Машинное обучение

А.М.Савчук
Факультет космических исследований МГУ
осень 2025

1 Введение

Что такое машинное обучение, чем оно отличается от классического анализа данных и глубокого обучения? Машинное обучение — это наука, которая позволяет некоторой компьютерной программе обучаться на загружаемых в нее данных, не будучи запрограммированной явно. Более точно. Предположим, что есть некоторый однородный набор данных D, целевая задача T и оценка качества P. Мы говорим, что наша программа есть ML–программа, если с поступлением новых данных она улучшает свое качество.

Исторически первыми программами (тогда еще программами на бумаге) были экспертные системы. Эти системы были от начала до конца составлены и «обучены» человеком. Например, экспертной системой была система классификации животных по фенотипу. Обитает в море или на суше? Умеет летать? Кормит своих детенышей молоком? И так далее, по заранее составленному исследователем списку вопросов система выдавала ответ (тоже составленный заранее). С точки зрения современного ML здесь человек составлял и сам метод работы = правила (в данном случае, дерево решений) и список признаков (летает или нет, в море или на суше). Такая программа программируется явно и, следовательно, к машинному обучению не относится.

Затем появились методы статистического анализа данных. Предположим, мы много раз подбросили монетку. Обработав результаты (например, взяв выборочное среднее, выборочную дисперсию, доказав соответствующую теорему и посчитав соответствующий доверительный интервал), мы теперь можем оценивать вероятность того или иного события, связанного с дальнейшим подбрасыванием нашей монеты. Это уже вполне современная задача, но она не относится к ML — у программы нет обучения, программа составлена экспертом заранее (на основе его теоретических знаний).

Затем появились методы регрессии и классификации. Здесь эксперт собирает данные, подготавливает их, потом придумывает модель анализа этих данных, подгоняет параметры этой модели подходящим образом и на выходе получает функцию f (собственно, модель). Это почти задача машинного обучения! Штука в том, что в те годы (примерно 1960–е) задача после построения модели считалась решенной. Оставалось только использовать модель для новых данных. Задачи такого типа сейчас принято называть задачами классического анализа данных. Целью работы эксперта здесь является сама модель! То есть, эксперт на основании своей модели делает разные качественные и количественные выводы и на этом задача считается решенной. Именно этим мы с вами занимались в прошлом семестре.

А теперь давайте поменяем нашу цель. Задача программы машинного обучения — качественно работать на новых данных и постоянно совершенствовать себя — менять параметры модели, но уже без участия эксперта. При этом, придумывание модели, первичную подгонку ее параметров и гиперпараметров (отличия обсудим чуть позже)

берет на себя эксперт. Науку, которая этим занимается, называют сейчас классическим машинным обучением.

Обратите внимание, что мне пришлось добавить слово «классический» дважды — не просто анализ данных, а классический анализ данных, не просто машинное обучение, а классическое машинное обучение. Дело в том, что термин «анализ данных» включает в себя машинное обучение, а машинное обучение включает в себя глубокое обучение. Для того, чтобы моя классификация наук была дизъюнктной, мне и пришлось говорить о классическом ... Глубокое обучение — совсем молодая наука (практические успехи в этой области начались с 2012 года). Здесь эксперт не составляет законы за счет своего опыта, не подбирает параметры вручную или на основе статистических тестов. Эксперт лишь составляет архитектуру нейронной сети, подгружает в нее свои данные и контролирует качество работы, подкручивая параметры архитектуры. Здесь модели нет вообще (точнее, есть, конечно, но такая сложная, что выписывать ее в аналитическом виде нет никакого смысла)! Нейронная сеть — это черный ящик, который работает (хорошо или плохо), но почему он работает, мы сказать уже не можем. Нейронные сети показали себя очень хорошо за прошедшие 12 лет (они существовали, конечно, и до того, но никак не получалось настроить их, чтобы они начали работать сопоставимо или лучше в сравнении с классическим ML). К сожалению, теория нейронных сетей катастрофически не поспевает за практикой, так что сейчас эта область знаний является типичной технической. Условно говоря, специалист по нейронным сетям — это инженер, который ходит вокруг своей сети, подкручивает в ней какие–то винтики, но почему она работает, сказать не может. Нейросетям будут посвящены учебные курсы в следующих семестрах (в восьмом и в девятом).

Давайте договоримся об обозначениях. Во–первых, все наши данные — это числа. Как и раньше, признаки (feature) типа число — это число; а признаки типа да/нет кодируются бинарной переменной 1/0. Признаки с большим числом вариантов (пусть k) кодируются $k-1$–ой бинарной переменной. Здесь значение j–ой переменной равно 1, если признак имеет значение j–го варианта. При этом, все остальные переменные принимают значение 0. В частности, если осуществляется вариант номер k, то обнуляются все наши переменные. Такое кодирование называют one–hot–кодированием. К сожалению, на практике приходится иметь дело и с более сложно устроенными признаками. Что делать, если вам надо анализировать качество ресторанов, а признаком является текстовый отзыв посетителя? В таком случае данные надо предобработать, чтобы выделить из них числовые признаки. Такой процесс называется feature engineering или feature extraction — конструирование признаков или извлечение признаков.

Далее, договоримся, что каждый образец данных (sample) содержит значения всех признаков. На практике, как вы знаете, это далеко не всегда так. Но мы будем считать, что предварительная обработка данных уже проведена. На практике можно: выкинуть те образцы, в которых присутствуют не все признаки; выкинуть те признаки, которые известны не для всех образцов; заполнить пропуски средним значением признака по выборке; заполнить пропуски модой признака по выборке; заполнить пропуски случайными значениями в разумном диапазоне; построить предварительную модель, которая предскажет пропуски тем или иным образом по другим признакам.

Договоримся записывать каждый образец в виде строки данных $\vec{x} = (x_1, \ldots, x_m)$.

Договоримся обозначать x_j значение j–го признака для нашего образца. Договоримся, что общее число признаков обозначается m, а общее число образцов — через n. Договоримся обозначать наши образцы $\vec{x}_i$, где $i \in [1, n]$, а координаты их через x_{ij}. Двумерный массив всех наших данных будем обозначать X (как вы понимаете — это матрица размера $n \times m$, в которой каждая строка — образец). Иными словами, мы будем работать с табличными данными. Договоримся обозначать признаки через X_j (получается, что это заголовки столбцов). Признаки, вошедшие в модель, будем называть предикторами (predictors).

В классическом анализе данных число признаков обычно невелико (впрочем, как и число образцов). Машинное обучение обычно работает с большими объемами данных. Например, классический тестовый набор MNIST (распознавание цифр) имеет параметры $n = 70000$, $m = 784$, и считается небольшим. В этом состоит еще одно отличие ML от классического анализа данных — задачи машинного обучения обычно содержат большое число образцов.

Числовые признаки

Пусть «сырой признак» принимает вещественные значения. Самое логичное — оставить его вещественным, но нормализовать. Есть два классических способа нормализации. Первый способ: вычислить среднее и среднее квадратическое признака, вычесть среднее и разделить на среднее квадратическое. Получим признак с нулевым средним и единичным средним разбросом. Второй способ: найти минимальное значение признака и его размах, вычесть минимум и разделить на размах. Получим признак в шкале $[0, 1]$. Перед применением второго способа надо обязательно проверить признак на наличие выбросов, ликвидировать их (удалением образцов или принудительным изменением, скажем, на среднее значение), и только потом проводить нормализацию.

Для ряда задач нас может не интересовать конкретное значение признака. Важно только, например, попадает ли он в референтные значения. Тогда признак можно заменить на бинарную переменную (это называется бинаризацией признака). Более сложный случай: нас вновь не интересует конкретное значение признака, но интересует, мал ли он, средний, большой и т.д. Тогда имеет смысл выделить интервалы и заменить значения признака, попавшие в интервал на середину интервала. Такой метод называется бинированием признака. Делить на интервалы можно по разному. Самое простое — разделить диапазон изменения признака на равные интервалы, а число интервалов подобрать из смысла задачи или из характера данных. Можно и сами интервалы подбирать из смысла задачи. Например, если речь идет о возрасте респондента, причем минимум по выборке равен 7, а максимум 77, то логично взять интервалы $[7, 17)$ (школьник), $[17, 24)$ (студент), $[24, 35)$ (молодой), $[35, 50)$ (средних лет, активный), $[50, 65)$ (еще работает), $[65, 77]$ (пенсионер). Третий способ (тоже часто используется) — разбить выборку на квартили.

Очень полезно проводить разведочный анализ признаков. При этом надо помнить, что большинство методов «любят» нормальные распределения, так что полезно корректировать признаки в этом направлении. Если какие–то признаки по смыслу задачи очень важны, или у вас много времени, то стоит посмотреть график их распределения. Признаки, которые имеют одностороннее распределение (особенно, если оно похоже на

экспоненциальное), логично прологарифмировать. Признаки с двумя пиками, разложить в сумму двух нормальных распределений (о том, как это сделать мы поговорим ниже) и так далее. Если данных у нас много и признаков тоже много, а значит, нет возможности построить для каждого признака гистограмму, обдумать распределение, протестировать его — тогда имеет смысл запустить трансформацию признаков с помощью преобразования Бокса–Кокса (см. курс анализа данных, кто забыл).

В задачах часто встречаются признаки времени (часы, дни недели, месяцы и т.д.). Конечно, всегда можно перевести эти признаки в один единственный числовой признак «время в секундах». Стандарт Unix Timestamp предполагает начало отсчета времени от 00.00 1 января 1970 года. Однако использование одного этого признака приведет к потере важной информации. Дело в том, что многие события в нашей жизни имеют цикличность. Например, число автомобилей на дорогах Москвы в зависимости от времени, очевидно имеет периодическую составляющую с периодом 24 часа (ночью автомобилей меньше, днем — больше), с периодом 7 дней (в выходные меньше, в будние дни — больше) и с периодом 1 год (летом меньше, в другие сезоны — больше). Добавить (кроме сквозного времени) признак t, равный времени внутри суток в диапазоне $[0, 24)$ — плохая идея. Дело в том, что значение такого признака 0 близко к 1 (и признак это отражает), но оно близко и к 23 (а признак это не отражает). Придется добавлять два признака. Стандартный способ — признаки $\sin(\pi t/12)$ $\cos(\pi t/12)$. При этом надо понимать, что такой метод может тоже создать проблемы. Например, деревья принятия решений делают сплиты по одному признаку, и теперь неясно, какой из двух использовать. Если речь идет именно про такие данные и именно про деревья, то лучше добавить признак $|t-3|$ (число 3 выбрано потому, что в три часа ночи трафик в Москве минимален).

Категориальные признаки

Категориальные признаки следует кодировать с помощью one–hot encoding. Например, если у вас собраны данные по автомобилям, и есть признак «Тип коробки передач», принимающий значения «Механика», «Автомат» и «Вариатор», то неверной идеей будет создать один признак X, приписав ему значения, например, 1, 2 и 3. Это приведет к зависимостям вида Механика + Автомат = Вариатор. Правильное решение — создать две бинарные переменные X_1 и X_2 — индикаторы, например, событий «Автомат» и «Вариатор». Тогда наши категории будут иметь кодировки «Механика»=(0,0), «Автомат»=(1,0), «Вариатор»=(0,1).

Этот метод, при всех его преимуществах, имеет один недостаток: вместо одного признака с p значениями мы получаем $p-1$ бинарный признак. Если p достаточно велико, а данные содержат много категориальных признаков, то после кодирования мы рискуем получить огромное число признаков. Поэтому, если между категориями все-таки прослеживается связь типа меньше/больше, лучше/хуже, то можно использовать ранговую переменную. Например, если вы анализируете данные с таргетом стоимость аренды квартиры в Москве, и у вас есть категория «Округ», то разумно будет взять среднюю цену по каждому округу и закодировать район этой ценой. Метод называется target encoding. У него тоже есть недостаток. Модели с таким признаком могут переобучаться — игнорировать все остальные признаки и строить прогноз только по этому одному

признаку. В такой ситуации помогает зашумление признака — добавление к нему случайного нормального шума.

Другая альтернатива one–hot кодированию — кодирование признака по числу образцов в данных (count encoding). Фактически, это скрытое кодирование рангами в ситуации, когда у нас нет времени разбираться с каждым признаком подробно. Представим, что вы обрабатываете данные пациентов диетолога и таргет — бинарная переменная, отражающая наличие у пациента диабета. У вас в данных присутствует ответ на вопрос «ваш любимый продукт». Понятно, что ответы будут самые разные, так что one–hot кодирование не разумно. Ясно, однако, что излишняя любовь к сладкому (например) как раз и провоцирует диабет. Так что среди пациентов–диабетиков будут часто встречаться определенные виды любимых продуктов. А люди, у которых любимые продукты не провоцируют диабет, просто не окажутся в числе пациентов и в данных не встретятся. В такой ситуации count encoding разумно.

Типы машинного обучения

Машинное обучение очень разнообразно. Его методы принято делить по разным признакам. Самое, пожалуй, важное деление — обучение с учителем или без учителя (на самом деле, бывает еще обучение с подкреплением, но это отдельный разговор, и в наш курс этот вид обучения не входит).

Предположим, что для каждого образца нам дополнительно известно значение y_i из некоторого множества значений. Тогда признак Y мы называем целевым (target) и говорим, что у нас задача обучения с учителем (т.е. y_i — это и есть оценка учителя для данного ему образца $\vec{x}_i$). Если же такого признака Y нет, но его программа должна создать сама, то у нас обучение без учителя. В реальности часто встречаются смешанные данные — значение признака y_i присутствует лишь для части данных (обычно небольшой по отношению к общему объему n). В такой ситуации говорят о частичном обучении; ту часть данных, для которых y_i известно, называют размеченными данными, а оставшуюся часть данных — неразмеченными.

Приведем основные примеры обучения с учителем. Прежде всего, это регрессия. Здесь наша задача составить отображение $f : \vec{x} \mapsto y$, $f : \mathbb{R}^m \to \mathbb{R}$. Второй не менее частый случай — это классификация. Задача бинарной классификации: составить отображение $f : \mathbb{R}^m \to \{0, 1\}$, т.е. отнести каждый вектор $\vec{x}$ к одному из двух классов C_0 или C_1. Часто встречается многоклассовая ситуация, в которой $f : \mathbb{R}^m \to \{1, 2, \ldots, k\}$ — разбиение множества векторов $\vec{x}$ на k классов. Естественно, при больших k такая задача приближается к задаче регрессии. И регрессия, и классификация могут проводится различными методами, о которых собственно и пойдет речь в курсе: параметрическая регрессия, логистическая регрессия, метод ближайших соседей, метод опорных векторов (для регрессии — SVR или для классификации — SVC), решающие деревья и т.д. Встречаются и более сложные множества образов у отображения f. Например, если мы учим программу искать на фотографиях людей из некоторого набора (например, узнавать на фотографиях членов семьи), то образом f являются все подмножества множества членов семьи. Действительно, на фотографии может оказаться любой состав из этих людей.

Наиболее частые задачи обучения без учителя — это кластеризация и снижение размерности. Кластеризация очень похожа на классификацию — каждый вектор $\vec{x}$ надо отнести к одному из кластеров $C_1, \ldots, C_k$ (при этом, число k может быть задано заранее и фиксировано, а может тоже быть целью кластеризации). Разница с классификацией очень проста — в кластеризации используются неразмеченные данные. Понижение размерности — это задача уменьшения числа m признаков. Например, мы можем заметить, что один из признаков X_j одинаков (или почти одинаков) для всех образцов — это очевидный повод отбросить данный признак. Или (другой пример) мы можем заметить, что между признаком X_j и признаком X_l наблюдается тесная линейная зависимость — это повод отбросить один из признаков или заменить эту пару признаков, например, на их полусумму. Кстати, задача нахождения таких зависимостей интересна и сама по себе. Такие задачи называют задачами нахождения ассоциативных правил. Вы сами встречали такие правила: перед оформлением покупки в интернет магазине может появиться строка «с этим товаром обычно покупают...».

Другое разбиение методов машинного обучения связано с их способностью учиться «на лету». При пакетном обучении система может обучаться только на полном пакете данных. Для улучшения работы такой системы требуется обучать ее «с нуля» на новом (обычно, старом дополненном) пакете данных. Это не является приговором для системы — мы всегда можем распараллелить процессы, одна копия предобученной системы работает, а другая в это время обучается, а после обучения заменяет собой рабочую копию системы. Но конечно, лучше, если система умеет учиться во время работы. Такие системы называют динамическими. Бывают и смешанные варианты — систему надо отправлять на дообучение на новом пакете данных (при этом система обучается не с нуля, а именно дообучается). Такое обучение называют обучением при помощи мини–пакетов. Типичный пример — система днем работает, а ночью дообучается на пакете данных, поступивших за прошедший день.

Общая структура проекта ML

ML–проект — это триада: данные, функция потерь (или качества), модель и алгоритм ее обучения.

Никакой системы машинного обучения не получится, пока вы не собрали достаточно много данных для обучения своей системы. Нет данных (или мало данных) — нет проекта. Еще раз повторим, что данные необходимо предобработать, что иногда совсем не тривиальная задача. Затем массив данных необходимо разбить на три части: обучающая выборка (training set или training data) — на этих данных вы будете учить свою систему; проверочный набор (validation set) — на этих данных вы будете оценивать, какой из методов обучение лучше, подбирать для него гиперпараметры и т.п.; испытательный набор (test set) — эти данные надо отложить с самого начала, никогда на них не смотреть и использовать только для финальной проверки системы перед тем, как запустить ее в работу.

Для рационального использования данных очень разумно применять перекрестную проверку (cross–validation). Здесь вы с самого начала откладываете в сторону испытательный набор. Оставшиеся данные разбиваете на несколько частей. На каждой

части данных обучаете некоторую систему, а проверяете ее на всех остальных частях. Таким образом, вы находите наилучший алгоритм (или наилучшие гиперпараметры алгоритма). Теперь, зафиксировав этот алгоритм и эти гиперпараметры, вы обучаете систему на всех данных (кроме испытательного набора), а затем испытываете на нем полученную систему.

Никакой системы машинного обучения не получится, пока вы не сформулируете целевую функцию. Например, вы хотите научить машину читать. Но что вы имеете в виду? Уметь выделять ключевые слова из текста? Уметь находить тексты с похожим смыслом? Уметь разбирать предложения на подлежащее, сказуемое и т.д.? Проект возможен только тогда, когда выбрана целевая функция, которая оценивает качество модели. Исторически сложилось, что вместо функции качества обычно используется функция потерь (loss function). В процессе обучения система будет стремиться минимизировать значение функции потерь. Потом, когда система будет готова и начнет работать, эта же функция потерь может стать оценкой качества системы. Но могут использоваться и другие функции оценки качества, это вовсе не запрещено.

И третье, без чего невозможен ML–проект — это модель, которой мы будем описывать данные и алгоритм обучения этой модели. Большинство моделей, которые используют сейчас, параметрические (например, регрессия или разделяющая поверхность в методе опорных векторов или решающее дерево, в каждом узле которого выбирается номер признака и его критическое значение). В таком случае целью обучения является подбор параметров модели, которые минимизируют функцию ошибок. Непараметрические модели обычно появляются в обучении без учителя (например, метод ближайших соседей). Обычно они используются на первом шаге построения ML–проекта — их результаты подаются на вход следующей модели, которая уже обучается. Алгоритм обучения — тоже очень важный аспект. Чаще всего сейчас, это те или иные модификации градиентного спуска

$$f(\vec{\theta}) \longrightarrow min,\ \vec{\theta} = (\theta_1, \ldots, \theta_m) \qquad \Longrightarrow \quad \vec{\theta}^{k+1} = \vec{\theta}^{k} - \eta \cdot \nabla f(\vec{\theta}^{k}).$$

Градиентный спуск

Для простоты будем пока говорить про обычный МНК без регуляризации. Как нам минимизировать функцию потерь — методом обратной матрицы или градиентным спуском?

Посчитаем вычислительную сложность первого метода. Рационально умножить вначале вектор Y^T на матрицу X, а затем транспонировать произведение. Сложность этих операций имеет порядок nm. Теперь перемножим матрицы X^T и X — сложность $O(m^2 n)$. Теперь обратим матрицу $X^T X$ — она имеет размер $m \times m$ и сложность обращения есть $O(m^3)$ (можно, правда, пользоваться итерационными методами обращения, но они могут и расходиться). Остается перемножить матрицу $(X^T X)^{-1}$ и вектор $X^T Y$ — сложность $O(m^2)$. Итого, получаем сложность порядка $O(nm^2 + m^3)$. На практике n может быть порядка 10^6, а m порядка 10^3, так что сложность нашего алгоритма имеет порядок 10^{12} — очень много. Вторая проблема аналитического решения — плохая обусловленность матрицы $X^T X$. Действительно, если два каких–то признака почти коллинеарны, то матрица X имеет число обусловленности близкое к нулю (кстати, что

такое число обусловленности?), а обусловленность X^TX равна квадрату обусловленности X, что совсем плохо.

Именно поэтому на практике применяется градиентный спуск. В данном случае у нас выпуклая вниз функция

$$S(\theta) = \sum_{i=1}^{n}(y_i - \theta_0 - \theta_1 x_{i1} - \ldots - \theta_m x_{im})^2.$$

Легко найти ее вектор градиента: j–ая координата этого вектора равна

$$2n\left(-\overline{X_jY} + \theta_0\overline{X_j} + \theta_1\overline{X_1X_j} + \ldots + \theta_m\overline{X_mX_j}\right).$$

Сложность вычисления, как видим равна $O(nm + mk)$, где k — число шагов спуска. Градиентный спуск лучше проводить с переменным шагом — начинать с больших шагов, а затем (при приближении к точке минимума) их уменьшать. Опасения вызывает скорее первый член в асимптотике: nm, поскольку объем выборки n может быть большим. Для ускорения можно проводить усреднения в формуле градиента не по всей выборке, а по случайно выбранной подвыборке. Такой метод градиентного спуска называют стохастическим градиентным спуском[1], а сами подвыборки называют мини–батчами (mini–batch). Чтобы не тратить время на датчик случайных чисел, выборку делят на эти мини–батчи заранее (перед этим выборку обязательно надо перемешать случайным образом — обычно собранные данные обладают «памятью» о порядке их сбора). Затем на каждый шаг спуска подается свой мини–батч. Через некоторое число шагов мы в результате пройдем по всей выборке — такой полный проход называют **эпохой обучения**. Впрочем, бывают такие объемы данных, что модель обучится, пока не закончится даже первая эпоха.

Для более сложных функций потерь в задачах регрессии, альтернатива та же — можно выписать аналитическое решение, но предпочтительнее использовать стохастический градиентный спуск. При этом может возникнуть проблема с функциями MAE и ЛАССО–регрессией, поскольку модель не имеет производной в точке излома. На самом деле, проблема не в этом — вычислительная погрешность не даст методу попасть в точности в точку излома модуля, а даже если это и произойдет, можно свободно считать в этой точке производную модуля равной любому числу между -1 и 1 (такой вектор называют субградиентом функции) — за счет большого числа данных произвол здесь не даст никакого эффекта. Проблема скорее в том, что спуск на таких функциях обычно начинает колебаться поперек линии излома. Действительно, если вы двигаетесь по графику модуля, то оказавшись справа от точки ноль, вы захотите сдвинуться к точке ноль, а так как производная равна 1 (она не мала), то скорее всего, вы перепрыгните ноль, окажитесь слева и т.д. Лечение этой проблемы простое — в таких функциях потерь надо следить за размером колебаний и вовремя уменьшать шаг спуска.

Функции MSE, MAE, их варианты со штрафами — все они выпуклы (некоторые нестрого выпуклы) и ограничены снизу (мы считаем очевидным, что при стремлении θ к бесконечности любым образом функция возрастает — подумайте, почему). Это означает,

[1]Кое–где используется слегка другая терминология. Стохастическим градиентным спуском называют спуск, в котором размер мини–батчей равен строго единице.

что минимум существует всегда, а в задачах с MSE и ее регуляризациями единственен. В задачах с MAE–функциями он может быть не единственен, но это очень редкий случай, а кроме того, значения целевой функции в различных точках минимума одинаковы.

Градиентный спуск позволяет нам ввести еще один метод регуляризации (кроме введения штрафов и того, что мы упоминали в первой лекции — постепенном вводе в метод признаков). Проводя спуск мы можем параллельно вычислять функцию потерь модели на валидационном наборе. В тот момент, когда эта функция потерь перейдет от убывания к росту, мы можем решить, что модель стала переобучаться на тренировочном наборе (на нем, разумеется, будет наблюдаться монотонное снижение потерь). В этот момент мы можем остановить обучение модели (так и не достигнув минимума). Такой метод называют ранним прекращением (early stopping).

Типичные проблемы машинного обучения

Перечислим их.

1. Недостаток данных. Здесь, кажется, все понятно.

2. Некачественные данные. Например, в данных может оказаться слишком много выбросов, чтобы мы могли построить хорошую модель. Тогда можно попробовать отфильтровать выборку, удалив выбросы. Или в данных присутствуют ошибки разметки. Здесь, скорее всего, придется просто собрать данные заново. Или наши данные могут оказаться слишком разнородными. Это сложная проблема и она обычно лечится мучительным подбором правильных признаков модели.

3. Плохая предобработка данных. Дело в том, что многие модели «любят» стандартизированные данные. Стандартизацию обычно проводят так: вычисляют по каждому признаку выборочное среднее и выборочную дисперсию, а затем вычитают из данных среднее и делят на среднее квадратичное отклонение. Альтернатива стандартизации — нормализация данных. Ее проводят так: вычитают из данных минимальное значение, а затем делят на размах (по каждому признаку отдельно, разумеется). Получаются данные в диапазоне $[0, 1]$. Нормализацию следует проводить только после того, как удалены выбросы (иначе вы получите выборку, сконцентрированную в точке 0.5).

4. Переобучение модели (имеется в виду overfitting, т.е. ситуация, когда мы слишком хорошо обучили модель). Действительно, здесь работает принцип «слишком хорошо — тоже плохо». Модель, которая слишком хорошо соответствует training set, может плохо сработать на test set. Типичный пример — полиномиальная регрессия. При попытке приблизить данные многочленом очень высокой степени вы получите удивительное поведение модели между точками интерполяции. Чтобы этого избежать, и придумана validation set. Продолжая последний пример, надо поступить так: объявить степень многочлена (или коэффициент штрафа за большое число ненулевых коэффициентов многочлена) гиперпараметром. Затем запустить обучение модели на training set и валидацию модели на validation set, причем гиперпараметр менять. Вы увидите, как меняется качество модели на validation set. Например,

при увеличении степени многочлена оно вначале будет улучшаться, а затем начнет ухудшаться. Так вы сможете подобрать походящее значение гиперпараметра и избежать переобучения. Альтернатива такому методу: выбрать степень многочлена, походящую по смыслу задачи.

Параметрическая регрессия

Параграф изучается самостоятельно, поскольку содержит материал прошлого семестра.

Этот метод вам отлично известен. У нас есть набор данных X из n образцов с m признаками каждый. Для каждого образца $\vec{x}_i$ известно значение цели y_i. Мы выдвигаем предположение, что $y_i = f(\vec{x}_i)+\varepsilon_i$, где ε_i — случайные величины (шум), а функция f лежит в некотором параметрически заданном семействе. Самый простой случай — линейный:

$$f(\vec{x}) = \theta_0 + \theta_1 x_1 + \cdots + \theta_m x_m = (1,\, x_1, \ldots,\, x_m) \cdot (\theta_0,\, \theta_1, \ldots,\, \theta_m)^T = \vec{x} \cdot \vec{\theta}^T$$

(для удобства мы добавили в данные столбец из единиц). Мы помним, что полиномиальная и логарифмическая модель — это тоже линейная модель. Просто в список ее атрибутов добавляются предикторы X_j^2, X_j^3 и т.д. или $\ln X_j$. Мультипликативные модели вида $\hat{y} = CX_1^{\alpha_1} \cdot X_m^{\alpha_m}$ тоже сводятся к линейному случаю преобразованием таргета вида $Z = \ln Y$. Мы помним (я надеюсь), что часто интересно вводить предикторы из дробей X_j/X_l.

Итак, будем работать с линейной моделью. Стандартное предположение — нормальная распределенность шума. Это действительно разумное предположение, поскольку в силу центральной предельной теоремы сумма большого числа независимых одинаково распределенных случайных величин близка к нормальному распределению, а шум логично считать результатом воздействия на данные большого числа независимых факторов. Правильным сдвигом (выбором θ_0) можно добиться $\varepsilon \sim \mathcal{N}(0, \sigma)$. Еще одно стандартное предположение — независимость случайных величин ε_i между собой. Вы помните, конечно, что после построения модели необходимо провести тесты остатков на нормальность, независимость и гомоскедастичность.

При таких предположениях разумно искать параметры модели θ_j методом наибольшего правдоподобия. Составляем функцию правдоподобия стандартным образом $L = \prod_{i=1}^n p(\varepsilon_i)$ (функция распределения независимых с.в. равна произведению отдельных функций распределения), подставляем $p(\xi) = \frac{1}{\sqrt{2\pi}\sigma} \exp\{-\xi^2/(2\sigma^2)\}$, берем логарифм $\ln L$ и максимизируем его. Отбрасывая постоянные слагаемые и множители получаем задачу $\sum_{i=1}^n \varepsilon_i^2 \to \min$, т.е. тот самый метод наименьших квадратов, который мы хорошо знаем.

Решение задачи минимизации здесь можно выписать явно. Действительно, составим функцию $S(\theta_0, \ldots, \theta_m) = \sum_{i=1}^n (y_i - \theta_0 - \theta_1 x_{i1} - \ldots - \theta_m x_{im})^2$, заметим, что она выпукла вниз, а значит единственная ее стационарная точка является точкой ее минимума. Теперь найдем все частные производные $\frac{\partial S}{\partial \theta_j}$ и приравняем их к нулю. Получим систему нормальных уравнений — линейную систему, которую легко решить. Опуская известные вам выкладки, получим $\vec{\theta} = (X^T X)^{-1} X^T Y$. Можно, однако решать задачу оптимизации и градиентным спуском — это уже вопрос скорости и точности вычислений.

Обратите внимание, что функция потерь $MRSE$ является прямым следствием предположения о нормальности остатков. Если в какой–то задаче мы придем к выводу о том, что остатки подчинены распределению Лапласа $\varepsilon_i \sim \exp\{-|\xi|/\sigma^2\}$, то повторяя рассуждения выше придем к другой задаче минимизации $\sum_{i=1}^n |\varepsilon_i| \to \min$. Это означает, что здесь нашей функцией потерь должна быть MAE. Здесь важен и вывод, и сам метод.

Итак, что мы делаем в реальности. Если нет никаких логических противоречий, то распределение остатков предполагаем нормальным, применяем метод наименьших квадратов, получаем модель, а потом проверяем остатки на нормальность. Если видим в остатках «тяжелые хвосты» и тесты на нормальность не проходят, то меняем метод на минимизацию MAE. Если распределение остатков одностороннее, то можно попробовать предположить их распределенность по Пуассону. И так далее, каждое предположение порождает свою функцию потерь.

Есть и другие популярные функции потерь. Предположим, например, что таргет y_i может принимать значения, различные по порядку. Тогда предположение о постоянстве дисперсий остатков нелогично. Логичнее считать, что $\varepsilon_i \sim \mathcal{N}(0, \sigma y_i)$, т.е. погрешность измерения пропорциональна порядку измеряемой величины. Повторяя рассуждения (составив функцию правдоподобия, ее логарифм и т.д.), придем к задаче

$$\frac{1}{n}\sum_{i=1}^{n}\frac{(y_i - \theta_0 - \theta_1 x_{i1} - \ldots - \theta_m x_{im})^2}{y_i^2} = \frac{1}{n}\sum_{i=1}^{n}\frac{(y_i - \hat{y}_i)^2}{y_i^2} \longrightarrow min.$$

Такую функцию потерь (ее принято усреднить делением на n) называют MSPE (mean squared percentage error). Наконец, если мы предположим, что $\varepsilon_i \sim Laplace(\sigma y_i)$, то получим функцию потерь mean absolute percentage error

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \theta_0 - \theta_1 x_{i1} - \ldots - \theta_m x_{im}}{y_i}\right| = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{y_i}.$$

Обе последние метрики имеют проблемы с делением на ноль в случае, если в выборке встречаются нулевые (или близки к нулевым) значения y_i. С этим можно бороться несколькими способами: вместо «обрезать» значения дробей на некотором фиксированном уровне или заменить метрику, например, на $WAPE = \frac{\sum_{i=1}^n |y_i - \hat{y}_i|}{\sum_{i=1}^n |y_i|}$.

