

Нейронные сети

Наумов Д.А., доц. каф. КТ

Экспертные системы и искусственный интеллект, 2019

Содержание лекции

1

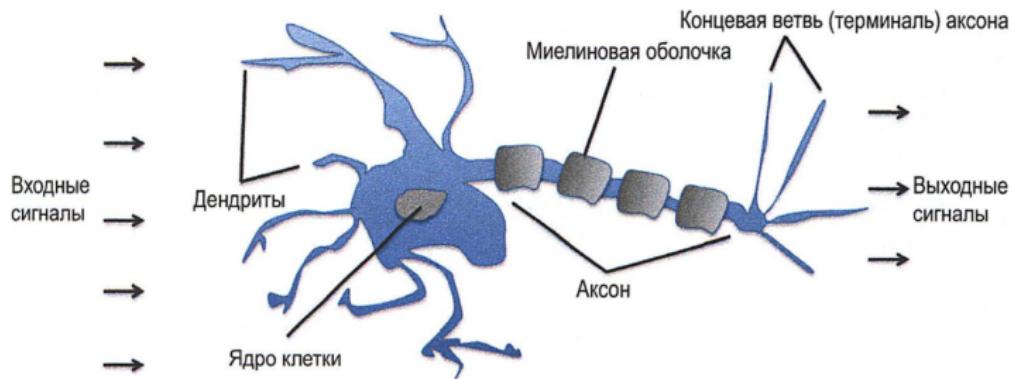
Тренировка алгоритмов машинного обучения для задач классификации

- Краткий обзор истории
- Реализация персептрана на Python
- Реализация алгоритма обучения и тренировка персептрана
- Аддитивные линейные нейроны и сходимость обучения
- Реализация адаптивного линейного нейрона
- Оптимизация на основе стохастического градиентного спуска

План изучения темы

- ① изучение первых алгоритмических методов классификации данных - персепtron и адаптивный линейными нейрон;
- ② пошаговая реализация персептрана на языке Python и его тренировки для решения;
- ③ задача классификации различных видов цветков из набора данных Ирисов Фишера;
- ④ базы оптимизации с использованием адаптивных линейных нейронов;
- ⑤ использование библиотек pandas, NumPy и matplotlib для чтения, обработки и визуализации данных;
- ⑥ реализация алгоритмов линейной классификации на Python.

Модель нейрона



- ① У. Маккалок и У. Питтс, «Логическое исчисление идей, относящихся к нервной активности», Bulletin of Mathematical Biophysics (Бюллетень математической биофизики), 5 (4):115-133, 1943);
- ② Ф. Розенблatt, «Лерсептрон, воспринимающий и распознающий автомат», Cornell Aeronautical Laboratory (Лаборатория аэронавтики Корнелльского университета), 1957).

Задача бинарной классификации

- задаются **два класса**: 1 (положительный класс), -1 (отрицательный класс);
- рассчитывается **чистый вход** - линейная комбинация весов и входных значений $z = w_1 \cdot x_1 + \dots + w_m \cdot x_m$;

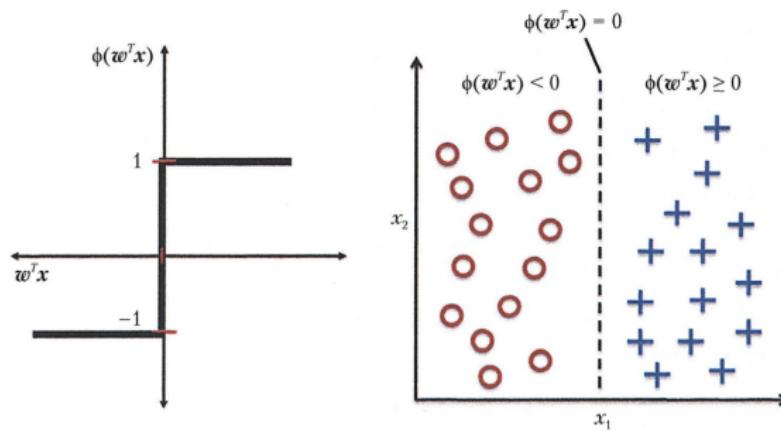
$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}.$$

