Содержание

1 Постановка задачи

Цель работы: Изучить метод опорных векторов (SVM) и его применение для построения траектории движения.

- Изучить метод опорных векторов для случая полностью разделимой выборки.
- Рассмотреть SVM метод в пространстве признаков.
- Рассмотреть случай неразделимой выборки.
- С помощью метода опорных векторов построить траекторию обхода препятствий.

2 Основная часть

2.1 Метод опорных векторов в случае полноостью разделимой выборке

Рассмотрим задачу бинарной классификации для полностью разделимой выборки $S = ((\vec{x_1}, y_1), (\vec{x_2}, y_2), ..., (\vec{x_l}, y_l))$, где $\vec{x_i} \in \Re^n$ и $y_i \in \{-1, 1\}$ попытаемся разделить её гиперплоскостью $(\vec{\omega} \cdot \vec{x}) - c = 0$.
Полностью разделимая выборка - это выборка, которую можно без ошибок разделить гиперплоскостью.

Наша основная цель в этой части найти оптимальную разделяющую гиперплоскость, которая будет находится точно в середине между ближайшими сверху и снизу точками положительной и отрицательной частей выборки, поэтому будем искать максимум непрерывной функции $\rho(\vec{\omega})$ на компакте $\{\vec{\omega} : \|\vec{\omega}\| \leq 1\}$.

Такая гиперплоскость действительно существует, в силу доказанных лемм в [1].

Найденную гиперплоскость $(\omega_0 \cdot \vec{x}) - c_0 = 0$ уже назовем оптимальной.

Теперь, выяснив что такое оптимальная гиперплоскость, построим метод построения оптималной гиперплоскости как задачу квадратичного программирования, рассмотрев её альтернативное математическое определение:

$$y_i((\vec{\omega} \cdot \vec{x}_i) + b) \geq 1 \tag{2.1.4}$$

где $i = 1, ..., l$, и величина $\|\vec{\omega}_0\|$ была бы минимальна при этих ограничениях.

Для такой задачи составим лагранжиан:

$$L(\vec{\omega}, b, \vec{\alpha}) = \frac{1}{2}(\vec{\omega} \cdot \vec{\omega}) - \sum_{i=1}^{l} \alpha_i(y_i((\vec{\omega} \cdot \vec{x}_i) + b) - 1) \tag{2.1.5}$$

Для поиска седловой точки лагранжиана данный функционал необходимо минимизировать по $\vec{\omega}$ и b, а после маскимизировать по множителям

Лагранжа, при условиях $\alpha_i \geq 0, i = 1, ..., l$. Возьмем производные:

$$\frac{\partial L(\vec{\omega}, b, \vec{\alpha})}{\partial \vec{\omega}} = \vec{\omega} - \sum_{i=1}^{l} \alpha_i y_i \vec{x}_i = \vec{0}, \tag{2.1.6}$$

$$\frac{\partial L(\vec{\omega}, b, \vec{\alpha})}{\partial \vec{b}} = \sum_{i=1}^{l} \alpha_i y_i = 0 \tag{2.1.7}$$

Определим

$$W(\vec{\alpha}) = L(\vec{\omega}, b, \vec{\alpha}) \tag{2.1.8}$$

Теперь подставим полученные производные в изначальный лагранжиан:

$$W(\vec{\alpha}) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{l} \alpha_i \alpha_j y_i y_j (\vec{x}_i \cdot \vec{x}_j) \tag{2.1.9}$$

Получается, что надо максимизировать функционал выше, при условии:

$$\sum_{i=1}^{l} \alpha_i y_i = 0 \tag{2.1.10}$$

И учитывать также, что $\alpha_i \geq 0$, $i = 1, 2, ..., l$

Допустим, мы нашли наши решения ω_0 и b_0. Но в задачах нелинейного программирования накладываются дополнительные необходимые условия Каруша-Куна-Таккера. В данной задаче:

$$\alpha_i^0 (y_i((\vec{\omega_0} \cdot \vec{x}_i) + b_0) - 1) = 0 \tag{2.1.11}$$

для $i = 1, ..., l$

Отсюда следует, что $\alpha_i^0 > 0$ может быть только для тех i, для которых $y_i((\vec{\omega_0} \cdot \vec{x}_i) + b_0) - 1 = 0$, т.е. для тех векторов, которые лежат на самих гиперплоскостях $(\vec{\omega_0} \cdot \vec{x}_i) + b_0 = \pm 1$. Такие векторы называются опорными векторами.

