Санкт-Петербургский государственный университет
Факультет прикладной математики — процессов управления
Кафедра технологии программирования

Соловьев
Сергей
Алексеевич

Оптимизация выбора начальных параметров в методе опорных векторов

Заведующий кафедрой,
кандидат физ.-мат. наук,
доцент — Сергеев С. Л.

Научный руководитель,
кандидат физ.-мат. наук,
доцент — Добрынин В. Ю.

Рецензент,
кандидат тех. наук,
доцент — Матросов А. В.

Санкт-Петербург
2010 г.

Содержание

Введение

Машинное обучение или обучение по прецедентам — активно развивающийся раздел математики, который позволяет решать широкий спектр прикладных задач: медицинская и техническая диагностика, предсказание свойств конечной продукции, классификация месторождений полезных ископаемых, автоматическое распознавание спама, рубрикация текстов, распознавание изображений, рукописных символов, речи, идентификация подписей, оценивание заемщиков (для банков), оценивание привлекательности инвестиционных проектов (для инвесторов), предсказание ухода клиентов, обнаружение мошенничества и т. д. Более подробно обо всех этих задачах рассказано в работе [4].

Одним из самых популярных и, по мнению некоторых исследователей, эффективных методов решения задач машинного обучения является метод опорных векторов. Первые шаги в разработке этого метода были предприняты еще в 60-е годы коллективом советских математиков под руководством В. Н. Вапника. В 90-е годы метод был значительно доработан и обобщен на нелинейный случай. После этих доработок метод получил мировую известность и стал называться «метод опорных векторов», также известный как “Support Vector Machines” или “SVM”. Описанию задач машинного обучения и метода опорных векторов посвящены разделы 1, 2 данной работы.

Автор использовал метод опорных векторов для решения задачи регрессии imat2009, предлагаемой компанией Yandex в рамках конкурса «Интернет-математика 2009» [5] (раздел 3.1). При решении задач машинного обучения в первую очередь производят сбор данных и приводят их к признаковой форме. Однако данный этап был сделан организаторами конкурса. Кроме того, уже на тот момент времени существовало несколько эффективных программных реализаций метода опорных векторов: SVM^{light} [10] и LIBSVM [7] (раздел 3.2). Таким образом, решение задачи imat2009 сводилось к поиску таких начальных параметров для метода опорных векторов, которые обеспечат наилучшее качество решения. Программый пакет LIBSVM содержит программу `gridregression.py`, которая позволяет подобрать оптимальный вектор начальных параметров на конечном множестве-сетке. Подход, лежащий в основе данный программы, называется в кругу исследователей метода опорных векторов coarse grid search [9].

Основная проблема, которая возникла при применении подхода coarse grid search к задаче imat2009 — это долгое время поиска оптимального вектора начальных параметров по причине большого объема данных. Так, на выбор начальных параметров для метода SVM при использовании одного персонального компьютера средней производительности потребовалось бы около года (раздел 5.1), что было недопустимо во временных рамках конкурса. Данная проблема решалась в двух направлениях:

- Разработка алгоритма, основанного на подходе coarse grid search, и позволяю-

щего выбирать начальные параметры значительно быстрее стандартного подхода (раздел 5).

- Алгоритм coarse grid search позволяет исследовать разные векторы начальных параметров независимо друг от друга, что дает возможность распределения и распараллеливания вычислений. Эта возможность реализована в программе `gridregression.py`, однако в данной работе разработана и использована более агрессивная стратегия параллельных и распределенных вычислений. Это также позволяет сократить время поиска начальных параметров.

Основная идея предлагаемого алгоритма заключается в поиске оптимального вектора начальных параметров на подмножествах обучающей выборки (раздел 5.3). Зависимость времени поиска начальных параметров от объема обучающей выборки нелинейная. Так, при поиске на подмножестве в два раза меньше обучающей выборки, время поиска сокращается более, чем в два раза (раздел 5).

Таким образом, основное исследование данной работы направлено на разработку, оценку и выявление недостатков альтернативного, более быстрого метода выбора начальных параметров метода опорных векторов.

1 Задачи машинного обучения

В этом разделе рассматриваются основные понятия машинного обучения: постановка задачи, обучающая выборка, обучаемый алгоритм, модель алгоритмов, функция потерь и функционал качества, оценка обобщающей способности, признаковое описание объектов и т. д. Затем эти понятия используются в описании метода опорных векторов для задач классификации и регрессии.

Описание данного раздела в значительной степение опирается на следующие публикации [4], [6].

1.1 Основные понятия

Пусть имеется множество *объектов* X, множество *ответов* Y, а также существует *целевая функция* $y^*\colon X \to Y$, значения которой известны на некотором конечном подмножестве объектов $\{x_1, \ldots, x_\ell\} \subset X$. Пары «объект-ответ» (x_i, y_i) называются *прецедентами*. Совокупность пар $T^\ell = \{x_i, y_i\}_{i=1}^{\ell} \subset (X \times Y)$ называется *обучающей выборкой*.

Задача *машинного обучения* заключается в том, чтобы восстановить функциональную зависимость между объектами и ответами, то есть построить отображение $a(x)\colon X \to Y$, которое удовлетворяет следующим требованиям [4]:

- Отображение $a(x)$ должно допускать эффективную компьютерную реализацию. По этой причине его называют *алгоритмом*.
- Алгоритм $a(x)$ должен воспроизводить на объектах обучающей выборки заданные ответы: $a(x_i) = y_i$, $i = 1, \ldots, \ell$. В зависимости от задачи равенство здесь может пониматься как точное или как приближенное.
- Алгоритм $a(x)$ должен обладать *обобщающей способностью*: достаточно точно приближать целевую функцию $y^*(x)$ не только на объектах обучающей выборки, но и на всем множестве X.
- На алгоритм $a(x)$ могут накладываться некоторые *априорные ограничения*: непрерывность, гладкость, монотонность, устойчивость, и т. п. В некоторых случаях задается функциональный вид (*модель*) алгоритма $a(x)$.[1]

1.2 Разновидности задач машинного обучения

В зависимости от вида множества ответов Y, существуют следующие задачи машинного обучения: *классификация на M непересекающихся классов*, *классификация на*

[1] В частности, функциональный вид алгоритма задается в методе опорных векторов.

M пересекающихся классов, *восстановление регрессии*, *прогнозирование*. В рамках данной работы будут рассмотрены следующие задачи машинного обучения:

- $Y = \{1, \ldots, M\}$ — задача классификации на M непересекающихся классов. В этом случае множество объектов X делится на классы $K_y = \{\, x \in X \mid y^*(x) = y \,\}$.
- $Y = \mathbb{R}$ — задача восстановления регрессии.

Определение. *Модель алгоритмов* — параметрическое семейство отображений A, из которого выбирается искомый алгоритм $a(x)$:

$$A = \{\, \varphi(x, \gamma) \mid \gamma \in \Gamma \,\}, \tag{1}$$

где $\varphi\colon X \times \Gamma \to Y$ — некоторый алгоритм, Γ — *пространство параметров*.

Процесс подбора параметров модели по обучающей выборке называется *настройкой* или *обучением* алгоритма. В результате настройки выбирается единственный алгоритм $a \in A$, который будет приближать целевую функцию y^*. Следует различать такие понятия, как *параметры модели алгоритмов* и *начальные параметры метода машинного обучения*. Основная часть исследований данной работы направлена на выбор последнего типа параметров.

Определение. *Методом обучения* называется отображение $\mu\colon T^\ell \to A$, которое произвольной конечной выборке $\{x_i, y_i\} \subset X \times Y$ ставит в соответствие алгоритм $a\colon X \to Y$. Метод обучения μ, как и сам алгоритм a, должен допускать эффективную программную реализацию.

Таким образом, в задачах машинного обучения различается два этапа:

1. *Обучение*. Метод μ по обучающей выборке T^ℓ строит алгоритм $a = \mu(T^\ell)$.
2. *Применение*. Алгоритм a получает на вход *новые* объекты $x \in X$ для получения ответов $y = a(x)$.

1.3 Функция потерь и функционал качества

Определение. *Функция потерь* — это неотрицательная функция $\mathcal{L}(a, x)$, которая показывает величину ошибки алгоритма a на объекте x. Если $\mathcal{L}(a, x) = 0$, то ответ $y = a(x)$ называется *корректным*.

Определение. Функционал

$$Q(a,\, T^\ell) = \frac{1}{\ell}\sum_{i=1}^{\ell} \mathcal{L}(a, x_i). \tag{2}$$

называется *функционалом качества* алгоритма a на выборке T^ℓ.

Ниже приведены примеры функции потерь:

- *Индикатор несовпадения с правильным ответом*:

$$\mathcal{L}(a, x) = [a(x) \neq y^*(x)]; \tag{3}$$

- *Индикатор существенного отклонения от правильного ответа*, где ε — заданный порог точности:

$$\mathcal{L}(a, x) = [\,|a(x) - y^*(x)| \geqslant \varepsilon\,]; \tag{4}$$

- *Квадратичная функция потерь*:

$$\mathcal{L}(a, x) = (a(x) - y^*(x))^2; \tag{5}$$

  Для квадратичной функции потерь функционал Q называется *средней квадратичной ошибкой* алгоритма a на выборке T^ℓ.

1.4 Проблема переобучения и обобщающая способность алгоритма

Если алгоритм a доставляет минимум функционалу $Q(a, T^\ell)$ на обучающей выборке T^ℓ, то это еще не гарантирует, что он будет хорошо приближать целевую зависимость на произвольной *контрольной выборке* $V^k = \{x'_i, y'_i\}_{i=1}^k$. Когда качество работы алгоритма на новых объектах, не вошедших в состав обучения, оказывается существенно хуже, чем на обучающей выборке, говорят об эффекте *переобучения*. *Обобщающая способность* метода μ характеризуется величиной $Q(\mu(T^\ell), V^k)$, при условии, что выборки являются представительными [4].

Существуют *теоретические* и *эмпирические* оценки обобщающей способности методов обучения μ. Некоторые теоретические оценки приведены в работе [4]: *состоятельность метода обучения* μ, *функционал среднего риска*.

Теоретические оценки позволяют предсказывать качество алгоритмов достаточно точно, но для многих практически интересных случаев эти оценки либо неизвестны, либо завышены. В тех случаях, когда не удается воспользоваться теоретическими

оценками, применяют эмпирические. В качестве эмпирической оценки часто используют *оценку скользящего контроля* (*cross-validation*).

1.5 Оценка скользящего контроля

Идея этого подхода заключается в следующем. Пусть дана выборка $T^L = \{x_i, y_i\}_{i=1}^L$. Ее разбивают N различными способами на две непересекающихся подвыборки: обучающую T_n^ℓ длины ℓ и контрольную V_n^k длины $k = L - \ell$. Для каждого разбиения $n = 1, \ldots, N$ строится алгоритм $a_n = \mu(T_n^\ell)$ по обучающей выборке T_n^ℓ, а затем вычисляется значение функционала качества $Q_n = Q(a_n, V_n^k)$ для контрольной выборки V_n^k. Усредненная сумма значений Q_n называется *оценкой скользящего контроля*:

$$CV(\mu, T^L) = \frac{1}{N} \sum_{i=1}^{N} Q(\mu(T_n^\ell), V_n^k). \tag{6}$$

Существенным недостатком этой оценки является тот факт, что задачу обучения приходится решать N раз, что сопряжено со значительными вычислительными затратами. Этот недостаток и является основной темой исследования данной работы.

Существует несколько типов оценки скользящего контроля в зависимости от способа разбиения выборки T^L. Ниже кратко рассмотрены два из них.

1.5.1 K-fold cross-validation

Выборка T^L разбивается на K непересекающихся подмножеств. Одно из подмножеств используется в качестве контрольной выборки, а объединение $K-1$ остальных подмножеств — в качестве обучающей выборки. Каждое из K подмножеств используется только один раз в качестве контрольной выборки. Таким образом, количество разбиений $N = K$.

1.5.2 Leave-one-out cross-validation

Этот вид оценки скользящего контроля является частным случаем метода K-fold cross-validation, при котором каждое подмножество, которое участвует в разбиении, состоит ровно из одного элемента T^L. В этом случае количество разбиений $K = \ell$.

1.6 Признаковое описание объектов

Объекты обучающей выборки часто имеют сложную природу. Примерами таких объектов могут выступать пациенты в больнице, торговые предприятия, тексты документов, запросы для поисковой машины. Для каждого обучающего объекта можно

определить набор *признаков*, которые позволяют представить этот объект в виде числового вектора.

Определение. *Признаком* (*feature*) называется отображение $f\colon x \in X \to D_f$, описывающее результат измерения некоторой характеристики объекта, где D_f — заданное множество.

В зависимости от множества допустимых значений D_f признаки делятся на следующие типы:

- *бинарный* признак: $D_f = \{0, 1\}$;
- *номинальный* признак: D_f — конечное множество;
- *порядковый* признак: D_f — конечное упорядоченное множество;
- *количественный* признак: $D_f = \mathbb{R}$.

Пусть имеется набор признаков $f_1, \ldots, f_n$. Вектор $\mathbf{x} = (f_1(x), \ldots, f_n(x))$ называется *признаковым описанием* объекта $x \in X$. Совокупность признаковых описаний всех объектов выборки T^ℓ, записанную в виде таблицы размером $\ell \times n$ называют *матрицей объектов-признаков*, которая является стандартным и наиболее распространенным способом описания объектов:

$$F = (f_j(x_i))_{\ell \times n} = \begin{pmatrix} f_1(x_1) & \ldots & f_n(x_1) \\ \ldots & \ldots & \ldots \\ f_1(x_\ell) & \ldots & f_n(x_\ell) \end{pmatrix} \tag{7}$$

Далее в рамках данной работы будем предполагать, что для обучающих объектов используется признаковое описание: $X \subset D_{f_1} \times \cdots \times D_{f_n}$, а обучающие объекты будут обозначаться как векторы.