Регуляризация

Вроде все понятно, но есть одна очень важная тонкость. Она состоит в целеполагании. Мы в прошлый раз уже говорили о том, что цель системы машинного обучения — давать хорошие прогнозы на новых данных, основываясь на собранных. А в рассуждениях этой лекции мы исходили из другой задачи — подобрать такие параметры, при которых собранные данные наиболее вероятны.

Давайте обозначим через D событие: результаты измерений признаков окажутся (или оказались) равными нашим данным X. Давайте обозначим через $p(\vec{\theta})$ априорное распределение коэффициентов модели (воспринимая эти коэффициенты как случайные величины). Тогда задача классического анализа данных выглядит так:

$$\hat{\theta} = \arg\max_{\theta} p(D|\theta)$$

(для упрощения записи я перестал писать стрелочку над вектором θ). А наша задача должна быть такой

$$p(y|x,D)\to\max \quad\Longleftrightarrow\quad \int_\Theta p(x,y,D,\theta)\,d\theta=\int_\Theta \frac{p(x,y,D,\theta)}{p(x,D,\theta)}\cdot\frac{p(x,D,\theta)}{p(\theta)}\cdot p(\theta)\,d\theta=$$
$$=\int_\Theta p(y|x,D,\theta)\cdot\frac{p(x)\cdot p(D,\theta)}{p(\theta)}\cdot p(\theta)\,d\theta=p(x)\int_\Theta p(y|x,\theta)\cdot p(D|\theta)\cdot p(\theta)\,d\theta\to\max.$$

Мы использовали то, что новый вектор x не зависит от предыдущих данных D и, разумеется, не зависит от того, какие параметры θ мы берем. Кроме того, мы использовали равенство $p(y|x,D,\theta)=p(y|x,\theta)$, поскольку это следует из сути нашего метода — по данным D мы подбираем параметр θ, а затем с помощью нашей модели и новых данных x строим прогноз y.

Множитель $p(y|x,\theta)$ на самом деле равен нулю в наших предположениях — это вероятность того, что остаток ε в модели $y=f_\theta(x)+\varepsilon$ обнуляется, а мы предположили, что $\varepsilon\sim\mathcal{N}(0,\sigma)$. Естественно, равна нулю и вероятность $p(y|x,D)$, которую мы собрались максимизировать. Действительно, вероятность предсказать значение цели y **в точности** и должна быть нулевой. Правильная постановка — максимизировать вероятность угадывания y с погрешностью a, т.е. вероятность события $p(|\hat y-y|<a|x,D)$. В такой постановке целевая функция задача становится осмысленной, а величина $p(y\in(\hat y-a,\hat y+a)|x,\theta)$ легко вычисляется — это вероятность события $p(|\varepsilon|<a)$ для с.в. $\varepsilon\sim\mathcal{N}(0,\sigma)$. Получаем постоянную величину (квантиль нормального распределения). Значит, множитель $p(y\in(\hat y-a,\hat y+a)|x,\theta)$ не влияет на оптимизацию и его можно отбросить (как и вероятность $p(D)$).

Итак, получаем задачу

$$\int_\Theta p(D|\theta)p(\theta)\,d\theta\to\max.$$

Получается, что при байесовском подходе вообще нет понятия «подходящая модель». Нам необходимо построить все возможные модели нашего класса, взять предсказания каждой из них, а затем усреднить эти предсказания по плотности $p(\theta)$. Вопрос — а что же мы тогда оптимизируем? Очевидно, параметры этой самой плотности (мы сейчас увидим, что это есть гиперпараметры модели). Поскольку теоретическое решение этой задачи нетривиально (и далеко не для всех моделей возможно), обратимся к практике. Упростим задачу: вместо максимизации интеграла будем максимизировать интегрируемую функцию, т.е. вместо B–оценки получим MAP–оценку. Получим

$$\hat\theta=\arg\max_\theta p(D|\theta)p(\theta).$$

Получается, что наш предыдущий метод совпадает с новым только если считать $p(\theta)=const$ (что заведомо невозможно, если мы считаем, что коэффициенты могут оказаться произвольными числами).

Внесем в наши рассуждения коррекции — построим линейную модель MAP–оценкой. Для этого нам надо выдвинуть какие–то предположения об априорных оценках наших параметров. Пусть, например, у нас уже есть результаты решения нашей задачи предыдущими исследователями. Или, возможно, есть какие–то теоретические

предположения о модели. Итак, считаем, что есть априорные оценки параметров $\theta = \mu$. Тогда логично предположить, что эти оценки μ_j есть средние значения случайных величин θ_j, а сами эти величины распределены нормально (почему нормально? — ну надо же что–то предположить) со средним квадратичным a_j. Будем также считать, что случайные величины θ_j попарно независимы. Действовать будем в стандартных предположениях (независимость, несмещенность, нормальность и гомоскедастичность остатков). Тогда функция правдоподобия примет вид

$$L(\theta) = p(\theta)p(D|\theta) = \prod_{j=0}^{m} p_j(\theta_j) \prod_{i=1}^{n} p_i(\varepsilon_i) =$$

$$= (2\pi)^{-(n+m+1)/2}\sigma^{-n} \prod_{j=0}^{m} a_j^{-1} \times \exp\left\{ -\frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 - \frac{1}{2} \sum_{j=0}^{m} \frac{(\theta_j - \mu_j)^2}{a_j^2} \right\}.$$

Возьмем логарифм и отбросим константы, не влияющие на оптимизацию:

$$\sum_{i=1}^{n} (y_i - \vec{x} \cdot \theta^T)^2 + \sum_{j=0}^{m} \frac{\sigma^2}{a_j^2} (\theta_j - \mu_j)^2 \longrightarrow min.$$

Мы пришли к ридж–регрессии! Функция потерь — это сумма квадратов отклонений модельных значений от верных, но к этой сумме добавлено слагаемое штрафа за отдаление коэффициентов θ_j от априорных своих значений μ_j.

Все это хорошо, но откуда мы возьмем значения коэффициентов штрафа — чисел $\lambda_j = \frac{\sigma^2}{a_j^2}$, которые, конечно, влияют на положение точки минимума? Штраф за коэффициент θ_0 не вводят — этот коэффициент отвечает за несмещенность модели, и нарушать ее нет смысла. Остальные коэффициенты λ_j часто берут одинаковыми (обозначим их λ). Для выбора λ обычно поступают утилитарно: обучают серию моделей с разными коэффициентами λ, а затем проверяют их на валидационном наборе, вычисляя там уже обычную нерегуляризованную MRSE. Финальной выбирается модель с тем λ, для которых MRSE минимально. Это и есть тот самый подбор гиперпараметров, про который мы говорили в прошлой лекции. По поводу чисел μ_j: если нам априори они не известны, а регуляризовать модель хочется, то можно выбрать их попросту нулями.

Верно ли, что мы только что предложили численное решение задачи байесовской оптимизации, с которой начали разговор о регуляризации? Не совсем. Вообще–то для каждого значения гиперпараметра мы должны строить ансамбль моделей с различными значениями θ, брать усредненное предсказание на валидационном наборе по всем моделям (усреднение надо проводить с нормальной плотностью, в которую входят наши гиперпараметры), а затем выбирать те значения гиперпараметров, которые обеспечивают максимальный результат по MRSE. Понятно, что это слишком затратно с вычислительной точки зрения. Запомним, однако, что необходимость использования ансамблей моделей теоретически обосновано.

Если мы предположим не нормальное распределение коэффициентов θ_j, а распределение Лапласа, то слагаемое штрафа поменяется на $\sum_{j=1}^{m} \lambda_j |\theta_j - \mu_j|$ и мы получим ЛАССО–регрессию. Такая регуляризация склонна обнулять (при $\mu_j = 0$) «лишние» коэффициенты модели (в то время как ридж–регрессия их не обнуляет, а

только уменьшает). Это несомненно достоинство ЛАССО–регрессии. На практике еще лучше работает смесь двух регрессий elastic net. Здесь штраф имеет вид

$$\sum_{j=1}^{m} \lambda_j \left(r|\theta_j - \mu_j| + \frac{1-r}{2}(\theta_j - \mu_j)^2 \right).$$

Гиперпараметр $r \in [0,1]$ отвечает за смешивание двух штрафов (при $r = 1$ получаем ЛАССО–регрессию, а при $r = 0$ — ридж). Его тоже подбирают на валидационной выборке.

Упражнение 1. *Какая регуляризация получится, если мы выдвинем априорное предположение о равномерном распределении коэффициентов $\theta_j \sim R[a_j, b_j]$?*

Упражнение 2. *Докажите, что если априорное распределение θ — нормальное $\mathcal{N}(\mu_0, \Sigma_0)$, а остатки независимы, несмещены, нормальны и гомоскедастичны, т.е. $\varepsilon_j \sim \mathcal{N}(0, \sigma)$, то апостериорное распределение параметров тоже нормально: $p(\theta|D) \sim \mathcal{N}(\mu, \Sigma)$. Проверьте, что*

$$\Sigma = \left(\Sigma_0^{-1} + \frac{1}{\sigma^2} X^T X \right)^{-1}, \qquad \mu = \Sigma \Sigma_0^{-1} \mu_0 + \frac{1}{\sigma^2} \Sigma X^T Y.$$

Эти формулы имеют не только теоретическую ценность. Пусть вы построили модель, а потом хотите ее дообучить на новых данных. Тогда числа μ_j разумно взять уже другими — теми, которые получаются по второй из приведенных формул!

2 Качество классификации

Для задач регрессии обычно все просто: качество модели характеризуется значением ее функции потерь. Впрочем, разумно использовать в качестве функции потерь на тестовой выборке регуляризованную функцию, а мерой качества сделать обычную RMSE или MAE на валидационной выборке. Есть, правда, одна функция, которую никогда не используют как функцию потерь, но используют как меру качества — доля предсказаний с абсолютными ошибками больше, чем d:

$$\frac{1}{n} \sum_{i=1}^{n} I_{|y_i - \hat{y}_i| > d}$$

(d — параметр).

Для того, чтобы привести меру качества к единой шкале используется коэффициент детерминации модели $R^2 = 1 - \frac{RSS}{TSS}$. Здесь $RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$, а $TSS = \sum_{i=1}^{n} (y_i - \overline{y})^2$. Для линейных моделей регрессии эти величины совпадают, когда $\theta_1 = \cdots = \theta_m = 0$ (что очевидно). В остальных случаях (если модель строилась методом наименьших квадратов) $RSS < TSS$ (подумайте, почему — на этот вопрос можно ответить без вычислений). Таким образом, для моделей, построенных по МНК $R^2 \in [0,1]$ и, действительно, показывает качество модели. Конечно, если метод был другим, то R^2 может оказаться и отрицательным, что тоже показывает качество (точнее, некачественность) модели.

Мы же поговорим сегодня об оценке качества бинарной классификации. Напомню, что задача классификации — по данным X и меткам $y_i \in \{0, 1\}$ научиться классифицировать новые данные $\vec{x}$ к первому или второму классу. В отличие от регрессии, с оценкой качества здесь все интереснее.

Введем обозначения. Предположим, что метка $y_i = 1$ и модель предсказывает $\hat{y}_i = 1$. Такой случай назовем верным положительным (true poritive) и число таких векторов обозначим TP. Если $y_i = 1$, а $\hat{y}_i = 0$, то скажем, что это ложная отрицательная классификация (false negative), число таких классификаций обозначим FN. Аналогично определим верные отрицательные (true negative) и ложные положительные (false positive) классификации, обозначим их число TN и FP. Матрицу $\left(\begin{smallmatrix} TN & FP \\ FN & TP \end{smallmatrix}\right)$ называют матрицей неточностей (confusion matrix).

На первый взгляд все просто: функцией качества должен быть процент верных ответов модели (accuracy — верность классификации) $Acc = \frac{TP+TN}{n}$. Проблема в том, что размеры классов $n_0 = TN + FP$ и $n_1 = TP + FN$ могут сильно различаться. Представим ситуацию, когда $n_0 = n_1$ (классы сбалансированы). Точность модели $Acc = 0.95$ означает здесь, что вероятность верной классификации наблюдения из класса C_j равна

$$p_0 = p(\hat{y} = 0|y = 0) = \frac{TN}{n_0} = 2\frac{TN}{n}, \quad p_1 = p(\hat{y} = 1|y = 1) = \frac{TP}{n_1} = 2\frac{TP}{n}, \quad \frac{p_0 + p_1}{2} = Acc = 0.95,$$

а значит, каждая из этих вероятностей не меньше 0.9. Представим теперь другую ситуацию, когда $n_0 = 0.95n$, а $n_1 = 0.05n$ (дисбаланс классов — класс C_1 встречается в 19 реже, чем C_0). Возьмем «глупую модель», которая всегда будет выдавать значение $\hat{y} = 0$. Очевидно, что $Acc = 0.95$, хотя вероятность верной классификации класса C_1 равна $p_1 = TP/n = 0$.

Как видим, метрика Acc недостаточна для оценки качества модели. Вспомним, что нам хорошо известны из статистики понятия ошибки первого рода: модель сказала «нет», хотя на самом деле «да» и ошибки второго рода: модель сказала «да», хотя на самом деле «нет». Обратите внимание, что при перестановке классов местами ошибка первого рода становится ошибкой второго рода, а второго рода — первого. Конечно, хотелось бы, чтобы наша модель не допускала ошибок ни первого, ни второго рода. Однако наши желания далеко не всегда совпадают с возможностями и нам приходится определяться в каждой конкретной задаче — ошибки какого рода для нас хуже. Представим, что наша система пытается определить, не болен ли пациент опасной болезнью. Очевидно, что здесь лучше перестраховаться и при положительной классификации задержать пациента в больнице, сделать подробные исследования и т.д. — это все неприятно, но гораздо лучше, чем выпустить больного: мы будем в первую очередь минимизировать ошибки первого рода. Теперь рассмотрим другую историю, когда наш классификатор осуждает виновного на смерть. Очевидно, что здесь надо всеми силами избегать ошибки второго рода.

В машинном обучении принято обозначать ошибку первого рода $fnr = \frac{FN}{TP+FN} = \frac{FN}{n_1}$ — false negative rate. Соответственно, ошибка второго рода $fpr = \frac{FP}{TN+FP} = \frac{FP}{n_0}$ — false positive rate. Величину $1 - fnr = \frac{TP}{TP+FN}$ — долю верной классификации на классе C_1 называют полнотой (recall, иногда еще sensetivity — чувствительность). Есть две крайности. Можно представить себе классификатор, который все векторы относит ко классу C_0. Тогда $TP = FP = 0$, поскольку случай positive classification не возможен. Для

такого классификатора получим $fpr = 0$, $recall = 0$. Другая крайность — классификатор, который все векторы относит ко классу C_1. Здесь $TN = FN = 0$ и тогда $fpr = recall = 1$. Все остальные классификаторы находятся «где–то между» этими крайностями.

Если мы стартуем от первого классификатора и постепенно начинаем увеличивать число точек, классифицируемых как C_1, то у нас будет расти величина TP. Но будет увеличиваться и величина FP (не ошибается только тот, кто ничего не делает). В результате будут постепенно увеличиваться обе величины: fpr и $recall$, меняясь от 0 к 1. На практике за изменения отвечает некоторый параметр (пороговый параметр — threshold) классификатора t. Таким образом, мы получаем параметрическую кривую $(fpr(t), recall(t))$, соединяющие точки $(0, 0)$ и $(1, 1)$ на плоскости $(fpr, recall)$. Эту кривую называют *ROC*–кривой (Receiver Operating Characteristic). Каждый алгоритм классификации имеет свою ROC–кривую (на фиксированном наборе данных разумеется). Выбор между способами классификации — это выбор ROC–кривой, а настройка параметров модели — выбор точки на этой кривой. В частности, выбор ROC–кривой можно осуществлять подбором гиперпараметров алгоритма.

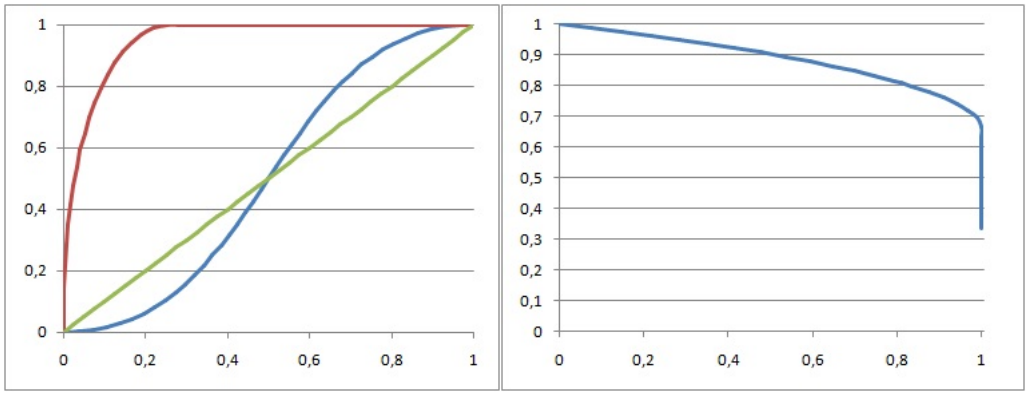
Пример 1. *Пусть надо классифицировать n точек, расположенных в квадрате $[-1, 1]^2$. Пусть точки, лежащие выше параболы $y = x^2$ помечены C_1, а ниже — C_0. Пусть распределение точек равномерно на квадрате, а n достаточно велико, чтобы мы могли рассуждать в средних значениях. Поскольку площадь фигуры $S_1 = \{(x, y) : \ y > x^2, -1 \le x, y \le 1\}$ составляет $1/3$ от площади всего квадрата, то доля точек, помеченных C_1 будет $n_1 \approx \frac{1}{3}n$. Тогда $n_0 \approx \frac{2}{3}n$.*

Предложим вначале очень простой классификатор. Не глядя на пометки точек, будем случайным образом относить точки ко классу C_1 с вероятностью p, а ко классу C_0 с вероятностью $1 - p$ (т.е. будем просто подбрасывать монетку). Число p будет параметром нашего классификатора (а гиперпараметров у него нет). Тогда математическое ожидание матрицы неточностей равно $n \begin{pmatrix} 2(1-p)/3 & 2p/3 \\ (1-p)/3 & p/3 \end{pmatrix}$ (это закон распределения двумерной случайной величины, полученной перемножением двух одномерных распределений — ведь наш классификатор независимо от меток выдает свой ответ). Легко найти, что здесь $fpr(p) = recall(p) = p$. ROC–кривая есть просто диагональ квадрата.

Теперь возьмем такой классификатор: проведем вертикальную прямую с абсциссой $t \in [-1, 1]$, все что лежит справа от прямой будем классифицировать как C_1, а все что слева — как C_0. Ясно, что этот классификатор совсем не походит к нашей задаче: событие «лежать выше параболы» никак не связано с событием «лежать правее». Считая, что n велико и вычисляя вероятности при помощи площадей фигур, получим

$$\frac{TN}{n} \approx \frac{t^3 + 3t + 4}{12}, \quad \frac{FN}{n} \approx \frac{-t^3 + 3t + 2}{12}, \quad \frac{FP}{n} \approx \frac{-t^3 - 3t + 4}{12}, \quad \frac{TP}{n} \approx \frac{t^3 - 3t + 2}{12}.$$

ROC–кривая этого классификатора приведена на рисунке (синий цвет). Видим, что она иногда уходит слегка выше диагонали, а иногда даже ниже нее. Такой классификатор нам не нужен!



Попробуем сделать другой классификатор: проведем горизонтальную кривую с параметром $t \in [-1, 1]$*, все что лежит выше прямой будем классифицировать как* C_1*, а все что ниже — как* C_0*. Сразу чувствуется, что такой классификатор сработает лучше — событие «лежать выше параболы» похоже на событие «лежать выше горизонтали». Опуская вычисления, получим*

$$\frac{TN(t)}{n} \approx \begin{cases} \frac{t}{2} + \frac{1}{2}, & t < 0, \\ \frac{t}{2} + \frac{1}{2} - \frac{t^{3/2}}{3}, & t \geq 0, \end{cases} \qquad \frac{FN(t)}{n} \approx \begin{cases} 0, & t < 0, \\ \frac{t^{3/2}}{3}, & t \geq 0, \end{cases}$$

$$\frac{FP(t)}{n} \approx \begin{cases} \frac{1}{6} - \frac{t}{2}, & t < 0, \\ \frac{1}{6} - \frac{t}{2} + \frac{t^{3/2}}{3}, & t \geq 0, \end{cases} \qquad \frac{TP(t)}{n} \approx \begin{cases} \frac{1}{3}, & t < 0, \\ \frac{1}{3} - \frac{t^{3/2}}{3}, & t \geq 0. \end{cases}$$

График ROC–кривой имеет красный цвет на рисунке. Видно, что эта кривая значительно отклоняется от диагонали. В точке $(0.25,\ 1)$ *она выходит на верхнюю границу квадрата — классификатор делает ошибку второго рода с вероятностью* 0.25*, а ошибок первого рода не делает вовсе. Это соответствует параметру* $t = 0$*. Если ошибки обоих родов нам одинаково важны, то можно выбрать* $t = 0.25$ *— вероятности ошибок обе окажутся равными* 0.125*. Ну а если нам поставлено требование допускать не более* 5 *процентов ошибок второго рода, то придется согласиться на вероятность* 0.35 *ошибки первого рода (это много, конечно).*

Ну и наконец предложим идеальный для данной задачи классификатор. Проведем параболу $y = t + x^2$ *с параметром* $t \in [-1, 1]$ *и все что лежит выше нее будем классифицировать как* C_1*, а все, что ниже — как* C_0*. Очевидно, что при* $t = 0$ *мы получим* $fpr = 0$*,* $recall = 1$ *— это идеальный случай. На самом деле, из–за статистических отклонений от нормального распределения на каждом конкретном наборе данных получится своя точка — близкая к* $(0, 1)$*, но не равная ей.*

Упражнение 3. *Постройте ROC–кривую для последнего классификатора, оперируя математическим ожиданием матрицы неточностей.*

Как уже говорилось, очень желательно перед началом построения классификатора определиться — какой класс C_1 или C_0 для вас характернее. Давайте договоримся, что мы будем вводить метки классов так, чтобы следить за классом C_1. Сосредоточимся на предсказаниях именно по этому классу. Тогда нас в первую очередь интересует величина $recall = \frac{TP}{TP+FN}$, которая является наблюдаемым значением случайной величины (модель

скажет«да» при условии, что в реальности «да»). Вспомним байесовский подход: мы сейчас сосредоточились на том, как подобрать модель, чтобы она получше предсказывала данные по классу C_1, а надо еще держать в голове апостериорные вероятности. Таки образом, нас должно интересовать событие (в реальности «да» при условии, что модель предсказывает «да»). Наблюдаемым значением вероятности этого события является $prec = \frac{TP}{TP+FP}$ — точность модели (precision, не путайте с accuracy — верность классификации).

Здесь тоже есть две крайности. Можно представить себе модель, которая всегда выдает $\hat{y}_i = 1$. Для нее $TN = FN = 0$, $TP = n_1$, $FP = n_0$, так что $recall = 1$, а $prec = \frac{n_1}{n}$ — наша модель максимизирует $recall$. Другая крайность — модель, которая всегда выдает $\hat{y}_i = 0$. Для нее $TP = FP = 0$, и тогда $recall = 0$, а точность не определена (точнее, ее надо считать переходом к пределу в неопределенности $\frac{TP}{TP+FP}$, в результате чего может получится любое число между 0 и 1, но для разумных моделей $FP << TP$ и в пределе $prec = 1$). Меняя параметр threshold классификатора, мы можем двигаться между этими двумя крайностями, получаю другую параметрически заданную кривую $(recall(t), prec(t))$. При этом $recall(t)$ строго понижается от 1 до 0, а $prec(t)$ обычно немонотонна, но в целом растет. Эту кривую называют PR–кривой. При движении по ней мы балансируем точность подгонки классификатора по классу C_1 на наших данных (recall) и надежность предсказания класса C_1 (precision).

Пример 2. *Для третьего нашего классификатора из предыдущего примера PR–кривая приведена на втором рисунке. Видим, что интересной оказывается точка $t = 1/3$, для которой $prec = recall = 0.81$. Кроме того, видно, что наш классификатор совсем не так хорош, как нам казалось, глядя на ROC–кривую.*

Чем лучше пользоваться — кривой ROC или PR? Есть эмпирическое правило: если класс C_1 является редким или ложноположительные результаты нежелательны (а ложноотрицательные допустимы), надо пользоваться кривой PR. В противном случае — кривой ROC. Впрочем, между ними есть связь.

Утверждение 1. *Если есть два классификаторы с монотонными ROC–кривыми, и известно, что для первого классификатора ROC–кривая проходит выше ROC–кривой второго (пересекаясь только в концах), то PR–кривая первого также пройдет выше PR–кривой второго.*

Доказательство. Будем обозначать характеристики первого классификатора индексом 1, а характеристики второго — индексом 2. Надо показать, что кривая $(recall_1(t), prec_1(t))$ проходит выше кривой $(recall_2(t), prec_2(t))$, т.е. для каждой точки, где $recall_1 = recall_2 = r$ выполнено $prec_1 > prec_2$. Пусть $recall_1(t)$ принимает значение r при threshold $t = t_1$, а $recall_2(t) = r$ при $t = t_2$. Пусть $fpr_1(t_1) = a$, $fpr_2(t_2) = b$. Так как ROC_1–кривая выше ROC_2–кривой, то $recall_2(t_1) = r'_0 < r_0$. В силу монотонности возрастание координаты $recall_2$ от r'_0 к r_0 должно сопровождаться возрастанием fpr_2 от a к b, т.е. $b = fpr_2(t_2) > a = fpr_1(t_1)$. Поскольку речь идет про классификацию одного и того же массива данных, то показатели n, n_0 и n_1 у обоих классификаторов совпадают. Значит, последнее неравенство означает $FP_2(t_2) > FP_1(t_1)$. Остается заметить, что равенство $recall_1 = recall_2$ влечет $TP_1(t_1) = TP_2(t_2)$. Отсюда выводим, что $prec_1(t_1) = \frac{TP_1}{TP_1+FP_1} > \frac{TP_2}{TP_2+FP_2} = prec_2(t_2)$. □

Кривые ROC и PR характеризуют классификатор «в целом». Если хочется дать числовую характеристику классификатора, то есть два стандартных способа — площадь под ROC–кривой (величина AUC — area under curve) и площадь под PR–кривой (величина AP — average precision).

Двигаясь по PR–кривой мы можем захотеть найти оптимальное значение порога t. Здесь можно ориентироваться на F_1–меру, которая равна среднему гармоническому precision и recall или, если важность precision и recall для вас различна, то можно взять взвешенное среднее гармоническое (F_β–мера)

$$F_1 = \frac{2}{\frac{1}{recall} + \frac{1}{precision}}, \qquad F_\beta = \frac{\beta + \beta^{-1}}{\frac{\beta}{recall} + \frac{\beta^{-1}}{precision}} = \frac{(\beta^2+1)\cdot precision \cdot recall}{\beta^2 precision + recall}.$$

Двигая параметр β по шкале от 0 до $+\infty$ можно увеличивать в мере значимость в recall (уменьшая значимость precision): $F_0 = prec$, $F_\infty = recall$.

Например, для нашего классификатора с горизонтальной отсечкой, максимальное значение $F_1 = 0,828$ получим при $t = 0,1715$, при котором $recall = 0,929$, а $precision = 0,7475$.

3 Простые классификаторы

Пусть нам дана задача классификации: собраны данные D из n образцов (samples) $\vec{x}_i$, каждый из которых имеет m координат (features). Пусть для каждого образца известна метка $y_i \in \{0, 1\}$, показывающая, к какому классу (C_0 или C_1) относится данный образец. Необходимо придумать правило, которое на основе собранных данных позволяло отнести любой новы образец $\vec{x}$ к тому или иному классу.

Применим байесовский подход. Предположим, что векторы $\vec{x}$ класса C_1 имеют распределение с плотностью $f_1(x) = p(x|C_1)$, а векторы класса C_0 — распределение с плотностью $f_0(x) = p(x|C_0)$ (эти плотности мы, конечно, не знаем). По формуле Байеса

$$r(\vec{x}) := p(\vec{x} \in C_1|\vec{x}) = \frac{p(x|C_1)p(C_1)}{p(x|C_1)p(C_1) + p(x|C_0)p(C_0)} = \frac{\pi_1 f_1(x)}{\pi_1 f_1(x) + \pi_0 f_0(x)}. \tag{1}$$

Для дискретных распределений на множестве $X = \{\vec{z}^1, \ldots, \vec{z}^s\}$ получим аналогично $p(\vec{x} = \vec{z}^j) = \frac{\pi_1 p_1(x=z^j)}{\pi_1 p_1(x=z^j) + \pi_0 p_0(x=x^j)}$. Предположим, что функция $r(\vec{x})$ нам известна. Тогда достаточно установить значение гиперпараметра $t \in (0, 1)$, и классифицировать образцы по правилу $\widehat{y}(\vec{x}) = I_{r(\vec{x})>t}$ (если значение функции $r(x)$ больше t, то относим образец $\vec{x}$ к классу C_1, а иначе — к C_0). В частности, при $t = 1/2$ построенный классификатор называется **байесовским классификатором** $\widehat{y}_b(\vec{x})$.

Будем сравнивать классификаторы по величине $Acc = \frac{1}{n}\sum_{i=1}^n I_{\widehat{y}_i \neq y_i}$.

Утверждение 2. *Байесовский классификатор имеет наибольшую Accuracy среди всех классификаторов.*

Доказательство. Рассмотрим произвольный классификатор $h(\vec{x})$. Тогда $Acc(\widehat{y}_b) - Acc(h) = \frac{1}{n}\left(\#\{i : \widehat{y}_b(\vec{x}_i) = y_i\} - \#\{i : h(\vec{x}_i) = y_i\}\right)$. Распишем выражение, находящееся в

скобках, для чего рассмотрим восемь случаев — в зависимости от значений $y,\ \widehat{y},\ h \in \{0,\ 1\}$. Получим

$$\#\{i:\ y=\widehat{y}=0,\ h=1\}+\#\{i:\ y=\widehat{y}=1,\ h=0\}-\#\{i:\ y=h=0,\ \widehat{y}=1\}-$$
$$-\#\{i:\ y=h=1,\ \widehat{y}=0\}=\sum_{\substack{i:r(\vec{x}_i)>1/2,\\ h(\vec{x}_i)=0}}(I_{y_i=1}-I_{y_i=0})+\sum_{\substack{i:r(\vec{x}_i)<1/2,\\ h(\vec{x}_i)=1}}(I_{y_i=0}-I_{y_i=1})\,.$$

Обозначим вероятность события $y_i = 0$ как $p(y_i = 0|\vec{x}_i, D)$ (условная вероятность на нашем массиве данных и при условии, что значения предикторов равны вектору $\vec{x}_i$). Аналогично для события $y_i = 1$. Тогда первая сумма с точностью до множителя $\#\{i:\ r(\vec{x}_i) > 1/2,\ h(\vec{x}_i) = 0\}$ равна $\sum_{\substack{i:r(\vec{x}_i)>1/2,\\ h(\vec{x}_i)=0}}(p(y_i = 1|\vec{x}_i, D) - p(y_i = 0|\vec{x}_i, D))$. Заметим, что неравенство $r(\vec{x}_i) > 1/2$ (просто по определению функции r) равносильно неравенству $p(y_i = 1|\vec{x}_i, D) > p(y_i = 0|\vec{x}_i, D)$, т.е. разность вероятностей в нашей сумме всегда положительна. Рассуждения для второй суммы аналогичны. □

Проблема в том, что мы не знаем ни чисел π_0, π_1, ни плотностей $f_0(\vec{x})$, $f_1(\vec{x})$.

