目的

パーセプトロンのアルゴリズムの改良版であるADALINE(ADAptive LInear NEuron)についてその学習アルゴリズムとコスト最小化について学び、後のロジスティック回帰やSVM等について深く学ぶための基礎知識を得る

目次

1. ADALINEについて

ADALINEはパーセプトロンが発表された数年後に発表された単一層ニューラルネットワークである. パーセプトロンとの主な違いは、パーセプトロンが重みの更新に単位ステップ関数を用いていたのに対して、ADALINEは活性化関数を使用して誤差を計算する. つまり、モデル誤差の計算と重みの更新に2値のクラスラベルを使用するのではなく、線形活性化関数から連続値の出力を使用する.最終的な出力は量子化気器(単位ステップ関数のようなもの)を通してクラスラベルを予測する.

2. 勾配降下法によるコスト関数の最小化

1. 重みの学習にコスト関数**J**を用いる.
    - 学習過程で最適化することが目標(コスト関数cost function)
2. コスト関数はパーセプトロンと同様に計算結果と正解ラベルとの誤差平方和(SSE: Sum of Squared Error)から求められる

$$\mathbf{J}(\mathbf{w}) = \frac{1}{2}\sum_{i}(y^{(i)} - \varphi(z^{(i)}))^2$$

    - wは重みを示す.
    - コスト関数J(w) は 訓練サンプルごとの正解ラベルyi と 線形活性化関数φを通した計算結果zi の差を2乗した和をとる.
3. パーセプトロンでは単位ステップ関数を使っていたため、計算結果zi は離散値(1/0)だった. しかし、線形活性化関数を用いることで連続値で計算結果を得ることが出来る. これによりコスト関数が微分可能になる.
4. コスト関数のもうひとつの利点はこの関数が凸関数になることである.このため、グラフの最大値/最小値を勾配降下法(gradient descent)を使うことでコスト関数を最小化する重みを見つけ出すことが出来る.

> 勾配降下法(バッチ勾配降下法) 学習率ηに負の勾配を掛けてコスト関数(重み)を計算する.
>
> $$\Delta\mathbf{w} = -\eta\nabla\mathbf{J}(\mathbf{w})$$
>
> 重みを一回更新するために全データセットの勾配を計算する.

3. Python における ADALINEの実装

パーセプトロンのクラスのfitメソッドを置き換えて実装する. 前述の通り、パーセプトロンとの差は活性化関数φ(w)を用いて重みを修正することである.(pp.35-36参照)

In [68]:

```
class AdalineGD(object):
    """
    ADAptive LInear NEuron 分類器

    パラメータ
    eta : float 学習率
    n_iter : int 訓練データの訓練回数

    w_ : 1次元配列 適合後の重み
    errors_ : list() 各エポック(トレーニング回数)での誤分類数
    """


    def __init__(self, eta=0.01, n_iter=50):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """訓練データに適合させる

        パラメータ
        X: {配列のようなデータ構造}, shape = {n_samples, n_features} 訓練データ
        y : 配列のようなデータ構造, shape = {n_samples} 目的変数(正解データ)

        返り値
        self.obj
        """
        self.w_ = np.zeros(1 + X.shape[1]) # 訓練データのサンプル数 + 1（結果）の0ベクトルを作る 0番
        self.cost_ = []

        for i in range(self.n_iter): # トレーニング回数分回す
            # 活性化関数の出力計算 φ(W^T x) = W^T x
            output = self.net_input(X) # Xと重みの総入力
            # 誤差の計算 y - φ(z)
            errors = (y - output)
            # 重みの更新
            # ΔW = η * Sigma(y - φ(z))^2 * Xj
            # X.T.dot(error)は 特徴行列X と 誤差ベクトルerrors との行列ベクトル積である.
            self.w_[1:] += self.eta * X.T.dot(errors)
            # w0の更新
            self.w_[0] += self.eta * errors.sum()
            # コスト関数の計算 J(W) = 1/2 * Sigma(error)^2
            cost = (errors**2).sum() / 2.0
            # コストの格納
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        """
        総入力を計算
        """
        return np.dot(X, self.w_[1:] + self.w_[0])

    def predict(self, X):
        """
        1ステップ後のクラスラベルを返す
        """
        # where で指定した条件に当てはまるインデックスを返す
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

収束に適した学習率ηを見つけるためには、何回か実験を行う必要がある。始めに学習率η=0.01とη=0.0001で計算してみる.その結果からどの程度ADALINEが効率よく学習するのかを確かめる。

In [19]:

