去します。

**X** T **Xw**=**X** T (log**y**−log(**1**−**y**))

両辺に(**X** T **X**) −1 をかけてやると、

**w**=(**X** T **X**) −1 **X** T (log**y**−log(1−**y**))

ADALINEの最小二乗法の時と似てますね。

# ３　ロジスティック関数



# ４　交差エントロピー