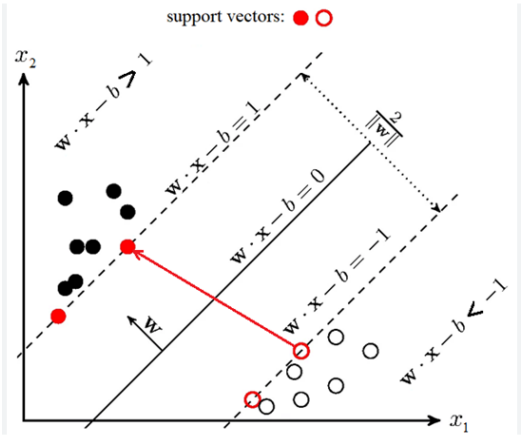

Рис. 1: Опорные векторы

ω_0 представляет собой линейную комбинацию опорных векторов $\vec{x}_{i_s}$, $s = 1, ..., k$, где k число опорных векторов:

$$\vec{\omega_0} = \sum_{s=1}^{k} \alpha_{i_s}^0 y_{i_s} \vec{x}_{i_s}. \tag{2.1.12}$$

А оптимальная гиперплоскость имеет вид:

$$\sum_{s=1}^{k} \alpha_{i_s}^0 y_{i_s} (\vec{x}_{i_s} \cdot \vec{x}) + b_0 = 0 \tag{2.1.13}$$

2.2 Метод опорных векторов в пространстве признаков.

В данной части мы будем рассматривать ту же выборку $S = ((\vec{x_1}, y_1), (\vec{x_2}, y_2), ..., (\vec{x_l}, y_l))$, где $\vec{x_i} \in \Re^n$ и $y_i \in \{-1, 1\}$. Но зададим некоторое нелинейное отображение из пространства признаков в пространство более высокой размерности:

$$\vec{x} = (x_1, ..., x_n) \to \vec{\phi} = (\phi_1(\vec{x}), ..., \phi_N(\vec{x})). \tag{2.2.1}$$

Отображение необязательно должно быть небратимым в данном случае.

И уже в новом пространстве признаков размерности $\Re^N$ будет строится оптимальная гиперплоскость, разделяющая векторы $\vec{\phi}(\vec{x}_1), ..., \vec{\phi}(\vec{x}_l)$.

Положим $z_j = \phi_j(\vec{x})$, где $j = 1, ..., N$, или в координатном виде $z_j \in \Re^N$.

Теперь будем строить гиперплоскость в простарнстве признаков $\Re^N$ по известному алгоритму из части 2.1. Эта гиперплоскость имеет своим прообразом в пространстве $\Re^N$ в общем случае нелинейную поверхность.

$$\sum_{j=1}^{N} \omega_j \phi_j(\vec{x}) + b = 0 \tag{2.2.2}$$

По аналогии с тем как мы составляли вектор весов $\vec{\omega}$ в части выше, запишем то же самое и для данного случая в координатном виде.

$$\omega_j = \sum_{i=1}^{l} \alpha_i^0 y_i \phi_j(\vec{x}_i) \tag{2.2.3}$$

Теперь подставим (2.1.15) в (2.1.14) и получим, опуская выкладки:

$$\sum_{j=1}^{N} \omega_j \phi_j(\vec{x}) + b = \sum_{i=1}^{l} \alpha_i^0 y_i (\vec{\phi}(\vec{x}) \cdot \vec{\phi}(\vec{x}_i)) + b = \sum_{i=1}^{l} \alpha_i^0 y_i K(\vec{x}, \vec{x}_i) + b = 0 \tag{2.2.4}$$

где

$$K(\vec{x}, \vec{x}_i) = (\vec{\phi}(\vec{x}) \cdot \vec{\phi}(\vec{x}_i)) \tag{2.2.5}$$

В конечном итоге, мы пришли к тому что можем разделять данные не только линейными поверхностями, но и нелинейными, путем замены скалярного произведеения на функцию ядра $K(\vec{x}, \vec{x}_i)$. При этом нам не нужно знать отображение $\vec{x} \to \vec{\phi}(\vec{x})$, а достаточно только знать ядро.

Запишем также вид нашей упрощенной функции $W(\alpha)$, которую мы минимизируем:

$$W(\alpha) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i=1}^{l} \alpha_i \alpha_j y_i y_j K(\vec{x}_i, \vec{x}_j). \tag{2.2.6}$$

На практике подбираются ядра, для которых соответствующая поверхность наилучшим образом разделяет обучающую выборку.

2.3 Ядра и спрямляющие пространства.

Исследуем новую задачу: Как строить функцию ядра? Джеймс Мерсер доказал теорему, которая даёт ответ.

Теорема. Функция $K(x, x')$ является ядром $\Leftrightarrow$ когда она симметрична и неотрицательно определена:

$$\int_X \int_X K(x, x') g(x) g(x') dx dx' \geq 0 \tag{2.3.1}$$

Функция $K(x, x')$ неотрицательно неопределена, если для любой конечной выборки $X^p = (x_1, ..., x_p)$ из X матрица $K = \|K(x_i, x_j)\|$ неотрицательно определена: $z^K z \geq 0 \ \forall z \in \Re^p$

Существуют конструктивные правила построения ядра [2], но всё же задача подбора ядра для какой-либо конкретной задачи открыта.