1.7 Некоторые эмпирические принципы машинного обучения

В данном подразделе будут кратко описаны некоторые эмпирические принципы, на которых основаны алгоритмы машинного обучения.

Все эти принципы в той или иной степени *эвристические*, то есть они опираются не только на строгие математические обоснования, но в значительной степени на соображения здравого смысла. Краткий обзор основных признаков приведен в работе [4, § 1.3].

Для описания метода опорных векоторов полезно рассмотреть следующие два принципа.

1.7.1 Принцип регуляризации

При использовании чрезмерно сложных моделей алгоритмов часто возникает эффект переобучения. С одной стороны, модели, обладающие избыточным числом свободных параметров, позволяют точнее воспроизводить ответы на материале обучения. С другой стороны, попытка описать обучающие данные точнее, чем в принципе позволяет суммарная погрешность измерений, может привести к катастрофическому снижению обобщающей способности [4].

Один из способов ограничения сложности состоит в том, чтобы отойти от *принципа минимизации эмпирического риска* и добавить к функционалу $Q(a, T^\ell)$ штрафное слагаемое, наказывающее чрезмерно сложные модели:

$$\mu(T^\ell) = \arg\min_{a \in A}(Q(a, T^\ell) + \tau C(a)) \tag{8}$$

Число τ называется *параметром регуляризации*, функционал $C(a)$ выражает сложность алгоритма a.

1.7.2 Принцип разделимости

Принцип разделимости относится к задачам классификации. Он предполагает, что объекты в пространстве X могут быть разделены некоторой поверхностью. Например, *линейная разделимость* двух классов в евклидовом пространстве $\mathbb{R}^n$ означает, что существует гиперплоскость, относительно которой точки одного класса лежат по одну сторону гиперплоскости.

Пусть множество ответов $Y = \{+1, -1\}$, а обучающие объекты описываны признаками: $\mathbf{x} = (f_1(x), \ldots, f_n(x))$. *Линейным разделяющим правилом* или *линейным классификатором* называется алгоритм классификации вида

$$a(\mathbf{x}) = \text{sign}(\alpha_1 f_1(x) + \cdots + \alpha_n f_n(x)), \tag{9}$$

где весовые коэффициенты $\alpha_1, \ldots, \alpha_n$ являются параметрами алгоритма и настраиваются по обучающей выборки T^ℓ.

Неявно принцип разделимости присутствует всегда, когда алгоритм классификации строится в виде $a(\mathbf{x}) = C(b(\mathbf{x}))$, где функция $b(\mathbf{x})$ дает числовую оценку принадлежности объекта $\mathbf{x}$ классу $y_i \in Y$ и называется *алгоритмическим оператором*; функция $C(b)$ отображает оценку принадлежности в номер класса и называется *решающим правилом*. В случае $Y = \{+1, -1\}$ естественно выбрать функцию $C(b) = \text{sign}(b)$.

Если $Y = \{+1, -1\}$, то величина $m_i = b(\mathbf{x}_i)y_i$ называется *отступом* (*margin*) объекта $\mathbf{x}_i$ от разделяющей поверхности. Отступ m_i отрицателен тогда, и только тогда, когда алгоритм допускает ошибку на объекте $\mathbf{x}_i$.

2 Метод опорных векторов

Рассматриваемый метод — нелинейное обобщение *метода обобщенного портрета* (*Generalized Portrait*), который был разработан в 60-е годы коллективом советских математиков под руководством В. Н. Вапника [19, 2]. Именно после значительных доработок и обобщения на нелинейный случай метод получил мировую известность и стал называться *методом опорных векторов* (*Support Vector Machines*) [18].

В основе метода лежит принцип разделимости (раздел 1.7.2). Идея заключается в следующем. Пусть объекты являются векторами евклидова пространства. Необходимо построить такую разделяющую гиперплоскость, которая наиболее удалена от обучающих объектов: *оптимальную разделяющую гиперплоскость*.

При более подробном рассмотрении метода будет видно, что он также тесно связян с принципом регуляризаци (раздел 1.7.1).

Метод опорных векторов позволяет решать как задачу классификации, так и задачу регрессии. Современное изложение метода можно найти в работах [6] (задача классификации) и [16] (задача регрессии).

2.1 Гиперплоскость в эвклидовом пространстве

Введённые ниже определения и утверждения основаны на материале книги [1].

Определение. *Гиперплоскостью* H в евклидовом пространстве $\mathbb{R}^n$ называется линейное подпространство пространства $\mathbb{R}^n$ размерности $n-1$.

Теорема (Векторное уравнение гиперплоскости)**.** Всякая гиперплоскость H в евклидовом пространстве $\mathbb{R}^n$ с вектором нормали $\mathbf{w}$, проходящая через точку $\mathbf{x}_0 \in \mathbb{R}^n$, задается векторным уравнением:

$$\mathbf{w} \cdot \mathbf{x} + b = 0, \tag{10}$$

где $b = -\mathbf{x}_0 \cdot \mathbf{w}$, $x \in \mathbb{R}^n$.

Следствие (Алгебраическое уравнение гиперплоскости)**.** Всякая гиперплоскость H в евклидовом пространстве $\mathbb{R}^n$ с вектором нормали $\mathbf{w} = (\alpha_1, \ldots, \alpha_n)$, проходящая через точку $\mathbf{x}_0 \in \mathbb{R}^n$, определяется алгебраическим уравнением:

$$\alpha_1 x_1 + \cdots + \alpha_n x_n - b = 0, \tag{11}$$

где $b = -\mathbf{x}_0 \cdot \mathbf{w}$, $x \in \mathbb{R}^n$.

Теорема. Расстояние от произвольной точки $\mathbf{x}_1 \in \mathbb{R}^n$ евклидового пространства до фиксированной гиперплоскости $H\colon \mathbf{w} \cdot \mathbf{x} - b = 0$ определяется по формуле:

$$d(\mathbf{x}_1, H) = \frac{|\mathbf{x_1} \cdot \mathbf{w} - b|}{\|\mathbf{w}\|}, \tag{12}$$

где $\|\mathbf{w}\| = \sqrt{\alpha_1^2 + \cdots + \alpha_n^2}$ — евклидова норма вектора $\mathbf{w}$.

2.2 Задача квадратичного программирования

Задача квадратичного программирования является частным случаем задачи *оптимизации.*

Определение. Задача поиска минимума функции $f(\mathbf{x})$, удовлетворяющей ограничениям

$$\begin{aligned} f(\mathbf{x}) &= \frac{1}{2}\,\mathbf{x}^{\mathrm{T}} Q \mathbf{x} + \mathbf{c}^{\mathrm{T}} \mathbf{x}, \\ g_i(\mathbf{x}) &\leqslant 0,\ i = 1, \ldots, m \\ h_j(\mathbf{x}) &= 0,\ j = 1, \ldots, k. \end{aligned} \tag{13}$$

где Q — матрица $n \times n$; $\mathbf{c}$ — вектор-столбец $n \times 1$; f, g_i, h_j — непрерывно-дифференцируемые функции, называется *задачей квадратичного программирования.*

Утверждение. Если матрица Q является *положительно-определенной*, а g_i, h_j — линейные функции, то существует *единственное* решение $\mathbf{x}^*$ задачи (13), а условия Каруша-Куна-Таккера (ККТ) [13] являются *необходимыми* и *достаточными* условиями точки минимума $\mathbf{x}^*$.

2.3 Случай линейной разделимости

Рассмотрим задачу классификации: пусть $T^\ell = \{\mathbf{x}_i, y_i\}_{i=1}^{\ell}$ — обучающая выборка, множество объектов $X \subset \mathbb{R}^n$ (объекты описаны при помощи признаков), $\mathbf{x} = (x^1, \ldots, x^n) \in X$, множество ответов $Y = \{+1, -1\}$. Пусть имеется гиперплоскость H, которая отделяет объекты двух классов («разделяющая гиперплоскость»), удовлетворяющая уравнению:

$$H\colon \mathbf{w} \cdot \mathbf{x} - b = 0, \tag{14}$$

где $\mathbf{w}$ — вектор нормали к гиперплоскости. Если такая гиперплоскость существуют, то выборку T^ℓ называют *линейно разделимой*, а в противном случае – *линейно неразделимой.*

Рассматривают линейный классификатор следующего вида:

$$a(\mathbf{x}) = \operatorname{sign}(\mathbf{w} \cdot \mathbf{x} - b) = \operatorname{sign}\left(\sum_{i=1}^{\ell} \alpha_i x^i - b\right) \tag{15}$$

В данном случае $\mathbf{w}$, b — параметры алгоритма.

Пусть d_+ (d_-) — расстояние от разделяющей гиперплоскости H до близжайшего объекта из положительного (отрицательного) класса:

$$\begin{aligned} d_+ &= \min_{i=1,\ldots,\ell} (\mathbf{x}_i \cdot \mathbf{w} - b) \quad \text{для } y_i = +1 \\ d_- &= -\max_{i=1,\ldots,\ell} (\mathbf{x}_i \cdot \mathbf{w} - b) \quad \text{для } y_i = -1 \end{aligned} \tag{16}$$

Тогда ни один объект $\mathbf{x}_i \in X$ не попадет внутрь *разделяющей полосы*

$$\{\, \mathbf{x} \mid -d_- \leqslant \mathbf{x} \cdot \mathbf{w} - b \leqslant d_+ \,\} \tag{17}$$

Из множества всех возможных *разделяющих* гиперплоскостей нас интересует оптимальная разделяющая гиперплоскость, которая наиболее удалена от объектов обоих классов. Поскольку гиперплоскости, определяющие границы разделяющей полосы параллельны, оптимальная гиперплоскость будет проходить ровно посередине разделяющей полосы. В этом случае $d_- = d_+$.

Для удобства производят следующую нормировку: делят вектор нормали $\mathbf{w}$ и постоянную b на $d_- = d_+$. Алгоритм $a(\mathbf{x})$ от этого не изменится. В формулах (16), (17) будет выполняться: $d_+ = d_- = 1$. Дальнейшие рассуждения будут проводиться для нормированных гиперплоскостей.

Условия того, что обучающие объекты не попадают внутрь разделяющей полосы, можно записать в следующем виде [3, 6]:

$$y_i(\mathbf{x}_i \cdot \mathbf{w} - b) \geqslant 1, \quad i = 1, \ldots, \ell \tag{18}$$

Ширина разделяющей полосы должна быть максимально велика, чтобы эффективнее классифицировать объекты. Ее определяют следующие параллельные (с общим вектором нормали $\mathbf{w}$) гиперплоскости:

$$\begin{aligned} H_- &= \{\, \mathbf{x} \mid \mathbf{w} \cdot \mathbf{x} - b = -1 \,\} \\ H_+ &= \{\, \mathbf{x} \mid \mathbf{w} \cdot \mathbf{x} - b = +1 \,\} \end{aligned} \tag{19}$$

Пусть $\mathbf{x}_- \in H_-$. Тогда $\mathbf{w} \cdot \mathbf{x}_- - b = -1$, $\mathbf{w} \cdot \mathbf{x}_- = b - 1$. Ширина разделяющей полосы

равна расстоянию от объекта $\mathbf{x}_-$ до плоскости H_+:

$$d(\mathbf{x}_-, H_+) = \frac{|\mathbf{x}_- \cdot \mathbf{w} - b - 1|}{\|\mathbf{w}\|} = \frac{|b - 1 - b - 1|}{\|\mathbf{w}\|} = \frac{2}{\|\mathbf{w}\|}. \tag{20}$$

Таким образом, ширина разделяющей полосы тем больше, чем меньше норма вектора $\mathbf{w}$.

С учетом линейных ограничений (18), получим следующую задачу квадратичного программирования для поиска оптимальной разделяющей гиперплоскости:

$$\begin{cases} \frac{1}{2}\|\mathbf{w}\|^2 \to \min\limits_{\mathbf{w},b}; \\ y_i(\mathbf{x}_i \cdot \mathbf{w} - b) \geqslant 1, \quad i = 1, \ldots, \ell. \end{cases} \tag{21}$$

Построим функцию Лагранжа для прямой задачи.

Это следует сделать хотя бы по двум причинам. Во-первых, ограничения-неравенства (18) заменятся множителями Лагранжа и с ними будет проще работать. Во-вторых, обучающие объекты будут присутствовать в задаче только в форме скалярного произведения векторов, что позволит удобно обобщить алгоритм на нелинейный случай [6].

$$L_P = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{\ell} \lambda_i y_i(\mathbf{x}_i \cdot \mathbf{w} - b) + \sum_{i=1}^{\ell} \lambda_i, \tag{22}$$

где λ_i — множители Лагранжа для ограничений (18), $\lambda = (\lambda_1, \ldots, \lambda_\ell)$. Тогда задача (21) сводится к следующей задаче минимизации функции Лагранжа:

$$\begin{cases} L_P(\mathbf{w}, b, \lambda, \mathbf{x}) \to \min\limits_{\mathbf{w},b,\lambda}; \\ \nabla_\lambda L_P = 0; \\ \lambda_i \geqslant 0, \quad i = 1, \ldots, \ell. \end{cases} \tag{23}$$

Здесь $\nabla_\lambda L_P$ — вектор-градиент функции L_P по переменным $\lambda = (\lambda_1, \ldots, \lambda_\ell)$, а равенство его нулю означает равенство нулю всех его компонент: $\frac{\partial L_P}{\partial \lambda_i} = 0$, $i = 1, \ldots, \ell$. Задача (23) является задачей *выпуклого программирования*, поэтому она имеет единственное решение, и для нее имеет место двойственная задача:

$$\begin{cases} L_D(\lambda, \mathbf{x}) \to \max\limits_{\lambda}; \\ \nabla_{\mathbf{w}} L_D = 0; \\ \nabla_b L_D = 0; \\ \lambda_i \geqslant 0, \quad i = 1, \ldots, \ell. \end{cases} \tag{24}$$

Здесь L_D – двойственная функция Лагранжа.