- определяется передаточная функция (функция активации) $\phi(z)$;
- в алгоритме персептрана функция активации - это простая ступенчатая функция Хевисайда;

$$\phi(z) = \begin{cases} 1, & \text{если } z \geq 0 \\ -1, & \text{иначе} \end{cases}$$

- активация отдельно взятого образца $x^{(i)}$, т. е. выход из $\phi(z)$, превышает заданный порог, то мы распознаем класс 1, в противном случае - класс -1;
- порог учитывается так: $w_0 = -\theta, x_0 = -1$;

$$z = w_0 x_0 + \dots + w_m x_m = \sum_{j=0}^m x_j w_j = \mathbf{w}^T \mathbf{x}.$$



Идея, лежащая в основе нейрона MCP и персептронной модели Розенблатта с порогом:

- ① инициализировать веса нулями либо малыми случайными числами;
- ② для каждого тренировочного образца $x^{(i)}$ выполнить следующие шаги:
 - вычислить выходное значение \hat{y} ;
 - обновить веса.

Выходное значение - метка класса, идентифицированная единичной ступенчатой функцией.

$$\omega_j := \omega_j + \Delta\omega_j$$

Значение $\Delta\omega_j$ вычисляется правилом обучения персептрана:

$$\Delta\omega_j := \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$

- η - темп обучения (константа 0.0..1.0);
- $y^{(i)}$ - истинная метка класса i-го тренировочного образца;
- $\hat{y}^{(i)}$ - идентифицированная метка класса.

Персептрон правильно распознает метку класса, веса остаются неизменными:

$$\Delta w_j = \eta(-1 - -1)x_j^{(i)} = 0;$$

$$\Delta w_j = \eta(1 - 1)x_j^{(i)} = 0.$$

В случае неправильного распознавания веса продвигаются в направлении соответственно положительного или отрицательного целевого класса:

$$\Delta w_j = \eta(1 - -1)x_j^{(i)} = \eta(2)x_j^{(i)};$$

$$\Delta w_j = \eta(-1 - 1)x_j^{(i)} = \eta(-2)x_j^{(i)}.$$

Замечания о сходимости:

- 1 сходимость персептрона гарантируется, только если эти два класса линейно разделимы и темп обучения достаточно небольшой.
- 2 если эти два класса не могут быть разделены линейной границей решения, необходимо установить максимальное число проходов по тренировочному набору данных (эпох) и/или порог на допустимое число случаев ошибочной классификации, иначе обновление весов будет продолжаться бесконечно.

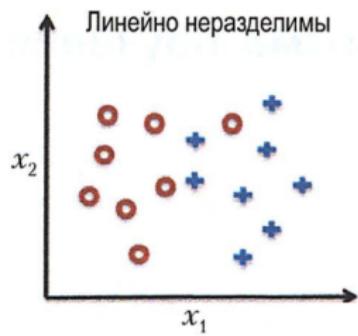
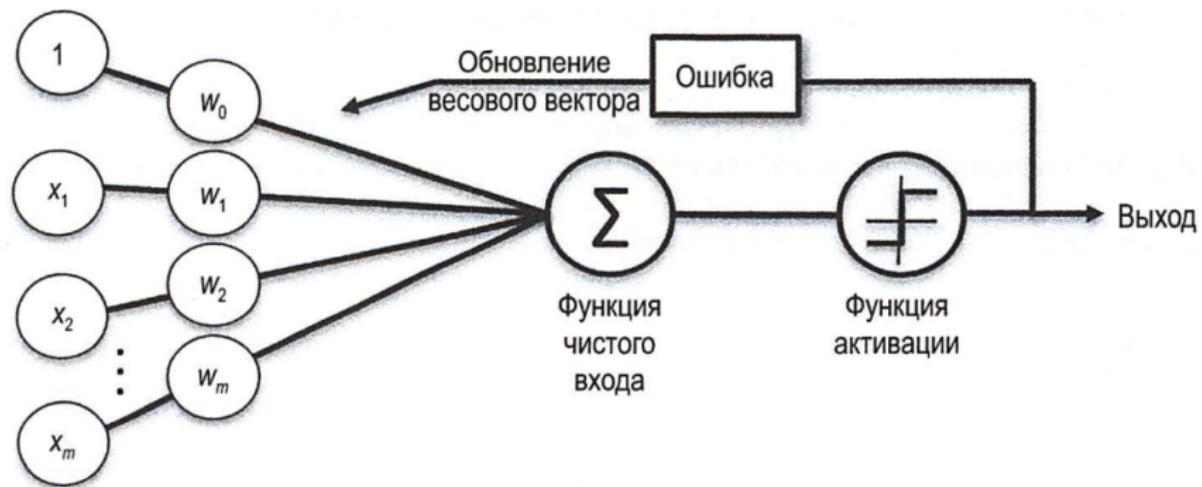