Наивный байесовский подход

Давайте оценим неизвестные нам величины так, как мы это делали в курсе статистики — методом наибольшего правдоподобия. Оценки чисел π_0, π_1 очевидны: $\hat{\pi}_0 = \frac{n_0}{n}$, $\hat{\pi}_1 = \frac{n_1}{n}$.

Для оценок плотностей нужно что-то о них предположить. Предположим для начала, что наши случайные величины дискретны и принимают значения в множестве $X = \{\vec{z}^1, \ldots, \vec{z}^s\}$. Такая задача нами в курсе статистики уже рассматривалась (правда, в одномерном случае, но это не важно). Ответ следующий: $\hat{p}_0(\vec{x} = \vec{z}^j) = \frac{\#\{i:\,\vec{x}_i=\vec{z}^j,\ y_i=0\}}{n_0}$, $\hat{p}_1(\vec{x} = \vec{z}^j) = \frac{\#\{i:\,\vec{x}_i=\vec{z}^j,\ y_i=1\}}{n_1}$. Подставим полученные выражения в формулу для $r(x)$ — получим оценку этой функции $\hat{r}(x)$, далее запустим классификатор.

Дискриминантный анализ

Теперь предположим, что плотности f_0 и f_1 являются нормальными плотностями с различными средними $\vec{\mu}_0$ и $\vec{\mu}_1$ и одинаковой ковариационной матрицей Σ т.е., что

$$f_k(\vec{x}) = \frac{1}{(2\pi)^{m/2}\sqrt{\det(\Sigma)}}\exp\left(-\frac{1}{2}\left\langle \vec{x}-\vec{\mu}_j, \Sigma^{-1}(\vec{x}-\vec{\mu}_j)\right\rangle\right).$$

Оценим параметры $\vec{\mu}_0$, $\vec{\mu}_1$ и Σ методом наибольшего правдоподобия на основе наших данных. Эти оценки нам хорошо известны — выборочные математические ожидания и выборочная дисперсия:

$$\hat{\mu}_0 = \frac{1}{n_0}\sum_{i:\ y_i=0}\vec{x}_i, \qquad \hat{\mu}_1 = \frac{1}{n_1}\sum_{i:\ y_i=1}\vec{x}_i,$$

$$\widehat{\Sigma} = \frac{1}{n}\left(\sum_{i:\ y_i=0}(\vec{x}_i - \hat{\mu}_0)(\vec{x}_i - \hat{\mu}_0)^T + \sum_{i:\ y_i=1}(\vec{x}_i - \hat{\mu}_1)(\vec{x}_i - \hat{\mu}_1)^T\right),$$

т.е. элемент матрицы $\hat{\Sigma}$ в строке α столбце β равен

$$\frac{1}{n}\sum_{i:\, y_i=0}(x_{i\alpha} - \hat{\mu}_{0\alpha})(x_{i\beta} - \hat{\mu}_{0\beta}) + \frac{1}{n}\sum_{i:\, y_i=0}(x_{i\alpha} - \hat{\mu}_{1\alpha})(x_{i\beta} - \hat{\mu}_{1\beta}).$$

Полученные оценки используем для вычисления функций $\hat{f}_0(\vec{x})$ и $\hat{f}_1(\vec{x})$ — получим оценку $\hat{r}(x)$. Полученный метод называется **линейным дискриминантным анализом** LDA.

Ослабим предположения — позволим плотностям f_0 и f_1 иметь разные ковариационные матрицы Σ_0 и Σ_1. Тогда оценки для этих матриц примут вид

$$\widehat{\Sigma}_0 = \frac{1}{n_0}\sum_{i:\ y_i=0}(\vec{x}_i - \hat{\mu}_0)(\vec{x}_i - \hat{\mu}_0)^T, \qquad \widehat{\Sigma}_1 = \frac{1}{n_1}\sum_{i:\ y_i=1}(\vec{x}_i - \hat{\mu}_1)(\vec{x}_i - \hat{\mu}_1)^T.$$

Полученный метод называют **квадратичным дискриминантным анализом** QDA.

Логистический классификатор

Получим оценку для $\hat{r}(\vec{x})$ из других соображений. Разделим числитель и знаменатель дроби в (1) на $p(x|C_0)\pi_0$ и обозначим $a = \ln\frac{p(x|C_1)\pi_1}{p(x|C_0)\pi_0}$ — получим $p(C_1|x) = \sigma(a) = \frac{1}{1+e^{-a}}$. Переменную a называют ненаблюдаемой или латентой переменной.

Попробуем использовать линейную модель для a, т.е. положим $a(\theta) = \theta_0 + \theta_1 x_1 + \cdots + \theta_m x_m$. Вот мы и получили логистический классификатор. Поскольку он получен из других соображений, то для него разумно разрешить менять параметр $1/2$ на другие значения t. Обратите внимание, что неравенство $\sigma(a) > t$ равносильно $a(\theta) > \ln\frac{t}{1-t}$, т.е. геометрически происходит следующее. В пространстве $\mathbb{R}^m$ проводится гиперплоскость, и все точки выборки, лежащие по одну сторону от нее, относятся ко классу C_1, а те, которые по другую сторону — ко классу C_0. Мы получили линейный классификатор (впрочем, как и все предыдущие).

Итак, модель выбрана. Теперь надо выписать функцию потерь, чтобы научиться подбирать параметры θ_j. Прежде всего, отметем все идеи об оптимизации accuracy — эта функция является кусочно–постоянной, так что градиентным спуском ее «не возьмешь»; кроме того, как мы уже говорили выше, она далеко не самый хороший показатель качества модели. К счастью, для случайных величин есть классическое понятие «расстояния» между ними — дивергенция Кульбака–Лейблера. Пусть есть истинное распределение P и его приближение — распределение Q. Тогда отклонение Q от P по мере P есть

$$KL_P(P,Q) = \int \ln\frac{dP}{dQ}dP = \sum_i p_i \ln\frac{p_i}{q_i} = \int_{\mathbb{R}} p(x)\ln\frac{p(x)}{q(x)}\,dx$$

(здесь приведена общая формула и ее частные случаи — для дискретных и для абсолютно непрерывных распределений). Наша цель — минимизировать эту дивергенцию, считая, что dP — это истинное распределение, т.е. наши данные y_i, а dQ — его приближение, т.е.

наши прогнозы $\hat{y}_i$. Поскольку $KL_P(P,Q) = \int \ln dP\, dP - \int \ln dQ\, dP$, а распределение dP задано, то минимизировать надо второе слагаемое $-\int \ln dQ\, dP$. Введем распределение P тривиальным образом: $p(Y=1|x_i) = y_i$, $p(Y=0|x_i) = 1-y_i$. Распределение Q нашего классификатора будет, какое получится. Будем воспринимать ответ классификатора $\hat{y}_i$ как вероятность события $p(y_i=1|x_i) = \hat{y}_i$. Соответственно, $p(y_i=0|x_i) = 1-\hat{y}_i$. Тогда для каждого $i \in [1,n]$ получаем для минимизации сумму $-y_i \ln \hat{y}_i - (1-y_i)\ln(1-\hat{y}_i)$. Остается просуммировать эти выражения и мы получаем функцию потерь

$$\ell(\theta) = -\frac{1}{n}\sum_{i=1}^{n}(y_i \ln \hat{y}_i(\theta) + (1-y_i)\ln(1-\hat{y}_i(\theta))).$$

Можно возразить, что в математике известно много других расстояний между распределениями — почему же мы выбрали именно KL–дивергенцию? Оказывается именно ее мы получаем методом максимального правдоподобия. В процессе вывода формулы логистической регрессии мы пришли к распределению Бернулли: для каждого вектора $\vec{x}_i$ вычисляется вероятность $\hat{y}_i = \sigma(\theta_0 + \theta_1 x_{i1} + \dots)$, а затем, фактически, подбрасывается монетка — с вероятностью $\hat{y}_i$ точка относится в класс C_1, а с вероятностью $1-\hat{y}_i$ — в класс C_0. Считая все измерения $\vec{x}_i$ независимыми, получим функцию правдоподобия

$$L(\theta) = p(y|X,\theta) = \prod_{i=1}^{n} p(Y_i = y_i|x_i,\theta) = \prod_{i=1}^{n} \hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i}.$$

Как обычно, берем логарифм и устремляем его к максимуму:

$$\ln L(\theta) = \sum_{i=1}^{n}(y_i \ln \hat{y}_i + (1-y_i)\ln(1-\hat{y}_i)) \longrightarrow max.$$

Усреднив и поменяв знак, получим ту же самую функцию потерь.

Поговорим о градиентном спуске в задачах логистической регрессии. Проверим функцию потерь на выпуклость. Для удобства вычислений будем применять метки классов не 0 и 1, а -1 и 1. Тогда

$$\ell(\theta) = -\ln\left(\prod_{y_i=1}\hat{y}_i \times \prod_{y_i=-1}(1-\hat{y}_i)\right) = -\sum_{y_i=1}\ln\frac{1}{1+e^{-a_i}} - \sum_{y_i=-1}\ln\frac{e^{-a_i}}{1+e^{-a_i}} =$$

$$= \sum_{y_i=1}\ln(1+e^{-a_i}) + \sum_{y_i=-1}\ln(1+e^{a_i}) = \sum_{i=1}^{n}\ln(1+e^{-y_i a_i}), \qquad a_i = \theta_0 + \theta_1 x_{i1} + \ldots + \theta_m x_{im},$$

$$\frac{\partial \ell}{\partial \theta_j} = -\sum_{i=1}^{n}\frac{y_i x_{ij} e^{-y_i a_i}}{1+e^{-y_i a_i}} = -\sum_{i=1}^{n}\frac{y_i x_{ij}}{1+e^{y_i a_i}}, \qquad \frac{\partial^2 \ell}{\partial \theta_j \partial \theta_k} = \sum_{i=1}^{n}\frac{y_i^2 x_{ij} x_{ik} e^{y_i a_i}}{(1+e^{y_i a_i})^2}.$$

Если теперь внимательно присмотреться, то можно увидеть, что частная производная по θ_j и θ_k равна скалярному произведению векторов X_j и X_k с весами $\frac{e^{y_i a_i}}{(1+e^{y_i a_i})^2}$. Введем в пространстве $\mathbb{R}^n$ такое новое скалярное произведение (веса все положительны, так что аксиомы выполнены) и тогда матрица вторых производных превратиться в матрицу Грама

относительно этого произведения, а матрица Грама всегда положительно определена. Значит, второй дифференциал функции ℓ положителен, а тогда она выпукла.

Выпуклая функция может иметь только одну точку минимума — отлично! Но наши рассуждения содержат пробел (весьма тонкий). Все это верно в области конечных чисел a_i. Но равенство $\hat{y}_i = 1$ соответствует $a_i = +\infty$, равно как $\hat{y}_i = 0$ соответствует $a_i = -\infty$. Это означает, что для линейно разделимых классов (когда есть семейство плоскостей, строго их разделяющих), наша функция примет значение ноль на всем этом семействе (это впрочем отлично видно из первой строки — если все $\hat{y}_i$ угаданы в точности, то в произведении стоят единицы и логарифм обнуляется). В этой ситуации наша функция выпукла нестрого и минимум не единственен.

Ситуация исправляется регуляризацией задачи, которую все равно проводить разумно — из байесовского подхода. Например, если у нас есть априорная информация о предполагаемом положении разделяющей (возможно, нестрого) плоскости $\theta_j = \mu_j$ и мы предположим нормальное распределение случайных величин θ_j, то к функции потерь добавится член $\sum_{j=0}^{m} \lambda_j(\theta_j - \mu_j)^2$ (см. вторую лекцию). Эта функция уже строго выпукла, так что минимум становится единственным.

Многоклассовая классификация

Мы обсудили бинарную классификацию. Рассмотрим ситуацию, когда классов больше: $C_1, C_2, \ldots, C_l$.

Метод логистической регрессии обобщается на многоклассовый случай способом, который называют многопеременной логистической регрессией (softmax regression). Вспомним, как мы с помощью формулы Байеса оценивали вероятности принадлежности классу и применим эту формулу для l классов:

$$p(C_k|x) = \frac{p(x|C_k)p(C_k)}{\sum_{r=1}^{l} p(x|C_r)p(C_r)} = \frac{e^{s_k}}{\sum_{r=1}^{l} e^{s_r}}, \qquad a_r = \ln p(x|C_r)p(C_r).$$

Идея проста: пусть модель создаст векторы $\theta^k = (\theta_j^k)_{j=1}^m$ для каждого класса C_k. Затем мы вычислим баллы каждого класса как обычно: $\hat{y} = s_k(\vec{x}) = \sum_{j=1}^{m} \theta_j^k x_j$. Теперь вычислим вероятности $y_k(\vec{x})$ того, что образец $\vec{x}$ относится ко классу C_k по формуле выше. Ну а затем выберем наибольшую вероятность и сделаем прогноз.

Остается вывести формулы для функции потерь (мы будем ее минимизировать при обучении softmax regression). Для двух классов мы сделали это двумя способами (они привели к одинаковому ответу). Здесь ограничимся одним — методом максимизации правдоподобия. Наши рассуждения о монетке теперь надо заменить на рассуждения о кубике с l гранями: модель вычисляет вероятности $p(C_k|\vec{x}_i)$, а затем подбрасывается кубик. Для удобства обозначений закодируем метки y_i не в виде чисел от 1 до l, а в виде векторов длины k. В этих векторах все координаты равны нулю, кроме k–ой, которая равна 1 (это означает, что $y_i = k$). Координаты векторов обозначим y_{ir}, $1 \le r \le l$. Тогда функция правдоподобия имеет вид

$$L = \prod_{i=1}^{n}\prod_{r=1}^{l} p(C_r|\vec{x}_i)^{y_{ir}} = \prod_{i=1}^{n}\prod_{r=1}^{l} (\hat{y}_r(\vec{x}_i))^{y_{ir}}, \quad -\ln L = -\sum_{i=1}^{n}\sum_{r=1}^{l} y_{ir} \ln \hat{y}_r(\vec{x}_i) \to min.$$

4 Предобработка данных: кластеризация (начало)

Часто при первичном анализе данных мы сталкиваемся с тем, что данные существенно неоднородны. Например, собрав данные о светимости, размерах, элементах орбиты и т.п. объектов ближнего космоса, мы можем заметить, что встречаются объекты с диаметром порядка сантиметра, а есть и объекты диаметром в десятки метров. Логично перед тем как начинать строить какую–либо модель разделить данные на группы. Возможно, для разных групп мы будем строить разные модели. Задача такого разбиения и является задачей кластеризации.

Исходными данными для кластеризации являются образцы x_i (пометок y_i нет — иначе мы говорим о классификации). Для того, чтобы проводить кластеризацию надо предварительно ввести метрику в пространстве $\mathbb{R}^m$ (вспомним, что каждый наш образец является вектором с m координатами). Например, логично использовать весовое евклидово расстояние $d(x,y) = \left(\sum_{s=1}^m w_s^2 |x_s - y_s|^2\right)^{1/2}$, где веса w_s зависят от размерностей признаков X_s и важности каждого из этих признаков. Другими популярными расстояниями являются ℓ_1 метрика $d(x,y) = \sum_{s=1}^m w_s|x_s - y_s|$ и равномерная метрика $d(x,y) = \max_s (w_s|x_s - y_s|)$. Итак, будем далее считать, что метрика d зафиксирована. Результатом разбиения будут являться пометки для каждого образца x_i вида $x_i \in S_k$, где S_k — k–ый кластер, т.е. метки, которые мы обозначим $a_i \in \{1, \ldots, k\}$.

На практике, помимо выбора расстояния, важно определиться с предполагаемым видом кластеров — требуем ли мы от кластеров шаровой формы (примерно, разумеется) или допускаем ленточные кластеры. Ответить на этот вопрос можно только исходя из смысла задачи. Возможно, следует построить несколько графиков — одномерных, двумерных или трехмерных проекций данных на те или иные координаты. Результатом ответа будет выбор метода кластеризации (см. ниже).

Качество кластеризации

Прежде, чем браться за задачу, хорошо бы сформулировать метрики качества, чтобы понимать, хорошо или нет задача решена. Проблема в том, что однозначной метрики выбрать в задаче кластеризации не получится. Действительно, например, что лучше — много маленьких шаровых кластеров или один большой ленточный? Ответ зависит от того, что мы дальше с этими кластерами собираемся делать. Тем не менее, определенные советы сформулировать можно.

Если число кластеров K заранее задано, то лучше всего ориентироваться на матрицу средних межкластерных расстояний:

$$dist(S_1, S_2) = \frac{\sum_{x_i \in S_1} \sum_{x_j \in S_2} d(x_i, x_j)}{\#S_1 \cdot \#S_2}.$$

Та же формула подходит и для среднего внутрикластерного расстояния (когда $S_2 = S_1$). Кластеризация считается хорошей, если внутрикластерные расстояния малы (т.е. кластеры достаточно кучные), а межкластерные — наоборот, велики. Таким образом, логично определить две метрики качества — общее среднее внутрикластерное и общее

среднее межкластерное расстояния

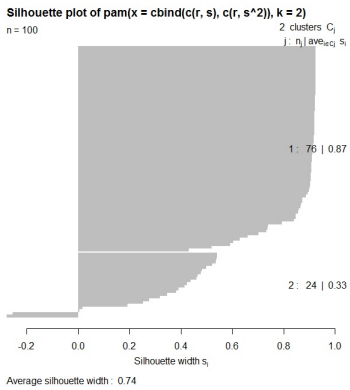
$$F_0 = \frac{\sum_{i=1}^n \sum_{j=1}^n d(x_i, x_j) I_{a_i=a_j}}{\sum_{i=1}^n \sum_{j=1}^n I_{a_i=a_j}} \longrightarrow min, \qquad F_1 = \frac{\sum_{i=1}^n \sum_{j=1}^n d(x_i, x_j) I_{a_i \neq a_j}}{\sum_{i=1}^n \sum_{j=1}^n I_{a_i \neq a_j}} \longrightarrow max.$$

Если $d(\cdot, \cdot)$ — евклидова (возможно, весовая), то в F_0 вместо суммы логичнее использовать сумму квадратов $\sum_{i,j} \|x_i - x_j\|^2$, которые стандартным приемом (прибавить, вычесть среднее, раскрыть скобки) приводятся к виду $\sum_i \|x_i - \mu\|^2$. Получаем метрику

$$WSS = \sum_{k=1}^K V(S_k) \longrightarrow min, \quad V(S_k) = \sum_{x_i \in S_k} \|x_i - \mu_k\|^2, \quad \mu_k = \frac{1}{\#S_k} \sum_{x_i \in S_k} x_i.$$

Аналогично, вместо F_1 разумно взять $\quad BSS = \sum_{k=1}^K \|\mu_k - \mu\|^2 \longrightarrow max, \quad \mu = \frac{1}{n} \sum_{i=1}^n x_i.$

Мы привели лишь несколько метрик качества (в литературе желающие найдут еще много аналогов). Если число кластеров K не определено, то лучше всего ориентироваться на коэффициент силуэта. Этот коэффициент лежит в диапазоне $[-1, 1]$ (сразу скажем — чем больше, тем лучше качество) и вычисляется для каждого элемента $x_i \in S_k$ каждого кластера. Результаты изображают для каждого кластера отдельно (получается силуэт кластера). А именно, их наносят в виде горизонтальных линий по убыванию коэффициентов (см. пример силуэтов двух кластеров ниже).



Результат, действительно, нагляден. Например, здесь видно, что первый кластер выделен отлично, а второй весьма проблемно. Алгоритм вычисления коэффициента $s(x_i)$ следующий. Вначале находим среднее расстояние $a(x_i)$ от x_i до остальных элементов кластера, в который попал x_i. Затем для каждого «чужого» кластера находим среднее расстояние от x_i до элементов этого кластера, а потом выбираем минимальное $b(x_i)$. Логично ожидать, что $b(x_i) > a(x_i)$ (для любого «чужого» кластера среднее расстояние больше, чем до «своего» кластера). В таком случае берем $s(x_i) = 1 - \frac{a(x_i)}{b(x_i)} \in [0, 1]$. Если же нашелся чужой кластер, который ближе своего, т.е. $b(x_i) \leq a(x_i)$, то берем $s(x_i) = -1 + \frac{b(x_i)}{a(x_i)} \in [-1, 0]$. Для одноэлементного кластера считаем $s = 0$. Коэффициенты, близкие к 1 означают, что элементы классифицированы верно. Близость к нулю означает неопределенность, а отрицательные коэффициенты показывают, что данные элементы классифицированы, вероятно, неверно. На графике силуэтов выводится также среднее значение коэффициентов по каждому кластеру.

Метод k–средних (k–means)

Пусть число кластеров K задано. Пусть зафиксирована весовая евклидова метрика d. Для каждого кластера тогда можно вычислить его центр μ_j, а затем и вариацию по каждому кластеру $V(S_k)$ и суммарную вариацию $WSS = \sum_{j=1}^k V_j$ Теперь будем искать такое разбиение на кластеры, чтобы величина WSS оказалась минимальной. Понятно,

что полный перебор имеет экспоненциальную сложность $O(n^k)$, где n — число образцов.
Алгоритм Lloyd.
1. Зафиксируем начальные точки μ_k (их можно указать вручную или выбрать случайным образом).
2. Для каждого образца найдем ближайшую точку μ_k и отнесем этот образец в кластер S_k.
3. Пересчитаем центры кластеров. Вернемся к шагу 2.
4. Цикл закончим, когда процесс стабилизируется или когда число шагов превысит заранее фиксированный предел.
Существует ускорение этого алгоритма — алгоритм Elkan. Ускорение основано на неравенстве треугольника $d(\mu_j, \mu_l) \le d(x_i, \mu_l) + d(x_i, \mu_j)$. Значит, если $d(x_i, \mu_k) < \frac{1}{2} d(\mu_k, \mu_l)$, то $d(x_i, \mu_l) \ge d(\mu_k, \mu_l) - d(x_i, \mu_k) > d(x_i, \mu_k)$ и считать расстояние $d(x_i, \mu_l)$ не нужно. Значит, на каждом шаге надо вначале вычислить все $d(\mu_k, \mu_l)$, а затем для каждого i посчитать расстояние от x_i до нового центра того кластера, в котором лежал x_i (скажем μ_k) и сравнить — скорее всего считать все остальные расстояния $d(x_i, \mu_l)$ не придется.
Другое ускорение — алгоритм MiniBatchKMeans: шаг 2 будем выполнять не для всех образцов, а только для небольшой случайной выборки; а остальные образцы будем оставлять в том кластеры, где они были. В любом случае, поскольку мы минимизируем суммарную дисперсию, то на выходе будем получать шарообразные кластеры — это отличительная особенность метода k–means.
Модификация — алгоритм k–medoids. Точки μ_k будем указывать не произвольно, а выбирая их из числа образцов. На первом шаге фиксируем μ_k из числа образцов случайно или по выбору пользователя. Пересчет центров кластеров будем делать не с помощью средних, а из соображений медианы:

$$\mu_k = \arg\min_{x_i \in S_k} \sum_{x_j \in S_k} d(x_i, x_j).$$

Это, конечно, тормозит алгоритм (вместо $O(\#S_k)$ операций получаем $O((\#S_k)^2)$), но зато теперь расстояние можно брать неевклидовым. Существует ускорение — будем перебирать не все $x_i \in S_k$, а только случайную выборку (детали, кто хочет, найдет).

EM–алгоритм

Полное название: Expectation–Maximization algorithm. Надо сказать, что этот метод применяется не только для кластеризации, в прошлом семестре мы говорили о нем в задаче о поиске латентных переменных. Алгоритм основан на предположении, что наши образцы представляют собой выборку из случайного вектора, равного смеси нескольких нормальных распределений (есть и другие разновидности алгоритма, но сосредоточимся на этой версии), т.е. наша плотность $f(x) = \sum_{j=1}^k w_j f_j(x)$, где $f_j(x) = \frac{1}{(2\pi)^{m/2}|\Sigma_j|^{1/2}} \exp\{-\frac{1}{2}(x-\mu_j)^t \Sigma_j^{-1}(x-\mu_j)\}$ — плотности нормальных векторов $\mathcal{N}(\mu_j, \Sigma_j)$. Задача алгоритма — найти оптимальные параметры распределения (набор чисел $w_j > 0$, $\sum w_j = 1$, центры μ_j и ковариационные матрицы Σ_j), а потом для каждого образца x_i найти набор вероятностей p_{ij} того, что x_i взят из с.в. с плотностью f_j. Тогда каждый образец помещается в тот кластер S_j, для которого вероятность p_{ij}

наибольшая. Заметим, что метод k–means является частным случаем EM–алгоритма, когда фиксируются $w_j = \frac{1}{k}$ и $\Sigma_j = I$, а ищутся только центры μ_j.

Будем действовать максимизацией логарифма функции правдоподобия

$$\mathfrak{L}(X) = \ln \prod_{i=1}^{n} f(x_i) = \sum_{i=1}^{n} \ln f(x_i) = \sum_{i=1}^{n} \ln \left(\sum_{j=1}^{k} w_j f_j(x_i) \right) \longrightarrow max.$$

К сожалению, полученная задача максимизации сложна и решить ее непосредственно трудно. Потому мы и будем применять двухшаговый итеративный метод (собственно, EM–метод). Обозначим условные вероятности

$$p_{ij} = \mathbb{P}(\text{точка взята из распределения } f_j | \text{это точка } x_i).$$

Используем выпуклость функции $\ln x$ вверх (точка на графике $y = \ln(\sum a_j x_j)$ лежит выше точки на хорде $\sum a_j \ln(x_j)$, где $a_j \geq 0$ и $\sum a_j = 1$):

$$\mathfrak{L}(X) = \sum_{i=1}^{n} \ln \left(\sum_{j=1}^{k} p_{ij} \cdot \frac{w_j}{p_i j} f_j(x_i) \right) \geq \sum_{i=1}^{n} \sum_{j=1}^{k} p_{ij} \ln \frac{w_j f_j(x_i)}{p_{ij}}.$$

Поставим цель E–шага: приблизить полученную нижнюю границу к значению $\mathfrak{L}(X)$. Неравенство $\ln(\sum a_j x_j) \geq \sum a_j \ln x_j$ обращается в равенство в точности тогда, когда все точки x_j равны между собой. Учитывая, что $\sum a_j = 1$, тогда они все должны равняться числу $f(x_i)$. Получаем

$$\frac{w_j}{p_{ij}} f_j(x_i) = f(x_i) \quad \Longleftrightarrow \quad p_{ij} = \frac{w_j f_j(x_i)}{f(x_i)}.$$

Итак, если параметры распределения w_j, μ_j, Σ_j известны, то на E–шаге мы можем найти нужные нам для ответа вероятности. Поставим цель M–шага: определить параметры w_j, μ_j и Σ_j оптимальным образом, т.е. максимизируя оценку снизу для $\mathfrak{L}(X)$ — величину $\sum_{i=1}^{n} \sum_{j=1}^{k} p_{ij} \ln \frac{w_j f_j(x_i)}{p_{ij}}$. Числа p_{ij} на этом шаге мы считаем известными, так что максимизировать остается

$$\sum_{i=1}^{n} \sum_{j=1}^{k} p_{ij} \ln w_j f_j(x_i) = \sum_{i} \sum_{j} p_{ij} \left(\ln w_j - \frac{1}{2} \ln |\Sigma_j| - \frac{1}{2} (x_i - \mu_j)^t \Sigma_j^{-1} (x_i - \mu_j) \right).$$

Видим, что максимизацию надо проводить независимо по w_j (при дополнительном ограничении $\sum w_j = 1$), по μ_j и по Σ_j. Первую задачу максимизации решаем методом множителей Лагранжа: $\sum_i \sum_j p_{ij} \ln w_j - \lambda(\sum w_j - 1) \to max$. Проверьте сами — получаем $w_j = \frac{1}{n} \sum_i p_{ij}$ (логично, получилась доля j–го кластера в общем объеме данных с учетом вероятностей p_{ij}). Проводя дифференцирование по μ_j, получим (при этом пользуемся невырожденностью матрицы Σ_j^{-1}) $\mu_j = \frac{\sum_{i=1}^{n} p_{ij} x_i}{\sum_{i=1}^{n} p_{ij}}$ (вновь логично — получилось взвешенное среднее с весами по j–му кластеру). Самое сложное технически — дифференцирование по элементам матрицы Σ_j. Опуская вычисления, получим матрицу взвешенных ковариаций с весами по j–му кластеру

$$\sigma_{j\alpha\beta} = \frac{\sum_{i=1}^{n} x_{i\alpha} x_{i\beta} p_{ij}}{\sum_{i=1}^{n} p_{ij}} - \mu_{j\alpha} \mu_{j\beta}.$$

Сам алгоритм теперь состоит в чередовании E– и M–шагов. Вначале инициализируем параметры (например, $w_j = \frac{1}{k}$, $\Sigma_j = I$, а μ_j задаются пользователем). Проводим E–шаг — находим p_{ij}. Затем проводим M–шаг и обновляем согласно полученным формулам w_j, Σ_j и μ_j. Затем снова на E–шаге находим p_{ij}, и так далее. Останавливаемся, когда рост функции $\mathfrak{L}$ замедляется (можно доказать, что на каждой итерации она не убывает) или когда число итераций превышает заранее заданный порог.

Иерархические методы

В предыдущих методах число кластеров k было задано. Поговорим о методах кластеризации, где k определяется в процессе. Иерархическим называется метод, который строит иерархию объединений/расщеплений (фактически, строится иерархическое дерево — дендрограмма). Такие методы делятся на разделяющие и объединяющие — в зависимости от способа построения дерева, от корня к листьям или наоборот. Поскольку разделяющие методы сейчас почти не применяются, сосредоточимся на объединяющих.

На нулевом шаге каждый образец считается отдельным кластером. Далее на каждом шаге мы решаем, какие кластеры следует объединить в один. Это решение принимается на основании близости кластеров. Например, можно ограничиться объединением на каждом шаге ровно двух кластеров — тех, расстояние между которыми самое маленькое из всех попарных расстояний. Тогда вопрос сводится только к выбору понятия расстояния между множествами. Перечислим самые частые из них.

1. Метод ближнего соседа. Расстояние между множествами есть расстояние между самыми близкими их точками: $d(S_1, S_2) = \min_{x_i \in S_1,\, x_j \in S_2} d(x_i, x_j)$. На практике кластеры получаются ленточными.

2. Метод дальнего соседа. Расстояние между множествами есть расстояние между самыми дальними их точками: $d(S_1, S_2) = \max_{x_i \in S_1,\, x_j \in S_2} d(x_i, x_j)$. Кластеры получаются небольшими, шарообразными и часто плохо отделимыми друг от друга «на глаз».

3. Метод средней связи. Расстояние между множествами есть среднее расстояние между их точками: $d(S_1, S_2) = \frac{1}{\#S_1 \cdot \#S_2} \sum_{x_i \in S_1,\, x_j \in S_2} d(x_i, x_j)$. Кластеру получаются округлыми и могут оказаться сильно разными по числу точек.

4. Метод центроидов. Расстояние между множествами есть расстояние между центрами этих множеств: $d(S_1, S_2) = d(\mu_1, \mu_2)$, где $\mu_k = \frac{1}{\#S_k} \sum_{x_i \in S_k} x_i$.