Ниже приведены примеры стандартных ядерных функций, в моей работе наиболее эффективное применение нашло **гауссово ядро RBF**.

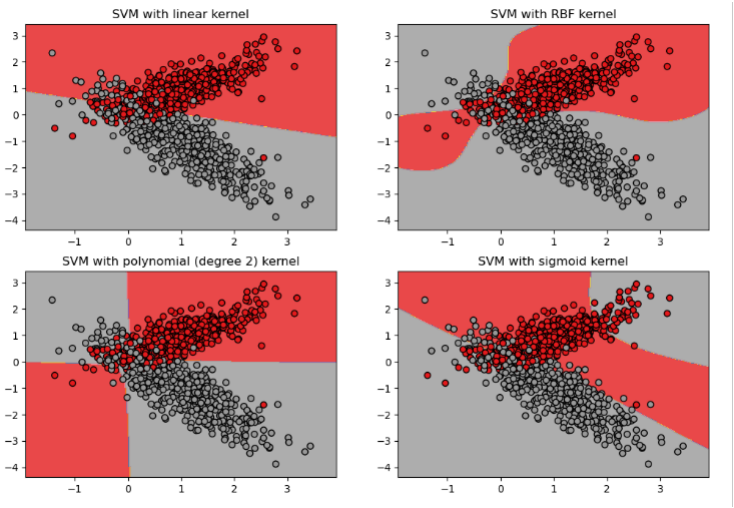

Рис. 2: Применение ядра

Линейное, просто вычисляет скалярное произведение векторов в исходном пространстве, что эквивалентно обычному линейному классификатору:

$$x_i^{'} x_j \tag{2.3.2}$$

Полиномиальное — способно улавливать более сложные зависимости между данными, создавая оптимальную разделяющую гиперплоскость в новом пространстве, однако требуется тщательный подбор параметров :

$$(\gamma x_i \cdot x_j + r)^d \tag{2.3.3}$$

Гауссовское RBF (радиально-базисная функция) — хорошо подходит для случаев, когда отношение в данных имеет сложную нелинейную форму и менее подвержено переобучению, поскольку учитывает не только значения признаков, но и их распределение:

$$exp(-\gamma \|x_i - x_j\|^2) \tag{2.3.4}$$

γ влияет на следующие аспекты: 1)Сила влияния расстояния: Чем больше γ, тем сильнее влияние расстояния между точками на значение ядра. Это означает, что точки, которые находятся ближе друг к другу, будут иметь более сильное влияние на решение задачи.
2)Ширина области влияния: Чем меньше γ, тем шире область влияния каждого опорного вектора. Это означает, что каждый опорный вектор будет иметь более широкое влияние на решение задачи в пространстве

признаков.
3)Чувствительность к шуму: Чем больше γ, тем более чувствительна модель к шуму в данных. Если γ слишком велико, модель может начать учитывать шум в данных, что может привести к переобучению.
4)Количество опорных векторов: Чем больше γ, тем меньше опорных векторов будет выбрано моделью. Это может привести к уменьшению сложности модели и времени ее обучения.
5)Область применимости: Чем меньше γ, тем более широкой будет область применимости модели. Это может быть полезно в случаях, когда необходимо предсказывать значения в областях, которые не были представлены в обучающей выборке.
6)Сглаживание решений: Чем меньше γ, тем более сглаженным будет решение задачи. Это может быть полезно в случаях, когда необходимо сгладить локальные минимумы и максимумы в решении.
По умолчанию используется $\gamma = \frac{1}{N}$, N - число признаков.

Сигмоидальное — применяет гиперболический тангенс к линейной комбинации векторов, имитируя использование двухслойной нейронной сети с сигмоидальной функцией активации, что также позволяет хорошо работать со сложными нелинейными случаями, однако это может привести к переобучению в случае появления шума и выбросов:

$$\tanh(\gamma x_i \cdot x_j + r) \tag{2.3.5}$$

2.4 Метод опорных веторов в случае неразделимой выборки. Случай линейной нормы.

В данной части будем рассматривать случай, когда наша выборка $S = ((\vec{x_1}, y_1), (\vec{x_2}, y_2), ..., (\vec{x_l}, y_l))$, где $\vec{x_i} \in \Re^n$ и $y_i \in \{-1, 1\}$ не разделимая, в таком случае вводятся вспомогательные переменные мягкого отступа ξ_i , $i = 1, ..., l$, которые будет как бы "штрафовать"объекты выборки за пересечение граничных гиперплоскостей их класса, при этом мы как бы разрешаем нашим объектам ошибаться.
На практике чаще используют линейную норму ошибок.

$$\frac{1}{2}(\vec{\omega} \cdot \vec{\omega}) + C \sum_{i=1}^{l} \xi_i \to \min_{\omega, \omega_0, \xi} \tag{2.5.1}$$

$$y_i((\omega, x_i) - b) \geq 1 - \xi_i, i = 1, ..., l \tag{2.5.2}$$

$$\xi_i \geq 0, i = 1, ..., l \tag{2.5.3}$$

Введём понятие отступа объекта x_i:

$$m_i = y_i((\omega, x_i) - b) \tag{2.5.4}$$

Алгоритм допускает ошибку на объекте x_i тогда и только тогда, когда отступ m_i отрицателен. Если $m_i \in (-1, +1)$, то объект x_i попадает внутрь разделяющей полосы. Если $m_i > 1$, то объект x_i классифицируется правильно, и находится на некотором удалении от разделяющей полосы.