Схема MCP



```
import numpy as np
class Perceptron(object):
    """Классификатор на основе персептрана.

    Параметры
    -----
    eta : float
        Темп обучения (между 0.0 и 1.0)
    n_iter : int
        Проходы по тренировочному набору данных.

    Атрибуты
    -----
    w_ : 1-мерный массив
        Весовые коэффициенты после подгонки.
    errors_ : список
        Число случаев ошибочной классификации в каждой эпохе.

    """
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """Выполнить подгонку модели под тренировочные данные.

        Параметры
```

```
-----
X : {массивоподобный}, форма = [n_samples, n_features]
    тренировочные векторы, где
    n_samples - число образцов и
    n_features - число признаков.
y : массивоподобный, форма = [n_samples]
    Целевые значения.

Возвращает
-----
self : object

"""
self.w_ = np.zeros(1 + X.shape[1])
self.errors_ = []

for _ in range(self.n_iter):
    errors = 0
    for xi, target in zip(X, y):
        update = self.eta * (target - self.predict(xi))
        self.w_[1:] += update * xi
        self.w_[0] += update
        errors += int(update != 0.0)
    self.errors_.append(errors)
return self

def net_input(self, X):
    """Рассчитать чистый вход"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def predict(self, X):
    """Вернуть метку класса после единичного скачка"""
    return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Тренировка персептронной модели на наборе данных цветков ириса

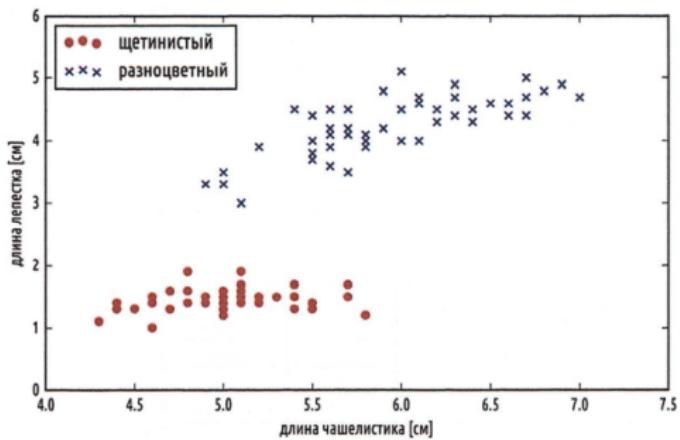
- 1 загрузим из набора данных ирисов два класса цветков: ирис щетинистый (`Iris setosa`) и ирис разноцветный (`Iris versicolor`).
- 2 в целях визуализации рассмотрим только два признака - длина чашелистика (`sepal length`) и длина лепестка (`petal length`).