5. Метод Уорда. Расстояние между множествами есть $d(S_1, S_2) = V(S_1 \cup S_2) - V(S_1) - V(S_2)$, где $V(S) = \sum_{x_i \in S} \|x_i - \mu\|^2$ (вариация), $\mu = \frac{1}{\#S} \sum_{x_i \in S} x_i$ (центр множества). Прекрасной новостью является то, что это выражение можно упростить к гораздо более быстро считаемому виду: $d(S_1, S_2) = \frac{1}{1/\#S_1 + 1/\#S_2} \|\mu_1 - \mu_2\|^2$ (упражнение: докажите это сами).

Очень важным подспорьем для ускорения вычислений является формула Ланса–Вильямса, которая позволяет вычислять расстояние от нового кластера, полученного

в результате объединения двух, до любого третьего кластера с помощью расстояний предыдущего шага:

$$d(S_1 \cup S_2, S_3) = \alpha_1 d_{13} + \alpha_2 d_{23} + \beta d_{12} + \gamma |d_{13} - d_{23}|, \qquad d_{ij} = d(S_i, S_j).$$

Формула работает для всех пяти методов при подходящем подборе α_i, β и γ.

Ближний сосед	Дальний сосед	Средняя связь	Центроиды	Уорд
$\alpha_1 = \alpha_2 = 0.5$	$\alpha_1 = \alpha_2 = 0.5$	$\alpha_i = \frac{\#S_i}{\#S_1 + \#S_2}$	$\alpha_i = \frac{\#S_i}{\#S_1 + \#S_2}$	$\alpha_i = \frac{\#S_i + \#S_3}{\#S_1 + \#S_2 + \#S_3}$
$\beta = 0$, $\gamma = -0.5$	$\beta = 0$, $\gamma = -0.5$			
Более просто:	Более просто:	$\beta = \gamma = 0$	$\beta = -\frac{\#S_1 \cdot \#S_2}{(\#S_1 + \#S_2)^2}$	$\beta = -\frac{\#S_3}{\#S_1 + \#S_2 + \#S_3}$
$\min(d_{13}, d_{23})$	$\max(d_{13}, d_{23})$		$\gamma = 0$	$\gamma = 0$

DBScan и OPTICS

Полное название алгоритма Density–based spatial clustering of applications with noise. Число кластеров алгоритм выбирает сам, зато у него есть два других гиперпараметра — число $\varepsilon > 0$ (радиус окрестности) и N — число соседей. Алгоритм является развитием старой идеи о кластеризации с помощью графа расстояний между образцами. Действительно, назовем два образца соседними, если расстояние между ними меньше ε. Соседние образцы соединим ребром (мысленно). Полученный граф распадется на связные компоненты — их и назовем кластерами. Это и есть основа для алгоритма DBScan, но есть еще важные нюансы.

Все образцы разделим на основные точки (они же, корневые, они же внутренние), граничные точки и выбросы (они же, шум). Точка называется основной, если в ее ε–окрестности найдется, как минимум, N других образцов (соседей). Точка называется граничной, если в ее ε–окрестности меньше N соседей, но есть хотя бы одна основная точка. Все остальные точки считаются выбросами, удаляются из числа образцов и вообще никак не кластеризуются. Все основные точки объединим в граф (ребро графа проводится между точками с расстоянием $< \varepsilon$). Теперь добавим в этот граф граничные точки в виде вершин степени 1 — соединим каждую граничную точку ровно с одной корневой — ближайшей (есть другая разновидность — берут первую найденную корневую из тех, что на расстоянии $< \varepsilon$). Полученный граф распадется на компоненты связности — они и будут нашими кластерами.

Выбор параметров ε и N весьма важен для корректной работы алгоритма. На практике, число N выбирают от 3 до 9, причем чем больше шума, тем больше N. Число ε советуют брать равным или близким к среднему расстоянию до N ближайших соседей: $\varepsilon \approx \frac{1}{n} \sum_{i=1}^{n} d_i$, где $d_i = \frac{1}{N} \sum_{x_j \in N(x_i)} d(x_i, x_j)$, $N(x_i)$ — множество N ближайших соседей к x_i. Если плотность данных различна в разных точках пространства, то лучше пользоваться модификацией DBScan (алгоритм Optics). Здесь число ε выбирается переменным для разных образцов в зависимости от чисел d_i.

OPTICS — Ordering Points To Identify the Clustering Structure. Обязательным параметром является только число соседей N. Параметр ε может быть задан, а может быть пропущен — тогда он выставляется равным $+\infty$. Рекомендуется, все–таки выбирать ε конечным — это ускоряет работу. Результатом работы OPTICS является не разбиение на кластеры, а два массива — порядок достижимости (order list) и значения достижимостей

(reachability), определения см. ниже. Кластеры извлекаются постобработкой этих массивов. Опишем работу алгоритма OPTICS по шагам.

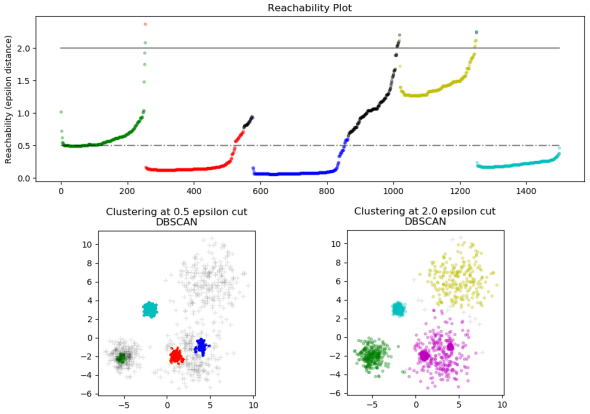
1. Инициализация. Каждой точке x_i мы припишем три числа: достижимость $r(x_i)$, базовое расстояние (core distance) $CD(x_i)$ и флаг в формате 0/1 — обработана точка или нет. Иниицализируем наши кортежи так: $(x_i, +\infty, +\infty, 0)$. Создаем список упорядочивания точек — order list (он вначале пуст, сюда будут переносится все обработанные точки). Создаем структуру данных «очередь» (будем называть ее очередью приоритетов), вначале очередь пуста.

2. Основной цикл. Выбираем очередную точку x_i. Если точка уже обработана, то пропускаем ее и переходим к следующей точке. Если x_i не обработана, то начинаем ее обрабатывать (и помечаем обработанной, добавляем ее в order list).

2а. Находим всех соседей x_i, находящихся на расстоянии $\leq \varepsilon$. Если соседей меньше N, то считаем точку x_i граничной или шумовой (это выяснится при постобработке) и определяем $CD(x_i) = +\infty$, переходим к следующей точке (т.е. возврат к п.2). Если соседей N или больше, то точка считается корневой — $CD(x_i)$ берем равным расстоянию до N–го ближайшего соседа, переходим к шагу 2б.

2б. Для каждой точки x_j — соседа x_i с флагом «не обработана» пересчитываем достижимость $r(x_j)$ (обработанные точки x_j не пересчитываем). Если достижимость $r(x_j) = +\infty$, то новое значение достижимости $r(x_j) := \varepsilon' = \max\{CD(x_i), d(x_i, x_j)\}$. Точку x_j при этом, добавляем в очередь приоритетов. Если же достижимость $r(x_j)$ конечна, то сравниваем $r(x_j)$ и ε': если $r(x_j) \leq \varepsilon'$, то не меняем $r(x_j)$, а иначе берем $r(x_j) := \varepsilon'$.

2в. Обрабатываем приоритетную очередь, пока она не опустеет. Выбираем из очереди точку x_l с наименьшей достижимостью, помечаем ее обработанной, добавляем в order list. Находим всех соседей x_l в шаре радиуса ε. Вычисляем $CD(x_l)$ так же, как в шаге 2а вычисляли $CD(x_i)$. Если x_l оказалась корневой точкой, то пересчитываем достижимости всех ее необработанных соседей, добавляя их в очередь приоритетов (как в пункте 2б). Переходим к следующей точке из очереди с наименьшей достижимостью.



Получив график достижимостей (по горизонтальной оси — номер точки в order list, по вертикальной — достижимости), мы должны постобработать его. Самая простая идея — выбрать отсечку ε и включить в кластеры точки с достижимостью $< \varepsilon$. Это, однако, фактически вернет нас к результатам DBScan (разве что, параметр ε будет грамотно выбран). Правильнее будет найти на графике долины (указывают на кластеры) и пики (границы между кластерами). Это можно сделать вручную или автоматически.

Спектральная кластеризация

Мы уже упоминали граф связей между образцами. Составим полный граф на n точках. Теперь припишем каждому его ребру вес w_{ij} — чем ближе точки x_i, x_j, тем больший вес имеет ребро (связь между точками прочная). Самый простой вариант:

$w_{ij} = \begin{cases} d(x_i, x_j), & \text{если } d(x_i, x_j) \le \varepsilon, \\ 0, & \text{иначе.} \end{cases}$ Часто используются веса $w_{ij} = \exp\{-\gamma \cdot d^2(x_i, x_j)\}$, полученные с помощью ядра Гаусса (оно же RBF–kernel). Есть, конечно, и другие варианты определения весов — для нашего метода нужно только $w_{ij} \ge 0$ и $w_{ij} = w_{ji}$.

Число кластеров K в нашем новом методе будет заранее задано. Наша идея — разорвать граф на компоненты, причем за разрыв каждого ребра берется штраф w_{ij}. Так вот, разрыв надо провести с наименьшим суммарным относительным штрафом:

$$N = \sum_{k=1}^{K} \frac{\sum_{x_i \in S_k} \sum_{x_j \notin S_k} w_{ij}}{\sum_{x_i \in S_k} \sum_{j=1}^{n} w_{ij}}.$$

Проведем сейчас рассуждения для $K = 2$ (разбиение графа на 2 части), а потом вывод обобщим на произвольное K уже без доказательства. Обозначим $d_i = \sum_{j=1}^{n} w_{ij}$, $1 \le i \le n$, и сформируем диагональную матрицу $D = diag(d_i)$. Обозначим матрицы $W = (w_{ij})$ и $L = D - W$, обозначим наши кластеры S и $\overline{S}$, обозначим суммы $\sum_{x_i \in S} d_i = s$, $\sum_{x_i \in \overline{S}} d_i = \widetilde{s}$. Сумма $s + \widetilde{s} = \sum_{i=1}^{n} d_i$ не зависит от разбиения на кластеры, так что вместо штрафа N можно минимизировать

$$N \cdot (s + \widetilde{s}) = \sum_{x_i \in S} \sum_{x_j \in \overline{S}} w_{ij} \frac{s + \widetilde{s}}{s} + \sum_{x_i \in \overline{S}} \sum_{x_j \in S} w_{ij} \frac{s + \widetilde{s}}{\widetilde{s}} = \sum_{x_i \in S,\, x_j \in \overline{S}} w_{ij} \left(2 + \frac{s}{\widetilde{s}} + \frac{\widetilde{s}}{s} \right) = \sum_{x_i \in S,\, x_j \in \overline{S}} w_{ij} \left(\frac{\sqrt{\widetilde{s}}}{\sqrt{s}} + \frac{\sqrt{s}}{\sqrt{\widetilde{s}}} \right)^2$$

Создадим вектор f с координатами $f_i = \begin{cases} \sqrt{\widetilde{s}}/\sqrt{s}, & \text{если } x_i \in S, \\ -\sqrt{s}/\sqrt{\widetilde{s}}, & \text{если } x_i \in \overline{S}. \end{cases}$ Тогда, продолжая равенство, получим $= \sum_{x_i \in S, x_j \in \overline{S}} w_{ij} (f_i - f_j)^2 = \sum_{i,j=1}^{n} w_{ij} (f_i - f_j)^2$, поскольку в случаях x_i, $x_j \in S$ и x_i, $x_j \in \overline{S}$ имеем $f_i - f_j = 0$. Теперь раскроем обратно скобки и заметим, что $\sum_{i,j} w_{ij} f_i^2 = \sum_i d_i f_i^2$ и аналогично для f_j, т.е. получим

$$N(s + \widetilde{s}) = 2 \sum_i d_i f_i^2 - 2 \sum_{i,j} w_{ij} f_i f_j = 2 f^t D f - 2 f^t W f = 2 f^t L f.$$

Итак, оказывается, мы ищем минимум квадратичной формы $\langle Lf, f \rangle$ симметричной матрицы. Тут, однако, надо заметить, что вектор f не пробегает всего пространства. В частности, если $e = (1, 1, \ldots, 1)$, то

$$\langle Df, e \rangle = \sum_i d_i f_i = \sum_{x_i \in S} d_i \frac{\sqrt{\widetilde{s}}}{\sqrt{s}} - \sum_{x_i \in \overline{S}} d_i \frac{\sqrt{s}}{\sqrt{\widetilde{s}}} = 0,$$

т.е. $Df \perp e$. Сделаем замену переменной $g = D^{1/2} f$, обозначим $A = D^{-1/2} L D^{-1/2}$. Тогда равенство $\langle Df, e \rangle = 0$ примет вид $\langle g, D^{1/2} e \rangle = 0$ — векторы g и $D^{1/2} e$ ортогональны, а задача минимизации примет вид $\langle Ag, g \rangle \to min$. Еще одно скрытое ограничение:

$$\|g\|^2 = \langle D^{1/2} f, D^{1/2} f \rangle = \langle Df, f \rangle = \sum_i d_i f_i^2 = \sum_{x_i \in S} d_i \frac{\widetilde{s}}{s} + \sum_{x_i \in \overline{S}} d_i \frac{s}{\widetilde{s}} = s + \widetilde{s} = \sum_i d_i.$$

Получаем задачу минимизации квадратичной формы симметрической матрицы с ограничением на норму вектора и на ортогональность вектору $D^{1/2}e$. Заметим еще, что вектор $D^{1/2}e$ является собственным: $AD^{1/2}e = D^{-1/2}Le$, $(Le)_i = d_i - \sum_j w_{ij} = 0$ с собственным значением 0. Поскольку штраф по определению не отрицательный, то остальные с.з. больше нуля. Вспомним теорему Гильберта–Шмидта из курса функционального анализа: в некотором базисе матрица A диагональна, а значит решение нашей задачи минимизации — это с.в. для второго (наименьшего положительного) с.з.

Вывод: надо найти второе с.з. матрицы A и соответствующий с.в., нормировать его — получим решение нашей экстремальной задачи. Есть однако подводный камень: искомый нами вектор f имел весьма специфические координаты (положительные и все равные между собой по кластеру S и отрицательные и тоже все равные между собой по кластеру $\overline{S}$). Значит, надо найденный с.в. изменить — сравнять все его положительные координаты и точно так же отрицательные. Но это и дает рецепт кластеризации: находим с.в. для второго с.з.; те x_i, для которых координаты с.в. больше нуля, помещаем в кластер S, а остальные — в кластер $\overline{S}$.

Теперь, как и собирались, перенесем метод (без доказательства) на большее число кластеров.
1. Находим с.в. для второго, третьего, ... , $K-1$ с.з. (обозначим их $f^1, \ldots, f^K$), нормируем их на единицу.
2. Составляем новые образцы $y_i = (f_i^1, \ldots, f_i^K)$ (получим n образцов в пространстве $\mathbb{R}^K$).
3. Проводим кластеризацию новых образцов (например методом средних или еще как–то).
4. От этой кластеризации берем только номера образцов по каждому классу — получаем ответ.

Плюс нашего метода — огромное понижение размерности: с m до K. Еще одно его достоинство — он срабатывает там, где визуально кластеры не видны. Вспомните, например, что амплитуда звукового сигнала с разбивкой по частотам ничего не дает для понимания (разве что, что звук есть), а спектрограмма звука сразу позволяет определить тембр.

5 Предобработка данных: понижение размерности

Метод главных компонент

Попробуем создать автоматический алгоритм, который отсечет «ненужные» факторы X_j. Самая простая идея связана исключительно с работой с данными. Прежде всего давайте центрируем наши данные: вычислим вектор $\vec{a} = \frac{1}{n}\sum_{i=1}^n \vec{x}_i$ и вычтем его из каждого вектора $\vec{x}_i$. Теперь наши данные центрированы на начало координат. Давайте найдем такое линейное подпространство L размерности K в пространстве $\mathbb{R}^m$ и такие точки $\vec{w}_i \in L$, которые наилучшим образом приближают векторы $\vec{x}_i$. Таким образом мы понизим размерность наших данных с m до K (останется только выбрать базис в L и записать точки $\vec{w}_i$ по этому базису векторами с K координатами). Наилучшее приближение будем понимать в смысле $\sum_{i=1}^n \|\vec{x}_i - \vec{w}_i\|^2 \to min$, где норма евклидова.

Можно, конечно, поспорить, почему мы берем именно сумму квадратов, а не сумму

норм или максимум норм по всем i. Приведем два довода. Если подпространство L уже выбрано, то у нас есть понятие наилучшего приближения вектора $\vec{x}_i$ по подпространству L — это вектор $\vec{w}_i = \arg\min_{\vec{w}\in L} \|\vec{x}_i - \vec{w}\|$, который (как известно из курса функционального анализа) является ортопроекцией $\vec{x}_i$ на L. Но тогда

$$\sum_{i=1}^{n} \|\vec{x}_i - \vec{w}_i\|^2 = \sum_{i=1}^{n} \langle \vec{x}_i - \vec{w}_i, \vec{x}_i - \vec{w}_i \rangle = \sum_{i=1}^{n} \langle \vec{x}_i - \vec{w}_i, \vec{x}_i + \vec{w}_i \rangle = \sum_{i=1}^{n} \|\vec{x}_i\|^2 - \sum_{i=1}^{n} \|\vec{w}_i\|^2.$$

Получается, что минимизация левой части равносильна максимизации суммы $\sum_{i=1}^{n} \|\vec{w}_i\|^2$. Учитывая, что наши данные центрированы, эта сумма с точностью до множителя n равна выборочной дисперсии выборки $\{\vec{w}_i\}_1^n$. Логично, что при потере информации (а проектирование, разумеется ведет к потере) мы хотим сохранить как можно больше информации о разбросе выборки, т.е. о ее вариации. Это был первый довод, а второй будет после того, как мы решим поставленную задачу.

Итак, решение. Выберем в подпространстве L ортонормированный базис $\{\vec{e}_k\}_{k=1}^K$. Разложение вектора $\vec{w}_i$ по этому базису имеет вид $\vec{w}_i = \sum_{k=1}^K \langle \vec{w}_i, \vec{e}_k \rangle \cdot \vec{e}_k$ (ряд Фурье). Из равенства Парсеваля $\|\vec{w}_i\|^2 = \sum_{k=1}^K |\langle \vec{w}_i, \vec{e}_k \rangle|^2$. Поскольку $\vec{x}_i - \vec{w}_i \perp \vec{e}_k$, то $\langle \vec{w}_i, \vec{e}_k \rangle = \langle \vec{x}_i, \vec{e}_k \rangle$. Получаем задачу

$$\sum_{i=1}^{n} \sum_{k=1}^{K} \langle \vec{x}_i, \vec{e}_k \rangle^2 \to max, \quad \|\vec{e}_k\| = 1, \quad \vec{e}_j \perp \vec{e}_k.$$

Вспомним, что в нашей матрице данных X векторы $\vec{x}_i$ есть строки с номером i. Тогда вектор $X\vec{e}_k$ имеет как раз координаты $\{\langle \vec{x}_i, \vec{e}_k \rangle\}_{i=1}^n$ (вектор $\vec{e}_k$ мы считаем вектором–столбцом, как обычно). Получаем переформулировку

$$\sum_{k=1}^{K} \|X\vec{e}_k\|^2 = \sum_{k=1}^{K} \langle X\vec{e}_k, X\vec{e}_k \rangle \to max.$$

Поиск подпространства L равносилен поиску векторов $\vec{e}_k$. Введем функцию Лагранжа

$$L = \sum_{k=1}^{K} \langle X\vec{e}_k, X\vec{e}_k \rangle - \sum_{k=1}^{K} \lambda_k^2 (\langle \vec{e}_k, \vec{e}_k \rangle - 1)$$

(мы учли ограничения $\|\vec{e}_k\|^2 = 1$, а квадраты у множителей Лагранжа поставлены из соображений, которые станут ясны ниже) и приравняем к нулю ее частные производные по векторам $\vec{e}_k$ (точнее вариации, поскольку производная берется по векторной величине, а не по скалярной). Имеем

$$L(\vec{e}_k + \vec{h}) - L(\vec{e}_k) = 2\langle X\vec{e}_k, X\vec{h} \rangle - 2\lambda_k^2 \langle \vec{e}_k, \vec{h} \rangle + O(\|\vec{h}\|^2).$$

Ортогональность произвольному приращению $\vec{h}$ дает равенство $X^*X\vec{e}_k = \lambda_k^2 \vec{e}_k$. Оказывается, что все векторы $\vec{e}_k$ должны быть собственными векторами матрицы X^*X. Подставляя эти равенства в целевую функцию, получим

$$\sum_{k=1}^{K} \lambda_k^2 \to max, \quad \vec{e}_j \perp \vec{e}_k.$$

Оператор X^*X симметричен: $(X^*X)^* = X^*X^{**} = X^*X$ и положителен: $\langle X^*X\vec{x},\vec{x}\rangle = \langle X\vec{x}, X\vec{x}\rangle > 0$. Тогда его собственные значения положительны и их можно занумеровать по убыванию. Повторять собственные значения в сумме (при отсутствии кратности) мы не можем (это запрещает условие $\vec{e}_j \perp \vec{e}_k$). Получается, что максимум достигается на наборе собственных векторов $\{\vec{e}_k\}_{k=1}^K$, отвечающим K наибольшим собственным значениям оператора X^*X (с учетом кратности). Это и есть ответ. Полученный метод называется методом главных компонент (principal component analysis = PCA).

Резюмируем, что нужно сделать с точки зрения вычислений. Находим SVD–разложение $X = V\Lambda U^*$. Здесь $\Lambda = \mathrm{diag}\{\lambda_k\}_1^m$, где $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_m \geq 0$ (без квадратов!), U — унитарная матрица размера $m \times m$, столбцы которой состоят из нормированных собственных векторов e_k матрицы X^*X, а V — тоже унитарная матрица (размера $n \times m$, ее столбцы будут нормированными собственными векторами матрицы XX^*). Теперь обрезаем все три матрицы — от U и V оставляем K первых столбцов, а от центральной матрицы — верхний левый $K \times K$ угол. Получаем произведение $W = V_K\Lambda_K(U_K)^*$ — матрицу размера $n \times m$. Ее строки $\vec{w}_i$, $1 \leq i \leq n$, — и есть искомые проекции векторов $\vec{x}_i$. Векторы $\vec{w}_i$ имеют по m координат, поскольку по–прежнему лежат в $\mathbb{R}^m$, но на самом деле, они лежат в K–мерном подпространстве. Для дальнейших вычислений лучше использовать их K–координатные разложения по базису $\{e_k\}_1^K$ (иначе мы не достигаем нашей начальной цели — уменьшения числа признаков). Эти разложения получаются как строки $\vec{z}_i$, $1 \leq i \leq n$, матрицы $Z = V_K\Lambda_K$ размера $n \times k$. Столбцы этой матрицы — наши новые признаки Z_k называют латентными (ненаблюдаемыми) признаками.

Вероятностный PCA

Другой подход приводит к почти таким же выводам, но основан на других идеях. Предположим, что у нас есть K латентных переменных Z_k, которые независимы и нормально распределены в пространстве $\mathbb{R}^K$, т.е. $Z \sim \mathcal{N}(0, E)$. Однако, мы их не наблюдаем, а наблюдаем их «искажение» — набор переменных X, полученный из Z с помощью некоторого линейного оператора A, переводящего $\mathbb{R}^K$ в подпространство $L \subset \mathbb{R}^m$, а потом еще и зашумления этих образов в $\mathbb{R}^m$. Получим $\vec{w}_i = \vec{z}_iA$, $\vec{x}_i = \vec{w}_i + \varepsilon_i$. Будем считать, что шум тоже распределен нормально $\varepsilon \sim \mathcal{N}(0, \sigma^2E)$. Тогда (см. курс теорвера — отображения нормальных векторов) получим $\vec{x}_i \sim \mathcal{N}(0, A^*A+\sigma^2E)$, т.е. имеют плотность распределения

$$p(\vec{x}) = \frac{1}{(2\pi)^{n/2}\det\Sigma}\exp\left\{-\frac{1}{2}\vec{x}\cdot\Sigma^{-1}\cdot\vec{x}^T\right\}, \qquad \Sigma = A^*A + \sigma^2E$$

(вектор $\vec{x}$ — строка). Поставим вопрос — можем ли мы по наблюдениям $\{\vec{x}_i\}_1^n$ оценить матрицу A, чтобы потом вернуться к латентным переменным Z_k?

Воспользуемся методом наибольшего правдоподобия. Составим функцию

$$\ln L(A,\sigma) = \ln\prod_{i=1}^n p(\vec{x}_i) = -n\ln\det(A^*A+\sigma^2E) - \frac{1}{2}\sum_{i=1}^n \vec{x}_i(A^*A+\sigma^2E)^{-1}\vec{x}_i^T + const.$$

Дифференцируя по матрице A и приравнивая производную к нулю, получим (вычисления опущены)

$$\hat{A} = V(\Lambda^2 - \sigma^2 E)^{1/2},$$

где матрицы Λ и V находятся из сингулярного разложения нашей матрицы данных: $X = V\Lambda U^*$ (матрицы — те же, что и раньше). Дифференцируя по параметру σ и приравнивая к нулю производную, получим

$$\hat{\sigma} = \frac{1}{m-K}\sum_{K+1}^{m}\lambda_j^2, \qquad \Lambda^2 = \mathrm{diag}\{\lambda_j^2\}_1^m,$$

где числа λ_j^2 упорядочены по убыванию с учетом кратности. Поскольку λ_j^2 — это те самые собственные значения, а $e_j = e_j^0 U^*$ (попросту столбцы матрицы U^*) — те самые собственные векторы, которые мы получали в методе PCA, то видим, что вероятностный PCA совпадает с обычным при $\sigma \to 0$.

Выбор правильного числа измерений

Мы по–прежнему считаем, что наши данные X центрированы. Величину

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\|\vec{x}_i - \vec{w}_i\|^2 = \frac{1}{n}\sum_{i=1}^{n}\|\vec{x}_i\|^2 - \frac{1}{n}\|\vec{w}_i\|^2 = D(X) - D(W)$$

называют ошибкой проекции. В терминах собственных значений матрицы X^*X эта величина равна (мы это получили выше, когда подставили собственные векторы в целевую функцию) $\frac{1}{n}\sum_{k=K+1}^{m}\lambda_k^2$. Легко видеть, что эта величина строго убывает по K, так что разумного способа выбрать K не проглядывается — чем больше K, тем меньше ошибка.

Можно поступить просто: зафиксировать уровень (например 0.95) и выбрать K минимальным числом, для которого ошибка проекции не превосходит 0.05 по отношению к исходной дисперсии:

$$\sum_{k=1}^{K}\lambda_k^2 \geq 0.95\sum_{k=1}^{m}\lambda_k^2.$$

Есть другой способ. Предположим, что часть собственных значений λ_k^2 с номерами $1 \leq k \leq K$ случайно выбраны из $\mathcal{N}(\mu_1, \nu^2)$, а остальные λ_k^2 выбраны из $\mathcal{N}(\mu_2, \nu^2)$. Тогда легко оценить параметры

$$\mu_1 = \frac{1}{K}\sum_{k=1}^{K}\lambda_k^2, \qquad \mu_2 = \frac{1}{m-K}\sum_{k=K+1}^{m}\lambda_k^2, \qquad \nu^2 = \frac{\sum_{k=1}^{K}(\lambda_k^2-\mu_1)^2 + \sum_{k=K+1}^{m}(\lambda_k^2-\mu_2)^2}{m}.$$

Теперь составим функцию правдоподобия

$$\ln L(K) = \frac{m}{2}\ln(2\pi) - m\ln\nu(K) - \sum_{k=1}^{K}\frac{(\lambda_k^2-\mu_1)^2}{\nu^2} - \sum_{k=K+1}^{m}\frac{(\lambda_k^2-\mu_2)^2}{\nu^2} = -m\ln\nu(K) + const$$

и максимизируем ее (т.е. решим задачу $\nu(K) \to min$ по параметру K).

Для визуализации данных строят график в пространстве 2 главных компонент. При этом можно использовать так называемый biplot, позволяющий выудить больше информации, чем просто двумерный график. Biplot добавляет к картинке в пространстве главных компонент m векторов h_j, каждый из которых представляет собой столбец DV^T, усеченный до первых k координат. При этом скалярные произведения h_j и u_i* аппроксимируют скалярные произведения исходных векторов u_i и столбцов DV^T, совпадающие с $x_{i,j}$.

Тем самым, мы можем проецировать векторы u_i^* в нашем подпространстве на векторы, добавленные в biplot, можно восстанавливать координаты исходных $x_{i,j}$ и рассматривать их. Схему, как реализовать biplot в питоне, можно взять https://sukhbinder.wordpress.com/2015/08/05/biplot-with-python/.

Нелинейный PCA

Обычный метод PCA хорошо работает, если выполнены два условия: векторы $\{\vec{x}_i\}$ образуют один кластер; этот кластер сгруппирован возле некоторого линейного подпространства. Если кластеров несколько, то, скорее всего, точки разных кластеров будут группироваться возле разных подпространств. Вывод: если есть подозрение, что наши данные разбиваются на несколько кластеров, то перед применением PCA надо провести кластеризацию. Затем данные каждого кластера спроектировать (будут получены латентные переменные — свои для каждого кластера). Затем надо строить модели для каждого кластера отдельно. Фактически, такой метод предполагает конвейер из трех моделей. На верхнем уровне находится дерево решений, которое разделяет данные по кластерам. Затем применяется PCA. Затем данные подаются на основную модель.

Бывает, однако, ситуации, когда кластер данных один, но данные группируются возле нелинейных многообразий. В таком случае надо применять нелинейную версию PCA. По хорошему, мы должны ввести нелинейные признаки (например, степени X_j^2), расширить ими пространство признаков, а затем уже в расширенном пространстве провести линейный PCA. Давайте рассмотрим отображение φ из $\mathbb{R}^m$ в гильбертово пространство H (конечномерное или бесконечномерное). В этом пространстве мы получим набор данных $\Phi = (\phi(\vec{x}_i))_{i=1}^n = (\phi_i)_{i=1}^n$. Пространство H может иметь координатное представление (тогда ϕ_i — векторы–строки) или функциональное представление (тогда ϕ_i — функции). Теперь мы проводим обычный линейный метод PCA для данных Φ в пространстве H, т.е. для вычислений используем скалярное произведение этого пространства. Опуская промежуточные шаги, приведем финальный алгоритм, как это было сделано для обычного PCA.

Составим матрицу скалярных произведений $K = (k_{ij})_{i,\,j=1}^n$, $k_{ij} = \langle \phi_i, \phi_j \rangle_H = K(\vec{x}_i, \vec{x}_j)$. Здесь происходит, так называемый, ядерный трюк — оказывается, что нам не нужно строить пространство H и отображение ϕ, достаточно работать с новым «скалярным произведением» $K(\vec{x}, \vec{y})$ вместо $\vec{x} \cdot \vec{y}^T$. Применим SVD–разложение к матрице K — найдем диагональную матрицу Λ^2 из собственных значений и унитарную матрицу V из нормированных собственных векторов. Теперь проекция на нелинейное многообразие задается формулой $Pr(\vec{x}_*) = k_* V_K \Lambda_K^{-1}$, где вектор–строка $k_* = [K(\vec{x}_*, \vec{x}_1), \ldots, K(\vec{x}_*, \vec{x}_n)]$, V_K — матрица, составленная из K первых собственных векторов, Λ_K — левый верхний

угол матрицы Λ размера $K \times K$.

Есть одна тонкость — в наших рассуждениях речь шла о центрированных данных, а после применения отображения ϕ центрированность пропадает. Для того, чтобы ее восстановить, необходимо центрировать матрицу K, перейдя к матрице $\tilde{K} = PKP$, где $P = E - \frac{1}{n}\mathbf{1}$. Здесь E — обычная единичная матрица (размера $n \times n$), а $\mathbf{1}$ — матрица, все элементы которой равны 1.