```
# Irisデータセットを取得する
import pandas as pd
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data',engine='
df.tail()
```

Out[19]:

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

In [20]:

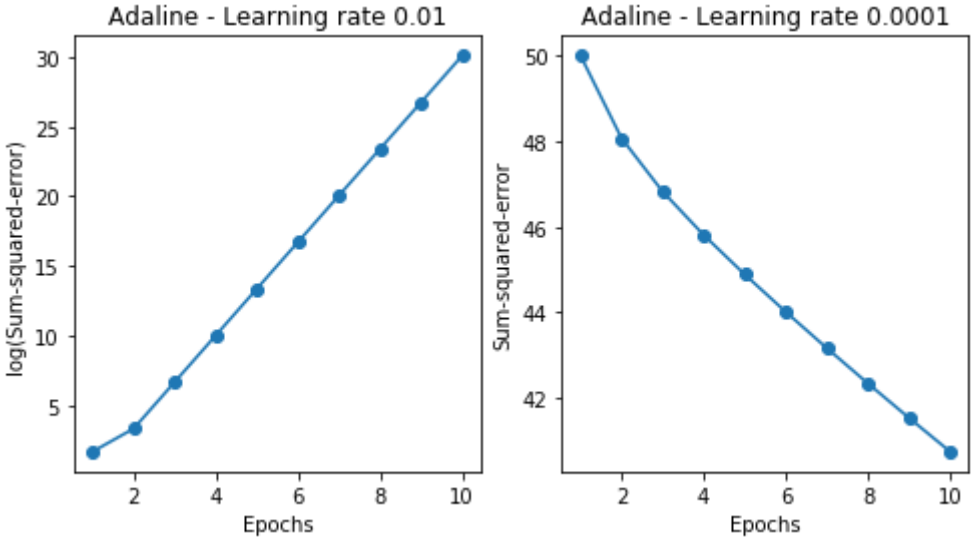
```
# データの整形

# 1-100行目の目的変数(正解ラベル)の抽出
y = df.iloc[0:100, 4].values
# Iris-setosa を -1, Iris-virginica を 1 に変換
y = np.where(y == 'Iris-setosa', -1, 1)

# 1-100行目の1,3行目(素性)の抽出
X = df.iloc[0:100, [0,2]].values
```

In [21]:

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

In [69]:

```
# 描画範囲を1行2列に分割する
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8,4))

# バッチ勾配降下法によるADALINEの学習(η = 0.01)
ada1 = AdalineGD(n_iter = 10, eta = 0.01).fit(X, y)
# エポック数とコストの関係を表す折れ線グラフをプロットする
ax[0].plot(range(1, len(ada1.cost_) + 1), np.log10(ada1.cost_), marker='o')
# 軸ラベル
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('log(Sum-squared-error)')
# タイトル
ax[0].set_title('Adaline - Learning rate 0.01')