Тогда можно ввести функционал числа ошибок нашего алогритма,который помимо прочего чувствителен к ошибкам:

$$Q(a, X^l) = \sum_{i=1}^{l} (1 - m_i)_+ + \tau \|\omega\|^2 \to \min_{\omega, b}. \tag{2.5.5}$$

Новый полученный лагранжиан:

$$L(\omega, \omega_0, \xi, \lambda, \eta) = \frac{1}{2}(\omega, \omega) - \sum_{i=1}^{l} \lambda_i (y_i((\omega, x_i) - b) - 1) - \sum_{i=1}^{l} \xi_i (\lambda_i + \eta_i - C) \tag{2.5.6}$$

где $\eta = (\eta_1, ..., \eta_l)$ — вектор переменных, двойственных к переменной $\xi = (\xi_1, ..., \xi_l)$. Как и в прошлый раз, условия Куна-Таккера сводят задачу

к поиску седловой точки функции Лагранжа:

$$\begin{cases} L(\omega, \omega_0, \xi, \lambda, \eta) \to \min\limits_{\omega,b,\xi} \max\limits_{\lambda,\eta} \\ \xi_i \geq 0, \lambda_i \geq 0, \eta \geq 0, i = 1, ..., l \\ y_i((\omega, x_i) - b) = 1 - \xi_i, i = 1, ..., l \\ \xi_i = 0, i = 1, ..., l \end{cases} \tag{2.5.7}$$

При подсчете производных получаем те же результаты что и в линейном случае, кроме:

$$\eta_i + \lambda_i = C, i = 1, ..., l \tag{2.5.8}$$

Из условий $\eta_i \geq 0$ следует ограничение $\lambda_i C$. И получается, что возможны три типа объектов.

- $\lambda_i = 0$; $\eta_i = C$; $\xi_i = 0$; $m_i = 1$: Объект x_i классифицируется правильно и находится далеко от разделяющей полосы.
- $0 < \lambda_i < C$; $0 < \eta_i < C$; $\xi_i = 0$; $m_i = 1$: Объект x_i классифицируется правильно и лежит в точности на границе разделяющей полосы.
- $\lambda_i = C$; $\eta_i = 0$; $\xi_i > 0$; $m_i < 1$: Объект x_i либо лежит внутри разделяющей полосы, но классифицируется правильно

  $$(0 < \xi_i < 1, 0 < m_i < 1)$$

  , либо попадает на границу классов ($\xi_i = 1, m_i = 0$), либо вообще относится к чужому классу ($\xi_i > 1, m_i < 0$).

$$\begin{cases} -L(\lambda) = -\sum\limits_{i=1}^{n} \lambda_i + \sum\limits_{i=1}^{n} \sum\limits_{j=1}^{n} \lambda_i \lambda_j y_i y_j (x_i \cdot x_j) \to \min\limits_{\lambda} \\ 0 \leq \lambda_i \leq C, (1 \leq i \leq n) \\ \sum\limits_{i=1}^{n} \lambda_i y_i = 0. \end{cases} \tag{2.5.9}$$

2.5 Разметка препятствий и запуск на библиотечной реализации SVM

Опираясь на статью[3] применим алгоритм построения оптимальной гиперплоскости для задачи планирования пути с учетом наличия препятствий. Суть алгоритма заключается в классификации препятствий на те, что следует обойти слева и те, что следует обойти справа. В этом случае разделяющая кривая будет являться путем, свободным от препятствий.

Алгоритм планирования пути можно описать следующим образом:

1) Разметка препятствий. Все препятствия должны быть помечены как положительные или отрицательные перед применением SVM. Один из способов -протестировать все возможные варианты меток и выбрать наилучший,который обеспечивает кратчайший путь или же использовать рандомизированный подход, при котором случайным образом выбирается и тестируется ограниченное количество шаблонов препятствий (обозначений). Однако наибольшую эффективность(особенно по времени работы) показал алгоритм, согласно которому проводится линия от начальной до конечной точки: препятствия, центр масс которых оказался выше проведенной линии относятся к классу "1 остальные - к классу 1".