6 Метод опорных векторов

Предложим еще один метод бинарной классификации. Предположим вначале, что наши данные линейно разделимы, т.е. существует такая гиперплоскость $\theta_0+\theta_1 X_1+\ldots+\theta_m X_m = 0$ (назовем ее π), что все векторы $\vec{x}_i$ с метками $y_i = 1$ попадают в гиперпространство $\theta^T\vec{x} > 0$, а векторы с метками $y_i = -1$ — в $\theta^T\vec{x} < 0$. Тогда разделяющих плоскостей много и нам хочется провести такую, для который ширина полосы вдоль π, свободной от точек $\vec{x}_i$, была бы наибольшей. Поворачивая плоскость наилучшим образом и расширяя полосу максимально мы, очевидно, наткнемся в конце концов на какие–то векторы $\vec{x}_i$, лежащие на границе полосы. Эти векторы называют **опорными**, а сам метод — метод опорных векторов.

Как мы помним из линейной алгебры, расстояние от точки до гиперплоскости равно $\rho(\vec{x}_i, \pi) = \frac{|\theta^T\vec{x}_i|}{\|\theta\|}$ (норма — евклидова). Раскрывая модуль как $y_i \cdot \theta^T\vec{x}_i$ (для меток $y_i = 1$ с плюсом, а для $y_i = -1$ с минусом), получим задачу

$$\min_{1\le i\le n} \frac{y_i \cdot \theta^T\vec{x}_i}{\|\theta\|} \longrightarrow max \quad \Longleftrightarrow \quad \begin{cases} \|\theta\|^2 = \theta^T\theta \to min, \\ y_i(\theta_0 + \theta_1 x_{i1} + \ldots + \theta_m x_{im}) \ge 1 \quad \forall i. \end{cases}$$

Мы воспользовались тем, что наша функция однородна по θ (не меняется от умножения θ на коэффициент), а значит можно сделать числитель равным единице. Дальнейшее понятно — решается задача минимизации квадратичной функции на линейном симплексе.

Проблема в том, что данные могут не быть линейно разделимыми (и это как раз обычно самый интересный случай). Тогда наша задача просто не будет иметь решения (область, заданная системой ограничений, будет пустой). Надо научиться «допускать ошибки». Ошибкой будет случай, когда выражение $1 - y_i(\theta_0 + \ldots + \theta_m x_{im})$ положительно. Тогда логично сделать мерой ошибки величину $\max\{0, 1-y_i(\theta_0+\ldots+\theta_m x_{im})\}$ (ее называют hinge loss — петлевая функция потерь), сложить их все и учесть в функции потерь с некоторым (неизвестным нам) коэффициентом, который характеризует величину штрафа за каждую ошибку. Получаем задачу

$$\theta^T\theta + C\sum_{i=1}^{n} \max\{0, 1-y_i(\theta_0+\ldots+\theta_m x_{im})\} \longrightarrow min \quad \Longleftrightarrow \quad \lambda\|\theta\|^2_{L_2} + \sum_{i=1}^{n} \max\{0, 1-y_i\theta^T\vec{x}_i\} \longrightarrow min.$$

В последней форме мы умножили целевую функцию на коэффициент $\lambda = C^{-1}$, чтобы неизвестный множитель стоял перед слагаемым с евклидовой нормой — это похоже на ридж–регрессию.

В отличие от логистической регрессии, здесь функция потерь не является гладкой, так что применять к ней градиентный спуск неразумно. Лучше переписать ее в виде задачи с ограничениями. Обозначим $\xi_i = \max\{0, 1 - y_i(\theta_0 + \ldots + \theta_m x_{im})\}$ (эти величины, очевидно, неотрицательны, и не меньше $1 - y_i\theta^T\vec{x}_i$). Тогда получим задачу

$$\begin{cases} \frac{1}{2}\theta^T\theta + C\sum_{i=1}^n \xi_i \longrightarrow min, \\ y_i(\theta_0 + \ldots + \theta_m x_{im}) \geq 1 - \xi_i \quad \forall i, \\ \xi_i \geq 0 \quad \forall i \end{cases}$$

(число C положительно и неизвестно, множитель $\frac{1}{2}$ поставлен для удобства, см. дальше). Получили отличную задачу для численного решения — на линейном симплексе мы минимизируем квадратичную выпуклую функцию. Она имеет единственное решение. К ней отлично применяется градиентный спуск. Надо только следить — при выходе на границу симплекса менять целевую функцию, учитывая соответствующее граничное условие. Кроме того, можно сводить задачу к системе Куна–Таккера.

Переход к двойственной задаче

Напомню, что если вы имеете задачу оптимизации $f(\vec{x}) \to extr$ с системой ограничений типа равенства $g_j(\vec{x}) = 0$, $1 \leq j \leq N$, то можно составить функцию Лагранжа

$$\mathcal{L}(\vec{x}, \lambda_1, \ldots, \lambda_N) = f(\vec{x}) + \sum_{j=1}^N \lambda_j g_j(\vec{x}),$$

найти ее стационарные точки — некоторые из них окажутся решением оптимизационной задачи. Этот же метод можно применять для ограничений типа неравенств.

Теорема 1 (Каруша–Куна–Таккера)**.** *Пусть дана задача на минимум $f(\vec{x}) \to min$ с ограничениями $g_j(\vec{x}) \leq 0$, $j = 1, \ldots, N$, и ограничениями $x_i \geq 0$. Пусть функции f и g_j, $j = 1, \ldots, N$ выпуклы (т.е. выпуклы вниз), а область, задаваемая ограничениями содержит хотя бы одну внутреннюю точку. Тогда задача имеет единственное решение — точку $\vec{x}$, которая находится как решение системы Куна–Таккера*

$$\mathcal{L}(\vec{x}, \lambda_1, \ldots, \lambda_N) = f(\vec{x}) + \sum_{j=1}^N \lambda_j g_j(\vec{x}), \qquad \begin{cases} \nabla_x \mathcal{L} = 0, \\ \lambda_j g_j(\vec{x}) = 0 \quad \forall j, \\ g_j(\vec{x}) \leq 0, \quad \forall j \\ \lambda_j \geq 0 \quad \forall j. \end{cases}$$

В записи системы равенство нулю градиента $\nabla\mathcal{L}$ предполагает равенство нулю производных только по переменным x_i. Вторая строка системы называется ***условиями дополняющей нежесткости****. Вместе с третьей и пятой строкой системы их надо читать так: для каждого j реализуется один из двух вариантов — либо $\lambda_j = 0$, а $g_j(\vec{x}) < 0$ (такие ограничения называют неактивными); либо $\lambda_j > 0$, а $g_j(\vec{x}) = 0$ (это*

активные ограничения).[2]

Эта теорема как раз подходит к нашей задаче. Сейчас будут некоторые вычисления и для удобства мы вернемся временно к задаче с жестким зазором. Итак, функция $\frac{1}{2}(\theta_1 + \ldots + \theta_m)^2 \to min$ выпукла, ограничения $1 - y_i(\theta_0 + \ldots + \theta_m x_{im}) \leq 0$ линейны, а значит выпуклы (теорема допускает нестрогую выпуклость). Получаем функцию Лагранжа и систему ККТ:

$$\mathcal{L} = \frac{1}{2}\sum_{j=1}^{m}\theta_j^2 + \sum_{i=1}^{n}\lambda_i(1 - y_i(\theta_0 + \ldots + \theta_m x_{im})), \qquad \begin{cases} \theta_j - \sum_{i=1}^{n}\lambda_i y_i x_{ij} = 0, \quad j = 1, \ldots, m, \\ -\sum_{i=1}^{n}\lambda_i y_i = 0, \\ 1 - y_i(\theta_0 + \ldots + \theta_m x_{im}) \leq 0 \quad \forall i, \\ \lambda_i(1 - y_i(\theta_0 + \ldots + \theta_m x_{im})) = 0 \quad \forall i, \\ \lambda_i \geq 0 \quad \forall i. \end{cases}$$

Выразим из первой строки все числа $\theta_j = \sum_{i=1}^{n}\lambda_i y_i x_{ij}$ (это не решает задачу — мы неизвестные доселе числа θ_j выразили через введенные нами и неизвестные числа λ_i). Теперь подставим эти равенства в задачу $\mathcal{L} \to min$ и учтем вторую строку $\sum_{i=1}^{n}\lambda_i y_i = 0$:

$$\mathcal{L} = -\frac{1}{2}\sum_{i=1}^{n}\sum_{l=1}^{n}\lambda_i\lambda_l y_i y_l \sum_{j=1}^{m} x_{ij}x_{lj} + \sum_{i=1}^{n}\lambda_i \to min.$$

Умножим на -2 и получим новую задачу

$$\sum_{i=1}^{n}\sum_{l=1}^{n}\lambda_i\lambda_l y_i y_l \sum_{j=1}^{m} x_{ij}x_{lj} + \sum_{i=1}^{n}\lambda_i \to max, \quad \lambda_i \geq 0 \ \forall i. \tag{2}$$

Сюда надо добавить еще две строки условий из системы ККТ. Для активных ограничений получаем равенства $1 - y_i\theta_0 - y_i\sum_{j=1}^{m}\sum_{l=1}^{n}\lambda_l y_l x_{ij}x_{lj} = 0$. Эти равенства в систему включать не надо — они определяют коэффициент θ_0 (если таких равенств несколько, то он должен удовлетворять каждому из них). Их используют, чтобы найти θ_0 после того, как задача на максимум решена. При этом, для уменьшения вычислительной погрешности разумно использовать усреднение по всем равенствам:

$$\theta_0 = \frac{1}{\#\{\lambda_i > 0\}}\sum_{i:\,\lambda_i>0}(y_i - \sum_{l=1}^{n}\lambda_l y_l \sum_{j=1}^{m} x_{ij}x_{lj}).$$

Ограничения, которые остались неактивными надо включать в задачу: для всех i, для которых $\lambda_i = 0$ появляется дополнительное ограничение

$$1 - y_i\theta_0 - y_i\sum_{l=1}^{n}\sum_{j=1}^{m}\lambda_l y_l x_{ij}x_{lj} < 0, \tag{3}$$

[2]Есть расширенный вариант теоремы. В нем в условия добавляются неравенства $x_i \geq 0$ для некоторого (произвольного) набора $\mathcal{I}$ переменных задачи. Тогда теорема сохраняется, но в системе ККТ надо сделать следующие изменения: вместо равенства нулю всех частных производных по переменным x_i потребовать равенство нулю только по переменным x_i, которые НЕ входят в $\mathcal{I}$; для индексов $i \in \mathcal{I}$ надо потребовать неотрицательность частных производных $\frac{\partial\mathcal{L}}{\partial x_i} \geq 0$, неотрицательность переменных $x_i \geq 0$ и потребовать, что если $x_i > 0$, то обнулялась бы частная производная, и наоборот.

где θ_0 выписано выше. Полученную задачу (целевая функция из (2) и ограничения (3)) называют **двойственной задачей**.

Упражнение 4. *Выпишите двойственную задачу для задачи с мягким зазором (используйте расширенный вариант теоремы ККТ).*

Решив двойственную задачу, легко получить решение прямой — достаточно вычислить $\theta_j = \sum_{i=1}^n \lambda_i y_i x_{ij}$ и $\theta_0 = \frac{1}{\#\{\lambda_i>0\}} \sum_{i:\,\lambda_i>0}(y_i - \sum_{l=1}^n \lambda_l y_l \sum_{j=1}^m x_{ij}x_{lj})$.

Двойственная задача в данном случае имеет такую же сложность, как и прямая (кроме редкого случая, когда $m > n$ и тогда переход к двойственной задаче снижает размерность). Гораздо важнее другое: двойственная задача выписывается **только через скалярные произведения данных**. Действительно, и в целевой функции, и в ограничениях числа x_{ij} фигурируют только в выражениях $\langle \vec{x}_i, \vec{x}_l \rangle = \sum_{j=1}^m x_{ij}x_{lj}$.

Ядерная модификация

Понятно, что бывают задачи, в которых хорошая разделяющая гиперплоскость просто не существует, но данные при этом хорошо разделяются нелинейным многообразием.

Пример 3. *Пусть точки класса C_0 располагаются в первой и третьей четвертях в $\mathbb{R}^2$, а точки класса C_1 — во второй и четвертой. Любая разделяющая прямая режет классы пополам, а вот две прямые (координатные оси) строго разделяют данные. Заметим, что в нашем примере можно поступить так: создадим третий признак $X_3 = X_1X_2$. Для точек класса C_0 этот признак положителен, а на классе C_1 отрицателен. Значит, в пространстве признаков (X_1, X_2, X_3) данные линейно разделяются плоскостью $X_3 = 0$.*

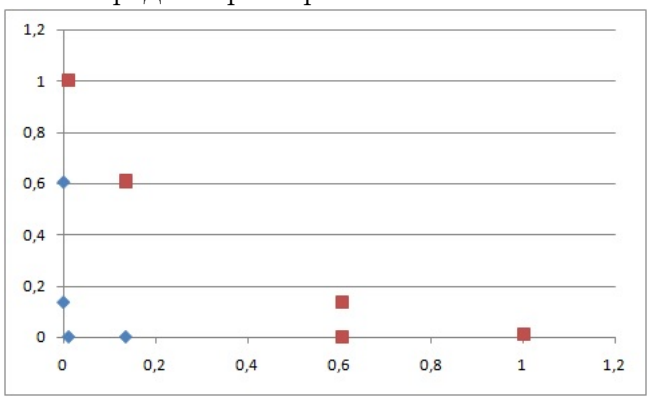
Этот пример приводит нас к идее полиномиального метода SVC. Например, если мы расширим пространство признаков квадратами имеющихся признаков, а также всеми их попарными произведениями, а затем применим метод SVC в полученном пространстве признаков, то мы проведем в результате квадратичную поверхность, разделяющую данные в исходном пространстве. Получили полиномиальный метод SVC порядка 2. Проблема в том, что при этом катастрофически растет число признаков: вместо m их станет $m + m(m+1)/2 \sim m^2/2$. Естественно, для полиномиального SVC степени p будет $O(m^p)$. К счастью, этого можно избежать.

Пусть мы решили добавить к признакам X_j, $j = 1, \ldots, m$, признаки X_j^2 и $\sqrt{2}X_jX_k$, $1 \le j < k \le m$ (множитель $\sqrt{2}$ выбран для красоты, см. ниже). Тогда скалярные произведения примут вид

$$\langle \vec{x}_i,\, \vec{x}_l \rangle = \sum_{j=1}^m x_{ij}x_{lj} + \sum_{j=1}^m x_{ij}^2 x_{lj}^2 + 2 \sum_{1\le j<k\le m} x_{ij}x_{ik}x_{lj}x_{lk} = \sum_{j=1}^m x_{ij}x_{lj} + \left(\sum_{j=1}^m x_{ij}x_{lj}\right)^2.$$

Если ввести функцию $K(\vec{a}, \vec{b}) = \langle \vec{a}, \vec{b} \rangle + \langle \vec{a}, \vec{b} \rangle^2$, не добавлять новые признаки, а просто заменить везде скалярные произведения $\langle \vec{x}_i, \vec{x}_l \rangle = \sum_{j=1}^m x_{ij}x_{lj}$ на $K\left(\vec{x}_i, \vec{x}_l\right)$ — мы получим те же выражения. Итак, можно взять двойственную задачу для исходных данных и только лишь поменять в ней все скалярные произведения на $K(\cdot)$, взятое от этих произведений.

Ядра, сосредоточенные около нуля (такие, как RBF и экспоненциальное), добавляют в число предикторов признаки близости.



Пример 4. *Рассмотрим очень простую ситуацию, когда* $m = 1$. *Пусть вектор–столбец данных равен* $X = X_1 = (-4\ \ -3\ \ -2\ \ -1\ 0\ 1\ 2\ 3\ 4)^T$, *а вектор* $Y = (0\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 0)^T$. *Понятно, что наши данные линейно не разделимы. Легко разделить их условием* $x^2 - 6.5 = 0$, *но мы пойдем сейчас другим путем. Поставим два ориентира, например в точках* $a = 1$ *и* $b = -2$. *А теперь построим два новых признака* X_2 *и* X_3 *для каждой точки: расстояние до первого и второго ориентира, посчитанные в смысле ядра RBF (параметр* γ *возьмем* 0.5*). Например, для точки* $x_{11} = -4$ *получим* $x_{12} = e^{-0.5\cdot(-4-1)^2} \approx 0$, $x_{13} = e^{-0.5\cdot(-4+2)^2} \approx 0.14$. *В результате на плоскости* (X_2, X_3) *получим облако точек, приведенное на рисунке. Теперь наши классы линейно разделимы.*

Здесь мы добавили два ориентира. Обычно RBF–метод добавляет каждое данное $\vec{x}_i$ как ориентир, так что число признаков оказывается равным n. Для того, чтобы такая модель не переобучалась, надо выбирать γ близким к нулю или включать не все $\vec{x}_i$ в список ориентиров, а только некоторую их часть (скажем α), которая выбирается случайным образом. Гиперпараметры γ и α настраиваются на валидационной выборке или методом кросс–валидации.

Метод опорных векторов (продолжение)

На прошлой лекции мы поговорили о методе опорных векторов: обсудили линейные задачи с жестким и мягким зазором, выяснили, как можно решать нелинейную задачу с жестким зазором. Выводить нелинейную версию задачи с мягким зазором не будем — это остается в качестве упражнения. Продолжим обсуждение ядерного метода SVC. Опишем его математическую постановку. Напомним формулы из прошлой лекции

$$\begin{cases} \sum_{i=1}^n \sum_{l=1}^n \lambda_i \lambda_l y_i y_l \langle \vec{x}_i, \vec{x}_l \rangle + \sum_{i=1}^n \lambda_i \to max, \quad \lambda_i \geq 0,\ i = 1, \ldots, n; \\ \lambda_i \geq 0, \qquad \sum_{i=1}^n \lambda_i y_i = 0, \\ \theta_j = \sum_{i=1}^n \lambda_i y_i x_{ij},\ j > 0; \qquad \theta_0 = y_i - \sum_{l=1}^n \lambda_l y_l \langle x_i, x_j \rangle\ \forall i :\ \lambda_i > 0. \end{cases}$$

Обозначим через H гильбертово пространство новых признаков (конечное или бесконечное ℓ_2 размерности N). Обозначим $\phi : \mathbb{R}^m \to H$ отображение, которое меняет старые признаки на новые (при этом, новые могут включать в себя старые, как для

квадратичного ядра). Применим к данным $\{h_i = \phi(\vec{x}_i)\}_1^n$ в пространстве H обычный метод SVC. Обозначим θ_j, $0 \le j \le N$, — коэффициенты разделяющего аффинного подпространства $H_0 = \{\vec{x} : \ h(x) = \sum_{j=1}^N \theta_j x(j) + \theta_0 = 0\}$ в H. Тогда для нового вектора $\vec{x}_{n+1}$ получим

$$h(\phi(\vec{x}_{n+1})) = \sum_{j=1}^N \theta_j \phi(\vec{x}_{n+1})(j) + \theta_0 = \langle \theta, \phi(\vec{x}_{n+1})\rangle_H + \theta_0.$$

Вектор θ найдем по формулам $\theta = \sum_{i=1}^n \lambda_i y_i \phi(\vec{x}_i)$ (мы эти формулы получили выше для конечномерного случая, но в бесконечномерной ситуации никаких отличий нет). Подставляя эти равенства, получим

$$h(\phi(\vec{x}_{n+1})) = \sum_{i=1}^n \lambda_i y_i \langle \phi(\vec{x}_i), \phi(\vec{x}_{n+1})\rangle_H + \theta_0 = \sum_{i=1}^n \lambda_i y_i K(\vec{x}_i, \vec{x}_{n+1}) + \theta_0.$$

Оказывается, для работы классификаторы не обязательно находить числа θ_j. Алгоритм Kernel SVC оказывается следующим.

1. Пишем двойственную задачу

$$\begin{cases} \sum_{i=1}^n \sum_{l=1}^n \lambda_i \lambda_l y_i y_l K(\vec{x}_i, \vec{x}_l) + \sum_{i=1}^n \lambda_i \to max, \quad \lambda_i \ge 0, \ i = 1, \dots, n; \\ \lambda_i \ge 0, \qquad \sum_{i=1}^n \lambda_i y_i = 0. \end{cases}$$

2. Решаем ее — находим числа λ_i, $i = 1, \dots, n$.
3. Вычисляем $\theta_0 = \frac{1}{\#\{\lambda_i > 0\}} \sum_{i:\, \lambda_i > 0} \left(y_i - \sum_{l=1}^n \lambda_l y_l K(\vec{x}_i, \vec{x}_l)\right)$.
4. Составляем правило классификации $\hat{y}(\vec{x}) = \text{sign}\left(\sum_{i=1}^n \lambda_i y_i K(\vec{x}, \vec{x}_{n+1}) + \theta_0\right)$.

В нашем выводе нелинейной задачи постулировалось наличие отображения ϕ, создающего дополнительные признаки. Это отображение, как мы видели, используется только для теоретического вывода — практически работа идет только с ядром $K(a, b) : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$. Возникает резонный вопрос — какие ядра допустимы?

Пример 5. *Рассмотрим ядро RBF* $K(\vec{x}, \vec{y}) = \exp\{-\frac{1}{2}\sum_{j=1}^m (x_j - y_j)^2\}$. *Наша задача — найти гильбертово пространство* H, *в котором это ядро имело бы вид скалярного произведения* $K(\vec{x}, \vec{y}) = \langle \phi(\vec{x}), \phi(\vec{y})\rangle_H$. *Решение этой задачи простое: возьмем от функции* $K(\vec{x}, \vec{y})$ *m–мерное преобразование Фурье по переменной* $\vec{x}$:

$$(2\pi)^{-m/2} \int_{\mathbb{R}^m} e^{-\frac{1}{2}\sum_{j=1}^m (x_j - y_j)^2} \cdot e^{-i\sum_{j=1}^m \lambda_j x_j} d\vec{x} = \ (x_j - y_j = \xi_j) \ =$$
$$= (2\pi)^{-m/2} \int_{\mathbb{R}^m} e^{-\sum_{j=1}^m (\xi_j^2/2 + i\lambda_j \xi_j)} d\xi_1 \dots d\xi_j \times e^{-i\sum_{j=1}^m \lambda_j y_j} = e^{-\frac{1}{2}\sum_{j=1}^m \lambda_j^2 - i\sum_{j=1}^m \lambda_j y_j}.$$

Теперь запишем формулу обратного преобразования Фурье:

$$K(\vec{x}, \vec{y}) = (2\pi)^{-m/2} \int_{\mathbb{R}^m} e^{i\sum_{j=1}^m \lambda_j x_j} \cdot e^{-i\sum_{j=1}^m \lambda_j y_j} \cdot e^{-\frac{1}{2}\sum_{j=1}^m \lambda_j^2} \, d\lambda_1 \dots d\lambda_m = \left\langle e^{i\sum_{j=1}^m \lambda_j x_j}, \, e^{i\sum_{j=1}^m \lambda_j y_j} \right\rangle_{L_2(\sigma)},$$

где $\sigma(\lambda_1, \dots, \lambda_m) = (2\pi)^{-m/2} e^{-\frac{1}{2}\sum_{j=1}^m \lambda_j^2}$. *Пространство* $H = L_2(\sigma)$ *построено. Отображение* ϕ *тоже видно невооруженным глазом:* $\phi(\vec{x}) = e^{i\sum_{j=1}^m \lambda_j x_j}$. *Теперь формула обратного преобразования Фурье принимает вид* $K(\vec{x}, \vec{y}) = \langle \phi(\vec{x}), \phi(\vec{y})\rangle_H$.

Теорема 2 (Мерсера)**.** [3] *Пусть ядро* $K(\vec{x}, \vec{y}) : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ *индуцирует в пространстве* $L_2(\mathbb{R}^m)$ *самосопряженный компактный оператор (например, подойдет ядро, которое непрерывно, симметрично, положительно определено и интегрируемо в квадрате на всем* $\mathbb{R}^m \times \mathbb{R}^m$*). Тогда найдется такое сепарабельное гильбертово пространство* H*, и отображение* $\phi : \mathbb{R}^m \to H$*, что* $K(\vec{x}, \vec{y}) = \langle \phi(\vec{x}), \phi(\vec{y}) \rangle_H$*.*

Перечислим те ядра, которые применяются.

1. Линейное ядро $K(a, b) = \langle a, b \rangle$ возвращает нас к линейному методу SVC.
2. Полиномиальное ядро $K(a, b) = (\gamma \langle a, b \rangle + r)^d$ (условиям теоремы Мерсера это ядро, кстати, не удовлетворяет; пространство H здесь предъявляется явно).
3. Гауссово ядро (оно же RBF) $K(a, b) = \exp\{-\gamma \|a - b\|^2\}$ и его вариации $K(a, b) = \exp\{-(a - b) \cdot \Sigma \cdot (a - b)\}$.
4. Сигмоидальное ядро $K(a, b) = tanh(\gamma \langle a, b \rangle + r)$.
5. Экспоненциальное ядро $K(a, b) = \exp\{-\gamma \|a - b\|\}$.
6. Периодическое ядро $K(a, b) = \exp\{-\gamma \sin^2(\omega \|a - b\|)\}$.

Обсудим, в каких ситуациях разумно применять то или иное ядро.

Заметим в конце, что всегда можно конструировать свои ядра из имеющихся. Ниже приведены несколько способов.

1. Суммирование $K(\cdot, \cdot) = K_1(\cdot, \cdot) + K_2(\cdot, \cdot)$.
2. Перемножение $K(\cdot, \cdot) = K_1(\cdot, \cdot) \times K_2(\cdot, \cdot)$.
3. Домножение на функцию $K(a, b) = f(a) K_1(a, b) f(b)$, где f произвольна.
4. Композиция $K(a, b) = q(K(a, b))$, где q раскладывается в ряд Маклорена с неотрицательными коэффициентами (например, $q(\xi) = \xi + \xi^2$ или $q(\xi) = e^{\xi}$).

Многоклассовый SVC

В отличие от логистической регрессии, где есть метод softmax, многоклассовая классификация методом SVC возможна только с помощью двух следующих способов. Они универсальны и позволяют настроить multiclass classifier на основе бинарной — не важно, каким способом проводится эта бинарная классификация.

Первый способ называется OvA (one versus all — один против всех). Для каждого класса C_k, $k = 1, \ldots, l$, обучим свой классификатор, относя при обучении все остальные классы в

[3] Для интересующихся. Доказательство несложное. Надо рассмотреть интегральный оператор $[Tf](\vec{x}) = \int_{\mathbb{R}^m} K(\vec{x}, \vec{y}) f(\vec{y})\, d\vec{y}$, проверить, что он компактен и самосопряжен, а затем применить теорему Гильберта–Шмидта, из которой следует разложение $K(\vec{x}, \vec{y}) = \sum_{j=1}^{N} \lambda_j e_j(\vec{x}) e_j(\vec{y})$ по собственным значениям λ_j и собственным функциям $e_j(\vec{x})$.

C_0. Теперь, когда нам надо классифицировать некоторый вектор $\vec{x}$, мы прогоним его через все классификаторы и посмотрим, какой классификатор поставит ему больше баллов — получим ответ.

Другой способ называется OvO (one versus one — один на один). Обучим $l(l-1)/2$ бинарных классификаторов для каждой пары классов k_1, k_2. Теперь, когда нам надо классифицировать некоторый вектор $\vec{x}$, мы прогоним через все классификаторы и выберем тот класс, который выиграл больше дуэлей. Обучать, конечно, придется много классификаторов, но каждый будет обучаться лишь на части данных — на тех, которые размечены именно двумя данными классами k_1 и k_2. С практической точки зрения — выбирая между OvA и OvO, надо посмотреть, как растет сложность вашего классификатора по параметру n. Есть еще один подводный камень: применяя метод OvA надо следить за балансом размерности классов.

Регрессия методом опорных векторов

Начальные условия стандартны: матрица X размера $n \times m$ и вектор Y высоты n. Перевернем задачу: будем искать линейную функцию $\hat{Y} = \theta_0 + \ldots + \theta_m X_m$ (в пространстве $(y, \vec{x})$ она задает гиперплоскость π) такую, что все данные расположены в полосе $\rho((y_i, \vec{x}_i), \pi) < d$ наименьшей ширины. Поскольку расстояние от точки до гиперплоскости имеет вид $\rho((y_i\vec{x}_i), \pi) = \frac{|y_i - \hat{y}_i|}{\sqrt{1+\theta_1^2+\ldots+\theta_m^2}}$, то обозначив знаменатель константой c, получаем задачу минимизации

$$\begin{cases} \max_{1\le i\le n} |y_i - (\theta_0 + \theta_1 x_{i1} + \ldots + \theta_m x_{im})| \to min, \\ \theta_1^2 + \ldots + \theta_m^2 = c^2 - 1. \end{cases}$$

Переходя к функции Лагранжа, получим

$$\max_{1\le i\le n} |y_i - (\theta_0 + \theta_1 x_{i1} + \ldots + \theta_m x_{im})| + C\sum_{j=1}^{m} \theta_j^2 \to min.$$

Мы пришли к ридж–регрессии с нестандартной функцией потерь. Честно говоря, эта функция потерь нехороша — она совершенно не умеет работать с выбросами, даже один выброс кардинально меняет задачу. Если немного подумать, то станет ясно отчего это произошло — мы поставили задачу с жестким зазором. Сделаем зазор мягким — разрешим некоторым данным выходить за пределы полосы. Назовем число d гиперпараметром и заменим $|y_i - \hat{y}_i|$ на $\max\{0, |y_i - \hat{y}_i| - d\}$. Это уже известная функция для построения регрессии — функция Хубера, она показывает величину штрафа за пересечение точкой полосы. Общий штраф составим как сумму всех штрафов (как и в методе SVC) и получим новую функцию потерь

$$\mathcal{L} = \frac{1}{2}\sum_{j=1}^{m} \theta_j^2 + C\sum_{i=1}^{n} \max\{0, |y_i - \hat{y}_i| - d\}$$

(функцию штрафа мы берем с некоторым коэффициентом — гиперпараметром, а тогда коэффициент при ридж–слагаемом можно взять $1/2$).

В принципе, на этом можно остановиться и далее решать задачу обычным градиентным спуском. Однако, наша функция потерь негладкая и ее поверхность содержит многочисленные плато. Действительно, очевидно, что если полоса уже построена, то малые изменения точек, оказавшихся внутри полосы, никак не меняют функцию $\mathcal{L}$. Поступим так же, как мы сделали в задаче классификации — вернемся от функции потерь к задаче с ограничениями. Положим $\xi_i^+ = \max\{0, y_i - \hat{y}_i - d\}$, $\xi_i^- = \max\{0, \hat{y}_i - y_i - d\}$. Получаем задачу

$$\begin{cases} \mathcal{L} = \frac{1}{2}\sum_{j=1}^m \theta_j^2 + C\sum_{i=1}^n (\xi_i^+ + \xi_i^-) \to min, \\ \xi_i^+ \geq 0, \quad \xi_i^- \geq 0 \qquad \forall i, \\ y_i - \hat{y}_i - d \leq \xi_i^+, \qquad \hat{y}_i - y_i - d \leq \xi_i^- \qquad \forall i. \end{cases}$$

Это уже классическая задача квадратичной минимизации на линейном симплексе. Ее можно решать стандартным методом квадратичного программирования.

