Классификация

Задача классификации в машинном обучении заключается в том, чтобы по набору признаков определить, к какому из заранее известных классов относится объект. Модель учится распознавать закономерности на примерах с известными метками и затем применяет эти знания для правильного отнесения новых, ранее не встречавшихся объектов к определённым категориям. Это позволяет автоматизировать процесс сортировки и принятия решений по многим реальным задачам, от диагностики заболеваний до фильтрации спама, где важно отнести объект к правильной группе, обладающей особыми характеристиками. Таким образом, классификация решает проблему распределения данных на дискретные категории для последующего анализа или действий с ними.

Пример задачи классификации — определение, является ли входящее письмо спамом или нет. Модель обучается на большом наборе писем, где известно, какие помечены как «спам», а какие — «не спам». Затем, получив новое письмо, классификатор решает, к какому из этих двух классов его отнести. Это помогает автоматически фильтровать нежелательную почту, улучшая качество работы с электронной корреспонденцией. Аналогично работают задачи распознавания болезней по медицинским данным, определение пола или возраста по фотографии и многие другие.

Логистическая регрессия — это статистический метод, предназначенный для решения задач классификации, который прогнозирует вероятность принадлежности объекта к одному из двух классов. В отличие от линейной регрессии, которая предсказывает числовое значение, логистическая регрессия использует сигмоидную функцию, преобразующую линейную комбинацию признаков в значение от 0 до 1, интерпретируемое как вероятность

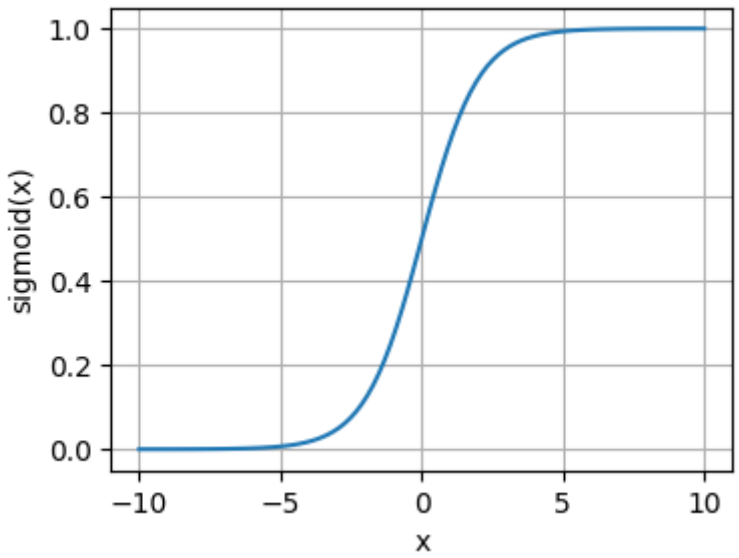
$$\sigma(z) = \frac{1}{(1+exp(-z))^{-1}} = \frac{1}{1+\frac{1}{exp(z)}} = \frac{exp(z)}{exp(z)+1}$$

$$\sigma(z) = \frac{exp(z)}{1+exp(z)}.$$

где z – это скалярное произведение весов на значение признака.

На основе этой вероятности принимается решение о принадлежности объекта к конкретному классу. Основное применение — бинарная классификация, например, оценка вероятности одобрения кредита или диагностика заболевания по медицинским данным.

График сигмоиды выглядит следующим образом



Функция потерь в бинарной классификации называется логарифмической функцией потерь (log loss) или кросс-энтропией и в общем виде записывается так:

$$-\ln L(X) = -\sum_{i=1}^{l}(y_i \ln \frac{1}{1+exp(-\langle w, x_i \rangle)} + (1-y_i)\ln(1-\frac{1}{1+exp(-\langle w, x_i \rangle)})).$$

$$-\ln L(X) = -\sum_{i=1}^{l}(y_i \ln(\sigma) + (1-y_i)\ln(1-\sigma)).$$

Пошаговый алгоритм логистической регрессии. Пусть у нас уже имеются подготовленные к моделированию данные, разбитые на обучающую и валидационную выборки

1 – реализуем функцию сигмоиды

```
def sigmoid_2(x):
    return np.exp(x) / (1 + np.exp(x))
```