# バッチ勾配降下法によるADALINEの学習(η = 0.0001)
ada2 = AdalineGD(n_iter = 10, eta = 0.0001).fit(X, y)
# エポック数とコストの関係を表す折れ線グラフをプロットする
ax[1].plot(range(1, len(ada2.cost_) + 1), ada2.cost_, marker='o')
# 軸ラベル
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('Sum-squared-error')
# タイトル
ax[1].set_title('Adaline - Learning rate 0.0001')
plt.show()
```



η=0.0001のとき、コストが減少しているのがわかるが学習率が低すぎるために相当数のEpochが必要になると考えられる. η = 0.01のときは常にコストが増加しており、ηで修正する勾配が大きすぎたために局所的最小値を超えてしまっている.

4. データの標準化とバッチ勾配降下法の問題点

このような結果になる理由の一つとしてサンプルごとのデータのばらつきが大きいことが挙げられる.改善方法として特徴量をスケーリングする.今回は標準化という手法をとる.

標準化とは

各特徴量の平均を0, 標準偏差を1にすること.

ある特徴量jを標準化するためには、サンプルの平均μ_jを全てのトレーニングサンプルX_jから引き、標準偏差δ_jでひけばいい.ただし、X_jはトレーニングサンプルのj番目の値からなるベクトルになる.

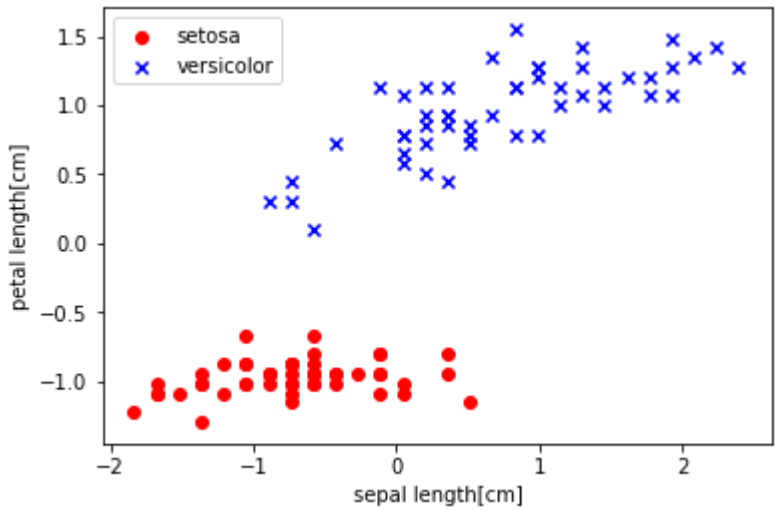
$$\mathbf{X}'_j = \frac{\mathbf{X}_j - \mu_j}{\sigma_j}$$

In [64]:

```
# 訓練データの標準化

# 訓練データをコピー
X_std = np.copy(X)
# 各特徴量を標準化
# meanメソッド : ベクトルの平均値を計算する
# stdメソッド : ベクトルの標準偏差を計算する.
X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

In [65]:

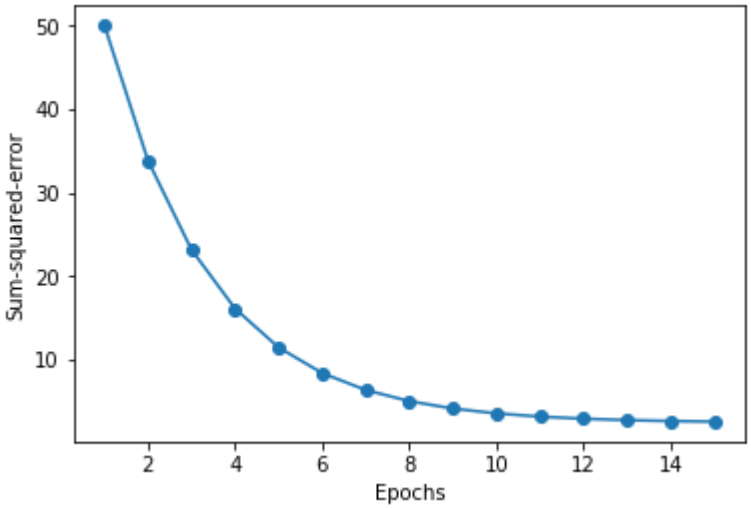
```
# 標準化したデータの表示
plt.scatter(X_std[:50,0], X_std[:50,1], color='red', marker='o', label='setosa')
plt.scatter(X_std[50:100,0], X_std[50:100,1], color='blue', marker='x', label='versicolor')

plt.xlabel('sepal length[cm]')
plt.ylabel('petal length[cm]')

plt.legend(loc = 'upper left')
plt.show()
```



上記の表から標準化してもデータの分布は変わらないが、データの範囲が大きく狭まったことがわかる.　次に、標準化された訓練データを使ってη=0.01に設定した場合のトレーニングを行う.