Пусть $(x_9, y_0) = (0, 0)$ - начальная точка, (x_1, y_1)- конечная точка. Уравнение прямой, соединяющей эти точки: $y = \frac{y_1}{x_1} \cdot x$

$c = (c_x, c_y) = (\frac{1}{n} \cdot \sum_i^n (x_i); \frac{1}{n} \cdot \sum_i^n (y_i))$ - центр масс препятствия,

$label = np.where(\frac{y_1}{x_1} \cdot c_x > c_y, 1, -1)$

2) Установливаем виртуальные препятствия вокруг начальной и конечной точек. (рис. 4)

3) Установливаются направляющие выборки, чтобы охватить возможную пройденную область, а так же минимизировать длину пути. В нашем коде направляющие выборки генерируются как линии(точнее, группы точек), параллельные прямой, соединяющей начальную и конечную точки. (рис. 4)

4) Применение SVM к сгенерированным выборкам и извлечение пути.

1. Запуск на библиотечной реализации SVM (rbf kernel)

```
rbf_svc = svm.SVC(kernel='rbf', C=10,
↪  probability=True).fit(x_train, label)
print('Accuracy of RBF kernel:', accuracy_score(label,
↪  rbf_svc.predict(x_train)))
```

Вывод: Accuracy of RBF kernel: 0.9901186071817193
Скорость работы: менее 1 секунды на датасете из 900 объектов.

Визуализация:

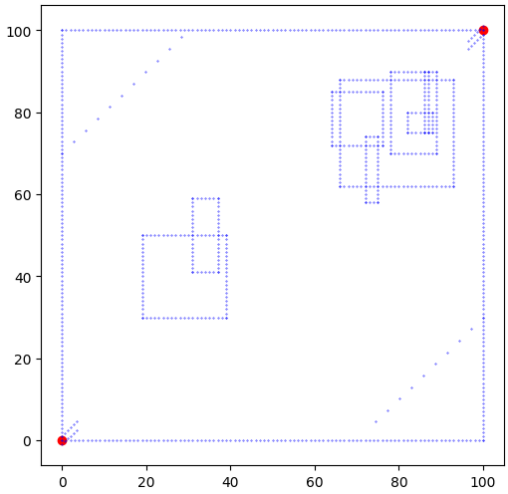

Рис. 3

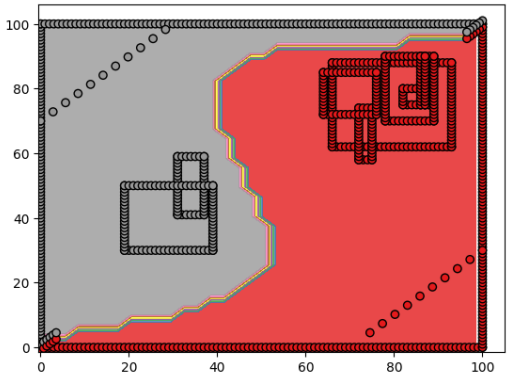

Рис. 4

Параметр C отвечает за важность штрафа за ошибки классификации в задаче оптимизации SVM. Чем выше значение C, тем более важным

становится штраф за ошибки классификации, и модель будет пытаться минимизировать количество ошибок классификации. Это может привести к переобучению модели. В нашей работе оптимальное $1 \leq C \leq 20$.

2.6 Реализация SVM с помощью svxopt

2. Теперь реализуем классификатор SVM. Система (2.5.9) представляет из себя задачу квадратичного программирования. Для ее решения воспользуемся библиотекой svxopt. Svxopt решает задачу, записанную в виде:

$$\begin{cases} L(\alpha) = -\frac{1}{2}\alpha^T P\alpha + \alpha^T E \\ 0 \leq \alpha \leq C \\ \alpha^T y = 0 \end{cases} \tag{2.7}$$

Учитывая систему(2.5.9) легко находим выражения для P и подставляем в метод cvxopt.solvers.qp.

```
def product(self,k,l): #Аналог скалярного произведения двух векторов
      return np.exp(-sum((k-l)**2)/(2*self.sigma_sq))

def gaussian_kernel(self,x1,x2):
    m1=x2.shape[0]
    n1=x1.shape[0]
    op=[[self.product(x1[x_index],x2[l_index]) for l_index in
    ↪  range(m1)] for x_index in range(n1)]
    #Находит все комбинации скалярных произведений векторов-объектов
    return np.array(op)

def fit(self,x,t):
    n_samples, n_features = x.shape
    y=np.copy(t)
    X=np.copy(x)

    #Compute the kernel matrix using the RBF kernel. The kernel
    ↪  matrix K has shape (n_samples, n_samples)
    K=self.gaussian_kernel(X,X)

    P = cvxopt.matrix(np.outer(y,y) * K)
    q = cvxopt.matrix(np.ones(n_samples) * -1)
    A = cvxopt.matrix(y, (1,n_samples))
    b = cvxopt.matrix(0.0)
    # soft-margin SVM
    G = cvxopt.matrix(np.vstack((np.diag(np.ones(n_samples) * -1),
    ↪  np.identity(n_samples))))
```

```
        h = cvxopt.matrix(np.hstack((np.zeros(n_samples),
        ↪  np.ones(n_samples) * self.C)))

        # solve QP problem. The solution is a dictionary containing the
        ↪  optimal values of the Lagrange multipliers.
        solution = cvxopt.solvers.qp(P, q, G, h, A, b)

        #Extract the Lagrange multipliers
        a = np.ravel(solution['x'])

        # Compute the weight vector
        self.W = ((y * a).T @ K)

        #Select the indices of the support vectors by thresholding the
        ↪  Lagrange multipliers.
        S = (a > 1e-4).flatten()

    def predict(self,x_train):
        X=np.copy(x_train)
        init=np.copy(self.initial)
        xpr=self.gaussian_kernel(X,init)
        weight=np.copy(self.W)
        predictin = xpr.dot(weight)
        return np.where(predictin > 0, 1, -1)
```