Пойдем дальше. Нам хотелось бы применить ядерный метод в задаче SVR. Как мы помним, для этого надо вначале перейти к двойственной задаче. Введем множители $\lambda_i^\pm$ и составим систему Куна–Таккера (нам нужен расширенный вариант теоремы)

$$\mathcal{L} = \frac{1}{2}\sum_{j=1}^m \theta_j^2 + C\sum_{i=1}^n (\xi_i^+ + \xi_i^-) +$$

$$+\sum_{i=1}^n \left(\lambda_i^+ (y_i - \theta_0 - \sum_{j=1}^m \theta_j x_{ij} - d - \xi_i^+) + \lambda_i^- (\theta_0 + \sum_{j=1}^m \theta_j x_{ij} - y_i - d - \xi_i^-) \right),$$

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = 0 \quad \Longleftrightarrow \quad \theta_j = \sum_{i=1}^n (\lambda_i^+ - \lambda_i^-) x_{ij}, \quad j = 1, \ldots, m, \qquad \sum_{i=1}^n (\lambda_i^+ - \lambda_i^-) = 0,$$

$$\begin{cases} \lambda_i^\pm \geq 0, \quad \xi_i^\pm \geq 0, \quad y_i - \theta_0 - \sum_{j=1}^m \theta_j x_{ij} - d \leq \xi_i^+, \quad \theta_0 + \sum_{j=1}^m \theta_j x_{ij} - y_i - d \leq \xi_i^-, \\ \lambda_i^+ \left(y_i - \theta_0 - \sum_{j=1}^m \theta_j x_{ij} - d - \xi_i^+ \right) = \lambda_i^- \left(\theta_0 + \sum_{j=1}^m \theta_j x_{ij} - y_i - d - \xi_i^- \right) = 0. \end{cases}$$

Подставляя выражения для θ_j в целевую функцию и в условия получим задачу оптимизации, где данные фигурируют исключительно в виде скалярных произведений $\langle \vec{x}_i, \vec{x}_l \rangle = \sum_{j=1}^m x_{ij} x_{lj}$:

$$\mathcal{L} = \frac{1}{2}\sum_{i=1}^n \sum_{l=1}^n \langle \vec{x}_i, \vec{x}_l \rangle + C\sum_{i=1}^n (\xi_i^+ + \xi_i^-) +$$

$$+\sum_{i=1}^n \sum_{l=1}^n (\lambda_i^- - \lambda_i^+)(\lambda_l^+ - \lambda_l^-)\langle \vec{x}_i, \vec{x}_l \rangle - d\sum_{i=1}^n (\lambda_i^- + \lambda_i^+) - \sum_{i=1}^n (\lambda_i^+ \xi_i^+ + \lambda_i^- \xi_i^-) \to min, \ldots$$

(систему ограничений выпишите сами). Теперь остается заменить все скалярные произведения на $K(\vec{x}_i, \vec{x}_j)$ и получаем ядерный метод SVR.

Упражнение 5. *Выведите формулы прогнозирования значения SVR на новом векторе $\vec{x}_{n+1}$ так, как это было проделано в прошлой лекции для SVC.*

Упражнение 6. *Предложите формулу для θ_0.*

7 Метрические методы

Суть метрических методов заключена в известной поговорке: «Скажи мне, кто твой друг, и я скажу, кто ты». У этих методов практически нет никакого обучения, за что их называют lazy learning. На этапе предсказания методы просто ищут объекты, похожие на целевой. Таким образом, они являются **локальными методами**. Поскольку эти методы не строят никаких моделей, не подбирают никаких параметров, то их еще называют **непараметрические методы**.

Метрические методы редко применяются в машинном обучении solo. Обычно они предваряют какой–то другой метод, т.е. метрический метод обычно — первый этап какой–то сложной модели, например, нейронной сети. Например, метрический метод на первом этапе применяется для того, чтобы в первом приближении кластеризовать данные. Затем для каждого кластера строится своя модель.

KNN = K nearest neighbourhood = метод k–ближайших соседей

Начнем с задачи бинарной классификации методом KNN. Пусть дана обучающая выборка (X, Y) объема n, где $\vec{x}_i \in \mathbb{R}^m$, а $y_i \in \{0, 1\}$. Зафиксируем некоторую метрику ρ в пространстве $\mathbb{R}^m$. Аксиомы метрики, вы, надеюсь, помните — положительность, симметричность и неравенство треугольника. Так вот, иногда разумно брать функции ρ, которые неравенству треугольника не удовлетворяют (см. примеры ниже), так что строго говоря, ρ может не быть метрикой (при этом, мы все равно будем ее так называть). Стадии обучения нет. Пусть нам необходимо классифицировать некоторый объект $\vec{x}_{n+1}$. Найдем k точек $\vec{x}_{i_s}$ из тренировочной выборки, ближайшие к $\vec{x}_{n+1}$ в смысле расстояния ρ. Теперь посчитаем, сколько из них принадлежат классу C_0, а сколько — классу C_1 (пусть это оказались числа k_0 и k_1). Теперь выдадим баллы для вектора $\vec{x}_{n+1}$ по классам: k_0/k и k_1/k. Отнесем $\vec{x}_{n+1}$ к тому классу, который набрал больше баллов.

Будьте осторожны! Числа k_0/k и k_1/k хотя и похожи на вероятности, но не являются хорошими статистическими оценками этих вероятностей. Поэтому я назвал их баллами. Очевидно, что метод легко обобщается на большое число классов $l > 2$. Если речь идет о задаче регрессии, то логично давать предсказания в виде $\hat{y}_{n+1} = \frac{1}{k}\sum_{s=1}^{k} y_{i_s}$ — мы получаем один из случаев локально–постоянной регрессии. Число k является гиперпараметром. При малых k метод склонен к переобучению — границы классов получаются весьма замысловатыми.

Какую метрику ρ выбрать? Это зависит от вашей задачи и от структуры ваших данных. Давайте считать, что вы уже привели ваши признаки к единой шкале, например, сделали нормализацию.

В подавляющем большинстве случаев хорошим выбором будет обычное **евклидово расстояние** $\rho_2(\vec{u}, \vec{v}) = \left(\sum_{j=1}^{m}(u_j - v_j)^2\right)^{1/2}$. Используются и другие метрики — перечислим их.

Манхеттенская метрика (она же ℓ_1–метрика) $\rho_1(\vec{u}, \vec{v}) = \sum_{j=1}^{m}|u_j - v_j|$. Эта метрика часто используется при больших m из–за лучшей устойчивости к выбросам. Представим себе два вектора u и v, которые близки по всем координатам $|u_j - v_j| \approx \varepsilon$, кроме одной

$|u_{j_0} - v_{j_0}| \approx d$. Скорее всего, эта координата является ошибкой измерения — выбросом и мы должны бы выдать расстояние между векторами $\widetilde{u}$ и $\widetilde{v}$ с исправленной координатой. Тогда отношение расстояний $\rho(u,v)$ к $\rho(\widetilde{u},\widetilde{v})$ по двум разным метрикам составит

$$\frac{\rho_2(u,v)}{\rho_2(\widetilde{u},\widetilde{v})} = \sqrt{1 + \frac{d^2}{(m-1)\varepsilon^2}} \sim \frac{d}{\sqrt{m-1}\varepsilon}, \qquad \frac{\rho_1(u,v)}{\rho_1(\widetilde{u},\widetilde{v})} = 1 + \frac{d}{(m-1)\varepsilon} \sim \frac{d}{(m-1)\varepsilon},\ d \to \infty.$$

Видно, что ℓ_1–метрика склонна нивелировать выбросы лучше, чем евклидова при больших m.

Метрика Минковского $\rho_p(\vec{u},\vec{v}) = \left(\sum_{j=1}^m |u_j - v_j|^p\right)^{1/p}$.

Косинусное расстояние $\rho_{cos}(\vec{u},\vec{v}) = 1 - \cos\theta_{u,v} = 1 - \frac{\langle\vec{u},\vec{v}\rangle}{\|\vec{u}\|\cdot\|\vec{v}\|}$. Оно применяется тогда, когда нам интересны векторы с точностью до положительного коэффициента. Фактически мы вводим расстояние между лучами, а не между точками или, что тоже самое, проектируем точки на сферу в $\mathbb{R}^m$ и измеряем расстояние между этими проекциями. Действительно, по теореме косинусов, расстояние между проекциями равно $\sqrt{1+1-2\cos\theta} = \sqrt{2\rho_{cos}}$. Легко проверить, что в такой постановке наша функция расстоянием является. Характерный пример использования косинусного расстояния — измерение различий между двумя текстами. Координаты векторов в таких задачах — это число появления того или иного слова в тексте. При этом, мы хотим замерять именно семантическое расстояние между текстами, т.е. если взять абзац некоторого текста, а затем приписать к нему похожий абзац, то текст изменится не сильно с нашей точки зрения.

Расстояние Жаккара $\rho(A,B) = 1 - \frac{|A\cap B|}{|A\cup B|}$ используется для измерения расстояния между изображениями (модулем обозначен m–мерный объем). Действительно, два множества «похожи», если их пересечение примерно совпадает с их объединением.

Если нормализацию мы делать не хотим, то разные признаки придется учитывать с разным весом. Тогда евклидова метрика превращается в весовую евклидову метрику — **расстояние Махаланобиса**

$$\rho_M(\vec{u},\vec{v}) = \langle M(\vec{u}-\vec{v}),(\vec{u}-\vec{v})\rangle^{1/2} = \langle M^{1/2}(\vec{u}-\vec{v}),M^{1/2}(\vec{u}-\vec{v})\rangle^{1/2},$$

где матрица M должна быть самосопряженной и положительной. Выбирая коэффициенты масштаба на диагонали M, можно провести нормализацию расстояния без нормализации самих данных.

У классического KNN есть недоработка — мы не учитываем близость соседей к вектору $\vec{x}_{n+1}$, а эта информация может быть полезной. Логично использовать **весовой KNN**. Для задач классификации будем вычислять баллы по каждому классу C не суммой дробей $\frac{1}{k}$, а суммой $\sum_{s=1}^k w_s I_{y_{i_s}\in C}$. Для задач регрессии составим прогноз

$$\hat{y}(x) = \frac{\sum_{i=1}^s w_s y(\vec{x}_{i_s})}{\sum w_s}.$$

Весь вопрос теперь в том, как подбирать веса. Есть несколько стандартных способов.

Можно отсортировать соседей $\vec{x}_{i_s}$ по возрастанию расстояния до $\vec{x}_{n+1}$ и взять большие веса для близких соседей и малые для дальних. Здесь встречаются два варианта — арифметическая прогрессия $w_s = \frac{k+1-s}{k}$ (первый сосед идет с весом 1, а последний с весом $1/k$) и геометрическая прогрессия $w_s = q^s$, $0 < q < 1$ — гиперпараметр метода.

Скользящее окно

Еще более логичная идея — формировать веса, основываясь не на порядковом номере соседа, а на расстоянии $\rho(\vec{x}_{i_s}, \vec{x}_{n+1})$. Тогда нам нужна функция $K(\xi)$, которая будет вычислять эти веса. Понятное дело, функция должна быть положительной, должна достигать максимума в нуле, а далее монотонно убывать по $|\xi|$. Такую функцию называют kernel function (ядро). При такой постановке речь уже не идет о k соседях. Фактически, мы рассматриваем некоторую окрестность точки $\vec{x}_{n+1}$ и учитываем все образцы $\vec{x}_i$, которые попали в эту окрестность (каждого со своим весом). Функция K не обязательно обнуляется вне окна $|\xi| < h$ (см. примеры ниже), так что окрестность может совпадать с $\mathbb{R}^m$ (тогда веса должны достаточно быстро убывать по $|\xi|$).

Для задач классификации получаем способ подсчета баллов по классу C:

$$\sum_{i=1}^{n} K\left(\frac{\rho(\vec{x}_i, \vec{x}_{n+1})}{h}\right) I_{y_i \in C}.$$

В задачах регрессии получаем формулу Надарая–Ватсона (метод называют ядерной регрессией — kernel regression)

$$\hat{y}(\vec{x}) = \frac{\sum_{i=1}^{n} K\left(\frac{\rho(\vec{x}_i, \vec{x})}{h}\right) y_i}{\sum_{i=1}^{n} K\left(\frac{\rho(\vec{x}_i, \vec{x})}{h}\right)}$$

Если речь идет о задачах регрессии, то гладкость функции $\hat{y}(\vec{x})$ напрямую зависит от гладкости ядра K. Популярные ядра перечислены ниже. В списке мы их нормализуем «на единицу», оставляя свободу в выборе параметра h — ширины окна. Кроме того, ядра в нашем списке нормированы на единичную площадь под графиком $K(\xi)$.

1. Прямоугольное ядро $K(\xi) = \frac{1}{2} I_{|\xi| \leq 1}$.
2. Треугольное ядро $K(\xi) = (1 - |\xi|) I_{|\xi| \leq 1}$.
3. Ядро Епанечникова $K(\xi) = \frac{3}{4}(1 - \xi^2) I_{|\xi| \leq 1}$.
4. Биквадратное ядро $K(\xi) = \frac{15}{16}(1 - \xi^2)^2 I_{|\xi| \leq 1}$.
5. Гауссовское ядро $K(\xi) = \frac{1}{\sqrt{2\pi}} \exp\{-\frac{\xi^2}{2}\}$.

Немного теории

Дадим математическое обоснование метода KNN и метода скользящего окна (пока что мы говорили на уровне эвристики). Предположим вначале, что мы говорим о задаче классификации. Для каждого класса C_k мы можем посчитать эмпирическую функцию распределения $F_k(\vec{x}) = \frac{\#\{\vec{x}_i \in C : \vec{x}_i < \vec{x}\}}{\#C}$ (неравенство надо понимать как систему неравенств по всем координатам, т.е. $x_{ij} < x_j \ \forall j$). Теперь с помощью дискретного дифференцирования

можно оценить плотность каждого класса. Дифференцируем функцию F по всем переменным x_j по формуле $\frac{\partial F}{\partial x_j} \approx \frac{1}{2h}(F(\dots, x_j + h, \dots) - F(\dots, x_j - h, \dots))$. Получаем

$$p(x|y=k, D) = \frac{1}{\#C_k \cdot (2h)^m} \sum_{i:\, y_i=k} I_{\forall j: |x_{ij} - x_j| \le h} = \frac{2}{\#C_k \cdot (2h)^m} \sum_{i=1}^{n} K\left(\frac{\rho(\vec{x}_i, \vec{x})}{h}\right) I_{y_i=k},$$

где ядро K надо взять прямоугольным, а метрику ρ — Минковского с индексом $p = \infty$. Байесовское правило классификации предполагает выбор класса, для которого вероятность $p(y = k|x, D) = \frac{p(x|y=k,D)p(y=k|D)}{p(x|D)}$ является наибольшей. Отбрасывая знаменатель (он одинаков для всех классов) и учитывая, что $p(y = k|D) = \#C_k$, получим правило: надо вычислить баллы по каждому классу

$$\sum_{i=1}^{n} K\left(\frac{\rho(\vec{x}_i, \vec{x})}{h}\right) I_{y_i=k}$$

и выбрать класс k, для которого эти баллы наибольшие. Получился метод скользящего окна с прямоугольным окном и метрикой Минковского $p = \infty$. Все остальные методы получаются выбором других формул для дискретного дифференцирования и/или других метрик ρ.

Чтобы от метода скользящего окна прийти к методу KNN надо «разморозить» в предыдущих рассуждениях параметр h. Действительно, мы считали его фиксированным и одинаковым для каждой точки $\vec{x}$. А теперь давайте выберем $h = \rho(x_{i_k}, \vec{x})$ равным расстоянию от точки $\vec{x}$ до k–го по счету соседа, считая их упорядоченными по возрастанию расстояния. Тогда в сумме будут участвовать только те индексы i_s, для которых $\rho(\vec{x}_{i_s}, \vec{x}) < \rho(\vec{x}_{i_k}, \vec{x})$, т.е. все соседи с меньшим расстоянием.

В задаче регрессии мы будем предсказывать математическое ожидание

$$\hat{y}(\vec{x}) = E(y|x, D) = \int y \cdot p(y|\vec{x}, D)\, dy = \int y \cdot \frac{p(\vec{x}, y|D)}{p(\vec{x}|D)}\, dy = \frac{\int y \cdot p(\vec{x}, y|D)\, dy}{\int p(\vec{x}, y|D)\, dy}.$$

Таким образом, нам надо лишь оценить совместную плотность вектора $(\vec{x}, y)$ на основании наших данных. Для простоты снова зафиксируем простейшую формулу дискретного дифференцирования и метрику Минковского ρ_∞. Получим оценку

$$p(\vec{x}, y|D) = \frac{1}{n \cdot (2h)^m} \sum_{i=1}^{n} I_{\forall j: |x_{ij} - x_j| \le h} \cdot I_{|y_i - y| \le h} = \frac{4}{n \cdot (2h)^m} \sum_{i=1}^{n} K\left(\frac{\rho(\vec{x}_i, \vec{x})}{h}\right) K\left(\frac{|y_i - y|}{h}\right).$$

Подставляя в верхнюю формулу, учитывая нормировку K на единичный интеграл и четность, имеем

$$\hat{y}(\vec{x}) = \frac{\sum_{i=1}^{n} K\left(\frac{\rho(\vec{x}_i, \vec{x})}{h}\right) \int y \cdot K\left(\frac{|y_i - y|}{h}\right) dy}{\sum_{i=1}^{n} K\left(\frac{\rho(\vec{x}_i, \vec{x})}{h}\right) \int K\left(\frac{|y_i - y|}{h}\right) dy} = \frac{\sum_{i=1}^{n} K\left(\frac{\rho(\vec{x}_i, \vec{x})}{h}\right) y_i}{\sum_{i=1}^{n} K\left(\frac{\rho(\vec{x}_i, \vec{x})}{h}\right)}.$$

Получили формулу Надарайя–Ватсона.

Локальная регрессия

Взглянем на оценку Надарайя–Ватсона с другой стороны. Отметим, что оптимальной постоянной оценкой регрессии является постоянная a, минимизирующая

$$\sum_{i=1}^{n}(Y_i - a)^2,$$

то есть $a = \overline{Y}$. Если же мы при некотором наборе $w_i(x)$ минимизируем

$$\sum_{i=1}^{n} w_i(x)(Y_i - a)^2,$$

то ответом будет

$$a = \sum_{i=1}^{n} \frac{w_i(x)Y_i}{\sum_{j=1}^{n} w_j(x)},$$

что при $w_i(x) = g_i(x)$ даст оценку Надарайя–Ватсона. Таким образом, это "локально постоянная" оценка, где в окрестности каждой точки x мы взвешиваем погрешности пропорционально их близости к точке x.

Данный подход естественно обобщается до минимизации

$$\sum_{i=1}^{n} w_i(x)(Y_i - a - b(X_i - x))^2$$

с теми же $w_i(x) = g_i(x)$. Взвешенная линейная модель дает нам

$$\begin{pmatrix} \widehat{a}(x) \\ \widehat{b}(x) \end{pmatrix} = (X^T W X)^{-1} X^T W \vec{Y}, \quad X = \begin{pmatrix} 1 & x_1 - x \\ \dots & \dots \\ 1 & x_n - x \end{pmatrix}, \quad W = diag(w_i(x)).$$

Определение 1. *Функция $\widehat{a}(x)$, получаемая в формуле выше, называется локально линейной оценкой.*

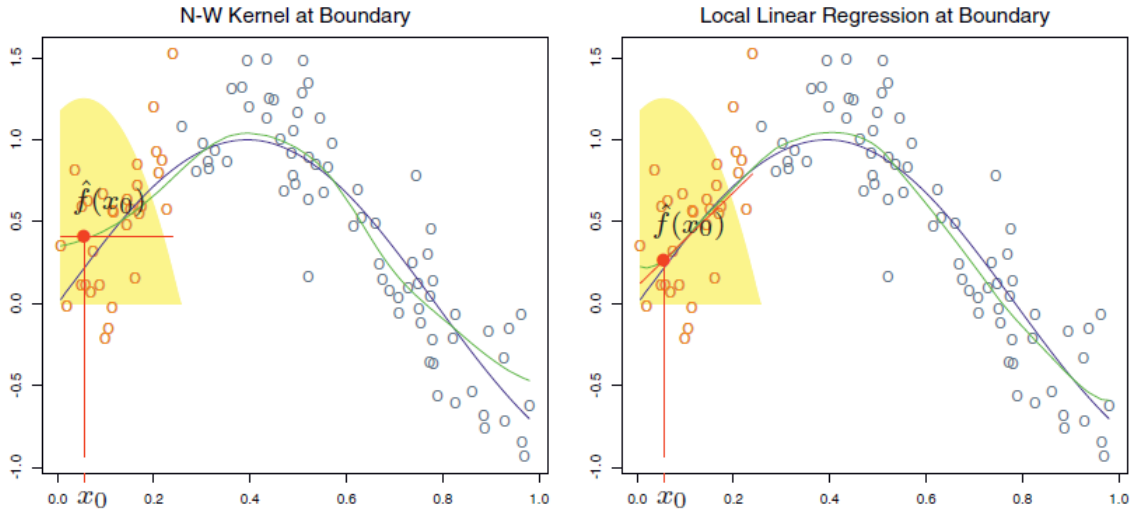
Аналогичным образом вводится локально полиномиальная оценка:

$$X = \begin{pmatrix} 1 & x_1 - x & \dots & (x_1 - x)^k \\ \dots & \dots & \dots & \dots \\ 1 & x_n - x & \dots & (x_n - x)^k \end{pmatrix}, \quad W = diag(w_i(x)),$$

Нередко используют именно полиномиальную оценку невысокого порядка, что дает и достаточную регулярность, и маленькое смещение: линейную или квадратичную оценку.

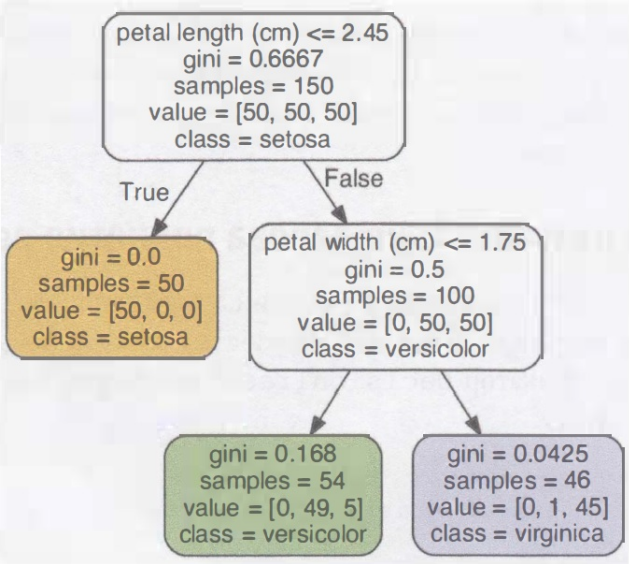
Важным преимуществом локально линейных и полиномиальных оценок является отсутствие так называемого "смещения на границе": из-за того, что в районе границы наша функция определена только с одной стороны, зачастую в граничных точках появляется большее смещение, чем в остальных местах. Локально полиномиальные оценки снижают этот эффект.

Рис. 1: Черным цветом изображена настоящая функция, справа зеленым локально линейная оценка, слева зеленым – оценка Надарайя–Ватсона.



8 Деревья принятия решений

Анализ данных в современном его виде начал появляться в 60-е годы (тогда его называли регрессионным анализом). И первыми методами были, разумеется регрессии (в том числе, логистическая). Затем на долгое время самым популярным методом стал SVM — там есть ядерная модификация, сам метод очень быстро работает (а компьютеры были на два порядка медленнее). На рубеже 80–90-х годов пальму первенства перехватили решающие деревья и случайные леса. Ну, а в 2012 году заработали наконец эффективно нейронные сети (сами сети появились еще в начале 90-х, но долгое время проигрывали в эффективности методам классического машинного обучения).



Чтобы понять метод decision tree, достаточно посмотреть на один пример. На рисунке приведено дерево для классического набора данных iris (мы его уже встречали в курсе анализа данных). Теперь дадим строгое определение.

Определение 2. *Бинарное решающее дерево — это бинарное помеченное дерево. Каждой внутренней вершине (узлу) v дерева приписан предикат $B_v : X \to \{0, 1\}$. Каждому листу v приписан прогноз: значение регрессии или номер класса (или баллы по каждому классу).*

Если дерево построено, то работа модели заключается в проходе по дереву от корня до листа, подчиняясь предикатам. Результатом работы является прогноз, приписанный полученному листу. Получается, что каждый образец $\vec{x}_i$ при обработке совершает путь по дереву от корня до некоторого листа. Для каждой вершины дерева таким образом, определен набор тех образцов, которые при обработке проходят через данную вершину. Для каждого узла дерева это множество образцов расщепляется при обработке — некоторые отправляются вниз влево, а некоторые вниз вправо. Из-за этого узлы дерева решений называют сплитами.

Вообще–то предикаты могут иметь произвольный вид, но на практике обычно это предикаты типа неравенств $B(\vec{x}) = [x_j \leq t]$. Иными словами, на каждом шаге принятия решения мы выбираем некоторый признак X_j, смотрим на значение этого признака у нашего образца (число x_j) и сравниваем это значение с порогом t. Из сказанного следует несколько важных выводов.

1. Области значений при классификации получаются типа m–мерных параллелепипедов, ребра которых параллельны координатным осям.

2. Выученная функция всегда кусочно–постоянна, так что о градиентных методах построения дерева можно забыть.

3. Дерево решений не может экстраполировать свои выводы за пределы области, в которой лежит обучающая выборка.

4. Если строить дерево без ограничения его высоты, то оно отнесет каждый отдельный образец в свой лист — идеально приблизит обучающую выборку, но наверняка плохо сработает на тестовой. Таким образом, метод склонен к переобучению.

Поясним еще некоторую информацию, которая видна на рисунке примера. Атрибут samples каждой вершины показывает, сколько образцов из тренировочной выборке относятся к данной вершине (проходят через данную вершину при классификации). Атрибут $value = [value_1, \ldots, value_l]$ вершины показывает абсолютную частоту распределения образцов в данной вершине по классам (в данном случае у нас три класса). Атрибут gini (индекс Джини) показывает загрязненность вершины. Формула для расчета:

$$G(v) = 1 - \sum_{k=1}^{l} \left(\frac{value_k(v)}{samples(v)} \right)^2 = 1 - \sum_{k=1}^{l} p_k^2,$$

где p_k — доли при распределении по классам образцов тренировочной выборки в данной вершине. Поскольку, очевидно, что дроби здесь правильные, а их сумма равна 1, то сумма их квадратов ≤ 1. Таким образом, $G(v) \in [0, 1]$, значение 0 соответствует чистой вершине (все образцы относятся к одному и тому же классу).

Упражнение 7. *Какое максимальное значение может принимать индекс Джини в задаче классификации по l классам?*

Информативность

Индекс Джини — это всего лишь один из вариантов. Для того, чтобы строить деревья, нам нужно заранее определиться с понятием информативности — оно для метода построения так же важно, как функция потерь для регрессии.

Будем обозначать через c значения признака Y. Для задач регрессии $c \in \mathbb{R}$, для задач классификации $c \in \{1, \ldots, l\}$. Если кроме указания класса–победителя требуется выдать баллы для каждого класса, то будем считать, что $c = (c_1, \ldots, c_l)$, где баллы отнормированы условием $\sum c_k = 1$. Зафиксируем некоторую функцию потерь L. Если мы хотим какую–то вершину дерева сделать листом, то логично приписать ей значение $H(v)$, которое минимизирует среднее значение функции потерь

$$H(v) = \min_{c \in Y} \frac{1}{samples(v)} \sum_{i: \vec{x}_i \in v} L(y_i, c)$$

(суммирование ведется по тем индексам, для которых вектор $\vec{x}_i$ тренировочной выборки попадает в вершину v при классификации).

Определение 3. *Величину $H(v)$ называют информативностью или загрязненностью вершины (impurity).*

Для задачи регрессии логично взять функцию потерь $L(y_i, c) = (y_i - c)^2$. Получаем задачу минимизации функции $\sum_i (y_i - c)^2$. Легко видеть, что здесь

$$c = \frac{\sum_{i: \vec{x}_i \in v} y_i}{samples(v)} = \overline{y}, \qquad H(v) = \sum_{i: \vec{x}_i \in v} \frac{(y_i - \overline{y})^2}{samples(v)} = D(v).$$

Итак, для квадратичной функции потерь, оценка для каждой вершины — это выборочное среднее значение по образцам, попавшим в вершину, а информативность — выборочная дисперсия по этим образцам.

Возьмем теперь функцию потерь $L(y_i, c) = |y_i - c|$. Получаем задачу минимизации функции $\sum_i |y_i - c|$.

Упражнение 8. *Проверьте, что точкой минимума является медиана $Med\{y_i : \vec{x}_i \in v\}$.*

Информативностью тогда оказывается абсолютное отклонение от медианы

$$H(v) = \sum_{i: \vec{x}_i \in v} \frac{|y_i - Med|}{samples(v)}.$$

Пусть теперь мы занимаемся мультиклассовой классификацией, причем предсказываем только номер класса, но не баллы по каждому классу. Простейший выбор функции потерь — это индикатор ошибки $L(y_i, c) = \begin{cases} 1, & \text{если } y_i \neq c, \\ 0, & \text{если } y_i = c. \end{cases}$. Тогда оптимальное c — это номер k самого частого в вершине v класса. Для информативности получаем

$$H(v) = \frac{\sum_{i: \vec{x}_i \in v} I_{y_i \neq k}}{samples(v)} = 1 - p_k.$$

Теперь предположим, что мы проводим мультиклассовую классификацию, причем предсказываем баллы $c_k(v)$ для каждого класса. Теперь метки $y_i = k$ нам придется кодировать векторами e_k (k–ый базисный орт с l координатами) — one–hot кодирование. Тогда логично использовать обычное расстояние между вектором $c = (c_k)$ и вектором $y_i = (y_{ik})$ (точнее квадрат расстояния). Получаем задачу на минимум

$$\sum_{i:\,\vec{x}_i \in v} \sum_{k=1}^{l} (c_k - y_{ik})^2 \to min.$$

Приравнивая к нулю все частные производные по c_k, получим ответ $c_k = p_k$. Тогда

$$H(v) = \frac{\sum_{i:\,\vec{x}_i \in v} \sum_{k=1}^{l} (p_k - y_{ik})^2}{samples(v)} = \sum_{k=1}^{l} p_k^2 - 2\sum_{k=1}^{l} p_k \frac{\sum_i y_{ik}}{samples(v)} + \frac{\sum_i \sum_{k=1}^{l} y_{ik}^2}{samples(v)} = \sum_{k=1}^{l} p_k^2 - 2\sum_{k=1}^{l} p_k^2 + 1$$

— получили тот самый показатель Джини.

Другая идея: в той же ситуации договоримся нормировать c_k условием $\sum_{k=1}^{l} c_k = 1$. Будем считать их вероятностями, а y_i — результатами независимых испытаний случайной величины c (т.е. проводятся испытания по схеме Бернулли). Тогда логично искать c_k максимизацией функции правдоподобия $\prod_i \prod_{k=1}^{l} c_k^{y_{ik}}$. Тогда логично взять $L(y_i, c)$ логарифм этой функции (с обратным знаком). Получаем задачу

$$-\sum_i \sum_{k=1}^{l} y_{ik} \ln c_k \to min, \quad \sum_{k=1}^{l} c_k = 1.$$

Составляем функцию Лагранжа, приравниваем к нулю производные и после несложных вычислений получаем $c_k = p_k$ (как и в предыдущем случае). Однако теперь

$$H(v) = \frac{-\sum_i \sum_{k=1}^{l} y_{ik} \ln p_k}{samples(v)} = -\sum_{k=1}^{l} p_k \ln p_k.$$

Полученная величина хорошо известна — это информационная энтропия Шеннона.

Построение дерева

Хорошо, пусть понятие информативности зафиксировано. Как построить дерево решений (по возможности, оптимальным образом)? Вначале продумаем, как построить пенек — дерево с одним узлом. Логично перебрать все признаки $1 \le j \le m$ и для каждого j найти оптимальный уровень t — такое число, чтобы взвешенная загрязненность двух новых вершин v_1 и v_2, которые получаться при разбиении с предикатом $x_{ij} < t$ была минимальной:

$$J(j,t) = \frac{samples(v_1)}{samples(v)} H(v_1) + \frac{samples(v_2)}{samples(v)} H(v_2) \to min.$$

Сложность решения такой задачи есть $O(m \cdot samples(v))$. Действительно, надо перебрать все m признаков и для каждого попробовать пороги t, порождающие разные разбиения (а

упорядоченное множество из $samples(v)$ элементов можно разбить на два подмножества с помощью неравенства ровно $samples(v) + 1$ способами). Еще придется для каждого разбиения посчитать два раза функцию H, а там сложность есть $O(samples(v))$ во всех случаях. Итого, получаем сложность $O(m \cdot samples^2)$.