In [70]:

```
# バッチ勾配降下法によるADALINEの学習(η = 0.01)
ada = AdalineGD(n_iter = 15, eta = 0.01).fit(X_std, y)
plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker = 'o')
plt.xlabel('Epochs')
plt.ylabel('Sum-squared-error')
plt.show()
```



標準化された特徴量でトレーニングした場合にはADALINEが収束していることがわかる.ただし、全てのサンプルが正しく分類されたとしても誤差平方和(y - φ(z))が0にはならない.

今回用いた訓練データは200サンプル程度で非常に小さいデータセットだが、数百万個以上のサンプルが含まれるデータセットでバッチ勾配降下法を行うことを考える.バッチ勾配降下法では**局所的最小値に1ステップ近づくごとに訓練データ全体で勾配を計算する必要があり、計算コストがかなり大きくなることが考えられる。**そこで、バッチ勾配降下法の代わりに確率的勾配降下法(stochastic gradient descent)を用いる。

5. 確率的勾配降下法(stochastic gradient descent)

バッチ勾配降下法では全てのサンプルXに対して蓄積された誤分類の合計を用いて重みを計算していた. 確率的勾配降下法は各サンプルごとに重みを更新していく、更新頻度が高いため遥かに高速に収束する.確率的勾配降下法を用いて正確な結果を得るためには、エポックごとにランダムな順序に並べ替えることが重要である.並べ替えないとならない理由はエポックごとに全く同じ訓練データの場合には、訓練データを往復するたびに重みベクトルが同じ値に戻ってしまうためである.

また、確率的勾配降下法の場合はオンライン学習に利用することが出来る.バッチ勾配降下法は訓練データ全体で勾配を計算するためにオンライン学習ができなかった. オンライン学習はWebアプリケーションの顧客データなどの大量のデータが蓄積されている場合に、新たな訓練データを追加することでシステムを更新することができる.

5.1 Python での実装

バッチ勾配降下法のプログラムを元に、いくつかのメソッドの追加と調整を行う.

- figメソッド : 訓練データ全体ではなく、各訓練サンプル毎に重みを更新する.

訓練後にコスト関数が収束したか否かは、訓練サンプルの平均コストとしてエポックごとの平均を計算する.また、訓練前に訓練データをシャッフルする.(参考文献ではオンライン学習に対応させるために重みを初期化しないpertial_fitメソッドを定義していた. 詳細は p.43)