Accuracy при C=1: 0.9979317476732161
Время работы: 10 минут на выборке из 900 объектов.
Визуализация:

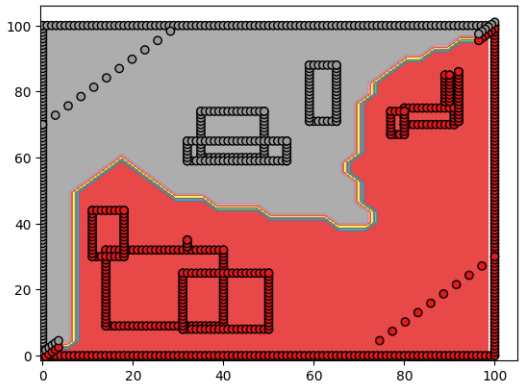

Рис. 5

Библиотека svxopt использует такой метод оптимизации как Interior Point Method (метод внутренней точки. Метод внутренней точки основан на следующих идеях: Вместо решения задачи оптимизации напрямую, мы решаем серию вспомогательных задач, которые приближаются к оригинальной задаче. В каждой вспомогательной задаче мы используем barrier-функцию, которая предотвращает выход из допустимой области. Barrier-функция выбирается так, чтобы она была гладкой и дважды дифференцируемой в допустимой области.

Barrier-функция имеет следующий вид:
$B(x, \mu) = f(x) - \mu \cdot \sum_{i=1}^{n} [ln(x_i - l_i) + ln(u_i - x_i)]$
где: f(x) - функция цели x - вектор переменных $l_i u_i$ - нижняя и верхняя границы для переменной x_i μ - параметр barrier-функции
Вспомогательная задача имеет следующий вид:
minimize $B(x, \mu)$
subject to $g_i(x) \leq 0, i = 1, ..., m$
$x \geq 0$
где: $g_i(x)$ - ограничения задачи m - количество ограничений

Алгоритм метода внутренней точки состоит из следующих шагов: Инициализация параметра μ и начального значения x. Решение вспомогательной задачи с помощью метода Ньютона или квази-Ньютона. Обновление параметра μ и значения x. Повтор шагов 2-3 до достижения оптимального решения. Подробнее данный алгоритм можно посмотреть в [4]

2.7 Реализация SVM с помощью SMO.

Используем алгоритм Sequential Minimal Optimization для решения задачи квадратичного программирования().Данный алгоритм используется в sklearn.svm.SVC.

ПМО разбивает эту задачу на ряд наименьших возможных подзадач, которые затем решаются аналитически. Из-за линейного равенства в ограничениях, которое включает лагранжные множители α_i наименьшая возможная задача включает два таких множителя. Тогда для любых двух множителей α_1, α_2.
$0 \leq \alpha_1, \alpha_2 \leq C$
$y_1\alpha_1 + y_2\alpha_2 = k$

и эту суженную задачу можно решать аналитически: нужно находить минимум одномерной квадратичной функции. k является суммой остальных членов в ограничении-уравнении с противоположным знаком, неизменной на каждой итерации.

Алгоритм действует следующим образом[5]: Найти лагранжей множитель α_1, который нарушает условия Каруша – Куна – Такера (ККТ) для задачи оптимизации. Выбрать второй множитель α_2 и оптимизировать пару (α_1, α_2) Повторите шаги 1 и 2 до совпадения. Когда все лагранжевые множители удовлетворяют условиям ККТ (в пределах определенного пользователем допуска), задача решена. Хотя этот алгоритм и совпадает гарантированно, для выбора пар множителей применяется эвристика, чтобы ускорить темп совпадения. Это критически для больших наборов данных, поскольку существует $n(n-1)$ возможных вариантов выбора α_i, α_j.