```
>>>
import pandas as pd
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'
df = pd.read_csv(url, header=None)
df.tail()
```

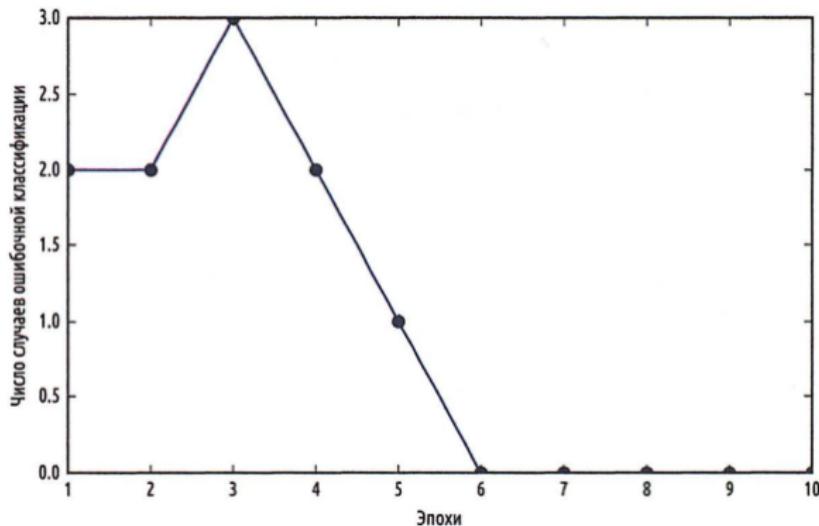
	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

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

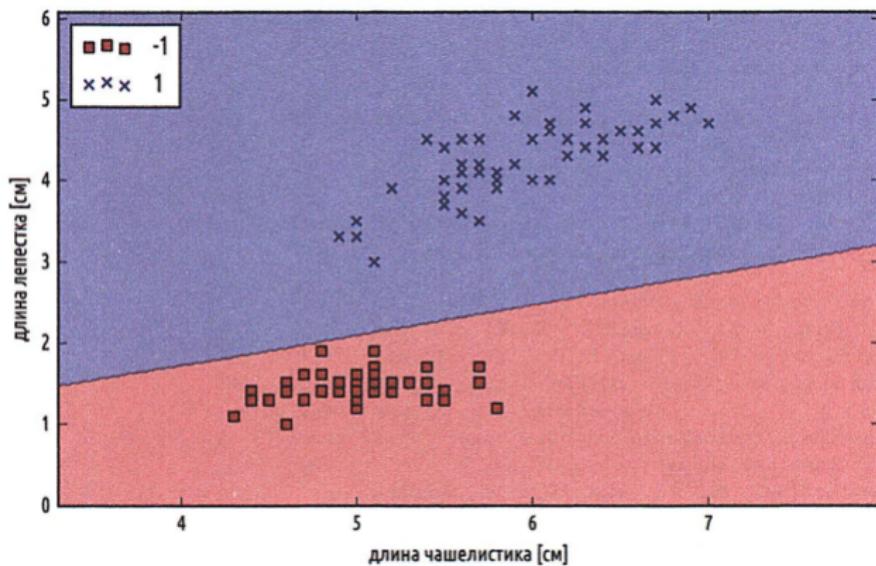
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)
X = df.iloc[0:100, [0, 2]].values
plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='щетинистый')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='x', label='разноцветный')
plt.xlabel('длина чашелистика')
plt.ylabel('длина лепестка')
plt.legend(loc='upper left')
plt.show()
```



```
ppn = Perceptron(eta=0.1, n_iter=10)
ppn.fit(X, y)
plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')
plt.xlabel('Эпохи')
# число ошибочно классифицированных случаев во время обновлений
plt.ylabel('Число случаев ошибочной классификации')
plt.show()
```

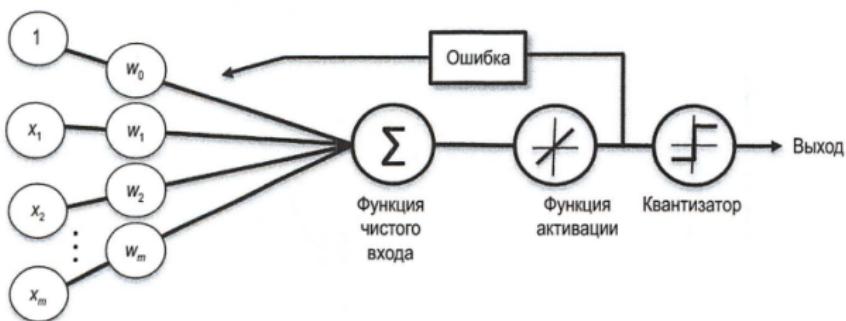


```
plot_decision_regions(X, y, classifier=ppn)
plt.xlabel('длина чашелистика [см]')
plt.ylabel('длина лепестка [см]')
plt.legend(loc='upper left')
plt.show()
```



ADALINE - ADaptive LInear NEuron, адаптивный линейный нейрон.