Необходимым условием точки *максимума* в задаче (24) является одновременное вы-

полнение условий $\nabla_{\mathbf{w}} L_P = 0$, $\nabla_b L_P = 0$. Это приводит нас к следующим соотношениям:

$$\begin{cases} \mathbf{w} = \sum\limits_{i=1}^{\ell} \lambda_i y_i \mathbf{x}_i; \\ \sum\limits_{i=1}^{\ell} \lambda_i y_i = 0. \end{cases} \tag{25}$$

Стоит заметить, что из первого из соотношений (25) следует, что вектор нормали оптимальной разделяющей гиперплоскости является линейной комбинацией объектов обучающей выборки $\mathbf{x}_i$.

При помощи соотношений (25) построим двойственную функцию Лагранжа L_D, зная вид функции L_P:

$$\begin{aligned} L_D &= \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{\ell} \lambda_i y_i (\mathbf{x}_i \cdot \mathbf{w} - b) + \sum_{i=1}^{\ell} \lambda_i \\ &= \frac{1}{2}\|\mathbf{w}\|^2 - \mathbf{w} \cdot \sum_{i=1}^{\ell} \lambda_i y_i \mathbf{x}_i + \sum_{i=1}^{\ell} \lambda_i y_i b + \sum_{i=1}^{\ell} \lambda_i \\ &= \frac{1}{2}\|\mathbf{w}\|^2 - \|\mathbf{w}\|^2 + \sum_{i=1}^{\ell} \lambda_i = -\frac{1}{2}\|\mathbf{w}\|^2 + \sum_{i=1}^{\ell} \lambda_i \\ &= -\frac{1}{2} \sum_{i=1}^{\ell} \lambda_i y_i (\mathbf{x}_i \cdot \mathbf{w}) + \sum_{i=1}^{\ell} \lambda_i = -\frac{1}{2} \sum_{i=1}^{\ell} \lambda_i y_i \mathbf{x}_i \cdot \sum_{j=1}^{\ell} \lambda_j y_j \mathbf{x}_j + \sum_{i=1}^{\ell} \lambda_i \\ &= -\frac{1}{2} \sum_{i,j=1}^{\ell} \lambda_i \lambda_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{\ell} \lambda_i \end{aligned}$$

Окончательно получим:

$$L_D = -\frac{1}{2} \sum_{i,j=1}^{\ell} \lambda_i \lambda_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{\ell} \lambda_i \tag{26}$$

Таким образом, задача поиска оптимальной разделяющей гиперплоскости в случае линейной разделимости сводится к двойственной задаче максимизации функции лагранжа L_D:

$$\begin{cases} -\frac{1}{2} \sum\limits_{i,j=1}^{\ell} \lambda_i \lambda_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum\limits_{i=1}^{\ell} \lambda_i \to \max\limits_{\lambda}; \\ \lambda_i \geqslant 0; \\ \sum\limits_{i=1}^{\ell} \lambda_i y_i = 0. \end{cases} \tag{27}$$

Итак, задача записана в такой форме, что объекты обучающей выборки входят в нее только в виде скалярного произведения.

С учетом соотношений (25) линейный классификатор (15) принимает следующий

вид:

$$a(\mathbf{x}) = \operatorname{sign}\left(\sum_{i=1}^{\ell} \lambda_i y_i \mathbf{x}_i \cdot \mathbf{x} - b\right) \tag{28}$$

Значение b можно вычислить из равенства $b = \mathbf{w} \cdot \mathbf{x}_i - y_i$, взяв произвольный обучающий объект $\mathbf{x}_i$.

2.4 Случай линейной неразделимости

Если попытаться решить задачу (27) для линейно неразделимой выборки, решение не будет найдено. Нужно позволить алгоритму допускать ошибки на обучающих объектах. Для этого вводят дополнительные переменные $\xi_i \geqslant 0$, характеризующие величину ошибки на объектах $\mathbf{x}_i$ [18]. При этом нужно потребовать, чтобы суммарная ошибка была минимальна: $\sum\limits_{i=1}^{\ell} \xi_i \to \min$.

С учетом вышесказанного, рассмотрим задачу поиска оптимальной разделяющей гиперплоскости в случае линейно неразделимой выборки:

$$\begin{cases} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum\limits_{i=1}^{\ell} \xi_i \to \min\limits_{\mathbf{w},b}; \\ y_i(\mathbf{x}_i \cdot \mathbf{w} - b) \geqslant 1 - \xi_i, \quad i = 1, \ldots, \ell; \\ \xi_i \geqslant 0. \end{cases} \tag{29}$$

В данном случае параметр регуляризации C является *начальным параметром* метода опорных векторов и выбирается пользователем. Он является одним из параметров, которые влияют на обобщающую способность алгоритма. «Большее значение C соответствует большему штрафу за ошибки и меньшему значению ширины разделяющей полосы.» [6]

Построим функцию Лагранжа для задачи (29):

$$L_P = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{\ell} \xi_i - \sum_{i=1}^{\ell} \lambda_i \left(y_i(\mathbf{x}_i \cdot \mathbf{w} - b) - 1 + \xi_i\right) - \sum_{i=1}^{\ell} \mu_i \xi_i, \tag{30}$$

где μ_i — множители Лагранжа для условий неотрицательности $\xi_i \geqslant 0$. Поскольку задача (29) является задачей *выпуклого квадратичного программирования*, условия

ККТ являются необходимым и достаточным условием оптимальной точки:

$$\nabla_{\mathbf{w}} L_P = w - \sum_{i=1}^{\ell} \lambda_i y_i \mathbf{x}_i = 0 \tag{31}$$

$$\nabla_b L_P = -\sum_{i=1}^{\ell} \lambda_i y_i = 0 \tag{32}$$

$$\nabla_\xi L_P = C - \lambda_i - \mu_i = 0 \tag{33}$$

$$y_i(\mathbf{x}_i \cdot \mathbf{w} - b) - 1 + \xi_i \geqslant 0 \tag{34}$$

$$\xi_i \geqslant 0 \tag{35}$$

$$\lambda_i \geqslant 0 \tag{36}$$

$$\mu_i \geqslant 0 \tag{37}$$

$$\lambda_i \left(y_i(\mathbf{x}_i \cdot \mathbf{w} - b) - 1 + \xi_i\right) = 0 \tag{38}$$

$$\mu_i \xi_i = 0 \tag{39}$$

Из соотношений $\xi_i \geqslant 0$ и $\lambda_i + \mu_i = C$ следует, что $0 \leqslant \lambda_i \leqslant C$. Из условий ККТ также следует, что существует три допустимых сочетация переменных ξ_i, λ_i, μ_i, поэтому все объекты $\mathbf{x}_i$ делятся на следующие три типа [3]:

1. $\lambda_i = 0$, $\mu_i = C$, $\xi_i = 0$, $y_i(\mathbf{x}_i \cdot \mathbf{w} - b) \geqslant 1$. Объект $\mathbf{x}_i$ классифицируется правильно и не влияет на решение. Такие объекты называются *периферийными*.

2. $0 < \lambda_i < C$, $0 < \mu_i < C$, $\xi_i = 0$, $y_i(\mathbf{x}_i \cdot \mathbf{w} - b) = 1$. Объект $\mathbf{x}_i$ классифицируется правильно и лежит в точности на границе разделяющей полосы. Такие объекты называются *опорными векторами* (*support vector*).

3. $\lambda_i = C$, $\mu_i = 0$, $\xi_i > 0$, $y_i(\mathbf{x}_i \cdot \mathbf{w} - b) < 1$. Объект $\mathbf{x}_i$ либо лежит внутри разделяющей полосы, но классифицируется правильно, либо попадает на границу классов, либо вообще относится к чужому классу. Во всех этих случаях объект $\mathbf{x}_i$ называется *опорным нарушителем*.

Определение. Если $\lambda_i > 0$, то объект обучающей выборки $\mathbf{x}_i$ называется *опорным вектором*.

В силу соотношения $\lambda_i + \mu_i = C$, в функции Лагранжа (30) обнуляются все члены, содержащие переменные ξ_i, μ_i, и двойственная задача для случая линейно неразделимой выборки приобретает вид, похожий на случай линейно разделимой выборки (27):

$$\begin{cases} -\frac{1}{2} \sum\limits_{i,j=1}^{\ell} \lambda_i \lambda_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum\limits_{i=1}^{\ell} \lambda_i \to \max\limits_{\lambda}; \\ 0 \leqslant \lambda_i \leqslant C; \\ \sum\limits_{i=1}^{\ell} \lambda_i y_i = 0. \end{cases} \tag{40}$$

Понятно, что способ поиска b и вид линейного классификатора (28) сохраняется в случае линейной неразделимости.

2.5 Метод опорных векторов в задаче регрессии

Пусть известны: $T^\ell = \{\mathbf{x}_i, y_i\}_{i=1}^\ell$ — обучающая выборка, множество объектов $X \subset \mathbb{R}^n$ (объекты описаны при помощи признаков), $\mathbf{x} = (x^1, \ldots, x^n) \in X$, множество ответов $Y = \mathbb{R}$. Рассмотрим задачу построения алгоритма $a(\mathbf{x})$, удовлетворяющего следующим условиям [16]:

1. алгоритм $a(\mathbf{x})$ вопроизводит ответы на обучающих объектах с ошибкой, не превосходящей ε. Стоить отметить, что это условие равносильно равенству нулю функционала качества (2), построенного для функции потерь (4):

$$Q(a, T^\ell) = \sum_{i=1}^{\ell} [\, |a(\mathbf{x}) - y^*(x)| \geqslant \varepsilon \,] = 0. \tag{41}$$

2. поверхность, описываемая уравнением $a(\mathbf{x}) = 0$ должна быть максимально *пологой*.[2]

В линейном случае, когда $a(\mathbf{x}) = \mathbf{x}_i \cdot \mathbf{w} - b$, пологость указанной выше поверхности означает минимизацию нормы вектора нормали [16]. С другой стороны, в силу первого из условий, которым должен удовлетворять алгоритм, значение функции $a(\mathbf{x})$ на объектах обучающей выборки не должно выходить за пределы ε-трубки. Это приводит к следующей задаче квадратичного программирования, похожей на (21):

$$\begin{cases} \frac{1}{2}\|\mathbf{w}\|^2 \to \min\limits_{\mathbf{w},b}; \\ y_i - \mathbf{x}_i \cdot \mathbf{w} - b \leqslant \varepsilon; \\ \mathbf{x}_i \cdot \mathbf{w} - b - y_i \leqslant \varepsilon. \end{cases} \tag{42}$$

Аналогично случаю линейной неразделимости в методе опорных векторов для задачи классификаци, в случае регрессии допускают возможность алгоритма $a(\mathbf{x})$ допускать ошибки, а именно: воспроизводить объекты обучающей выборки за пределами ε-трубки. Для этого аналогичным образом вводятся переменные ошибок $\xi_i, \xi_i^* \geqslant 0$, а в минимизируемый функционал добавляется суммарная ошибка, умноженная на

[2] В англоязычной литературе степень, с которой некоторая поверхность приближается (аппроксимирует) к плоскости, называется *flatness*. Перевода этого термина на русский не найдено.

параметр регуляризации C, имеющий такой же смысл, как в случае классификации:

$$\begin{cases} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{\ell}(\xi_i + \xi_i^*) \to \min\limits_{\mathbf{w},b}; \\ y_i - \mathbf{x}_i \cdot \mathbf{w} - b \leqslant \varepsilon + \xi_i; \\ \mathbf{x}_i \cdot \mathbf{w} - b - y_i \leqslant \varepsilon + \xi_i^*; \\ \xi_i, \xi_i^* \geqslant 0. \end{cases} \tag{43}$$

В случае классификации рассматривалась функция Лагранжа для прямой и двойственной задачи оптимизации. Проводя аналогичные рассуждения для задачи (43), получим следующую двойственную задачу выпуклого квадратичного программирования для задачи регрессии:

$$\begin{cases} -\frac{1}{2}\sum_{i,j=1}^{\ell}(\lambda_i - \lambda_i^*)(\lambda_j - \lambda_j^*)\mathbf{x}_i \cdot \mathbf{x}_j \\ -\varepsilon\sum_{i=1}^{\ell}(\lambda_i + \lambda_i^*) + \sum_{i=1}^{\ell} y_i(\lambda_i - \lambda_i^*) \to \min\limits_{\lambda,\lambda^*}; \\ \sum_{i=1}^{\ell}(\lambda_i - \lambda_i^*) = 0; \\ 0 \leqslant \lambda_i, \lambda_i^* \leqslant C. \end{cases} \tag{44}$$

Здесь λ_i, λ_i^* — двойственные переменные функции Лагранжа.

Алгоритм $a(\mathbf{x})$ будет иметь следующий вид:

$$a(\mathbf{x}) = \sum_{i=1}^{\ell}\left((\lambda_i - \lambda_i^*)\,\mathbf{x}_i \cdot \mathbf{x}\right) - b \tag{45}$$

Способы вычисления значения b описаны в [16].

2.6 Разновидности метода опорных векторов

В кругах исследователей метода опорных векторов задачу (27) часто называют *"C-SV Classification"* (*"C-SVC"*), а задачу (44) — *"ε-SV Regression"* (*"ε-SVR"*). Существуют еще несколько разновидностей метода опорных векторов для задач классификации и регрессии, и ниже для них будут приведены постановки без выкладок.