из ограничения $\sum_{i=1}^{n}(\alpha_i y_i) = 0$ получаем $y_1\alpha_1 + y_2\alpha_2 = -\sum_{i=3}^{n}(\alpha_i y_i) = k$
$y_i = -1, 1$
$0 \leq \alpha_i \leq C$ $\alpha_1\alpha_2$ α_2 может быть вычислен с помощью первой производной целевой функции, чтобы найти ее экстремум $\alpha_2^{new} = alpha_2^{old} + y_2 \cdot \frac{E_2 - E_1}{\nu}$
$E_i = y_i^- - y_i$
$\nu = K_{11} - 2K_{12} + K_{22}$
$K_{ij} = K(x_i, x_j)$
Если $y_1 \neq y_2$:
$L = max(0, \alpha_2 - \alpha_1)$
$H = min(C, C + \alpha_2 - \alpha_1)$
Если $y_1 = y_2$:
$L = max(0, \alpha_2 + \alpha_1 - C)$

$H = min(C, \alpha_2 + \alpha_1)$
Тогда:

$$\alpha_2^{new} = \begin{cases} L, alpha_2^{new} \leq L \\ \alpha_2^{new}, L \leq alpha_2^{new} \leq H \\ H, H \leq alpha_2^{new} \end{cases}$$
$$\alpha_1^{new} = \alpha_1^{old} + y_1 y_2 (\alpha_2^{old} - \alpha_2^{new})$$

```
def take_step(self, i1, i2):
        if (i1 == i2):
            return 0
        x1 = self.X[i1, :]
        x2 = self.X[i2, :]
        y1 = self.y[i1]
        y2 = self.y[i2]
        alpha1 = self.alphas[i1]
        alpha2 = self.alphas[i2]
        b = self.b
        E1 = self.get_error(i1)
        E2 = self.get_error(i2)
        s = y1 * y2
        if y1 != y2:
            L = max(0, alpha2 - alpha1)
            H = min(self.C, self.C + alpha2 - alpha1)
        else:
            L = max(0, alpha2 + alpha1 - self.C)
            H = min(self.C, alpha2 + alpha1)

        if L == H:
            return 0

        k11 = self.kernel_func(x1, x1)
        k12 = self.kernel_func(x1, x2)
        k22 = self.kernel_func(x2, x2)
        eta = k11 + k22 - 2 * k12
        if eta > 0:
            alpha2_new = alpha2 + y2 * (E1 - E2) / eta
            if alpha2_new >= H:
                alpha2_new = H
            elif alpha2_new <= L:
```

```
        alpha2_new = L
else:
    # Abnormal case for eta <= 0, treat this scenario as no
    ↪ progress
    return 0
# Numerical tolerance
# if abs(alpha2_new - alpha2) < self.eps:   # this is slower
# below is faster, not degrade the SVM performance
if abs(alpha2_new - alpha2) < self.eps * (alpha2 +
↪ alpha2_new + self.eps):
    return 0
alpha1_new = alpha1 + s * (alpha2 - alpha2_new)
# Numerical tolerance
if alpha1_new < self.eps:
    alpha1_new = 0
elif alpha1_new > (self.C - self.eps):
    alpha1_new = self.C
# Update threshold
b1 = b - E1 - y1 * (alpha1_new - alpha1) * k11 - y2 *
↪ (alpha2_new - alpha2) * k12
b2 = b - E2 - y1 * (alpha1_new - alpha1) * k12 - y2 *
↪ (alpha2_new - alpha2) * k22
if 0 < alpha1_new < self.C:
    self.b = b1
elif 0 < alpha2_new < self.C:
    self.b = b2
else:
    self.b = 0.5 * (b1 + b2)
# Update weight vector for linear SVM
if self.is_linear_kernel:
    self.w = self.w + y1 * (alpha1_new - alpha1) * x1 \
                    + y2 * (alpha2_new - alpha2) * x2
self.alphas[i1] = alpha1_new
self.alphas[i2] = alpha2_new
# Error cache update
## if alpha1 & alpha2 are not at bounds, the error will be 0
self.error[i1] = 0
self.error[i2] = 0
i_list = [idx for idx, alpha in enumerate(self.alphas) \
              if 0 < alpha and alpha < self.C]
for i in i_list:
    self.error[i] += \
```

```
                    y1 * (alpha1_new - alpha1) * self.kernel_func(x1,
                    ↪  self.X[i,:]) \
                  + y2 * (alpha2_new - alpha2) * self.kernel_func(x2,
                  ↪  self.X[i,:]) \
                  + (self.b - b)
        return 1

 def examine_example(self, i2):
    y2 = self.y[i2]
    alpha2 = self.alphas[i2]
    E2 = self.get_error(i2)
    r2 = E2 * y2
    if ((r2 < -self.tol and alpha2 < self.C) or (r2 > self.tol and
    ↪  alpha2 > 0)):
        if len(self.alphas[(0 < self.alphas) & (self.alphas <
        ↪  self.C)]) > 1:
            if E2 > 0:
                i1 = np.argmin(self.error)
            else:
                i1 = np.argmax(self.error)
            if self.take_step(i1, i2):
                return 1
        # loop over all non-zero and non-C alpha, starting at a
        ↪  random point
        i1_list = [idx for idx, alpha in enumerate(self.alphas) \
                        if 0 < alpha and alpha < self.C]
        i1_list = np.roll(i1_list,
        ↪  np.random.choice(np.arange(self.m)))
        for i1 in i1_list:
            if self.take_step(i1, i2):
                return 1
        # loop over all possible i1, starting at a random point
        i1_list = np.roll(np.arange(self.m),
        ↪  np.random.choice(np.arange(self.m)))
        for i1 in i1_list:
            if self.take_step(i1, i2):
                return 1
    return 0

def fit(self):
    loop_num = 0
    numChanged = 0
    examineAll = True
```

```
while numChanged > 0 or examineAll:
    if loop_num >= self.max_iter:
        break
    numChanged = 0
    if examineAll:
        for i2 in range(self.m):
            numChanged += self.examine_example(i2)
    else:
        i2_list = [idx for idx, alpha in enumerate(self.alphas)
        ↪  \
                       if 0 < alpha and alpha < self.C]
        for i2 in i2_list:
            numChanged += self.examine_example(i2)
    if examineAll:
        examineAll = False
    elif numChanged == 0:
        examineAll = True
    loop_num += 1
```