- для обновления весов используется линейная функция активации $\phi(\omega^T x) = \omega^T x$, а не единичная ступенчатая, как в персептроне;
- с целью распознавания меток классов используется квантизатор, аналогичный встречавшейся ранее единичной ступенчатой функции.

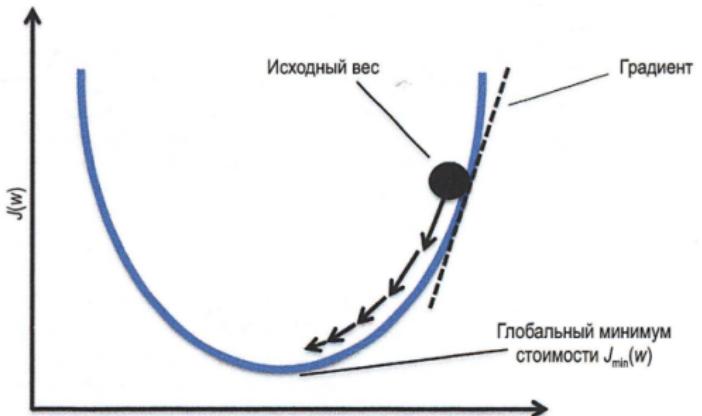


Б. Видроу и др., Adaptive «Adalineineuron using chemical «memistors» («Адаптивный нейрон "Adaline" с использованием химических "мемисторов--"). Number Technical Report Stanford Electron . Labs, Стэнфорд, Калифорния, октябрь 1960)

Ключевая составляющая алгоритмов машинного обучения с учителем: задание целевой функции, которая подлежит оптимизации во время процесса обучения.

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

Применяемый алгоритм оптимизации - **алгоритм градиентного спуска** для нахождения весов, которые минимизируют функцию стоимости.



Обновление веса на основе градиентного спуска путем выполнения шага в противоположную сторону от градиента функции стоимости

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}.$$

$$\Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w}).$$

Для расчета градиента функции стоимости нужно вычислить частную производную функции стоимости относительно каждого веса:

$$\Delta w_j = -\eta \frac{\delta J}{\delta w_j} = \eta \sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}.$$