Особенностью приведенных ниже методов является введение параметра ν, который обладает следующими свойствами:

- $\nu \in [0, 1]$;
- ν является верхней границей доли ошибок алгоритма;
- ν является нижней границей доли опорных векторов;

Более подробно оба алгоритма описаны в статье [15].

ν-SV Classification

$$\begin{cases} -\frac{1}{2}\sum\limits_{i,j=1}^{\ell}\lambda_i\lambda_j y_i y_j \mathbf{x}_i\cdot\mathbf{x}_j \to \max\limits_{\lambda}; \\ 0 \leqslant \lambda_i \leqslant \frac{1}{\ell}; \\ \sum\limits_{i=1}^{\ell}\lambda_i y_i = 0; \\ \sum\limits_{i=1}^{\ell}\lambda_i \geqslant \nu. \end{cases} \tag{46}$$

ν-SV Regression

$$\begin{cases} -\frac{1}{2}\sum_{i,j=1}^{\ell}(\lambda_i-\lambda_i^*)(\lambda_j-\lambda_j^*)\mathbf{x}_i\cdot\mathbf{x}_j \\ +\sum_{i=1}^{\ell} y_i(\lambda_i-\lambda_i^*) \to \min\limits_{\lambda,\lambda^*}; \\ 0 \leqslant \lambda_i, \lambda_i^* \leqslant \frac{C}{\ell}; \\ \sum_{i=1}^{\ell}(\lambda_i-\lambda_i^*) = 0; \\ \sum_{i=1}^{\ell}(\lambda_i+\lambda_i^*) \leqslant C\nu \end{cases} \tag{47}$$

2.7 Нелинейный случай

Существует еще один подход к решению задачи машинного обучения в случае линейно неразделимой выборки: разделять классы при помощи *нелинейной поверхности*.

В этом разделе будем обозначать пространство объектов буквой $\mathfrak{L}$. Рассмотрим отображение Φ исходного пространства объектов $\mathfrak{L}$ в евклидово пространство (возможно даже бесконечномерное) $\mathcal{H}$:

$$\Phi\colon \mathfrak{L} \mapsto \mathcal{H} \tag{48}$$

Алгоритм $a(\mathbf{x})$ зависит от обучающих объектов $\mathbf{x}$ только в форме скалярных произведений $\mathbf{x}\cdot\mathbf{x}_j$, где $\mathbf{x}$ — классифицируемый объект, а $\mathbf{x}_j$ — обучающий. Значит в пространстве $\mathcal{H}$ алгоритм $a(\mathbf{x})$ будет зависеть от обучающих объектов в форме скалярных произведений следующего вида: $\Phi(\mathbf{x})\cdot\Phi(\mathbf{x}_j)$. Таким образом, в пространстве $\mathcal{H}$ должна быть определена операция скалярного произведения. Для этого подойдет, например, евклидово или гильбертово пространство.

Если теперь определить *ядро* (*kernel function*) $K(\mathbf{x},\mathbf{y}) = \Phi(\mathbf{x})\cdot\Phi(\mathbf{y})$, то его можно использовать в обучаемом алгоритме, даже не имея представления о явном виде функции $\Phi(\mathbf{x})$. Тогда алгоритм (28) будет представлен следующим образом:

$$a(\mathbf{x}) = \operatorname{sign}\left(\sum_{i=1}^{\ell}\lambda_i y_i K(\mathbf{x}_i,\mathbf{x}) - b\right) \tag{49}$$

Определение. Функция $K\colon \mathfrak{L}\times\mathfrak{L} \to \mathbb{R}$ называется *ядром*, если она представима в виде $K(\mathbf{x},\mathbf{y}) = \Phi(\mathbf{x})\cdot\Phi(\mathbf{y})$ при некотором отображении $\Phi\colon \mathfrak{L}\mapsto\mathcal{H}$, где $\mathcal{H}$ — пространство со скалярным произведением.

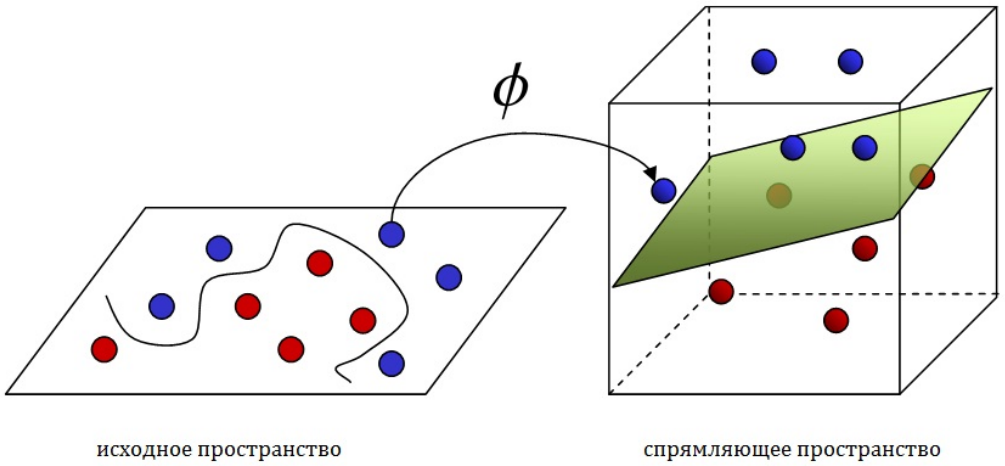

Рис. 1: Иллюстрация нелинейного ядра в методе опорных векторов

Пространство $\mathcal{H}$ обладает размерностью, как правило, большей, чем $\mathcal{L}$, поэтому можно надеяться, что в $\mathcal{H}$ выборка окажется линейно разделимой. По этой пространство $\mathcal{H}$ иногда называют *спрямляющим пространством.*

Пример. Пусть $\mathcal{L} = \mathbb{R}^2$, $\mathbf{x} = (x_1, x_2) \in \mathcal{L}$, $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})^2$. Построим отображение $\Phi\colon R^2 \mapsto \mathcal{H}$:

$$\begin{aligned} K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})^2 &= (x_1, x_2) \cdot (y_1, y_2) \\ &= (x_1 y_1 + x_2 y_2)^2 = x_1^2 y_1^2 + x_2^2 y_2^2 + 2 x_1 x_2 y_1 y_2 \\ &= (x_1^2, x_2^2, \sqrt{2} x_1 x_2) \cdot (y_1^2, y_2^2, \sqrt{2} y_1 y_2). \end{aligned} \tag{50}$$

Итак, $\Phi(x_1, x_2) = (x_1^2, x_2^2, \sqrt{2}\, x_1 x_2)$. Значит в пространстве $\mathcal{H}$ разделяющая поверхность будет иметь вид (гипер)плоскости:

$$\alpha_1 h_1(x_1, x_2) + \alpha_2 h_2(x_1, x_2) + \alpha_3 h_3(x_1, x_2) = b, \tag{51}$$

где $\alpha_1, \alpha_2, \alpha_3$ — координаты вектора нормали $\mathbf{w}$; $(h_1, h_2, h_3) \in \mathcal{H}$. В пространстве $\mathcal{L}$ разделяющая поверхность — это кривая второго порядка:

$$\alpha_1 x_1^2 + \alpha_2 x_2^2 + \sqrt{2}\, \alpha_3 x_1 x_2 = b, \tag{52}$$

где $\alpha_1, \alpha_2, \alpha_3$ — некоторые параметры, описывающие кривую (нетрудно показать, что в общем случае это эллипс).

Уравнения (51), (52) — суть одно и тоже, но в разных пространствах описывают разные поверхности. Таким образом, данное ядро позволяет разделить внутреннюю и наружную часть эллипса, что невозможно сделать при помощи линейного классификатора.

В случае замены в алгоритме $a(\mathbf{x})$ скалярного произведения объектов $\mathbf{x}_i \cdot \mathbf{x}_j$ на функ-

цию ядра $K(\mathbf{x}_i, \mathbf{x}_j)$, мы строим оптимальную разделяющую гиперплоскость в спрямляющем пространстве $\mathcal{H}$, которой в пространстве $\mathfrak{L}$ соотвествует некоторая поверхность, в общем случае отличная от плоскости.

2.7.1 Класс ядер

Необходимое и достаточное условие принадлежности функции $K : \mathfrak{L} \times \mathfrak{L} \to \mathbb{R}$ к классу ядер дает теорема Мерсера [14]. Однако эта теорема не дает способ определения ядра для данной конкретной задачи. Ядро K, как и параметр регуляризации C, определяют по методу скользящего контроля, что не является оптимальным решением. «На сегодняшний день проблема выбора ядра, оптимального для данной конкретной задачи, остается открытой теоретической проблемой.» [3]

На практике, как правило, в качестве ядра выбирают одну из функций, о которой известно, что она является ядром, а затем подбирают параметры этого ядра методом скользящего контроля.

2.7.2 Примеры ядер

Следующие ядра часто используются в кругах исследователей и пользователей метода опорных векторов:

- *Линейное.* $K(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y}$;
- *Полиномиальное (однородное).* $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})^d$;
- *Полиномиальное (неоднородное).* $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^d$;
- *Радиальная базисная функция (RBF).* $K(\mathbf{x}, \mathbf{y}) = e^{-\gamma \|\mathbf{x}-\mathbf{y}\|^2}$, $\gamma > 0$;
- *Радиальная базисная функция Гаусса.* $K(\mathbf{x}, \mathbf{y}) = e^{-\frac{\|\mathbf{x}-\mathbf{y}\|^2}{2\sigma^2}}$.

3 Задача imat2009: постановка задачи и способы решения

В данном разделе будет дана математическая постановка задачи imat2009 [5], а также предложены способы решения этой задачи. Далее будет описан стандартный подход для выбора начальных параметров, предлагаемый авторами LIBSVM и будет показано, почему такой подход неэффективен для задачи imat2009.

3.1 Математическая постановка задачи

Объекты обучающей выборки представляют собой пары «запрос-документ»:

$$X = \{(q, d)\}_{i=1}^{\ell}, \tag{53}$$

где объем выборки $\ell = 97290$. Запросы и документы находятся в соотношении «один к многим». Общее количество запросов составляет 9124.

Множество ответов обучающей выборки представляет собой набор релевантностей для пар (q, d):

$$Y = [0, 4], \tag{54}$$

где значение 4 соответствует высокой релевантности, а значение 0 — для нерелевантных пар (q, d).

Для каждого обучающего объекта определены 245 признаков, каждый из которых либо бинарный, либо непрерывный. Значения непрерывных признаков нормированы на интервал $[0, 1]$.

Для проверки результатов и определения рейтинга специалистами Yandex предоставлена контрольная выборка V^k объемом в 115643 элементов.

Требуется решить задачу регрессии для обучающей выборки $T^\ell = \{r_i, (q_i, d_i)\}_{i=1}^{\ell}$.

3.2 Выбор программного обеспечения для метода опорных векторов

Исчерпывающий список ПО для метода опорных векторов можно найти по адресу [17]. Среди всех программ особого внимания заслуживают SVM^{light} [10] и LIBSVM [7], поскольку обладают достаточно эффективной реализацией.

Обе программы умеют решать все задачи машинного обучения, упомянутые в данной работе: C-SVC, ν-SVC, ε-SVR, ν-SVR; поддерживают нелинейный случай для всех типов ядер, описанных в разделе 2.7, оценку скользящего контроля, а также много других особенностей, выходящих за рамки данной работы.

Каждая из программ обладает рядом особенностей, важных для решения текущей задачи. Ниже перечислены преимущества LIBSVM и недостатки SVM^{light}, которые определили выбор ПО в пользу LIBSVM:

- LIBSVM:
  - Поставляется с богатым набором программ, которые позволяют упростить решение задачи. Самые важные из них в рамках текущей задачи — программы, реализующие алгоритм выбора параметров coarse grid search для

задач классификации и регрессии, которые поддерживают *распределенное вычисление*. Ниже они будут описаны подробнее.

- Реализует оценку скользящего контроля K-fold cross validation для *всех* типов задач машинного обучения, перечисленных выше.
- Поддерживает *распараллеливание вычислений* при помощи OpenMP.
- Поставляется с подробной документацией, в которой описаны практические подходы к решению задач методом опорных векторов: подготовка данных, выбор ядра и параметров.

- SVM^{light}:
  - Реализует оценку скользящего контроля leave-one-out cross validation только для задач классификации.
  - Поставляется с набором научных статей, описывающих алгоритм, однако самые важные сведения находятся в отдельной книге [11].

Ниже будут подробно описаны особенности решения текущей задачи средствами LIBSVM, а также будут рассмотрены подходы к оптимизации поиска параметров.

3.3 Практические рекомендации от авторов LIBSVM

Авторы LIBSVM предлагают ряд практических рекомендаций, собранных в алгоритм решения задач классификации методом опорных векторов[3] [9]:

1. Перед *обучением* алгоритма данные необходимо нормировать, иначе признаки с более широким диапазоном значений будут иметь бóльшую значимость. Кроме того, большие значения признаков могут приводить к *вычислительным* проблемам. Рекоммендуется нормировать данные на интервал $[0, 1]$ или $[-1, 1]$.
2. Рекоммендуется в первую очередь исследовать ядро RBF:
   - Ядро RBF (как и любое другое нелинейное ядро) позволяет устранить проблему *линейной* неразделимости посредством *нелинейной* разделяющей поверхности в исходном пространстве $\mathfrak{L}$.
   - В работе [12] показано, что при некоторых значениях (C, γ) ядро RBF[4] ведет себя также, как и линейное ядро с некоторым параметром регулярициии $\tilde{C}$.