При $C = 1, kernel =' rbf', max_iter = 10, tol = 1e-1, eps = 1e-1$
Accuracy:0.9921436588103255
Время работы: 10 минут на выборке из 900 объектов.
Визуализация:

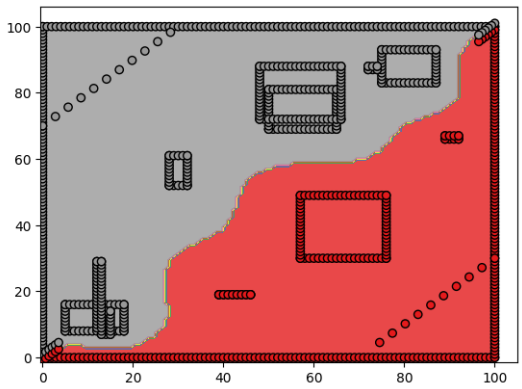

Рис. 6

2.8 Извлечение координат траектории и подсчет точек пути.

Алгоритм извлечения пути(рис. 7):

1)(x_0, y_0) - начальная точка. Задаем константу радиуса поиска R.
2)
$x_{i+1} = x_i + Rcos(\phi)$
$y_{i+1} = y_i + Rsin(\phi)$
$0 \leq \phi \leq 2\pi$
$(x_i - x_{i-1}, y_i - y_{i-1}) \cdot (x_{i+1} - x_i, y_{i+1} - y_i) \geq 0$ - путь не может резко повернуть более чем на 90 градусов. Ограничение на скалярное произведение необходимо для продвижения по траектории строго в одном направление.
3)prob= model.predict$_{proba}$([x$_{i+1}$, y_{i+1}])[0] Если точка лежит на разделяющей поверхности, $prob \approx 0.5$. Для различных ϕ ищем точку с prob максимально близким к 0.5 и сохраняем ее в массив.

Этот метод можно использовать как для получения координат траектории так и для сравнения длин траекторий(рис. 8).

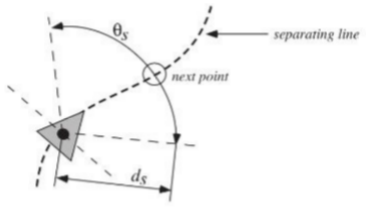

Fig. 3. Search for the next point on the separating surface.

Рис. 7

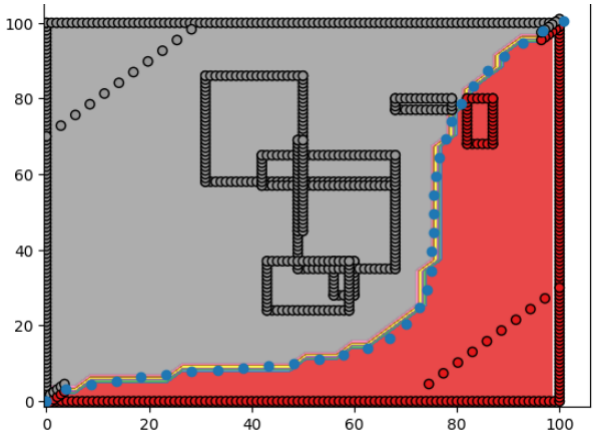

Рис. 8

3 Библиографические ссылки.

1. notebook:https: //colab.research.google.com/drive/1xfIpMxopuhnPYiKf56cVe8Jb0FnPwS0
2. В.В.Вьюгин, Математические основы теории машинного обучения и прогнозирования - 2013;
3. К.В.Воронцов, Лекции по методу опорных векторов - 2007;
4. Jun Miura, Support Vector Path Planning - 2006;
5. Курс: Методы оптимизации в машинном обучении, 2012, Методы внутренней точки
6. https://lucien-east.github.io/2022/07/30/Implement-SVM-with-SMO-from-scratch/