- значение $\phi(z^{(i)})$ - вещественное число, а не целочисленная метка класса;
- обновление веса вычисляется на основе всех образцов в тренировочном наборе, и поэтому такой подход называется "пакетным"(batch) градиентным спуском.

```
def fit(self, X, y):
    """ Выполнить подгонку под тренировочные данные.

Параметры
-----
X : {массивоподобный}, форма = [n_samples, n_features]
    Тренировочные векторы, где
        n_samples - число образцов и
        n_features - число признаков.
y : массивоподобный, форма = [n_samples]
    Целевые значения.

Возвращает
-----
self : объект

"""
    self.w_ = np.zeros(1 + X.shape[1])
    self.cost_ = []

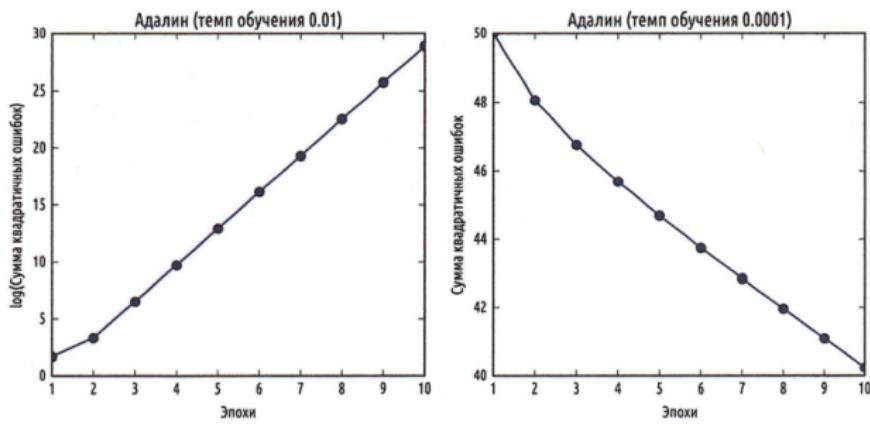
    for i in range(self.n_iter):
        output = self.net_input(X)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()
        cost = (errors**2).sum() / 2.0
        self.cost_.append(cost)

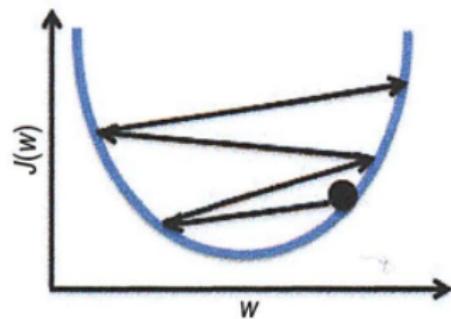
    return self
```



График стоимости в зависимости от числа эпох

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))
ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
ax[0].plot(range(1, len(ada1.cost_) + 1), np.log10(ada1.cost_), marker='o')
ax[0].set_xlabel('Эпохи')
ax[0].set_ylabel('log(Сумма квадратичных ошибок)')
ax[0].set_title('ADALINE (температура обучения 0.01)')
ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
ax[1].plot(range(1, len(ada2.cost_) + 1), ada2.cost_, marker='o')
ax[1].set_xlabel('Эпохи')
ax[1].set_ylabel('Сумма квадратичных ошибок')
ax[1].set_title('ADALINE - (температура обучения 0.0001)')
plt.show()
```



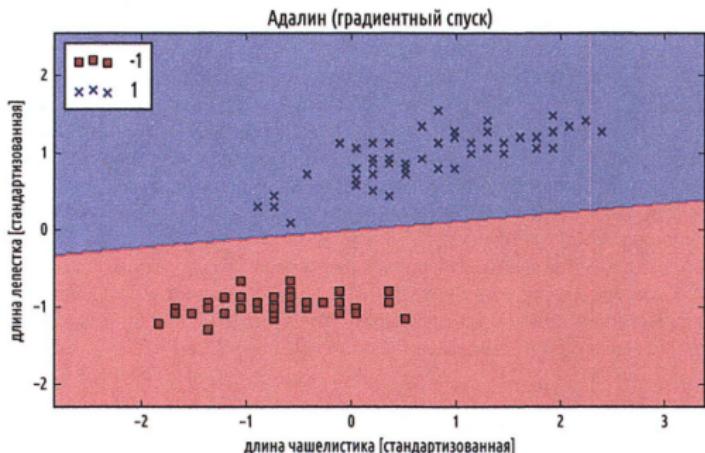


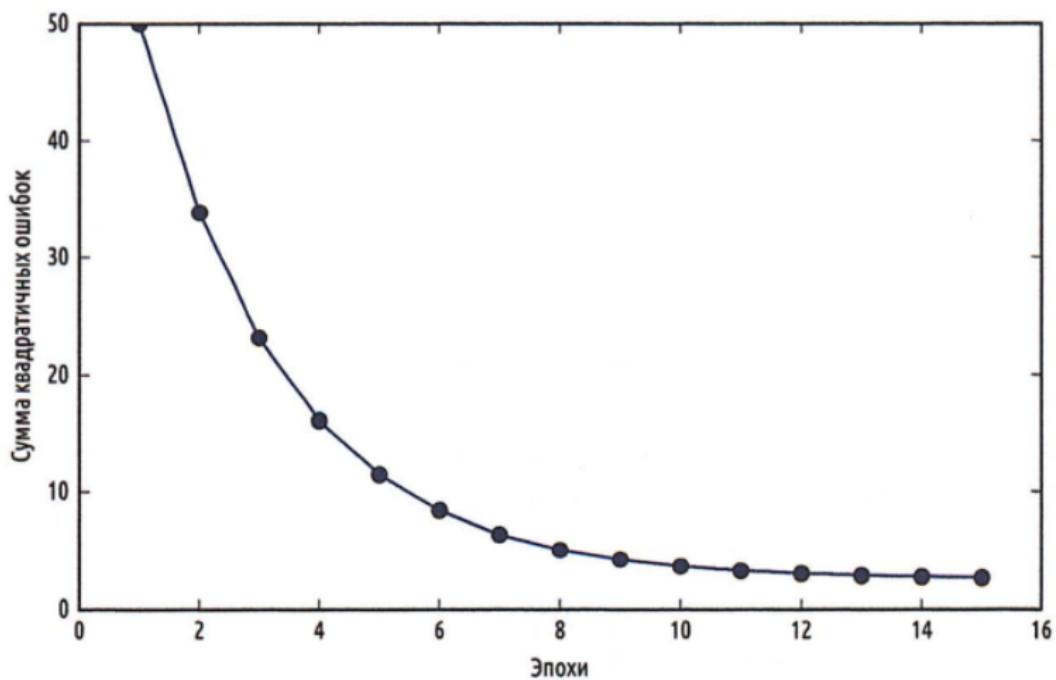
Алгоритм градиентного спуска можно улучшить, выполнив предварительное масштабирование признаков.

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}.$$