2 – Реализуем вычисление функции потерь логистической регрессии log loss и её градиента для оптимизации весов.

```
def log_loss(w, X, y):
    m = X.shape[0] #число примеров в выборке
    A = sigmoid(np.dot(X, w))

    loss = -1.0 / m * np.sum(y * np.log(A) + (1 - y) * np.log(1 - A))
    grad = 1.0 / m * X.T @ (A - y)

    return loss, grad
```

3 – Градиентный спуск

```
def optimize(w, X, y, n_iterations, eta):
    # потери будем записывать в список для отображения в виде графика
```

```
    losses = []

    for i in range(n_iterations):
        loss, grad = log_loss(w, X, y)
        w = w - eta * grad

        losses.append(loss)

    return w, losses
```

4 – Функция предсказаний. Полученные из сигмоиды вероятности принадлежности к целевому классу пропускаем через пороговое значение 0,5. Будем считать, все что больше 0,5 является объектом первого класса, если вероятность меньше либо равна 0,5 – объект принадлежит классу 0.

```
def predict(w, X):
    m = X.shape[0]
    y_predicted = np.zeros(m)
    A = np.squeeze(sigmoid(np.dot(X, w)))
    # За порог отнесения к тому или иному классу примем вероятность 0.5
    y_predicted = [1 if x > 0.5 else 0 for x in A]
    return y_predicted
```

Метрики в задачах классификации

Метрики классификации в машинном обучении служат для количественной оценки качества работы модели. Доля верно предсказанных объектов (accuracy) измеряет долю правильных предсказаний от общего числа и подходит при сбалансированных данных. Точность (precision) показывает, какая часть положительных предсказаний оказалась верной, а полнота (recall) отражает, какую часть реальных положительных объектов модель смогла найти. F1-мера объединяет точность и полноту, учитывая баланс между ложноположительными и ложноотрицательными ошибками. Для более детального анализа часто используют ROC-AUC, которая показывает способность модели отделять классы при различных порогах решения.

Разумный выбор метрик зависит от специфики задачи и распределения классов, поэтому обычно применяют несколько показателей для комплексной оценки модели.

Доля верно предсказанных объектов (accuracy)

$$accuracy(a, x) = \frac{1}{l}\sum_{i=1}^{l}[a(x_i) = y_i]$$

является недостоверной метрикой при дисбалансе классов.

Матрица ошибок (confusion matrix) — это таблица, которая показывает, как классификатор распределяет объекты по классам, сравнивая фактические метки с предсказанными. Для бинарной классификации она состоит из четырёх элементов: истинно положительные (True Positive, TP) — правильные

положительные предсказания, ложно положительные (False Positive, FP) — ошибки первого рода, когда модель ошибочно предсказывает положительный класс, ложно отрицательные (False Negative, FN) — ошибки второго рода, когда модель пропускает положительный класс, и истинно отрицательные (True Negative, TN) — правильные отрицательные предсказания. Матрица ошибок помогает не только оценить общую точность модели, но и понять, какие именно ошибки она допускает, что важно для корректировки и улучшения алгоритма.

	$y = +1$	$y = -1$
$a(x) = +1$	True Positive TP	False Positive FP
$a(x) = -1$	False Negative FN	True Negative TN

Точность (precision) представляет из себя долю истинных срабатываний от общего количества срабатываний. Она показывает, насколько можно доверять алгоритму классификации в случае срабатывания

$$precision(a, X) = \frac{TP}{TP + FP}.$$

Полнота (recall) считается как доля объектов, истинно относящихся к классу "+1", которые алгоритм отнес к этому классу

$$recall(a, X) = \frac{TP}{TP + FN};$$

TP+FN как раз будут вместе составлять весь список объектов класса "1".

Есть различные варианты объединения их в одну метрику, одним из наиболее удобных из них является F-мера, которая представляет собой среднее гармоническое между точностью и полнотой

$$F = \frac{2 \cdot precision \cdot recall}{presision + recall}$$

В отличие от, например, среднего арифметического, если хотя бы один из аргументов близок к нулю, то и среднее гармоническое будет близко к нулю.