In [72]:

```
from numpy.random import seed
```

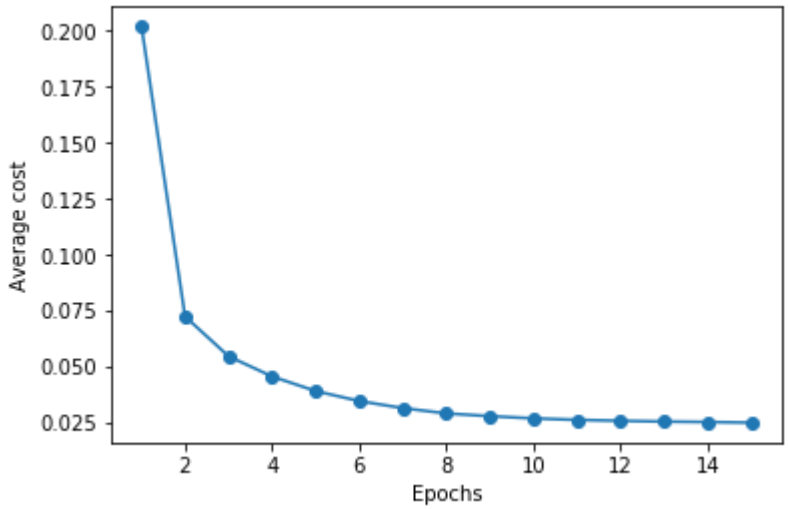
In [80]:

```
class AdalineSGD(object):
    """
    ADAptive LInear NEuron 分類器
    """

    def __init__(self, eta=0.01, n_iter=50, shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False # 重みの初期化フラグ(Trueの場合初期化しない)
        self.shuffle = shuffle # エポック毎に訓練データをシャッフルするか否か
        if random_state: seed(random_state)

    def fit(self, X, y):
        """訓練データに適合させる

        パラメータ
        X: {配列のようなデータ構造}, shape = {n_samples, n_features} 訓練データ
        y : 配列のようなデータ構造, shape = {n_samples} 目的変数(正解データ)

        返り値
        self.obj
        """

        # 重みベクトルを生成する
        self._initialize_weights(X.shape[1])
        # 各エポックごとのコストを格納するリストの作成
        self.cost_ = []
        # 訓練回数分だけ訓練データをまわす
        for i in range(self.n_iter):
            # シャッフルする
            if self.shuffle: X,y = self._shuffle(X,y)
            # 各サンプルごとのコストを格納するリスト
            cost = []

            # 各サンプルごとに重みを計算
            for xi, target in zip(X, y):
                # 特徴量Xi と 目的変数y を用いた重みの更新、コスト計算
                cost.append(self._update_weights(xi, target))
            # 各サンプルのコストの平均値を計算(ただのリストなのでnumpyは使えない)
            avg_cost = sum(cost)/len(y)
            # 平均コスト
            self.cost_.append(avg_cost)
        return self

    def _shuffle(self, X, y):
        """訓練データをシャッフルする"""
        r = np.random.permutation(len(y))
        return X[r], y[r]

    def _update_weights(self, xi, target):
        """ADALINEの学習規則で重みを更新"""
        # 活性化関数の出力 φ(W^T x) = W^T x
        output = self.net_input(xi)
        # 誤差の計算
        error = (target - output)
        # 重みw1,...,wmの更新
        self.w_[1:] += self.eta * xi.dot(error)
        # 重みw0の更新
        self.w_[0] += self.eta * error
```

```
        # コスト計算

        cost = error**2 / 2.0

        return cost

    def _initialize_weights(self, m):
        """重みを0に初期化する"""
        self.w_ = np.zeros(1 + m)
        self.w_initialized = True

    def net_input(self, X):
        """総入力を計算"""
        # Xと重みの総入力を計算し、w0を連結する
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """線形活性化関数の出力を計算"""
        return self.net_input(X)

    def predict(self, X):
        """
        1ステップ後のクラスラベルを返す
        """
        # where で指定した条件に当てはまるインデックスを返す
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

In [83]:

```
# ADALINEの学習
ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1).fit(X_std, y)

plt.xlabel("Epochs")
plt.ylabel("Average cost")
plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker = "o")
plt.show()
```



上記の表から、平均コストがバッチ勾配降下法に比べて早く減少することがわかる. 15エポックよりあとの最終的なコストはバッチ勾配降下法と同様の値になる.

まとめ

ADALINEはパーセプトロンのアルゴリズムを元に改良を加えた単一層ニューラルネットワークである.主な変更点はパーセプトロンは活性化関数にステップ関数を用いていたのに対して、ADALINEは線形活性化関数J($w^T x$)を用いていることである.線形関数を用いることでコスト関数の微分が可能になり、勾配降下法によるコスト関数の最小化ができるようになる.

この章では勾配降下法は2種類紹介されている. 1つ目はバッチ勾配降下法で、1エポックごとに全ての訓練データの勾配をとり重みを更新する. 2つ目は確率的勾配降下法で, 1サンプルごとに勾配をとり、重みを更新する. バッチ勾配降下法は全ての訓練データの勾配を計算するため、訓練データが増加するほど計算コストも大きくなる. 一方で確率的勾配降下法では1サンプルごとに更新するため、計算の頻度が高く遥かに高速で減少する.

次回

これまで学んだパーセプトロンとADALINEのアルゴリズムを含めた分類器はPythonのscikit-learnライブラリに存在する. 次章からはscikit-learnを使った分類を行う.