[3] Эти рекомендация относятся к задачам *классификации*, но многие из них справедливы в случае *регрессии*.

[4] На самом деле в [12] рассматривается гауссово ядро, но нетрудно видеть, что ядро RBF получается из гауссова ядра заменой $\sigma^2 = \frac{1}{2\gamma}$.

- Значение RBF ядра ограничено: $0 < e^{-\gamma\|\mathbf{x}-\mathbf{y}\|^2} \leqslant 1$; $\mathbf{x}, \mathbf{y} \in \mathfrak{L}$, поэтому с ним возникает меньше вычислительных затруднений, чем, например, с полиномиальным ядром.

3. Использовать оценку скользящего контроля для нахождения наилучшей пары параметров (C, γ).

В задаче imat2009 шаг 1 уже сделан компанией Yandex.

3.4 Класс алгоритмов выбора начальных параметров

Ранее было указано, что в случае метода опорных векторов существует два вида параметров: параметры, которые настраиваются на этапе обучения посредством метода обучения μ и параметры, выбираемые пользователем (начальные параметры). Начальными параметрами в случае метода опорных являются параметр регуляризации C, порог точности ε (в задачах регрессии), а также набор параметров ядра (в нелинейном случае).

Для дальнейшего исследования расширим понятие *модели алгоритмов* (1), введенной в разделе 1.2, так, чтобы оно включало в себя начальные параметры:

$$A = \{\, \varphi(x, \gamma, \omega) \mid \gamma \in \Gamma, \omega \in \Omega\}, \tag{55}$$

где $\varphi\colon X \times \Gamma \times \Omega \to Y$ — некоторый алгоритм, Γ — *пространство параметров, определяемых на этапе обучения*, $\Omega \subset \mathbb{R}^n$ — *пространство начальных параметров*, n — количество начальных параметров.

Метод обучения обучения μ в этом случае будет иметь следующий вид:

$$\mu\colon (T^\ell, \omega) \in (X \times Y) \times \Omega \to a \in A \tag{56}$$

Метод обучения ставит в соответствие обучающей выборке T^ℓ и набору начальных параметров ω некоторый алгоритм $a(x)$.

Задача выбора начальных параметров заключается в том, чтобы для данной обучающей выборки T^ℓ найти такой ω^* в заранее выбранном подмножестве Ω, что алгоритм $a^* = \mu(T^\ell, \omega^*)$ обладает наилучшей обобщающей способностью. Множество всевозможных отображений $(T^\ell, \Omega) \to A$, которые для различных подмножеств Ω определяют *оптимальный вектор начальных параметров* ω^*, назовем *классом алгоритмов выбора начальных параметров*.

Алгоритм $a^* \in A$ обладает наилучшей обобщающей способностью на подмножестве Ω (далее будет также обозначаться Ω), если он доставляет минимум оценке скользящего

контроля:

$$\omega^* = \arg\min_{\omega \in \Omega} CV(\mu(T^\ell, \omega), T^\ell) \tag{57}$$

4 Задача imat2009: выбор начальных параметров

В данном разделе будут описаны алгоритмы выбора начальных параметров, которые предлагают авторы LIBSVM для задач классификации и регрессии. Будет оценено время, которое потребуется этим алгоритмам для задачи imat2009.

4.1 Алгоритм выбора начальных параметров в задаче классификации

Алгоритм выбора начальных параметров в LIBSVM для задач классификации реализован в программе `grid.py`. Этот алгоритм исследует RBF-ядро для различных значений параметра регуляризации C и параметра ядра γ.

Рассмотрим множество, представляющее собой равномерное разбиение отрезка на n частей:

$$R(x_0, \delta x, n) = \{\, x \in \mathbb{R} \mid x = x_0 + (i-1)\delta x, i = 1, \ldots, n\} \tag{58}$$

Здесь x_0 — левый конец отрезка, δx — расстояние между точками разбиения, n — количество точек разбиения.

Далее построим множество-сетку (grid), на котором и выбираются начальные параметры в программе `grid.py`:

$$G_2 = \{\, (C_i, \gamma_i) \in \mathbb{R}^2 \mid C_i = 2^\alpha, \gamma_i = 2^\beta, (\alpha, \beta) \in R(c_0, \delta c, n_c) \times R(g_0, \delta g, n_g)\} \tag{59}$$

В описываемом алгоритме значения переменных по умолчанию таковы: $c_0 = -1$, $\delta c = 1$, $n_c = 8$, $g_0 = 0$, $\delta g = -1$, $n_g = 9$.

Итак, алгоритм, реализованный в `grid.py`, является, согласно введенной терминологии, алгоритмом выбора начальных параметров, в котором $\Omega = G_2$, $\omega = (C, \gamma)$. В качестве оценки скользящего контроля используется K-fold cross validation.

Поскольку всевозможные пары начальных параметров $(C, \gamma) \in G_2$ независимы, решение задачи выбора наилучшей пары можно распределить на несколько вычислительных машин. Так, программа grid.py позволяет распределить вычисления в кластере.

4.2 Алгоритм выбора начальных параметров в задаче регрессии

В задачах регрессии добавляется еще один начальный параметр — порог точности ε. Разумно ли наряду с параметрами C и γ включить в поиск еще и ε? С одной стороны, смысл гиперпараметра ε в том, чтобы гарантировать, что отклонение алгоритма $a(\mathbf{x})$ от целевой зависимости y^* не превосходит значение этого параметра. С другой стороны, значение оценки скольщяего контроля меняется при разных значениях ε: $CV = CV(C, \gamma, \varepsilon)$. Авторы LIBSVM сочли разумным включить порог точности ε в пространство начальных параметров Ω алгоритма выбора начальных параметров. Программа, реализующая этот алгоритм, называется `gridregression.py`.

В качестве пространства начальных параметров Ω рассматривается множество следующего вида:

$$G_3 = \{\, (C_i, \gamma_i, \varepsilon) \in \mathbb{R}^3 \mid C_i = 2^\alpha, \gamma_i = 2^\beta, \varepsilon_i = 2^\eta, \\ (\alpha, \beta, \eta) \in R(c_0, \delta c, n_c) \times R(g_0, \delta g, n_g) \times R(e_0, \delta e, n_e)\} \tag{60}$$

Для параметра порога точности задаются следующие начальные значения: $e_0 = -8$, $\delta e = -1$, $n_e = 8$. Для параметра регуляризации и параметра RBF-ядра задаются такие же значения, как и в случае классификации.

4.3 Оценка времени работы алгоритмов выбора начальных параметров LIBSVM для задачи imat2009

Для решения задачи imat2009 использовался кластер, состоящий из 10 вычислительных машин. Время, которое требуется для решения задачи imat2009 при помощи программы `svm-train`, входящей в состав LIBSVM составляет приблизительно три часа. Нетрудно при этом оценить, сколько раз придется решать задачу для выбора начальных параметров в случае регресии: $Kn_cn_gn_e$, где K — параметр для оценки скользящего контроля K-fold cross validation. Рекоммендуемое значение параметра $K = 5$. В то же время $n_c = 8$, $n_g = 9$, $n_e = 8$. Тогда задачу придется решать 2880 раз, а время, которое потребуется на решение, составит приблизительно один год.

Приведенные выше оценки позволяют заключить, что применение алгоритма выбора начальных параметров для задачи регресии в его превоначальном виде непригодно для конкурса. Поэтому ниже будет предложен альтернативный алгоритм, который позволит сократить время поиска начальных параметров.

Объем выборки v	Время обучения $t(v)$ секунд
1000	0.985
5000	32.585
10000	95.77
20000	504.853
40000	1491.037
60000	3328.48
97290	9243.36

Таблица 1: Время обучения в зависимости от объема выборки

5 Алгоритм быстрого выбора начальных параметров

Общая идея альтернативных подходов в выборе начальных параметров заключается в выделении подмножества исходной обучающей выборки и проведения выбора на нем. Это позволяет сузить множество для поиска начальных параметров, а тем самым и сократить время поиска. В этом разделе будет показано, насколько время обучения на подмножестве отличается от времени обучения на исходной выборке. Далее будет предложен алгоритм, который использует эту особенность.

5.1 Оценка времени обучения на подмножестве обучающей выборки для задачи imat2009

В состав пакета LIBSVM входит программа `subset.py`, которая позволяет из обучающей выборки объемом ℓ построить подмножество объемом $\ell' \leqslant \ell$, в котором элементы выбираются из обучающей выборки *случайным образом*.

При помощи этой программы из обучающей выборки строим подмножества следующих объемов: $1000, 5000, 10000, 20000, 40000, 60000$. Для каждого подмножества объемом v_i измеряем время обучения алгоритма t_i (Таблица 1), а затем, проинтерполировав набор пар $\{(v_i, t_i)\}$ строим график зависимости времени обучения от объема выборки (Рис. 2). Вычисления производились на комьютере с двумя четырехядерными процессорами Intel® Xeon® CPU E5440 2.83GHz для ядра RBF, параметров (C, γ, ε), устанавливаемых программой `svm-train` по умолчанию, оперативной памятью, отводимой для вычислений, размером 500 мегабайт.

Из таблицы 1 видно, что время, требуемое на обучение полной выборки, в 284 раза превосходит время обучения подмножества этой выборки, состоящего из 5000 элементов. При этом объем выборок отличается только в 19 раз. Аналогичные выводы можно сделать для подмножества любого объема: проигрыш во времени значительно превосходит проигрыш в размере.

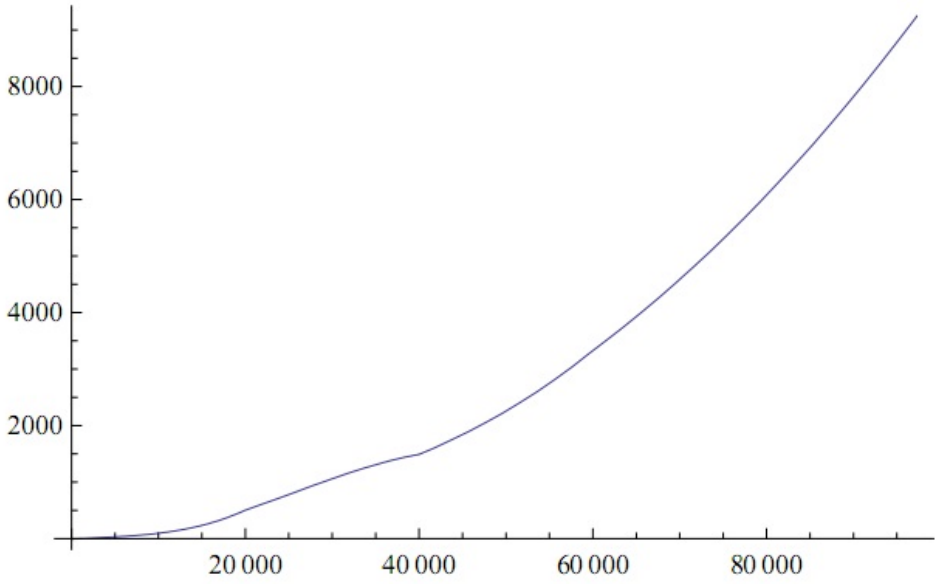

Рис. 2: График интерполяции функции времени обучения алгоритма от объема обучающей выборки

Из сказанного выше заключаем, что использование подмножеств обучающей выборки является чрезвычайно эффективным подходом для для уменьшения временных затрат.

5.2 Распределение и распараллеливание вычислений

В разделе 4.1 было сказано, что программа `grid.py` (как и `gridregression.py`) поддерживают распределенные вычисления. Это реализовано следующим обзразом: сначала строится список наборов начальных параметров Ω, затем извлекаются n элементов из этого списка и для извлеченных элементов запускаются оценки скользящего контроля на каждой из n вычислительных машин. Как только какая-то вычислительная машина завершает задачу и предоставляет результаты в виде средней квадратичной ошибки, из списка Ω извлекается еще одна задача и передается на решение это вычислительной машине. Так происходит до тех пор, пока список Ω не станет пустым.

Такую стратегию можно оптимизировать с учетом того, что каждый запуск оценки скользящего контроля можно распараллелить на многоядерной или многопроцессорной вычислительной машине.

Итак, пусть количество нод (вычислительных машин) в кластере равно s, количество процессорных ядер на i-ой ноде равно c_i, а количество памяти, отводимой в i-ой ноде на вычисления равно m_i. Будем использовать следующую стратегию для параллельных вычислений:

- Если количество задач $|\Omega| > s$, то распределяем задачи по нодам таким образом, чтобы на одно процессорное ядро каждой ноды приходилось по одной задаче, а количество отводимой памяти на одну задачу равнялось m_i/c_i. В этом случае количество задач, решаемых одновременно, не превосходит $\sum\limits_{i=1}^{s} c_i$.

- Если количество задач $|\Omega| \leqslant s$, то распределяем по одной задаче на каждую ноду, а также распараллеливаем эту задачу на все процессорные ядра этой ноды при помощи OpenMP. В этом случае на одну задачу приходится m_i памяти, а количество задач, решаемых одновременно, не превосходит s.

5.3 Описание алгоритма

Далее будем предполагать, что множество Ω всегда имеет вид (59) или (60), то есть представляет собой равномерную сетку на $\mathbb{R}^n$, у которой каждая компонента всех векторов пропотенцирована по основанию 2.