Метод опорных векторов

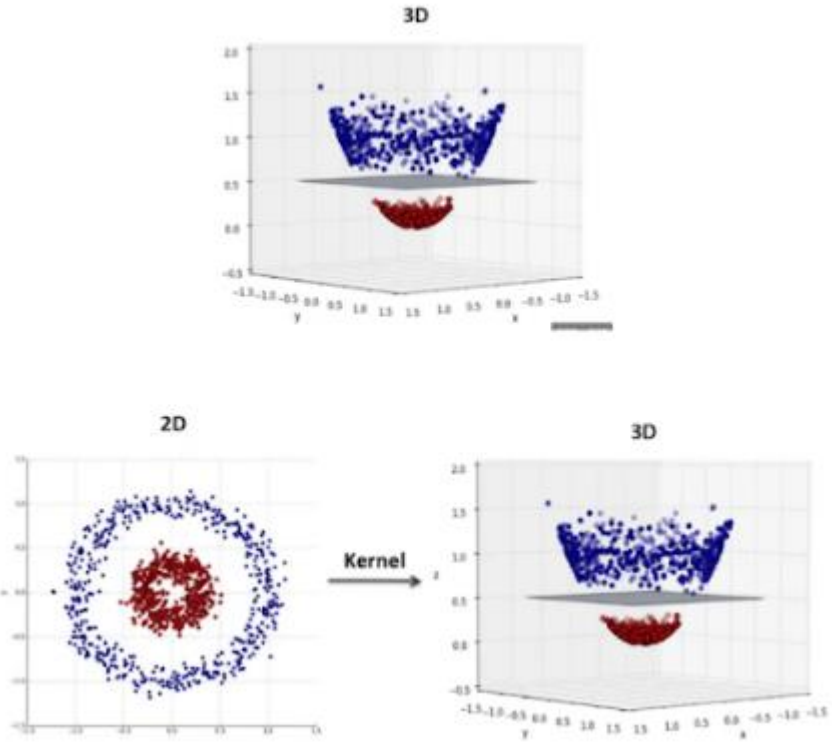
Метод опорных векторов (SVM, Support Vector Machine) – это один из самых мощных и универсальных алгоритмов машинного обучения с учителем, применяемый для задач классификации, регрессии и обнаружения аномалий. SVM особенно популярен при работе с высокоразмерными данными и сложными классификационными задачами.

Главная задача SVM – найти такую разделяющую гиперплоскость (или линию в двумерном случае), которая максимально отделяет объекты разных классов. При этом SVM стремится не просто найти любое разделение; алгоритм ищет гиперплоскость с максимально широким зазором (margin) между ближайшими точками разных классов. Эти точки называются опорными векторами и играют решающую роль в определении положения разделяющей поверхности.

Если данные линейно неразделимы, SVM использует специальные ядра (kernel trick) для преобразования исходного пространства признаков в пространство большей размерности, где гиперплоскость уже может быть найдена. Это позволяет разделять даже очень сложные, нелинейно разделимые данные.

Применение ядер (Kernel trick)

Если данные нельзя разделить прямой или плоскостью, SVM использует трюк с ядром: исходные данные отображаются в другое пространство более высокой размерности, где классы делимы линейно. Это делается эффективно, без явного пересчёта координат для всех точек, с помощью специальных функций (ядер), что позволяет SVM решать нелинейные задачи.



Популярные ядра:

Линейное ядро (Linear kernel) – это самое простое ядро, которое строит гиперплоскость для разделения данных. Оно часто используется в задачах с

линейно разделимыми данными. В математическом смысле линейное ядро вычисляет скалярное произведение между векторами признаков объектов.

Ядро Radial Basis Function (RBF) – это наиболее распространенное ядро, которое может разделять данные, не являющиеся линейно разделимыми. Оно создает границу принятия решений в виде радиально-симметричного колокола.

Ядро с полиномиальной функцией (Polynomial kernel) – это ядро, которое вводит полиномиальную функцию в пространство признаков для разделения данных. Это может быть полезно для данных, которые не могут быть разделены гиперплоскостью.

Ядро с сигмоидной функцией (Sigmoid kernel) – это ядро, которое используется для моделирования нейронных сетей. Оно может работать с нелинейными данными, но не так эффективно, как RBF-ядро.

Преимущества SVM

Эффективен в пространствах с большим количеством признаков;

Мало подвержен переобучению (при правильной настройке);

Универсальность – решает задачи как классификации, так и регрессии;

Может работать с линейными и нелинейными разделениями.