Мы посчитали сложность одного шага построения дерева. К сожалению, построение полностью оптимального дерева — NP—полная задача, т.е. ее сложность растет по параметру n как $\exp(n)$. Самое логичное, что можно тогда предложить — жадный алгоритм. Будем для каждого узла построенного дерева проверять, не устраивает ли нас уже его загрязненность $H(v) < \varepsilon$. Если устраивает, то не будем делить этот узел. Если же нет, то разделим его оптимальным образом — сложность этого деления есть $O(m \cdot samples^2(v))$. И так далее. Несложно видеть, что общая сложность такого алгоритма есть $O(hn^2m)$, где h — высота дерева. Это немало, но приемлемо.

На самом деле, внимательный анализ показывает, что сложность легко снизить до $O(hmn \log n)$. Если согласиться на небольшую (обычно) потерю качества разбиения, то можно дойти до сложности $O(mn \log n + hmn)$, но это уже не очень существенно, поскольку высокие деревья выращивать неразумно — начинается переобучение.

Вообще надо отметить, что качество модели дерева решений очень сильно зависит от правильного подбора параметров. Основных параметров здесь три: высота дерева h (обычно ее приходится ограничивать сверху, чтобы избежать переобучения); уровень загрязненности ε, при достижении которого вершина уже не делится; минимальное число семплов на листе, при достижении которого вершина тоже далее не делится. Подбирать эти параметры на тренировочной выборке нельзя — начнется переобучение. Необходимо работать на валидационной выборке или применять кросс–валидацию. Поскольку параметров три, то подбирать оптимальное их сочетание вручную проблематично. Тогда применяют grid search — автоматический перебор параметров на сетке их возможных значений.

9 Бэггинг

Разложение ошибки на смещение и разброс = Bias–Variance Decomposition

Начнем с простого примера из параметрической статистики. Пусть вам надо оценить некоторый параметр θ, у которого есть истинное значение θ^*. Оценку вы проводите на основании выборки $\{x_i\}_{i=1}^n$ из распределения $X \sim p_{\theta^*}(x)$. На основании этой выборки вы составляете оценку вашего параметра $\hat{\theta}$ по некоторой формуле $\hat{\theta} = f(x_1, \ldots, x_n)$. Поскольку X — случайная величина, то и $\hat{\Theta}$ — тоже случайная величина (а результат ваших подсчетов — число $\hat{\theta}$ есть ее наблюдаемое значение). Дисперсия ошибки вашего прогноза равна $E\left[(\hat{\Theta} - \theta^*)^2\right]$. Матожидание берется по распределению $p_{\theta^*}(x)$, т.е. $E\left[(\hat{\Theta} - \theta^*)^2\right] = \int_{\mathbb{R}^n}(f(t_1, \ldots, t_n) - \theta^*)^2 \prod_{i=1}^n p_{\theta^*}(t_i)dt_i$, как и все матожидания далее.

У вашей случайной величины $\hat{\Theta}$ тоже есть матожидание, которое я обозначу $\overline{\theta} = E[\hat{\Theta}]$. Хорошо, когда $\overline{\theta} = \theta^*$ — тогда ваша оценка называется несмещенной, а иначе разность $\overline{\theta} - \theta^* = Bias$ — ее смещение. Еще у вашей оценки есть вариация $V = E\left[(\hat{\theta} - \overline{\theta})^2\right]$. Хорошая оценка должна иметь вариацию поменьше. Например, оценивая матожидание нормального распределения, принято брать выборочное среднее. А ведь можно было бы взять, например, просто первый элемент выборки — легко проверить, что эта оценка несмещенная. Однако так не делают, поскольку у второй оценки большая вариация, а у первой маленькая. Ну и для самых прилежных — из курса статистики вы возможно помните, что вариацию нельзя сделать меньше грани Крамера–Рао $V \geq Cn^{-1}$, где константа C есть информационная константа Фишера, которая явным образом вычисляется по функции p_{θ^*}.

Так вот, разложение ошибки (точнее, дисперсии ошибки) на вариацию и квадрат смещения имеет вид

$$E\left[(\hat{\Theta} - \theta^*)^2\right] = E\left[((\hat{\Theta} - \overline{\theta}) + (\overline{\theta} - \theta^*))^2\right] = E\left[(\hat{\Theta} - \overline{\theta})^2\right] + \\ + 2(\overline{\theta} - \theta^*)E\left[\hat{\Theta} - \overline{\theta}\right] + (\overline{\theta} - \theta^*)^2 = V + Bias^2.$$

Представим теперь, что мы собрали выборку объема kn. Давайте разобьем ее на k подвыборок объема n, построим наши оценки $\hat{\theta}_j$ по каждой подвыборке, а затем общую оценку соберем как среднее арифметическое $\hat{\theta} = \frac{1}{k}\sum_{j=1}^k \hat{\theta}_j$. Посмотрим, какое будет смещение

$$Bias(\hat{\Theta}) = E[\hat{\Theta}] - \theta^* = E\left[\frac{1}{k}\sum_{j=1}^k \hat{\Theta}_j\right] - \theta^* = E[\hat{\Theta}_j] - \theta^* = Bias(\hat{\Theta}_j),$$

где j произвольно. Итог: смещение не изменилось. Теперь посчитаем вариацию

$$V\left[\hat{\Theta}\right] = V\left[\frac{1}{k}\sum_{j=1}^k \hat{\Theta}_j\right] = \frac{1}{k^2}\sum_{j=1}^k V[\hat{\Theta}_j] = \frac{V[\hat{\Theta}_j]}{k}.$$

Вариация уменьшилась в k раз (мы использовали факт независимости данных в выборке)!

То же самое верно и для наших моделей. Возьмем, например, регрессию. Пусть для простоты x, $y \in \mathbb{R}$, $y = f(x) + \varepsilon$, где f — детерминированная функция, а ε — случайная величина (шум) со свойствами $E\varepsilon = 0$, $V\varepsilon = E\varepsilon^2 = \sigma^2$. Квадратичная функция потерь на одной тестовой точке дает ошибку

$$MSE = (y(x) - \widehat{y}(x))^2, \qquad \widehat{y}(x) = \widehat{y}(x, X),\ X = ((x_1, y_1), \ldots, (x_l, y_l)), \qquad y(x) = y(x, \varepsilon).$$

Мы учли, что значение таргета зависит не только от переменной x, но и от шума ε. Кроме того, учли, что наша модель обучалась вначале на блоке данных X. Тогда с вероятностной точки зрения функция потерь равна

$$Q(\widehat{y}) = E_x E_{X,\varepsilon}(y(x, \varepsilon) - \widehat{y}(x, X))^2$$

— вначале мы учим нашу модель, а для ответа берем усреднение; итоговое качество мы оцениваем средним по тестовой выборке. Преобразуем внутреннее матожидание:

$$E_{X,\varepsilon}(y(x, \varepsilon) - \widehat{y}(x, X))^2 = E_{X,\varepsilon}(f(x) - \widehat{y}(x, X))^2 + 0 + E_\varepsilon(\varepsilon^2) = E_X(f(x) - \widehat{y}(x, X))^2 + \sigma^2$$

— мы учли, что с.в. X и ε независимы, откуда $E_{X,\varepsilon} = E_X E\varepsilon$, а $E_\varepsilon(\varepsilon) = 0$. Теперь повторим прием, показанный выше:

$$E_X(f(x) - \widehat{y}(x, X))^2 = E_X(f(x) - E_X\widehat{y} + E_X\widehat{y} - \widehat{y}(x, X))^2 = (f(x) - E_X\widehat{y}(x, X))^2 + V_X(\widehat{y}(x, X)).$$

Получаем итоговое разложение

$$Q(\widehat{y}) = [bias_X \widehat{y}(x, X)]^2 + V_X(\widehat{y}(x, X)) + \sigma^2.$$

Отсюда мораль: если мы наши данные разобьем на k массивов данных, проведем построение базовой модели на каждом, а затем усредним результаты, то мы не изменим смещение, но в k раз уменьшим вариацию. Такой метод называют pasting.

Бэггинг

Может показаться, что мы изобрели волшебный способ улучшать качество моделей. Это не так. Наше улучшение липовое, поскольку для снижения вариации в k раз пришлось в k раз увеличить выборку. Обратите внимание, что смещение вообще нельзя уменьшить, собрав побольше данных. Более того, нас обычно интересует средняя ошибка модели, т.е. корень из дисперсии, а он вообще уменьшится только в $\sqrt{k}$ раз.

Однако, можно предложить модификацию нашего метода — давайте не разбивать выборку на k частей, а извлекать из выборки наборы образцов (причем извлекать с возвращением). Затем будем проделывать все то же — строить базовую модель, а потом ответы усреднять. Такой метод называют бэггинг (bagging = bootstrap aggregating). Теперь, однако, мы не можем гарантировать уменьшения вариации в k раз, поскольку выборки стали зависимыми (данные в них стали повторяться), а значит, стали зависимыми наши случайные величины $\hat{\Theta}_j$. Тогда давайте проведем еще одну модификацию: будем получать оценки на подвыборках разными методами. Это обратно

повысит независимость наших оценок и вариация получит шанс все–таки уменьшится в k раз. Точного коэффициента уменьшения, конечно, не оценить — степень зависимости разных методов величина эфемерная.

Агрегированный прогноз вырабатывают довольно просто. В задачах регрессии вместо финального усреднения параметров усредняют прогнозы каждого метода (для линейных регрессий это, очевидно, одно и то же). В задачах классификации возможны варианты. Если классификаторы нашего ансамбля умеют только определять номер класса, то вариант один — устроить голосование членов ансамбля. Если же кроме номера класса мы имеем доступ к баллам по каждому классу, то лучше устроить мягкое голосование: привести баллы в единую шкалу масштабированием, затем их усреднить, а затем уже выбрать класс — победитель. Такую процедуру называют мягким голосованием (soft voting) и она обычно эффективнее обычного голосования, поскольку придает больший вес голосам с высоким доверием.

У бэггинга есть еще два приятных свойства. Во–первых, обучение моделей можно проводить параллельно, что очень важно для задач с большими данными. Во–вторых, при выборке с повторениями у нас будут оставаться ни разу не выбранные образцы, из которых потом логично составить тестовую выборку (внимание! не общую, а свою для каждого метода). Значит, можно не отщеплять тестовую выборку заранее и не заниматься кросс–валидацией — это повышает число тренировочных образцов. Может показаться, что невыбранные образцы — маловероятное явление. Это не так. Пусть общий объем выборки n, и мы провели выборку с повторениями объема n. Тогда вероятность того, что данный образец на не попался ни разу, равна $(1 - n^{-1})^n \approx e^{-1} \approx 0.37$.

Случайный лес

Прием бэггинга отлично показал себя на решающих деревьях. Дело в том, что здесь достаточно легко сделать алгоритмы (решающие деревья), работающие на разных подвыборках, не похожими друг на другу. Как уже было сказано, это понижает вариацию прогнозов. Вот алгоритм выращивания случайного леса.

1. Строим дерево номер i, пока $i \leq k$, где k (число деревьев в лесу) определено заранее.
   (a) Из общей выборки X выбирается подвыборка X^i того же размера, что X, но с повторениями.
   (b) При обучении дерева в каждой вершине проводится не полный перебор признаков $1 \leq j \leq m$, а выбирается случайным образом некоторый поднабор из $m' < m$ признаков (фактически, получается, что мы проектируем выборку X^i на случайным образом выбранное координатное подпространство). Оптимальный сплит ищется только среди них.
2. Теперь, когда деревья построены, вырабатываем общий прогноз: для регрессии усредняем ответы, для классификации устраиваем голосование (жесткое или мягкое — оно для деревьев возможно, см. прошлую лекцию).

Заметим, что у случайного леса есть несколько очень важных гиперпараметров: число деревьев k, высота деревьев h и размерность проектора m'. Поговорим о выборе этих

параметров, начнем с параметра h. Невысокие деревья не улавливают тонкостей обучающей выборки. Попробуйте сами загадать известного человека и предложите своему другу отгадать, задавая вопросы, на которые можно ответить только «да» или «нет». Скорее всего, первые вопросы будут примерно такими: «Это мужчина или женщина?», «Этот человек существовал в реальности?», «Он уже умер?», «Он говорит по–русски?». То есть за небольшое число сплитов алгоритм успевает выучить только статистики верхнего уровня. Измените тестовые данные, и вы увидите, что предсказание такого дерева будет достаточно стабильным (продолжая пример, вы можете заменить персонажа, а первые ответы почти не изменятся). Это означает, что *низкие деревъя имеют малую вариацию*. Но ведь именно ее мы хотим уменьшать с помощью бэггинга! Значит, нам надо выращивать высокие деревья. Действительно, у них вариация высокая — переобучение, к которому приходят все высокие деревья, в том и заключается, что дерево хорошо выучивает тестовую выборку, а на других выборках дает большой разброс в прогнозах. Вывод: в случайном лесу надо выращивать высокие деревья. Эмпирическое правило такое: в задачах классификации строить максимально высокие деревья — пока на каждом листе не окажется по одному объекту; в задачах регрессии — пока на каждом листе не окажется по пять объектов.

Второй вопрос — как выбрать параметр m'. Чем больше m' — тем больше корреляция между деревьями, а значит, тем меньше эффект бэггинга. С другой стороны, малое m' ведет к низкому качеству деревьев. Есть хорошее эмпирическое правило: $m' \approx \sqrt{m}$ в задачах классификации, а для задач регрессии: $m' \approx m/3$.

Третий вопрос — сколько деревьев выращивать в лесу? Здесь все просто — при $k \to \infty$ вариация оценки стремится к нулю. Однако остается ошибка смещения (на самом деле, остается еще ошибка случайного шума, просто в начале лекции я не вводил его в рассмотрение). Поэтому разумно увеличивать число k, проверяя качество на валидационной выборке, до тех пор, пока это качество не перестанет уменьшаться. Вероятность того, что образец не попал на обучение случайному лесу, равна $(1 - n^{-1})^{kn} \approx e^{-k} \to 0$ при увеличении k. Таких объектов совсем мало, чтобы составить из них выборку для оптимизации числа k. Однако, можно поступить хитрее — откажемся вообще от принципа — валидационная выборка не участвует в обучении. При валидации будем прогонять метод по всем образцам обучающей выборки, но ошибку модели для каждого образца будем вычислять с помощью прогноза, построенного только по тем деревьям, которые при обучении не использовали данный образец:

$$OOB = \sum_{i=1}^{n} L(y_i, \hat{y}_i) = \sum_{i=1}^{n} L\left(y_i, \frac{\sum_t \hat{y}_t(\vec{x}_i)}{T(\vec{x}_i)}\right).$$

Еще раз: суммирование в формуле идет только по тем деревьям $t \in [1, k]$, которые при обучении не использовали образец $\vec{x}_i$; в знаменателе стоит число таких деревьев $T(\vec{x}_i)$; через $\hat{y}_t(\vec{x}_i)$ обозначен прогноз дерева с номером t на образце $\vec{x}_i$. Такая ошибка называется out–of–bag–error.

Упражнение 9. *Проверьте, что математическое ожидание случайной величины T равно $k(1 - n^{-1})^n \approx 0.37k$, т.е. прогноз $\hat{y}_i$ в OOB вполне репрезентативен.*

Стекинг и блендинг

Если вы используете действительно «сильно разные» алгоритмы (например, регрессия, SVR и решающее дерево), то можно вообще не проводить выборку, а просто подать каждому алгоритму весь набор данных. Тогда говорят, что используется стекинг методов. При стекинге агрегирование результатов проводят не обязательно с помощью усреднения, а подают эти результаты на вход еще одной финальной модели (ее называют мета–модель). Приведем алгоритм обучения с использованием стекинга.

1. Делим общую выборку на тренировочную, валидационную и тестовую.
2. Тренировочную выборку делим на k фолдов.
3. Для каждого фолда фиксируем некоторую модель, обучаем ее на всей тренировочной выборке за исключением данного фолда, а затем выдаем с помощью нее прогнозы для каждого образца из нашего фолда.
4. Полученные прогнозы (они теперь доступны для каждого образца тренировочной выборки) подаем для обучения мета–модели. В мета–модель можно в качестве признаков подавать и какие–то исходные признаки.
5. Валидируем мета–модель на валидационной выборке, тестируем на тестовой.

Если данных достаточно много, то тренировочную выборку можно поделить на две. На первой учить k моделей, а на второй учить мета–модель с помощью прогноза уже построенных моделей. Такой метод называют блендингом. При блендинге разумно подавать в мета–модель не только прогнозы базовых моделей, но и наиболее важные исходные признаки. Например, для решающего дерева можно рассчитать feature importance для каждого признака по формуле

$$R_j(T) = \frac{\sum_{v \in nodes(T)} Imp_v \cdot I_{B_v = B_v(X_j)}}{\sum_{v \in nodes(T)} Imp_v},$$

$$Imp_v = \frac{samples(v)}{n} H(v) - \frac{samples(v_1)}{n} H(v_1) - \frac{samples(v_2)}{n} H(v_2).$$

Суммирование здесь проходит по внутренним вершинам v дерева T, т.е. тем вершинам, для которых определен разделяющий предикат B_v. Далее, для суммирования в числителе дроби мы выделяем ровно те внутренние вершины, предикат которых использует для разделения признак X_j. Суммируем мы величины Imp_v — важность вершины v. Эту важность мы подсчитываем как уменьшение загрязненности — вместо $H(v)$ после деления вершины стало $H(v_1)+H(v_2)$, но загрязненность при этом, берем с весами — относительное число образцов для вершин. Собственно, легко видеть, что при построении дерева мы выбираем предикат сплита в вершине v, максимизируя Imp_v. Легко видеть, что $\sum_{j=1}^{m} R_j = 1$, так что теперь мы можем видеть, какой вклад вносят те или иные признаки в уменьшение загрязненности. Теперь общую важность признака проводим усреднением $R_j(T)$ по всем k деревьям T в нашем лесу.

10 Бустинг

Общий градиентный бустинг

Классический пример бустинга — игра в гольф (кажется, этот пример приводят все учебники:). Действительно, нанося каждый новый удар, мы пытаемся уменьшить погрешность, которая получилась в результате предыдущих ударов.

Опишем бустинг для регрессии. Пусть мы хотим построить модель $\hat{y}(\vec{x})$, минимизируя квадратичную функцию потерь $L = \sum_{i=1}^{n}(y_i - \hat{y}(\vec{x}_i))^2$. Предположим, мы построили (неважно пока что, каким методом) первую модель (их сейчас будет много, они называются базовыми моделями, так что давайте их обозначим $b_k(\vec{x})$) $b_1(\vec{x})$. Давайте теперь посчитаем остатки $r_i^{(1)} = y_i - b_1(\vec{x}_i)$ и построим новую модель $b_2(\vec{x})$, которая будет предсказывать уже эти самые остатки, т.е. решим задачу $\sum_{i=1}^{n}(r_i^{(1)} - b_2(\vec{x}))^2 \to min$. Теперь наша новая модель: $b_1(\vec{x}) + b_2(\vec{x})$, а новые остатки равны $r_i^{(2)} = y_i - (b_1(\vec{x}_i) + b_2(\vec{x}_i))$. Продолжим, и решим задачу $\sum_{i=1}^{n}(r_i^{(2)} - b_3(\vec{x}))^2 \to min$, и так далее. Финальная модель (когда функция потерь стала достаточно малой, или число k стало уже неприлично большим) вычисляется как сумма $\hat{y}(\vec{x}) = \sum_{l=1}^{k} b_l(\vec{x})$.

Возникает резонный вопрос — почему бы нам сразу не построить суммарную модель? Честно говоря, так и произойдет в самом классическом случае — когда мы ищем линейную параметрическую регрессию $\hat{y} = \sum_{j=0}^{m} \theta_j X_j$ (если пренебречь ошибкой вычислений коэффициентов модели). Действительно, если мы правильно минимизировали функцию потерь при поиске модели b_1, то оптимальная модель уже получена. Тогда все остальные функции $b_j(\vec{x})$ будут тождественно нулевыми. В такой ситуации бустинг — бессмысленное дело (точнее, он просто является градиентным спуском).

Попробуем идею бустинга для других известных нам методов (например, для логистической регрессии). Вначале порассуждаем абстрактно. Заметим, что остатки, которые мы использовали выше

$$r_i^{(k)} = y_i - a_k(\vec{x}_i) = -0.5 \left.\frac{\partial (z_i - y_i)^2}{\partial z_i}\right|_{z_i = a_k(\vec{x}_i),} \qquad a_k(\vec{x}_i) = \sum_{l=1}^{k} b_l(\vec{x}_i),$$

равны антиградиенту квадратичной функции потерь, который надо вычислять в точке предсказания. Тогда проглядывается обобщение нашего метода. На каждом шаге мы делаем движение в сторону уменьшения градиента функции потерь. Это означает, что «остатки» $r_i^{(k)}$ для следующей модели надо брать пропорциональными производным $-\frac{\partial L}{\partial z_i}|_{z_i = a_k(\vec{x}_i)}$. Кстати, не обязательно делать шаги, строго равные длине вектора градиента. Введем в наш метод длины шагов γ_l, т.е. после k шагов будем вычислять модель как $a_k(\vec{x}) = \sum_{l=1}^{k} \gamma_l b_l(\vec{x})$. Получаем следующий алгоритм (он называется gradient boosting — градиентный бустинг).

1. Вычислим «нулевую модель» $b_0(\vec{x})$. В задачах регрессии можно, например, взять $b_0 \equiv 0$ или $b_0 \equiv \overline{y_i}$. В задачах классификации можно взять $b_0 \equiv mod(y_i)$ (номер самого популярного класса).

2. Предположим, что модель $a_{k-1}(\vec{x}) = \sum_{l=0}^{k-1} \gamma_l b_l(\vec{x})$ уже построена. Мы хотим подобрать числа $b_k(\vec{x}_i)$ и множитель γ_k так, чтобы минимизировать

выражение $\sum_{i=1}^n L(y_i, a_{k-1}(\vec{x}_i) + \gamma_k b_k(\vec{x}_i))$. Выберем в качестве таргетов «остатки» $R^{(k)} = r_i^{(k)} = -\frac{\partial L(y_i,z)}{\partial z}|_{z=a_{k-1}(\vec{x}_i)}$, которые мы нормируем (разделим на корень из суммы их квадратов). Теперь построим модель $b_k(\vec{x})$, по данным $(X, cR^{(k)})$. Построение будем вести минимизацией суммы квадратов (почему, объясним ниже), т.е.

$$b_k(\vec{x}) = \arg\min_{b\in\mathcal{A}} \sum_{i=1}^n (b(\vec{x}_i) - cr_i^{(k)})^2,$$

где $\mathcal{A}$ — некоторое множество базовых алгоритмов, которое мы решили использовать в нашей задаче (например, логистическая регрессия). После того, как функция $b_k(\vec{x})$ найдена, оптимизируем длину шага:

$$\gamma_k = \arg\min_{\gamma\in\mathbb{R}} \sum_{i=1}^n L(y_i, a_{k-1}(\vec{x}_i) + \gamma b_k(\vec{x}_i)).$$

Теперь, если требуемая точность не достигнута, или если мы не хотим прервать обучение модели, перейдем к построению слагаемого $\gamma_{k+1}b_{k+1}(\vec{x})$.

Отметим еще раз: невзирая на функцию потерь L для построения $b_k(\vec{x})$ применяется метод наименьших квадратов. Дело в том, что остатки $r_i^{(k)}$ не имеют никакого жизненного или вероятностного смысла — это просто вектор в пространстве $\mathbb{R}^n$. Мы хотим приблизить направление этого вектора, а значит должны минимизировать угол φ между векторами или, что тоже самое, максимизировать косинус этого угла. Но поскольку векторы нормированы (точнее, вектор таргетов нормирован, а вектор модели к нему близок, и потому почти нормирован), то по теореме косинусов квадрат расстояния между векторами равен $2(1 - \cos\varphi)$. Впрочем, особо педантичные люди используют именно косинусное расстояние между векторами.

Таким образом, нам остается только подавать на вход формулы для градиента ∇L. Например, для логистической регрессии (для случая двух классов с метками $y_i \in \{\pm 1\}$) имеем

$$L(y_i, z_i) = \sum_{i=1}^n \ln(1 + e^{-y_i z_i}), \qquad \nabla L = \left(\frac{-y_i e^{-y_i z_i}}{1 + e^{-y_i z_i}}\right)_{i=1}^n.$$

Отметим интересную возможность ускорения для конкретно этой ситуации. Если $z_i = a_k(\vec{x}_i)$ имеет тот же знак, что и y_i, и слегка отделена от нуля, то экспонента в знаменатели дроби мала по сравнению с единицей. Тогда дробь можно приблизительно заменить на $-y_i e^{-y_i z_i}$. Если же знаки $a_k(\vec{x}_i)$ и y_i противоположны, а величина $a_k(\vec{x}_i)$ вновь отделена от нуля, то дробь примерно равна $-y_i$.

Упражнение 10. *Выпишите ∇L для функции потерь (минус логарифм функции правдоподобия) многоклассовой регрессии.*

У градиентного бустинга есть две проблемы. Если наши базовые алгоритмы будут слишком простыми, то они будут приводить нас в точки, далекие от целевой, а в этих точках вектор градиента может сильно промахиваться мимо цели. Тогда наша модель с ростом k будет метаться, не попадая в цель. Обратная проблема — слишком

сложные базовые алгоритмы. Здесь срабатывает принцип «слишком хорошо — тоже плохо». Представьте, что преподаватель дает задачу, а какой–то ученик сразу хорошо ее решает. В результате он не дает обучиться другим ученикам — им не хватает задач. Так и здесь, если базовые алгоритмы переобучаются на тестовой выборке, то метод сходится слишком быстро и эффекта бустинга не возникает. После этого на валидационной выборке такой метод показывает результаты хуже, чем хорошо сбалансированный бустинг.

Обе проблемы лечатся одним приемом — введением скорости обучения $\eta \in (0, 1]$. Помещая этот коэффициент перед оптимальным коэффициентом γ мы будем замедлять наш метод. В результате первая проблема будет решена, так как мы запретим «резкие прыжки», а вторая будет решена, поскольку мы не дадим «слишком умному» базовому алгоритму быстро решить задачу.

GBDT = gradient boosting on decision trees

Прием бустинга особенно хорошо себя показал на решающих деревьях. Он настолько хорош, что даже сейчас на некоторых задачах побеждает нейронные сети. Например, с помощью GBDT работает алгоритм ранжирования Яндекса. У этого метода есть свои особенности.

Вспомним, что дерево решений разбивает все пространство $\mathbb{R}^m$ на непересекающиеся области, для каждой из которых предлагает ответ в виде константы (это верно и для задач классификации, и для регрессии). Пусть модель $a_k = \sum_{l=1}^{k} \gamma_l b_l$ уже построена, а мы добавляем дерево b_{k+1} с N листьями. Тогда у нас появляются набор $\Omega_\nu^{(k)}$ областей, где $\nu = 1, \ldots, N$, и константы $\omega_\nu^{(k)}$ — ответы дерева b_{k+1} для каждой области. Получается, что функция $a_{k+1}(\vec{x})$ есть сумма функции $a_k(\vec{x})$ и $\gamma_{k+1} \sum_{\nu=1}^{N} \omega_\nu^{(k)} I_{\Omega_\nu^{(k)}}(x)$. Видно, что в этой формуле множитель γ_{k+1} излишен. Получается, что задача добавления $k+1$–го дерева равносильна задаче оптимизации

$$\sum_{i=1}^{n} L\left(y_i, a_k(\vec{x}_i) + \sum_{\nu=1}^{N} \omega_\nu I_{\Omega_\nu}\right) \xrightarrow{\omega_\nu, \Omega_\nu} min.$$

Метод MART = multivariate additive regression trees предлагает следующий алгоритм. Фиксируем функцию потерь L, строим $k+1$–е дерево решений стандартным образом, затем оставляем области Ω_ν, но забываем ответы дерева ω_ν; эти ответы подбираем из последней оптимизационной задачи. Решать такие оптимизационные задачи уже не так страшно, поскольку области $\Omega_\nu^{(k)}$ не пересекаются, так что оптимизация сводится к решению N независимых задач

$$\sum_{i:\, \vec{x}_i \in \Omega_\nu} L\left(y_i, a_k(\vec{x}_i) + \omega\right) \xrightarrow{\omega} min, \qquad \nu = 1, \ldots, N.$$

Например, для квадратичной функции потерь получаем задачу $\sum_{i:\, \vec{x}_i \in \Omega_\nu} (y_i - a_k(\vec{x}_i) - \omega)^2 \to min$. Ее решение легко найти (дифференцированием): $\omega_\nu = \dfrac{\sum_{i:\, \vec{x}_i \in \Omega_\nu} (y_i - a_k(\vec{x}_i))}{\#\{i :\, \vec{x}_i \in \Omega_\nu\}}$. Для функции потерь $L(y_i, z_i) = |y_i - z_i|$ задача тоже решается явно. Проверьте сами, что

здесь ω_ν есть медиана набора чисел $\{y_i - a_k(\vec{x}_i)\}_{i:\,\vec{x}_i\in\Omega_\nu}$. А вот для логистической функции L аналитическое решение получить невозможно. Здесь получается задача

$$\sum_{i:\,\vec{x}_i\in\Omega_\nu} \ln\left(1+e^{-y_i a_k(\vec{x}_i)+\omega}\right) \stackrel{\omega}{\to} min \qquad \Longleftrightarrow \qquad \sum_{i:\,\vec{x}_i\in\Omega_\nu} \frac{p_i}{1+\xi\cdot p_i} = 0,\ \xi = e^{\omega},\ p_i = e^{-y_i a_k(\vec{x}_i)}.$$

На практике, впрочем, речь обычно идет о малых поправках, т.е. $\omega \approx 0$. В таком случае дроби можно разложить по малому параметру ω и получить приближенное решение

$$\omega \approx \frac{\sum_{i:\,\vec{x}_i\in\Omega_\nu} \frac{p_i}{1+p_i}}{\sum_{i:\,\vec{x}_i\in\Omega_\nu} \left(\frac{p_i}{1+p_i}\right)^2}.$$

Понятное дело, важно то, какой высоты деревья мы будем выращивать. Некоторые другие особенности деревьев тоже влияют на скорость методов. Классически сложились три разные подхода, которые в последнее время обменялись идеями и сейчас не очень уже сильно различаются. Но, тем не менее, принципы построения деревьев у них разные.

Основной принцип метода LightGBM — на каждом шаге находим сплит, деление которого даст максимальное снижение потерь. Критерий остановки: достижение максимального разрешенного числа листов. Деревья получаются несимметричными — например от корня идет левая ветка глубины 2, а глубина правой 15. Чтобы избежать переобучения, алгоритм делает еще одно ограничение на высоту дерева. Алгоритм XGBoost ограничивает не число листов, а высоту дерева (сверху, разумеется) и число объектов на каждом листе (снизу, чтобы избежать малочисленных листов). Деревья получаются почти сбалансированные. Алгоритм CatBoost строит только сбалансированные деревья, ограничивая только их высоту. Кроме того, на каждом уровне дерева все сплиты проводятся по одному и тому же предикату (одинаковый признак и одинаковый уровень). Это сильно ускоряет применение модели. В остальном, каждый из методов способен применять разные дополнительные тюнинги, перечисленные ниже.

1. Регуляризация: к функционалу потерь добавляются слагаемые γJ, где J — число листов дерева, и/или $\lambda \sum \omega_\nu^2$.