Введем *функцию оценки меры расстояния* между двумя векторами начальных параметров $\omega_1, \omega_2 \in \Omega$. Если в качестве такой функции взять евклидову меру расстояния, то параметр C будет иметь большую значимость, чем, например, ε. Действительно, при изменении c от -1 до 6 значение 2^c изменяется от 0.5 до 64 (63.5); при изменении же e от -8 до -1 значение 2^e изменяется от 0.00390625 до 0.5 (0.49609375). Если C и ε прологарифмировать по основанию 2, то они будут иметь одинаковую значимость. С учетом сказанного, данная функция оценки меры расстояния будет иметь следующий вид:

$$\rho_\Omega(\omega_1, \omega_2) = \sqrt{\log_2^2 \frac{C_1}{C_2} + \log_2^2 \frac{\gamma_1}{\gamma_2} + \log_2^2 \frac{\varepsilon_1}{\varepsilon_2}}. \tag{61}$$

Для дальнейших исследований необходимо доказать, что введенная выше функция (61) является функцией расстояния (метрикой). Тогда для любых точек $\omega_1, \omega_2, \omega_p \in \Omega$ эта фукнция должна удовлетворять следующим условиям:

1. Положительная определенность
   $\rho_\Omega(\omega_1, \omega_2) \geqslant 0$,
   $\rho_\Omega(\omega_1, \omega_2) = 0 \Leftrightarrow \omega_1 = \omega_2$;

2. Симметрия
   $\rho_\Omega(\omega_1, \omega_2) = \rho_\Omega(\omega_2, \omega_1)$;

3. Неравенство треугольника
   $\rho_\Omega(\omega_1, \omega_2) \leqslant \rho_\Omega(\omega_1, \omega_p) + \rho_\Omega(\omega_p, \omega_2)$;

Первые два свойство имеют очевидные доказательства. Для доказательства свойства «Неравенство треугольника» запишем неравенство Коши в следующей форме:

$$\sqrt{\sum_{i=1}^{3}(a_i+b_i)^2} \leqslant \sqrt{\sum_{i=1}^{3}a_i^2} + \sqrt{\sum_{i=1}^{3}b_i^2} \tag{62}$$
$$a_i, b_i \in \mathbb{R}^3$$

Представим вектора $\omega_1, \omega_2, \omega_p$ в следующем виде:

$$\omega_1 = (\omega_1^1, \omega_1^2, \omega_1^3)$$
$$\omega_2 = (\omega_2^1, \omega_2^2, \omega_2^3)$$
$$\omega_p = (\omega_p^1, \omega_p^2, \omega_p^3)$$

В данном случае мы заменили: $\omega_i^1 = \log_2 C_i$, $\omega_i^2 = \log_2 \gamma_i$, $\omega_i^3 = \log_2 \varepsilon_i$, $i = 1, 2, p$. Далее, делая следующую замену в формуле (62)

$$a_i = \omega_1^i - \omega_p^i$$
$$b_i = \omega_p^i - \omega_2^i$$

и учитывая, что $\Omega \subset \mathbb{R}^3$, мы и получим неравенство треугольника для функции ρ_Ω. Таким образом, мы доказали, что ρ_Ω является метрикой в пространстве Ω.

Введем *функцию сходства* двух подмножеств $\mathcal{A}, \mathcal{B}$ некоторого множества $\mathcal{C}$ следующим образом:

$$\rho(\mathcal{A}, \mathcal{B}) = \frac{2\mu(\mathcal{A} \cap \mathcal{B})}{\mu(\mathcal{A}) + \mu(\mathcal{B})}, \tag{63}$$

где $\mu(\cdot)$ — мера множества. В случае конечных множеств, $\mu(\mathcal{A}) = |\mathcal{A}|$. Значение введенной выше функции удовлетворяет условию $0 \leqslant \rho(\mathcal{A}, \mathcal{B}) \leqslant 1$. Действительно, пусть $\mathcal{A} \cap \hat{\mathcal{A}} = \varnothing$; $\tilde{\mathcal{A}} \subset \mathcal{A}$. Тогда $0 = \rho(\mathcal{A}, \hat{\mathcal{A}}) \leqslant \rho(\mathcal{A}, \tilde{\mathcal{A}}) \leqslant \rho(\mathcal{A}, \mathcal{A}) = 1$.

Описанный ниже алгоритм опирается на следующую *гипотезу*: чем ближе обучающие множества $\mathcal{A}, \mathcal{B} \subset T^\ell$ в смысле функции (63), тем меньше расстояние между оптимальными векторами для этих множеств $\omega_{\mathcal{A}}^*$, $\omega_{\mathcal{B}}^*$ в смысле метрики (61).

Стоит заметить, что если $\mathcal{A} \subseteq \mathcal{B}$, и разность $|\mathcal{B}| - |\mathcal{A}|$ невелика, то эти множества в достаточной степени похожи.

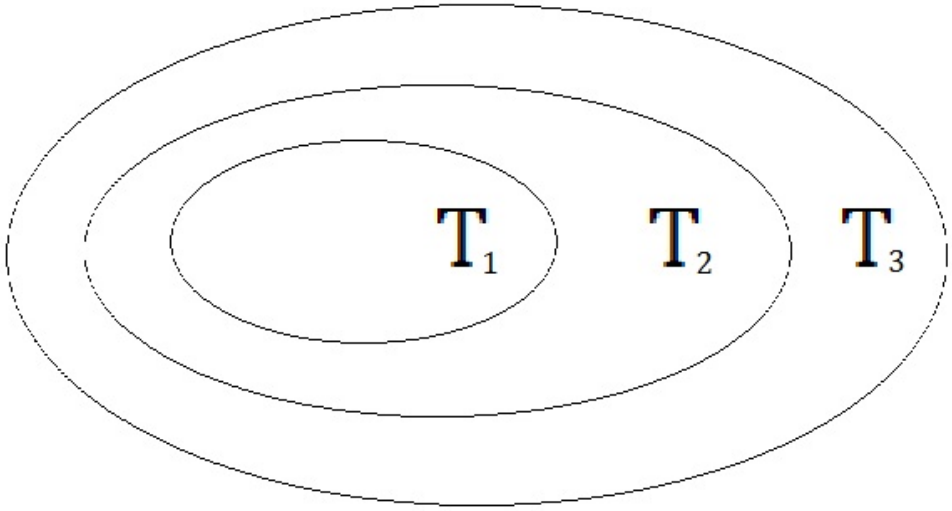

Рис. 3: Порядок вложенности обучающих подмножеств в случае $q = 3$

Введем следующий вид множества начальных параметров Ω:

$$\begin{aligned} S(c,g,e) = \{\, (C,\gamma,\varepsilon) \mid & C = 2^{c+i}, \gamma = 2^{g+j}, \varepsilon = 2^{e+k}; \\ & i = \{-\delta c, 0, \delta c\} \\ & j = \{-\delta g, 0, \delta g\} \\ & j = \{-\delta e, 0, \delta e\} \,\} \end{aligned} \tag{64}$$

По сути, введенное выше множество имеет такую же структуру, как и (60), но состоит всего из 27 элементов, сосредоточенных вокруг некоторой точки (C, γ, ε).

Основная идея алгоритма следующая. Исходную обучающую выборку разбиваем на последовательность схожих подмножеств. Зная оптимальный вектор начальных параметров для одного из подмножеств (например, меньшего по размеру), можно, на основании введенной выше гипотезы утверждать, что оптимальный вектор начальных параметров для соседнего подмножества (например, большего по размеру) находится достаточно близко, а это значительно сужает область поиска. В этом случае в качестве области поиска можно использовать множество S, а не G_3.

Обучающую выборку T^ℓ будем разбивать на q подмножеств так, что каждое следующее подмножество T_k является подмножеством предыдущего T_{k+1}: $T_k \subset T_{k+1}$ (см. рис. 3). При этом множество T_k состоит из некоторого количества (далее мы определим это количество) элементов T_{k+1}, выбранных *случайным образом.*

Пусть количество элементов в каждом подмножестве T_k отличается от количества элементов в T_{k+1} в $\varkappa$ раз:

$$|T_k| = \varkappa |T_{k+1}| \tag{65}$$

Ясно, что $\varkappa$ лежит в интервале $(0, 1)$. Эта переменная показывает, насколько *сходны* каждое множество и его подмножество, согласно введенной выше метрике (63): чем ближе $\varkappa$ к 1, тем более они сходны.

Зафиксируем $\varkappa$ и максимально допусутимый объем $B \in \mathbb{N}$ выборки[5] T_1 и найдем q. Легко видеть, что $|T_1| = \varkappa^{q-1}|T_q|$, где $|T_q| = |T^\ell|$ — объем исходной обучающей выборки. Решая неравенство $\varkappa^{q-1}|T_q| < B$ и принимая во внимание, что $q \in \mathbb{N}$, находим:

$$q = \left\lceil 1 + \log_\varkappa \left(\frac{B}{|T_1|} \right) \right\rceil \tag{66}$$

Оптимальный вектор начальных параметров ω_1^* для выборки T_1 (первое приближение к результату) будем искать на множестве $\Omega_1 = G_3$, а все остальные векторы $\omega_k^*, k = 2, \ldots, q$ — на множестве $\Omega_k = S(\omega_{k-1}^*)$

Алгоритм быстрого выбора начальных параметров:

Вход: Обучающая выборка: T^ℓ; множество поиска оптимальных векторов начальных параметров: $\Omega_1 = G_3$; максимально допустимый объем выборки T_1: $B < |T_q|$; коэффициент сходства подмножеств: $\varkappa \in (0, 1)$;

Выход: ω^* — оптимальный вектор начальных параметров для множества T^ℓ

Шаг 0**:** Разбиваем обучающую выборку T^ℓ на последовательность подмножеств $\{T_k\}_{k=1}^q$ указанным выше способом. Находим ω_1^* как оптимальный вектор начальных параметров на множестве $\Omega_1 = G_3$ для обучающей выборки T_1.

Шаг k**,** $k = 2, \ldots, q$**:** Находим ω_k^* как оптимальный вектор начальных параметров на множестве $\Omega_k = S(\omega_{k-1}^*)$ для обучающей выборки T_k. При $k = q$ получаем результирующий вектор начальных параметров $\omega^* = \omega_q^*$

Таким образом, мы определили алгоритм быстрого выбора начальных параметров для метода опорных векторов, который, по нашим оценкам, должен значительно сократить время поиска. Данный алгоритм был реализован на языке Python (см. приложение А), и в его реализации заложена стратегия распределенных вычислений, описанная в разделе 5.2.

5.4 Испытание алгоритма

Теперь необходимо испытать вышеописанный алгоритм на задаче imat2009, с целью получить вектор начальных параметров за приемлемое время.

[5]При анализе алгоритма будет показано, что именно T_1 оказывает наибольшее влияние на качество найденного вектора начальных параметров.

Так, данный алгоритм был испытан для следующих начальных данных:

$$G_3 = \{\, (2^c, 2^g, 2^e) \mid c = -1, 0, \ldots, 6; g, e = -8, -7, \ldots, 0 \,\}$$
$$\varkappa = \frac{2}{3}$$
$$B = 6000$$

В результате вычислений алгоритма получены следующие начальные параметры:

$$C^* = 1.0$$
$$\gamma^* = 0.0625$$
$$\varepsilon^* = 0.25$$

Средняя квадратичная ошибка для найденных начальных параметров на полной выборке составила 0.517832. Время, затраченное на вычисления, составило приблизительно двое суток.

Согласно оценкам, сделанным в разделе 5.1, время, которое потребовалось алгоритму быстрого выбора, значительно меньше времени, которое потребовалось бы стандартному алгоритму. Таким образом, нам удалось добиться одной из целей, а именно: выбрать начальные параметры за доступное время. Открытым же остается вопрос о том, насколько качественный конечный результат. В частности, нас интересует, насколько значение средней квадратичной ошибки, найденной алгоритмом быстрого поиска будет отличаться от значения этой ошибки, найденного с помощью стандартного алгоритма. Но проблема в том, что даже в рамках длительного исследования у нас нет возможности потратить год на испытание стандартного алгоритма, чтобы затем сравнить его с новым.

Другой проблемный фактор — это рандомизация данных в новом алгоритме: ведь элементы для каждого подмножества T_k в новом алгоритме выбираются случайным образом. Это значит, что, запустив новый алгоритм дважды, есть вероятность получить разные результаты.

В следующем разделе мы попытаемся сформировать такие метрики качества нового алгоритма, которые потребуют минимальных вычислительными затрат и позволят оценить качество найденного вектора начальных параметров.

5.5 Оценка качества алгоритма

Введем следующую *функцию оценки качества* нового алгоритма:

$$E = \frac{\rho_\Omega(\omega_1^*, \omega_q^*)}{(q-1)\sqrt{\delta c^2 + \delta g^2 + \delta e^2}} \tag{67}$$

Номер попытки	Ошибка	Функция E
1	8.2833	0.3795
2	12.5350	0.7563
3	9.0500	0.5844
4	8.2833	0.3514
5	8.2833	0.2940
6	9.2953	0.6186

Таблица 2: Зависимость между функцией и средней квадратичной ошибки нового алгоритма для задачи comp-activ

Здесь ω_1^* — начальное приближение к оптимальному вектору начальных параметров, ω_q^* — результирующий вектор начальных параметров, $\rho_\Omega(\omega_1^*, \omega_q^*)$ — расстояние между этими двумя векторами в смысле метрики (61).

Максимальное количество перемещений вектора ω_k^* при изменении k от 1 до q составляет $q-1$, а максимальная длина одного такого перемещения — $\sqrt{\delta c^2 + \delta g^2 + \delta e^2}$, где $\delta c, \delta g, \delta e$ — шаги разбиения для переменных C, γ, ε соответственно (см. формулу (58)). Поэтому значения данной функции нормированы на интервал $[0; 1]$

Функция E показывает, насколько сильно отличается вектор начальных параметров, выбранный на первой итерации алгоритма, от результирующего вектора начальных параметров. Значения данной функции, близкие к 1, говорят о значительных различиях между этими векторами.