Недостатки

Может быть медленным на очень больших датасетах;

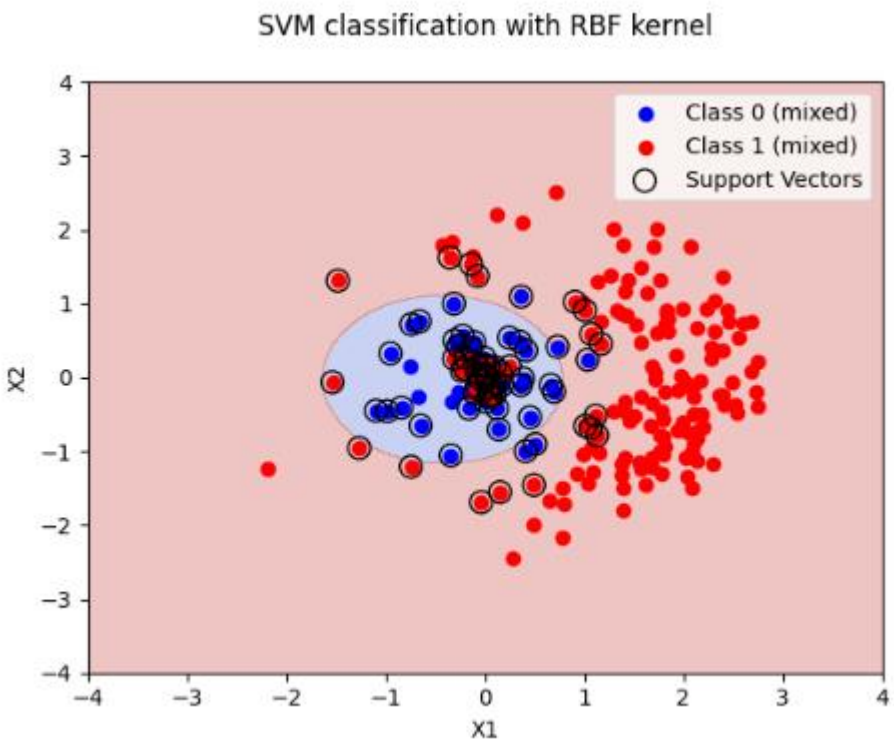
Требует настройки параметров ядра и регуляризации (например, параметра "C");

Плохо работает, если классы сильно пересекаются или если данные плохо масштабированы.

Пример. Классифицировать точки на плоскости, которые расположены по окружностям – класс 0 внутри круга радиуса 1, класс 1 – снаружи (от 1.5 до 2.5 радиуса). Очевидно, что эти классы невозможно разделить прямой линией – нужна нелинейная граница.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

np.random.seed(42)

# Класс 0 — внутри круга
class0 = np.random.randn(100, 2) * 1.4
class0 = class0[class0[:,0]**2 + class0[:,1]**2 < 1.2**2]

# Класс 1 — кольцо
```

```
class1 = (np.random.randn(250, 2) * 1.2) + np.array([1.7, 0])
mask = (class1[:,0]**2 + class1[:,1]**2 > 1.2**2) & (class1[:,0]**2 +
class1[:,1]**2 < 2.8**2)
class1 = class1[mask]

# Немного смешаем классы: добавим немного шума к классам, чтобы точки
пересекались
noise0 = 0.15 * np.random.randn(20, 2)
noise1 = 0.15 * np.random.randn(20, 2)
class0 = np.vstack([class0, noise1])  # добавляем несколько точек из класса 1
в класс 0
class1 = np.vstack([class1, noise0])  # добавляем несколько точек из класса 0
в класс 1

# Объединяем данные
X = np.vstack([class0, class1])
y = np.hstack([np.zeros(class0.shape[0]), np.ones(class1.shape[0])])

# Обучаем SVM с RBF ядром
model = svm.SVC(kernel='rbf', gamma='scale')
model.fit(X, y)

# Визуализируем результат
xx, yy = np.meshgrid(np.linspace(-4, 4, 500), np.linspace(-4, 4, 500))
grid_points = np.c_[xx.ravel(), yy.ravel()]
Z = model.predict(grid_points)
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.coolwarm)
plt.scatter(class0[:, 0], class0[:, 1], color='blue', label='Class 0
(mixed)')
plt.scatter(class1[:, 0], class1[:, 1], color='red', label='Class 1 (mixed)')

# Опорные векторы
sv = model.support_vectors_
plt.scatter(sv[:, 0], sv[:, 1], s=100, facecolors='none', edgecolors='k',
label='Support Vectors')

plt.legend()
plt.title('SVM classification with RBF kernel\n')
plt.xlabel('X1')
plt.ylabel('X2')
plt.show()
```


SVM classification with RBF kernel
Class 0 (mixed)
Class 1 (mixed)
Support Vectors
X1
X2
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4