2. Рандомизация: как в случайном лесу, при поиске сплита выбирается случайный поднабор семплов и/или случайный набор признаков.

3. Квадратичное приближение вместо линейного: наилучшим направлением при бустинге берется не градиент, а решение квадратного уравнения, в котором член второго порядка появляется из разложения Тейлора функции потерь до степени 2.

4. Пресортировка образцов и разные другие мелкие (но иногда важные) детали.

Адаптивный бустинг = Adaboost

Это альтернативная идея бустинга. Представьте, что учитель дает ученику на дом набор заданий на разные приемы. Ученик приносит свою работу, и учитель видит, что некоторый

прием у ученика не получился. Логично, что учитель обращает на это внимание ученика и заставляет того отработать данный прием. На этой идеи и основан Adaboost.

Возьмем данные и обучим первую базовую модель b_1 (для примера, пусть это обычная линейная регрессия). Посмотрим на остатки $r_i^{(1)}$ и запустим обучение модели b_2, где функция потерь будет весовой $L = \sum_{i=1}^{n} L(y_i, b_2(x_i))|r_i^{(1)}|$, т.е. большим остаткам отвечает большой вес — модель будет стараться исправиться на данных семплах. Получим новый набор остатков $r_i^{(2)}$ и используем их как веса для третьей модели. И так далее. Когда обучено достаточно много моделей, то устроим на них ансамбль. Например, можно просто взять взвешенную сумму их прогнозов, причем голоса учитывать с весами — в зависимости от того, насколько хорошо модели обучились. Для нашего примера с линейной регрессией можно взять веса $\alpha_k = R^2_{det}$.

11 Нейронные сети — архитектура

Перцептрон Розенблатта

Этот перцептрон был описан Фрэнком Розенблаттом еще в 1950-х годах. Пусть нам поставлена задача бинарной классификации, причем метки классов 1 и -1. Наши данные такие же, как и выше: матрица X размера $n \times m$ и вектор меток Y. На современном языке, перцептрон Розенблатта — это линейная модель, т.е. все данные она разделяет линейной гиперплоскостью в пространстве $\mathbb{R}^m$ признаков. Конкретнее, наша модель будет иметь вид

$$\hat{y} = \operatorname{sign}(\theta_0 + \theta_1 x_1 + \ldots + \theta_m x_m) = \operatorname{sign}(\vec{\theta} \cdot \vec{x}^T).$$

Как и раньше, наши семплы — это строки, мы добавляем для удобства обозначений признак $X_0 \equiv 1$, а вектор $\vec{\theta}$ — вектор из коэффициентов $(\theta_0, \ldots, \theta_m)$, который мы и ищем. Новое для нас — это выбор функции потерь

$$\mathcal{L}_P(\vec{a}) = \sum_{i=1}^{n} \mathcal{L}_i(\vec{\theta}), \qquad \text{где } \mathcal{L}_i(\vec{\theta}) = \begin{cases} 0, & \text{если } y_i = \hat{y}_i, \\ |\theta_0 + \theta_1 x_{i1} + \ldots + \theta_m x_{im}|, & \text{если } y_i \neq \hat{y}_i. \end{cases}$$

Если ввести обозначение $y_i^* = \theta_0 + \theta_1 x_{i1} + \ldots + \theta_m x_{im}$ (мы уже говорили, что эту переменную называют ненаблюдаемой, латентной), то выражение во второй строке есть $-y_i y_i^*$.

Естественно, мы стараемся подобрать такие коэффициенты модели, чтобы $\mathcal{L}_P$ была минимальной. Иными словами, мы минимизируем суммарное отклонение наших ответов от гиперплоскости, но смотрим только на неверные ответы — верный ответ не вносит ничего в функцию ошибки. Функция $\mathcal{L}_P$ дифференцируема (за исключением точек на самой разделяющей гиперплоскости, но это не страшно — погрешность вычислений никогда не даст нам попасть на нее в точности), так что для минимизации можно применять обычный градиентный спуск. Сам Розенблатт предлагал для обучения применять метод, который мы теперь называем спуском на мини–батчах. Алгоритм такой: мы последовательно проходим семплы из обучающей выборки: $\vec{x}_1$, $\vec{x}_2$ и т.д. Если очередной семпл $\vec{x}_i$ модель классифицирует верно, то коэффициенты a_j не меняются. Если же неверно, то делается поправка

$$\vec{\theta}_{new} = \vec{\theta}_{old} - \eta \operatorname{sign}(\vec{\theta}_{old} \cdot \vec{x}_i^T) \cdot \vec{x}_i.$$

Несложно видеть, что эта поправка есть $-\eta \cdot \nabla \mathcal{L}_i(\vec{\theta}_{old})$, так что получаем типичный градиентный спуск. Множитель η отвечает за скорость обучения. Описанное правило так и называют — *правило обучения перцептрона.*

Современный перцептрон

Из перцептронов Розенблатта не удастся создать содержательную нейронную сеть (это станет скоро понятно). Поэтому усложним модель — добавим *функцию активации.* Самая классическая функция для задач бинарной классификации — это логистическая кривая $\sigma(\xi) = \frac{1}{1+e^{-\xi}}$. Теперь наша модель принимает вид

$$\hat{y} = \sigma(\theta_0 + \theta_1 x_1 + \ldots + \theta_m x_m) = \sigma(\vec{\theta} \cdot \vec{x}^T),$$

т.е. это самая обычная логистическая регрессия. Обучается она тоже как обычно, функция потерь тоже стандартная

$$\mathcal{L}(\vec{\theta}) = \sum_{y_i=1} \ln(1 + e^{-y_i^*}) + \sum_{y_i=-1} \ln(1 + e^{y_i^*}) = \sum_{i=1}^{n} \ln\left(1 + e^{-y_i y_i^*}\right).$$

Обратите внимание, что если $y_i \neq \operatorname{sign} y_i^*$, то наше выражение есть $\ln(1 + e^{|y_i^*|})$, что при больших y_i^* примерно равно $|y_i^*|$ — как у Розенблатта. Если же $y_i = \operatorname{sign} y_i^*$, то получаем $\ln(1 + e^{-|y_i^*|})$, что при больших y_i^* примерно равно нулю — снова как у Розенблатта.

Функция активации $\sigma(\xi)$ (сигмоид — так ее называют) не единственная, которую используют (и сейчас не самая популярная). Поскольку из перцептронов будет собрана большая сеть, то функция активации должна, во–первых быть достаточно простой, чтобы ее можно было вычислить быстро. Во–вторых, у нее должна быть столь же легко вычисляемая производная. Для сигмоида имеем $\sigma'(\xi) = \sigma(\xi)(1 - \sigma(\xi))$, т.е. вычислив сам сигмоид, можно очень быстро найти производную. Очень похож по своим свойствам на сигмоид гиперболический тангенс $\operatorname{th}\xi = \frac{e^{2\xi}-1}{e^{2\xi}+1}$. Он тоже гладкий, а его производная тоже легко считается: $(\operatorname{th}\xi)' = 1 - \operatorname{th}^2\xi$.

Самая часто используемая сейчас функция активации

$$ReLU(\xi) = \begin{cases} 0, & \text{если } \xi < 0, \\ \xi, & \text{если } \xi \geq 0 \end{cases} = \max\{0, \xi\} = \xi_+.$$

Ее начали применять позже. Точнее, дело было так — ее начали применять в 1980-х, а потом забросили, и снова нашли только в 2010-х. Ее первое преимущество перед сигмоидом — скорость вычислений. Чтобы вычислить саму функцию, надо только одно сравнение — переменная y_i^* больше нуля (оставляем y_i^*) или нет (выдаем 0). Чтобы вычислить производную $ReLU'(\xi)$ тоже нужно только одно сравнение. Это сильно отличается от вычисления экспоненты для сигмоида.

Второе преимущество в том, что у ReLu, в отличие от сигмоида не происходит насыщения. Сигмоид все–таки, слишком похож на ступеньку. Действительно, $\sigma(0) = 0.5$, $\sigma(1) = 0.73$, $\sigma(2) = 0.88$, $\sigma(3) = 0.95$. Иными словами, все что больше, чем 3, сигмоид считает фактически единицей — различия стираются. А мы, ведь, собираемся передавать ответы одних перцептронов другим. Такая неразборчивость сигмоида будет останавливать обучение нейронной сети. Попробуем избавиться от этого эффекта — возьмем функцией активации сумму двух сигмоидов $\sigma(x)+\sigma(x-1)$. Для этой функции насыщение наступает где–то после точки $x = 4$: $\sigma(4)+\sigma(3) \approx 1.92 \approx 2 = \sigma(+\infty)+\sigma(+\infty)$. В попытке избавиться от насыщения добавим еще одно слагаемое — получим функцию $\sigma(x)+\sigma(x-1)+\sigma(x-2)$. Этот процесс приведет нас к функции активации $f(x) = \sum_{n=0}^{\infty} \sigma(x - n)$. Несложно видеть, что она равна интегральной сумме с шагом разбиения 1 при подсчете интеграла $\int_0^{+\infty} \sigma(x - y)\, dy$. Этот интеграл вычисляется:

$$\int_0^{\infty} \sigma(x - y)\, dy = \int_0^{\infty} \frac{dy}{1 + e^{y-x}} = \int_0^{\infty} \frac{e^{-y} dy}{e^{-y} + e^{-x}} =$$
$$= -\ln(e^{-y} + e^{-x})\Big|_0^{\infty} = \ln(1 + e^{-x}) - \ln(e^{-x}) = \ln(e^x + 1) \approx ReLu(x),$$

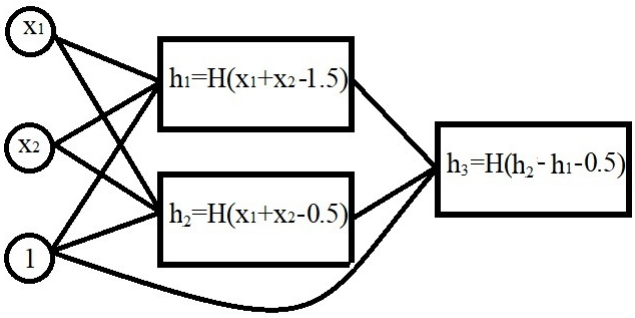
и мы пришли к функции типа $ReLu(x)$. Третий довод в пользу $ReLU$ — функция активации биологических нейронов похожа на $ReLU$ и не похожа на $\sigma(\cdot)$.

Используются, конечно, модификации функции $ReLU$. Недостаток функции $ReLu$ в том, что она постоянна при $\xi < 0$. Тогда можно изменить ее там и взять, например 0.1ξ (слабо убывающая функция). Такую функцию называют Leaking ReLU. Более правильно объявить коэффициент при ξ при $\xi < 0$ гиперпараметром и обучать его параллельно с обучением самой сети.

Нейронная сеть

Один нейрон (вне зависимости от выбора функции активации) остается очень простой моделью. Например, в задаче классификации он создает разделяющую гиперплоскость, а значит не способен разделить данные, которые не разделимы линейно. Это было отмечено самим Розенблатом на примере функции XOR.

Пример 6. *Пусть наш массив данных содержит ровно два признака X_1 и X_2, которые принимают значения 0 и 1 каждый. Пусть метки Y соответствуют логической функции $Y = XOR(X_1, X_2)$, т.е. $Y = 1$, если ровно один из признаков X_1, X_2 равен 1, а в остальных случаях $Y = 0$. Несложно видеть, что такие данные линейно не разделимы.*



Можно, однако легко построить правило разделения, если допускается использование композиции нейронов. Заметим, что $Y = (X_1 \vee X_2) \wedge \overline{(X_1 \wedge X_2)}$. Заведем нейрон h_1, который будет выдавать выход по правилу $h_1(x_1, x_2) = H(x_1 + x_2 - 3/2)$ (здесь $H(\cdot)$ — функция Хевисайда, равная 1 при $x > 0$ и 0 при $x \leq 0$). Легко видеть, что $h_1(x_1, x_2) = x_1 \wedge x_2$. Далее, заведем нейрон h_2, который будет работать по правилу $h_2(x_1, x_2) = H(x_1 + x_2 - 1/2) = x_1 \vee x_2$. Чтобы взять отрицание достаточно вычесть значение из единицы: $\overline{x_1 \wedge x_2} = 1 - h_1(x_1, x_2)$. Теперь остается подать выходы нейронов h_1 и h_2 на третий нейрон $h_3(h_1, h_2) = H(h_2 + (1 - h_1) - 3/2) = H(h_2 - h_1 - 1/2)$ — получим функцию XOR. Несложно видеть, что в результате мы построили нейронную сеть с двумя слоями.

Итак, идея нейронной сети очень проста. Назовем наши признаки $X_0, \ldots, X_m$ входами первого слоя (мы предполагаем, что $X_0 \equiv 1$). Посадим на этот первый слой много перцептронов, а их предсказания передадим на следующий слой. Теперь у нас есть выбор — можно закончить построение сети. Тогда соединим выходы перцептронов первого слоя в один общий (как в нашем примере) и получим финальный ответ. Построенная сеть называется сетью с одним скрытым слоем. Но можно поступить иначе — предсказания нейронов первого слоя сделаем входами (признаками) для каждого нейрона второго слоя. Далее процесс повторяем, пока не решим, что построение сети закончено и предсказания последнего слоя соединим в общий ответ.

При этом можно, разумеется, строить граф из нейронов более избирательно. Не обязательно передавать выходы всех нейронов слоя j на вход все нейронам слоя $j+1$. Архитектура сети — великое дело. Собственно, именно она во многом определяет успешную или неудачную работу сети. Однако, основный принцип работы ясен.

Определение 4. *Нейронная сеть — это ориентированный граф без циклов с $m+1$ входами $1, X_1, \ldots, X_m$ и одним выходом Y. Каждый вход имеет только исходящие ребра, которые ведут в вершины, где расположены нейроны первого слоя. Из этих вершин ведут ребра в вершины второго слоя, и так далее. Последний слой состоит только из одной вершины — выхода Y. В вершинах каждого слоя находятся функции вида $f(w_1x_1+\cdots+w_nx_n)$, где $x_1, \ldots, x_n$ — значения, приписанные входящим в вершину ребрам. Значение функции f приписывается ребрам, выходящим из вершины.*

Лучше один раз увидеть, чем сто раз услышать — откройте сеть по адресу https://playground.tensorflow.org/

Определение 5. *Числа w_j (их набор различен в каждой вершине) называются весами сети или ее коэффициентами. Обучением нейросети называется процесс подбора ее весов.*

Возникает резонный вопрос — всегда ли можно так подобрать веса сети, чтобы она выдавала требуемое нам значение $Y = f(X_0, \ldots, X_m)$?

Теорема 3 (Г.Цыбенко, 1989)**.** *Для любой непрерывной функции $f : \mathbb{R}^m \to \mathbb{R}$ и для любого $\varepsilon > 0$ найдется число N, числа $w_{11}, \ldots, w_{Nm}$ и числа $\alpha_1, \ldots, \alpha_N$ такие, что*

$$\max_{0\le x_1,\ldots,x_m\le 1} \left| f(x_1, \ldots, x_m) - \sum_{j=1}^{N} \alpha_j \sigma(\langle \vec{x}, \vec{w}_j \rangle) \right| < \varepsilon.$$

Здесь $\vec{w}_j = (w_{1j}, \ldots, w_{mj})$ — векторы из $\mathbb{R}^m$, составленные из коэффициентов сети, а $\sigma(\xi) = \frac{1}{1+e^{-\xi}}$ — сигмоид.

Следствие 1. *Нейронной сетью с одним скрытым слоем и с функцией активации $\sigma(\xi)$ можно сколь угодно хорошо приблизить любой конечный набор данных.*

Доказательство. Для любого конечного набора точек $\vec{x}_i = (x_{i1}, \ldots, x_{im}, y_i)$ найдется непрерывная функция $y = f(x_1, \ldots, x_m)$, которая удовлетворяет равенствам $f(\vec{x}_i) = y_i$. Далее воспользуемся теоремой. □

Замечание 1. *Хотя теорема Цыбенко и показывает, что всегда можно обойтись нейросетью с одним скрытым слоем и с сигмоидальной функцией активации, это не означает, что следует строить именно такие сети. Более сложная архитектура сети (большее число слоев) дает больше возможностей для подбора весов, так что следует ожидать, что такие сети будут учиться быстрее.*

Здесь мы приходим к вопросу, как учить сеть (как подбирать веса), над которым поработаем в следующей лекции.

Замечание 2. *Нейросеть гарантировано приближает данные только в той области, в которой находится тренировочный набор данных. Таким образом, использовать построенную нейросеть следует только в этой области.*

Нейронные сети — обучение

Определение 6. *Процесс подбора весов $\{w_{j,k}\}$, где k — номер нейрона, а j — номер веса в функции $f(w_0 + w_1x_1 + \cdots + w_nx_n)$, приписанной этому нейрону, называется процессом обучения сети.*

Для обучения используется принцип обратного распространения ошибки. Рассмотрим его на примере конкретной сети. Будем решать задачу бинарной классификации с метками классов $y_i \in \{\pm 1\}$. В нашем примере сеть будет состоять всего из двух слоев. Пусть на верхнем (втором) слое один перцептрон с логистической функцией активации, а на первом (нижнем) K перцептронов с активацией $ReLU$. Подадим в сеть очередной тренировочный семпл. Теперь посмотрим на ответы верхнего слоя. Ответ нашего единственного перцептрона этого слоя можно записать в виде

$$f(\vec{x}) = \sigma(\vec{\tau} \cdot \vec{z}^T),$$

где $\vec{z}$ — входы верхнего слоя, т.е. выходы предыдущего слоя. Эти выходы в свою очередь являются функциями от входов предыдущего слоя, т.е.

$$z_k = ReLU(\vec{\theta}_k \cdot \vec{x}^T) = |\vec{\theta}_k \cdot \vec{x}^T|_+$$

(мы использовали короткое обозначение для ReLU). Таким образом,

$$f(\vec{x}) = \sigma\left(\tau_0 + \sum_{k=1}^{K} \tau_k z_k\right) = \sigma\left(\tau_0 + \sum_{k=1}^{K} \tau_k \cdot ReLU\left(\theta_{k,0} + \sum_{j=1}^{m} \theta_{k,j} x_j\right)\right).$$

Функция ошибки у нас будет стандартной

$$\mathcal{L}(\tau, \theta) = \sum_{i=1}^{n} \ln(1 + e^{-y_i y_i^*}), \qquad y_i^* = \tau_0 + \sum_{k=1}^{K} \tau_k \left|\theta_{k,0} + \sum_{j=1}^{m} \theta_{k,j} x_{i,j}\right|_+ .$$

Наша задача — взять градиент функции $\mathcal{L}$ по всем коэффициентам модели $\theta_{k,j}$ и τ_k. Взяв этот градиент, мы начнем проводить обычный градиентный спуск, стараясь минимизировать функцию потерь. Отличная новость состоит в том, что мы можем вычислять этот градиент сверху вниз (от верхнего слоя к нижнему), основываясь на предсказаниях модели. А именно,

$$\frac{\partial \mathcal{L}}{\partial \tau_k} = \sum_{i=1}^{n} \frac{-y_i e^{-y_i y_i^*}}{1 + e^{-y_i y_i^*}} \cdot \frac{\partial y_i^*}{\partial \tau_k}, \qquad \frac{\partial y_i^*}{\partial \tau_k} = z_k(\vec{x}_i) = |\vec{\theta}_k \cdot \vec{x}^T|_+, \quad \frac{\partial y_i^*}{\partial \tau_0} = 1,$$

$$\frac{\partial \mathcal{L}}{\partial \theta_{k,j}} = \sum_{i=1}^{n} \frac{-y_i e^{-y_i y_i^*}}{1 + e^{-y_i y_i^*}} \cdot \frac{\partial y_i^*}{\partial \theta_{k,j}} = \sum_{i=1}^{n} \frac{-y_i e^{-y_i y_i^*}}{1 + e^{-y_i y_i^*}} \cdot \tau_k \cdot H\left(z_k(\vec{x}_i)\right) x_{i,j} \tag{4}$$

(при $j = 0$ считаем $x_{i,0} = 1$). Градиент выписан явно. Обратите внимание, что для его подсчета у нас есть все данные — это результаты прямого прогона модели — числа z_k, y_i и предыдущие коэффициенты модели τ_k. Кроме того, обратите внимание, что для

подсчета частных производных на нижнем уровне нам достаточно взять формулы частных производных верхнего уровня и заменить в них выражение z_k (выходы нижнего уровня) на производные этих выходов — числа $\tau_k \cdot H(z_k(\vec{x}_i))x_{i,j}$. И так будет всегда, поскольку это есть просто правило дифференцирования сложной функции. Если теперь у нас под нижним слоем есть еще один (скажем, нулевой) слой, то нам достаточно заменить множители $x_{i,j}$ в формулах из второй строки на соответствующие производные, и мы получим производные по коэффициентам этого нулевого слоя.

Конечно, формулы для частных производных становятся все сложнее при переходе к каждому нижнему слою. Однако можно легко сообразить, как эти производные усложняются. На верхнем слое мы имеем выражения вида $\sum_i a_i z_k(\vec{x}_i)$. На втором слое получим $\sum_i a_i b_{ik} x_{ij}$ — матрицу (b_{ik}) умножают на вектор (a_i). Легко понять, что и дальше с производными будет происходить то же самое — производные нижнего слоя получаются из производных верхнего с помощью матричного умножения. Это вновь совершенно абстрактный факт, напрямую вытекающий из формулы производной сложной функции. Если какая–то функция f зависит от k аргументов, а каждый аргумент есть функция g_k, зависящая от m аргументов x_j, то

$$\left(\frac{\partial f}{\partial x_j}\right)_{j=1}^{m} = \left(\sum_{k=1}^{K} \frac{\partial f}{\partial g_k} \cdot \frac{\partial g_k}{\partial x_j}\right)_{j=1}^{m} = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \cdots & \frac{\partial g_K}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_1}{\partial x_m} & \cdots & \frac{\partial g_K}{\partial x_m} \end{pmatrix} \begin{pmatrix} \frac{\partial f}{\partial g_1} \\ \vdots \\ \frac{\partial f}{\partial g_K} \end{pmatrix}.$$

Получили формулу произведения матрицы на вектор.

Итак, выписать в явном виде производные функции потерь по всем весам сети $\frac{\partial \mathcal{L}}{\partial w_{jk}}$ мы можем. Тогда для обучения сети можно применять обычный градиентный спуск. Проблема в том, что оптимизация проводится в пространстве большого числа переменных, функция $\mathcal{L}$ имеет много точек минимума и применяя спуск без каких–либо ухищрений, мы получим один из них, скорее всего, далекий от оптимального.

Методы обучения. Инициализация

Понятно, что поскольку у функции потерь много точек локального минимума, то простой градиентный спуск будет находить ту из них, которая ближе всего к начальной точке. Значит, от выбора начальных значений весов $w_{j,k}$ зависит очень многое.

Определение 7. *Выбор начальных значений чисел $w_{j,k}$ называется инициализацией сети.*

Давайте продумаем, что произойдет, если сеть инициализирована нулями. Это легко понять, посмотрев на формулы (4): если все $\theta_k = 0$, то $\frac{\partial y_i^*}{\partial \tau_k} = 0$ за исключением производной по τ_0, которая равна 1. Кроме того, так как равны нулю все τ_k, то обнуляются и все производные $\frac{\partial \mathcal{L}}{\partial \theta_{k,j}}$. Вывод — градиентный спуск сосредоточится на подборе константы τ_0. Это совсем не то, что мы хотим.

Попробуем инициализировать сеть случайными весами. Точнее, будем инициализировать веса $w_{j,k}$ случайно, кроме свободной константы $w_{0,k}$, которую будем брать нулем (как

мы поняли, градиентный спуск склонен в первую очередь оптимизировать именно $w_{0,k}$). Теперь попробуем выбрать распределение, из которого будем генерировать $w_{j,k}$.

Вначале разберем случай, когда функция активации — сигмоид или tanh. Логично взять искомое распределение симметричным относительно нуля (у нас нет никаких аргументов в пользу положительного или отрицательного сдвига). Рассмотрим один перцептрон с выходом $y = f(w_0 + \sum_{j=1}^{J} w_j x_j)$. Раз на предыдущих слоях веса выбирались случайно, то следует считать, что входы нейрона x_j также являются случайными величинами. Вполне логично, что с.в. w_j и x_j независимы. В окрестности нуля функция f похожа на линейную с коэффициентом наклона 1. Тогда

$$E(y) \approx \sum_{j=1}^{J} E(x_j w_j) = \sum_{j=1}^{J} E(w_j)E(x_j) = 0, \qquad \operatorname{Var}(y) \approx \sum_{j=1}^{J} \operatorname{Var}(w_j x_j) = \sum_{j=1}^{J} E(w_j^2 x_j^2).$$

Далее, $E(w_j^2 x_j^2) = E(w_j^2)E(x_j^2)$, а поскольку распределение у всех w_j мы считаем одинаковым, то первый множитель не зависит от j. Мы уже поняли, что $E(y) = 0$, а поскольку перцептроны на предыдущем слое были инициализированы так же, то $E(x_j) = 0$ и тогда $E(x_j^2) = \operatorname{Var}(x_j)$, так что окончательно

$$\operatorname{Var}(y) \approx J \operatorname{Var}(w) \operatorname{Var}(x).$$

Вывод: при переходе к новому слою дисперсия выхода каждого нейрона умножается на коэффициент $J \operatorname{Var}(w)$, где J — число нейронов на предыдущем слое. Логично выбирать распределение так, чтобы этот коэффициент равнялся единице — иначе дисперсия нейронов от слоя к слою либо будет затухать к нулю, либо расти к бесконечности.

Определение 8. *Инициализацию весов случайной выборкой из $\mathcal{N}\left(0, \frac{1}{\sqrt{n_{in}}}\right)$, где n_{in} — число нейронов предыдущего слоя, называется calibrated random numbers intialization.*

С другой стороны, посмотрим на производные функции потерь. Пусть на нашем слое расположены нейроны $y_1, \ldots, y_K$. Тогда

$$\frac{\partial \mathcal{L}}{\partial x_j} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial y_k} \cdot \frac{\partial y_k}{\partial x_j} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial y_k} \cdot f'(y_k^*) w_{j,k}.$$

Значение f' вновь будем считать примерно единицей. При случайной инициализации производные функции потерь тоже являются случайными величинами. Тогда, как и раньше,

$$E\left(\frac{\partial \mathcal{L}}{\partial x_j}\right) \approx \sum_{k=1}^{K} E(w_{j,k}) E\left(\frac{\partial \mathcal{L}}{\partial y_k}\right) = 0, \qquad \operatorname{Var}\left(\frac{\partial \mathcal{L}}{\partial x_j}\right) \approx \sum_{k=1}^{K} E(w_{j,k}^2) E\left(\frac{\partial \mathcal{L}}{\partial y_k}\right)^2 = K \operatorname{Var}(w) \operatorname{Var}\left(\frac{\partial \mathcal{L}}{\partial y_k}\right).$$

Градиенты не должны ни затухать ни взрываться, так что приходим к равенству $K \operatorname{Var}(w) = 1$. Если число нейронов меняется от слоя к слою, то оба равенства соблюсти не получится — возьмем нечто среднее.

Определение 9. *Инициализацией весов выборкой из* $\mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}+n_{out}}}\right)$ *или из* $R\left(-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right)$ *называется инициализацией Ксавье.*

В наших рассуждениях спрятано предположение о симметричности функции активации (мы заменили $E(x_j^2)$ на $\mathrm{Var}(x_j)$), то есть для функции $ReLU$ инициализация должна быть другой. Поскольку $E(y^*) = \sum_j E(w_j x_j) = 0$, а $y = |y^*|_+$, то в силу симметричности распределения $E(y^2) = \frac{1}{2}\mathrm{Var}(y^*)$. Остальное сохраняется:

$$\mathrm{Var}(y^*) = \sum_j \mathrm{Var}(x_j w_j) = \sum_j \left(E(x_j^2 w_j^2) - E^2(x_j)^2 E(w_j)\right) = \sum_j E(x_j^2)E(w_j^2) = n_{in}E(x^2)\,\mathrm{Var}\,w.$$

Таким образом, $\mathrm{Var}(y^*) = \frac{n_{in}}{2}\,\mathrm{Var}\,w\,\mathrm{Var}(x^*)$ и $E(y^2) = \frac{n_{in}}{2}\,\mathrm{Var}\,w\,E(x^2)$. Таким образом, здесь калибровка должна быть $\mathcal{N}\left(0, \frac{\sqrt{2}}{\sqrt{n_{in}}}\right)$.

Методы обучения. Скорость обучения

Итак, мы инициализировали сеть. Теперь надо запускать градиентный спуск. Подставляем очередной семпл, двигаясь по графу слева направо, вычисляем последовательно значения всех нейронов. Последний слой дает нам ответ $\widehat{y}_i$. Сравниваем этот ответ с истинным, т.е. со значением y_i, вычисляем значение функции потерь $\mathcal{L}(y_i, \widehat{y}_i)$. Частные производные этой функции по весам $w_{k,j}$ нейронов последнего слоя нам известны (они хранятся как аналитические выражения) — достаточно подставить туда значения нейронов. Частные производные функции $\mathcal{L}$ по весам $w_{k,j}$ предыдущего слоя можем вычислить с помощью произведения матрицы якобиана на посчитанный вектор градиента (якобиан тоже предварительно надо посчитать, подставив значения нейронов), и так далее. В результате мы получаем значения всех частных производных. Теперь делаем шаг градиентного спуска

$$w_{k,j}^{new} = w_{k,j}^{old} - \eta \nabla_{w_{k,j}} \mathcal{L}|_{w^{old}}.$$

Вопрос — насколько большой шаг нам сделать, т.е. каким взять коэффициент η? Этот параметр называется *скоростью обучения*. Понятно, что делая большие шаги мы рискуем «проскочить» точку минимума, а делая маленькие шаги будем «топтаться на месте». Стандартных выборов два.

Определение 10. *Эпоха обучения — последовательность шагов градиентного спуска, при которых все семплы тренировочной выборки просматриваются по одному разу каждый.*

Обычно договариваются не менять коэффициент η в течение эпохи. На следующей эпохе скорость η подправляем. Можно просто договориться о последовательном уменьшении по закону $\eta_t = \eta_{t-1} \cdot 2^{-1/T}$, где T — заранее выбранный гиперпараметр (число эпох обучения, после которых скорость обучения падает вдвое). Можно пойти другим путем и менять η адаптивно: будем в конце каждой эпохи обучения вычислять функцию потерь на валидационном множестве (будем брать среднее значение функции потерь). Если уменьшения по сравнению с предыдущей эпохой не произошло, то откатим все изменения весов назад и уменьшим число η в два раза (или в α раз, где α — гиперпараметр).

Попытка проводить обучение с большой скоростью часто приводит к тому, что значения коэффициентов начинают «дергаться вперед–назад». Вместо того, чтобы жестко уменьшить скорость обучения, поступим более мягко. Представим, что наша точка — это материальный шарик, который катиться по холмистой поверхности. В каждой точке у нас есть вектор силы (собственно, вектор антиградиента). Но кроме него есть текущая скорость (инерция). Обозначим эту скорость u. Тогда получаем шаг метода:

$$w_{k,j}^{new} = w_{k,j}^{old} - u^{new}, \qquad u^{new} = \gamma \cdot u^{old} + \eta \cdot \nabla_w \mathcal{L}|_{w^{old} - \gamma u^{old}}.$$

Здесь коэффициент $\gamma \in (0, 1)$ является гиперпараметром и показывает, какую долю предыдущего вектора скорости мы хотим оставить. Обратите внимание на ту точку, в которой мы вычисляем градиент: она отличается от предыдущей точки, мы берем ее той точкой, куда привела бы нас инерция, не будь градиентной поправки. Получилась модификация градиентного спуска — *метод Нестерова*. Этот метод регуляризует градиентный спуск — страхует его от колебаний «вперед–назад», а γ — коэффициент регуляризации.