```
X_std = np.copy(X)
X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

```
ada = AdalineGD(n_iter=15, eta=0.01)
ada.fit(X_std, y)
plot_decision_regions(X_std, y, classifier=ada)
plt.title('ADALINE (градиентный спуск)')
plt.xlabel('длина чашелистика [стандартизованная]')
plt.ylabel('длина лепестка [стандартизованная]')
plt.legend(loc='upper left')
plt.show()
plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Эпохи')
plt.ylabel('Сумма квадратичных ошибок')
plt.show()
```





- ADALINE сходится после тренировки на стандартизованных признаках с темпом обучения $T = 0.01$;
- сумма квадратичных ошибок (SSE) остается ненулевой даже при том, что все образцы были классифицированы правильно.

Для больших наборов данных выполнение пакетного градиентного спуска может быть в вычислительном плане довольно дорогостоящим, поскольку необходимо выполнять переоценку всего тренировочного набора данных каждый раз, когда мы делаем один шаг к глобальному минимуму.

Стохастический градиентный спуск

вместо обновления весов, основываясь на сумме накопленных ошибок по всем образцам, обновляет веса инкрементно по каждому тренировочному образцу:

$$\eta(y^{(i)} - \phi(z^{(i)})x^{(i)}).$$

- намного быстрее достигает сходимости;
- лучше выходит из мелких локальных минимумов;
- важно перемешивать тренировочный набор в каждой эпохе;
- возможность тренировки "на лету".

Корректировки алгоритма ADALINE

- в методе `fit` будут обновляться веса после каждого тренировочного образца;
- реализован дополнительный метод частичной подгонки `partial_fit`, который повторно не инициализирует веса, чтобы учесть динамическое обучение;
- для проверки, что алгоритм сходился после тренировки, вычисляется стоимость как усредненную стоимость тренировочных образцов в каждой эпохе;
- опцию `shuffle` - перемешивание тренировочных данных перед каждой эпохой для предотвращения зацикливания во время оптимизации функции стоимости.

```

ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
ada.fit(X_std, y)
plot_decision_regions(X_std, y, classifier=ada)
plt.title('ADALINE (стохастический градиентный спуск)')
plt.xlabel('длина чашелистика [стандартизованная]')
plt.ylabel('длина лепестка [стандартизованная]')
plt.legend(loc='upper left')
plt.show()
plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Эпохи')
plt.ylabel('Средняя стоимость')
plt.show()

```