Для исследования возьмем выборку, размер которой значительно меньше, чем выборка imat2009 — comp-activ[6] [8]. Небольшой объем выборки позволит нам испытать оба алгоритма и сравнить результаты.

Сначала был испытан стандартный алгоритм. Средняя квадратичная ошибка на результирующем векторе начальных параметров составила 8.609520. Для испытания быстрого алгоритма, последний был запущен несколько раз, так как, в силу рандомизации данных при применении быстрого алгоритма, результаты могут получаться разными при разных запусках. Результаты испытания внесены в таблицу 2. Графа «Ошибка» в этой таблице означает среднюю квадратичную ошибку.

Из таблицы 2 мы можем видеть, что в большинстве случаев качество нового алгоритма сравнимо с качеством стандартного алгоритма. Более того, в трех случая из шести новый алгоритм обыграл стандартный. Таким образом, в случае задачи comp-activ, новый алгоритм, затрачивая меньше временных ресурсов, достигает аналогичных результатов.

Далее проанализируем значения функции E. По сути, нам необходимо выявить корелляцию между следующими двумя переменными:

[6]Объем выборки для задачи comp-activ составляет 8192, а количество признаков — 27.

- разностью между средними квадратичными ошибками, вычисленными с помощью нового и стандартного алгоритма; чем больше эта разность (с учетом знака разности), тем предположительно бóльшие значения должны быть у функции E, так как они гипотетически характеризуют ошибку
- собственно значениями функции E

На самом деле, коэффициент корелляции, вычисленный для *разности средних квадратичных ошибок* и E, будет равен коэффициенту корелляции между *средней квадратичной ошибкой нового алгоритма* и E, так как в случае стандартного алгоритма средняя квадратичная ошибка постоянна. Это обстоятельство позволяет исследовать введенные выше метрики, вообще не запуская стандартный алгоритм.

Итак, по таблице 2 было вычислено значение коэффициента корелляции между функцией E и средней квадратичной ошибкой, полученной на новом алгоритме: 0.86.

Таким образом, существует сильная корреляция между средней квадратичной ошибкой нового алгоритма и функцией E, хотя эти парамеры, казалось бы, имеют мало общего. Значит функцию E можно применять для оценки качества нашего алгоритма.

Здесь можно сделать еще один интересный вывод. Тот факт, что функция E сильно кореллирует с качеством нового алгоритма указывает на то, что в последовательности подмножеств $T_1, \ldots, T_q$ наибольшее значение имеет самое первое подмножество T_1. По сути, функция оценки качества указывает, повезло ли нам с выбором подмножества T_1 при текущем запуске алгоритма.

6 Заключение

В результате данной работы был разработан и испытан алгоритм быстрого выбора начальных параметров. Ниже приведены основные положения о достигнутых результатах.

- Разработанный алгоритм позволяет выиграть по времени стандартный алгоритм на два порядка при аналогичном качестве.
- Этот алгоритм может обыгрывать по качеству стандартный алгоритм с большой вероятностью.
- Введена и проанализирована функция оценки качества нового алгоритма.
- Существенным недостатком алгоритма является тот факт, что заранее неизвестно, насколько качественно сработает алгоритм. Функцию оценки качества

можно применять только по окончании работы алгоритма. Данный недостаток компенсируется высокой скоростью работы алгоритма.

- Исследование функции качества E, произведенные в разделе 5.5, открывает горизонт для дальнейших исследований предложенного алгоритма. Так, было показано, что наибольшее значение имеет выбор самого первого подмножества T_1 в последовательности подмножеств обучающей выборки. Именно в этом направлении можно продолжить дальнейшие исследования.

A Исходный код программы быстрого выбора начальных параметров

```
#!/usr/bin/env python

from __future__ import with_statement
from subprocess import Popen, PIPE
import os, sys
import Queue
from sys import exit
import string
from string import find, split, join
from threading import Thread, Lock
from time import sleep
from signal import signal, SIGINT
from math import log, sqrt, log10, ceil
import re

is_win32 = (sys.platform == 'win32')
keep = True

svmtrain_exe = "/home/zeally/imat2009/libsvm-2.89/openmp/svm-train"
log_filename = 'gridfast.log'
log_file = open(log_filename, 'w')

def info(label = None, log = True, **info):
    global log_file
    pairs = ["%s=%s" % (k, info[k]) for k in info.keys()]
    str = ''
    if label:
        str = "[%s] %s" % (label, string.join(pairs))
    else:
        str = "%s" % string.join(pairs, ' ')
    if log:
        log_file.write(str + '\n')
    return str
```

```
def emph(str):
    return '{ %s }' % str

class Task:
    def __init__(self, C, gamma, eps, folds, dataset_path):
        self.C = C
        self.gamma = gamma
        self.eps = eps
        self.folds = folds
        self.dataset_path = dataset_path

    def dist(t1, t2):
        return sqrt((t1.C - t2.C)**2 + (t1.gamma - t2.gamma)**2
                + (t1.eps - t2.eps)**2)

    def __str__(self):
        return info(None, False, C=self.C, gamma=self.gamma,
                eps=self.eps, dataset=self.dataset_path)

    def __hash__(self):
        return hash(self.C) ^ hash(self.gamma) ^ hash(self.eps) ^
                hash(self.folds) ^ hash(self.dataset_path)

    def __cmp__(self, o):
        return cmp(self.__hash__(), o.__hash__())

class Result:
    def __init__(self, task, worker, mse, error_message = None):
        self.task = task
        self.worker = worker
        self.mse = mse
        self.error_message = error_message

class EventDispatcher:
    def __init__(self):
        self.listeners = {}

    def subscribe(self, event, callback):
        if not self.listeners.has_key(event):
            self.listeners[event] = []
        self.listeners[event].append(callback)

    def notify(self, event, *args):
        if self.listeners.has_key(event):
            for callback in self.listeners[event]:
                callback(self, *args)

class ComputationError(Exception):
    def __init__(self, message):
```

```
        self.message = message
    def __str__(self):
        return repr(self.message)

class RemoteSvrSolver(EventDispatcher):
    lock = Lock()
    stop_lock = Lock()

    def __init__(self, name, host, memory, cache):
        EventDispatcher.__init__(self)
        self.name = name
        self.memory = memory
        self.cache = cache
        self.subprocess = None
        self.host = host
        self.pid = None
        self.cwd = os.getcwd()
        self.status = 'not started'

    def solve(self, task, cores = 1):
        if self.status == 'abort':
            print info('aborted on solve')
            return
        thread = Thread(name = self.name,
                target = self.do_solve, args = (task, cores))
        self.status = 'starting'
        thread.start()

    def stop(self):
        with RemoteSvrSolver.lock:
            if self.status in ('not started', 'starting', 'started'):
                self.fail()
            elif self.status == 'failed':
                self.status = 'killing'
                print info('killing', node=self.name, pid=self.pid)
                Popen('ssh %s "kill -15 %s"' %
                        (self.host, self.pid), shell=True, stdout = PIPE)
                self.status = 'killed'
                print info('killed', node=self.name, pid=self.pid)

    def fail(self):
        if self.status == 'started':
            Popen('kill -15 %s' % self.subprocess.pid, shell=True)
        if self.status == 'starting' or 'not started':
            print info('failing not started',
                    status=self.status, pid=self.pid)

    def fetch_pid(self):
        pidof_process = Popen('ssh %s "pidof svm-train"' %
```

```
                self.host, shell=True, stdout = PIPE)
        pids_str = pidof_process.stdout.readline()
        if pids_str == '': return
        return max(map(int, pids_str.split()))

    def do_solve(self, *args, **kwargs):
        self.task, cores = args[0], args[1]

        # check cached solution
        mse = None
        try:
            with RemoteSvrSolver.lock:
                self.subprocess = Popen(self.cmdline(self.task, cores),
                        shell=True, stdout = PIPE)
                sleep(0.1) # wait for svm-train to start on self.host
                self.pid = self.fetch_pid() # fetch pid for stop() method
                if not self.pid:
                    self.status = 'bad pid'
                    raise Exception(emph(
                        'fail to fetch pid for worker %s' % self.name))
                self.status = 'started'
                print info('started solving',
                        worker=self.name, pid=self.pid,
                        task=emph(self.task))
            for line in self.subprocess.stdout.readlines():
                if find(line, "Cross") != -1:
                    mse = float(split(line)[-1])
                    break
            if not mse:
                raise Exception(emph(
                    'fail to retrieve mse value from svm-train output'))
            result = Result(self.task, self, mse)
            self.notify("success", result)
        except Exception, message:
            if self.status == 'bad pid':
                self.status = 'killed'
                print info('fail', message=message)
            if not self.status in['killing', 'killed']:
                result = Result(self.task, self,
                        mse, error_message = message)
                self.status = 'failed'
                self.notify('fail', result)

    def cmdline(self, task, cores):
        ssh_command = "%s -s 3 -c %s -g %s -p %s -v %s -m %s %s" % (
            svmtrain_exe,
            task.C, task.gamma, task.eps,
            task.folds,
            self.memory,
```

```
            task.dataset_path)
        return 'ssh %s "cd %s; export OMP_NUM_THREADS=%s; %s"' %
                (self.host, self.cwd, cores, ssh_command)

class Node(EventDispatcher):
    lock = Lock()

    def __init__(self, host, memory, cores, cache):
        EventDispatcher.__init__(self)
        self.memory = memory
        self.cache = cache
        self.host = host
        self.cores = cores
        self.used_cores = 0
        self.workers = []

    def worker_name(self, core=None):
        if core:
            return '%s:core:%s' % (self.host, core)
        else:
            return '%s:core:*' % self.host

    def subscribe_to_workers(self):
        for w in self.workers:
            self.subscribe_to_worker(w)

    def subscribe_to_worker(self, worker):
        worker.subscribe('success', self.on_success)
        worker.subscribe('fail', self.on_fail)

    def solve_all_cores(self, task):
        if self.used_cores > 0:
            raise Exception('Number of used cores must be 0')
        self.used_cores = self.cores
        worker = RemoteSvrSolver(
                self.worker_name(), self.host, self.memory, self.cache)
        self.workers.append(worker)
        self.subscribe_to_workers()
        worker.solve(task, self.used_cores)

    def solve_one_core(self, task):
        if self.used_cores == self.cores:
            raise Exception(
            'Number of used cores must be less than number of cores')
        self.used_cores += 1
        worker = RemoteSvrSolver(self.worker_name(self.used_cores),
                self.host, int(self.memory / self.cores), self.cache)
        self.workers.append(worker)
        self.subscribe_to_worker(worker)
```

```
            worker.solve(task)

        def fail_all(self):
            for w in self.workers:
                w.fail()

        def update(self, worker):
            # remove worker:
            for w in self.workers:
                if w.name == worker.name:
                    self.workers.remove(w)
                    break

            if len(self.workers) == 0:
                self.used_cores = 0
            else:
                self.used_cores -= 1

        def on_success(self, sender, result):
            # with Node.lock:
            Node.lock.acquire()
            worker = sender
            self.update(worker)
            result.worker = worker
            self.notify('success', result)
            Node.lock.release()

        def on_fail(self, sender, result):
            with Node.lock:
                worker = sender
                self.update(worker)
                print info('fail', node=self.host,
                           worker=worker.name, message=result.error_message)
                worker.stop()
                self.notify('fail', result)

    class MseMinimizer(EventDispatcher):
        def __init__(self, name, nodes, cache, dataset_path):
            EventDispatcher.__init__(self)
            self.dataset_path = dataset_path
            self.nodes = nodes
            self.name = name
            self.cache = cache
            self.task_queue = [] # Queue.Queue(0)
            self.lock = Lock()
            self.mse_values = []
            self.subset_path = ''

        def start(self, tasks, size, cache_file = None):
```

```
        self.n_tasks = len(tasks)

        self.subset_path = self.create_subset(self.dataset_path, size)
        if not cache_file:
            self.cache_file = open(self.subset_path + '.cache', 'w')
        else:
            self.cache_file = cache_file

        print info('started miminizing mse', name=self.name,
                dataset=self.subset_path, tasks=len(tasks))

        self.tasks = tasks
        for task in self.tasks:
            task.dataset_path = self.subset_path
            self.task_queue.append(task)

        # subscribe to nodes' events:
        for node in self.nodes:
            node.subscribe("success", self.on_success)
            node.subscribe("fail", self.on_fail)

        self.distr()

    def distr(self):
        abs_free_nodes = len([n for n in self.nodes if n.used_cores == 0])

        if abs_free_nodes >= len(self.task_queue):
            for node in self.nodes:
                if len(self.task_queue) == 0: break
                node.solve_all_cores(self.task_queue.pop(0))
        else:
            # distribute for all cores
            def cmp_nodes(node1, node2):
                return node1.used_cores - node2.used_cores

            node_count = 0
            pushed = False
            self.nodes.sort(cmp_nodes) # optimize
            while len(self.task_queue) > 0:
                # print 'hi there'
                node = self.nodes[node_count]
                if node.used_cores < node.cores:
                    node.solve_one_core(self.task_queue.pop(0))
                    pushed = True
                node_count += 1
                if node_count == len(self.nodes):
                    if not pushed: break # all busy
                    node_count = 0
                    pushed = False
```

```
    def busy_workers(self):
        return sum([node.used_cores for node in self.nodes])

    def find_argmin(self, min_mse):
        mse = min([self.cache[task] for task in self.cache])
        return [task for task in self.cache if self.cache[task] == mse]

    def on_success(self, sender, result):
        global keep
        with self.lock:
            node = sender
            worker = result.worker
            print info('finished solving', name=self.name,
                    node=node.host, worker=worker.name,
                    task=emph(result.task), mse=result.mse)
            self.cache[result.task] = result.mse # cache data
            self.cache_file.write(str(result.task) +
                    '=>' + str(result.mse) + '\n')
            self.cache_file.flush()
            self.mse_values.append(result.mse)

            if len(self.mse_values) == self.n_tasks:
                min_mse = min(self.mse_values)
                print info('finished minimization of mse',
                        name=self.name, mse=min_mse,
                        argmins=[emph(t) for t
                            in self.find_argmin(min_mse)])
                self.notify('finish',
                        {'argmins': self.find_argmin(min_mse),
                            'mse': min_mse})
            elif keep:
                self.distr()

    def on_fail(self, sender, result):
        with self.lock:
            node = sender
            if keep:
                self.task_queue.append(result.task)

    def create_subset(self, dataset_path, size = None):
        if not size:
            return dataset_path # solve on original data
        else:
            return dataset_path + "." + str(size)

def task_cube(C_range, gamma_range, eps_range, folds):
    tasks = []
    for C in C_range:
```

```
        for gamma in gamma_range:
            for eps in eps_range:
                tasks.append(Task(2 ** C,
                    2 ** gamma, 2 ** eps, folds, None))
    return tasks

def task_surrounding(log2c, log2gamma, log2eps, folds):
    # note: task data are in logarithmic scale
    # here we must exponentiate it
    # create surrounding of (C, gamma):
    # ((C - 1, gamma - 1) (C - 1, gamma) (C - 1, gamma + 1))
    # ((C,     gamma - 1) (C,     gamma) (C,     gamma + 1))
    # ((C + 1, gamma - 1) (C + 1, gamma) (C + 1, gamma + 1))
    return [Task(
        2 ** (log2c + c_i),
        2 ** (log2gamma + g_i),
        2 ** (log2eps + e_i), folds, None)
            for c_i in [-1, 0, 1]
            for g_i in [-1, 0, 1]
            for e_i in [-1, 0, 1]]

def parse_range(str):
    begin, end, step = map(int, str.split(','))
    return range(begin, end + step, step)

def nodes():
    return [Node("node%s" % i, 1000, 4, cache) for i in range(1, 11)]

def create_subsets(sizes):
    '''
        Creates subsequent subsets for the given datasets
    '''
    sizes.sort(); sizes.reverse()
    global dataset_path
    script = r"/home/zeally/imat2009/libsvm-2.89/tools/subset.py" # bad
    print info('cleaning subsets...')
    # os.popen('rm %s.*' % dataset_path)
    for i, size in enumerate(sizes):
        if not size:
            break
        superset_path = ''
        if size == sizes[0]:
            superset_path = dataset_path
        else:
            superset_path = dataset_path + '.' + str(sizes[i - 1])
        subset_path = dataset_path + '.' + str(sizes[i])
        print info('building subset', subset=subset_path)
        if not os.path.exists(subset_path):
            cmdline = "%s -s 1 %s %s %s" %
```

```
                (script, superset_path, size, subset_path)
            os.popen(cmdline, 'r')
            print info('created')
        else:
            print info('use existed')
    sizes.reverse()

def sizes(dataset_path, maxT1 = 0, kappa=2.0/3.0):
    maxT1 = float(maxT1)
    kappa = float(kappa)
    # Get Tn from dataset file:
    cmdout = os.popen('wc -l %s' % dataset_path, 'r')
    result_line = cmdout.readlines()[0]
    Tn = float(result_line.split()[0])
    # Compute maxT1 if default value (0) is provided:
    if maxT1 == 0:
        maxT1 = Tn / 20.0
    # Compute n = n(maxT1, Tn, kappa):
    n = ceil(log10(maxT1 / Tn) / log10(kappa) + 1)
    return map(lambda k: int(Tn * kappa ** (k - 1)), range(2, n + 1))

def get_last_subset_size():
    def get_size(line):
        size_re = re.search('(\d+)\.cache', line)
        return int(size_re.groups(1)[0])
    global dataset_path
    cmdout = os.popen('ls %s.*.cache' % dataset_path, 'r')
    result_lines = cmdout.readlines()
    if len(result_lines) > 0:
        return max([get_size(line) for line in result_lines])
    else:
        return

##################################################################
##################################################################

from optparse import OptionParser
parser = OptionParser(usage="usage: %prog [options] dataset")
parser.add_option(
        "-v", "--folds",
        dest="folds", type="int",
        default="5",
        help="number of folds. (default: %default)")
parser.add_option(
        "--log2c", dest="log2c",
        default="-1,6,1",
        help="range of C in log scale (default: %default)")
parser.add_option(
        "--log2gamma", dest="log2gamma",
```

```
        default="0,-8,-1",
        help="range of gamma in log scale (default: %default)")
parser.add_option(
        "--log2eps", dest="log2eps",
        default="-8,-1,1",
parser.add_option(
        "-m", type="int", dest="cache_size",
        default="1000",
parser.add_option(
        "--out", dest="output_path",
        help="path of output file")
parser.add_option(
        "--png",
        dest="png_path",
        help="path of png file")

(opts, args) = parser.parse_args()
if len(args) != 1:
    parser.error("incorrect number of arguments")

# from opts/args:
dataset_path = args[0]
folds = opts.folds
C_range = parse_range(opts.log2c)
gamma_range = parse_range(opts.log2gamma)
eps_range = parse_range(opts.log2eps)

# initial state:
cache = {}
iter = 0
argmin_f = None # open(dataset_path + '.argmins', 'w')
minimizer = MseMinimizer('minX', nodes(), cache, dataset_path)
argmin = None
argmin_track = []

sizes = sizes(dataset_path)
print info('sizes', sizes=sizes)
create_subsets(sizes)
last_size = get_last_subset_size()

def on_finish(sender, result):
    global iter, cache, folds, sizes, argmin_f, argmin, argmin_track

    print info('iter %s finished' % iter)
    print '-'*80
    cache = {}

    # print info('try to set argmin...')
    # print info('argmins: %s' % result['argmins'])
```

```
    if iter == 0:
        argmin = result['argmins'][0]
    else:
        argmin = min(result['argmins'], key=lambda t: Task.dist(t, argmin))
    # print info('argmin set to: %s' % argmin)

    iter += 1
    argmin_f.write('#%s\nmse=%s\nnearest=%s\nall=%s\n\n' %
            (iter, result['mse'], str(argmin),
                [str(task) for task in result['argmins']]))
    argmin_f.flush()

    argmin_track.append(argmin)

    if iter > len(sizes):
        path_quality = 0.0
        for i in range(len(argmin_track) - 1):
            path_quality += Task.dist(argmin_track[i], argmin_track[i + 1])
        path_quality /= (float(len(argmin_track) - 1))
        path_quality /= sqrt(3.0)
        print info('path error normilized', quality=path_quality)
        argmin_f.write('path error: %s' % path_quality)
        argmin_f.close()
        return

    tasks = task_surrounding(log(argmin.C, 2),
            log(argmin.gamma, 2), log(argmin.eps, 2), folds)

    if iter == len(sizes):
        mins = MseMinimizer('min' + str(iter), nodes(), cache, dataset_path)
        mins.subscribe('finish', on_finish)
        mins.start(tasks, None)
    else:
        mins = MseMinimizer('min' + str(iter), nodes(), cache, dataset_path)
        mins.subscribe('finish', on_finish)
        mins.start(tasks, sizes[iter])

argmin_path = dataset_path + '.argmins'

def argmin_f_non_empty():
    global argmin_path
    if os.path.exists(argmin_path):
        cmdout = os.popen('wc -l %s' % argmin_path)
        line_count = int(cmdout.readlines()[0].split()[0])
        if line_count > 0:
            return True
    return False

if argmin_f_non_empty() and sizes.count(last_size) != 0:
```

```
print info('proceed computation')
argmin_f = open(argmin_path, 'a+')
# restore argmin_path and argmin
while True:
    first = argmin_f.readline()
    if first == '':
        break
    mse_line = argmin_f.readline()
    argmin_line = argmin_f.readline()
    argmin_f.readline()
    argmin_f.readline()

    mse = float(mse_line.split('=')[1])
    c_m = re.search('C=([\d.]+)', argmin_line)
    C = float(c_m.group(1))
    gamma_m = re.search('gamma=([\d.]+)', argmin_line)
    gamma = float(gamma_m.group(1))
    eps_m = re.search('eps=([\d.]+)', argmin_line)
    eps = float(eps_m.group(1))
    dataset_m = re.search('dataset=(.*)\n', argmin_line)
    dataset = dataset_m.group(1)
    argmin = Task(C, gamma, eps, folds, dataset)
    argmin_track.append(argmin)
# restore iter:
iter = sizes.index(last_size)
# restore cache:
cache_f = open('%s.%s.cache' % (dataset_path, last_size), 'a+')
finished_tasks = []
for cache_line in cache_f.readlines():
    mse = float(cache_line.split('=>')[1])
    c_m = re.search('C=([\d.]+)', cache_line)
    C = float(c_m.group(1))
    gamma_m = re.search('gamma=([\d.]+)', cache_line)
    gamma = float(gamma_m.group(1))
    eps_m = re.search('eps=([\d.]+)', cache_line)
    eps = float(eps_m.group(1))
    task = Task(C, gamma, eps, folds, None)
    finished_tasks.append(task)
    cache[task] = mse
# get not finished tasks from last_tasks and task_surrounding
tasks_all = task_surrounding(log(argmin.C, 2),
        log(argmin.gamma, 2), log(argmin.eps, 2), folds)
if len(argmin_track) > iter:
    iter += 1
    not_finished_tasks = tasks_all
    cache_f = {}
    if iter > len(sizes) - 1:
        sizes.append(None)
else:
```

```
        not_finished_tasks = list(set(tasks_all) - set(finished_tasks))
    # proceed:
    minimizer = MseMinimizer('minRestore', nodes(), cache, dataset_path)
    minimizer.subscribe('finish', on_finish)
    minimizer.start(not_finished_tasks, sizes[iter], cache_f)
else:
    init_tasks = task_cube(C_range, gamma_range, eps_range, folds)
    argmin_f = open(argmin_path, 'w')
    minimizer = MseMinimizer('minInit', nodes(), cache, dataset_path)
    minimizer.subscribe('finish', on_finish)
    minimizer.start(init_tasks, sizes[iter])


# trick:
# avoid catching keyboard interrupting by threads
def interrupt_handler(signum, frame):
    global keep
    keep = False
    if frame.f_code.co_name == 'wait':
        # fail threads, remaining alive:
        for node in minimizer.nodes:
            node.fail_all()

signal(SIGINT, interrupt_handler)
```

Список литературы

[1] *Еремеев, . .* Геометрия: учебник для вузов / . . Еремеев, . . Кузютин, . . Зенкевич. — Лань, 2003.

[2] *Вапник, . .* Теория распознавания образов / . . Вапник, . . Червоненкис. — М.: Наука, 1974.

[3] *Воронцов, . .* Лекции по линейным алгоритмам классификации / . . Воронцов. — 2009.

[4] *Воронцов, . .* Вычислительные методы обучения по прецедентам. Введение / . . Воронцов. — 2009.

[5] Интернет-математика 2009. `http://company.yandex.ru/grant/2009/datasets`.

[6] *Burges, C. J. C.* A tutorial on support vector machines for pattern recognition / C. J. C. Burges // *Data Mining and Knowledge Discovery.* — 1998. — Vol. 2, no. 2. — Pp. 121–167. `citeseer.ist.psu.edu/burges98tutorial.html`.

[7] *Chang, C.-C.* — LIBSVM: a library for support vector machines, 2001. — Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[8] Computer activity dataset. http://www.cs.toronto.edu/~delve/data/comp-activ/desc.html.

[9] *Hsu, C. W.* A practical guide to support vector classification: Tech. rep. / C. W. Hsu, C. C. Chang, C. J. Lin. — Taipei: 2003. http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf.

[10] *Joachims, T.* Making large-scale support vector machine learning practical / T. Joachims // *Advances in kernel methods: support vector learning.* — 1999. — Pp. 169–184.

[11] *Joachims, T.* Learning to Classify Text Using Support Vector Machines: Methods, Theory and Algorithms / T. Joachims. — Springer, 2002.

[12] *Keerthi, S. S.* Asymptotic behaviors of support vector machines with gaussian kernel / S. S. Keerthi, C.-J. Lin // *Neural Comput.* — 2003. — July. — Vol. 15, no. 7. — Pp. 1667–1689. http://dx.doi.org/10.1162/089976603321891855.

[13] *Kuhn, H. W.* Nonlinear programming / H. W. Kuhn, A. W. Tucker // *Berkeley: University of California Press.* — 1951.

[14] *Mercer, J.* Functions of positive and negative type and their connection with the theory of integral equations / J. Mercer // *Philos. Trans. Roy. Soc. London.* — 1909. — Vol. A, no. 209. — Pp. 415–446.

[15] New support vector algorithms / B. Scholkopf, B. S. Gmd, A. J. Smola et al. — 1998.

[16] *Smola, A.* A tutorial on support vector regression: Tech. Rep. NeuroCOLT2 NC2-TR-1998-030 / A. Smola, B. Schoelkopf: 1998. citeseer.ist.psu.edu/smola98tutorial.html.

[17] Svm software. http://www.support-vector-machines.org/SVM_soft.html.

[18] *Vapnik, V.* Support-vector networks / V. Vapnik, C. Cortes // *Machine Learning.* — 1995. — Vol. 20, no. 3. — Pp. 273–297. citeseer.ist.psu.edu/cortes95supportvector.html.

[19] *Vapnik, V.* Pattern recognition using generalized portrait method / V. Vapnik, A. Lerner // *Automation and Remote Control.* — 1963. — Vol. 24.