Методы восстановления регрессии

Постановка задачи

Данная лекция посвящена методам восстановления регрессии. На предыдущих двух лекциях уже были рассмотрены два алгоритма для восстановления регрессии. Для алгоритма многомерной линейной регрессии уже было замечено, что алгоритм практически такой же, как и алгоритм линейной классификации, но примененный к задаче регрессии. И на данной лекции мы вспомним, что в целом было пройдено за этот семестр и как методы, которые были придуманы для задач классификации, можно применить к задаче регрессии.

Напомним постановку задачи регрессии.

Есть множество объектов X из R^n:

$$X - \text{объекты в } \mathbb{R}^n;$$

Также есть множество ответов Y. В отличие от задачи классификации Y это не какое-то фиксированное множество, а какие-то действительный числа:

$$Y - \text{ответы в } \mathbb{R}$$

Имеется обучающая выборка размера l, для которой уже известны ответы:

$$X^l = (x_i, y_i)_{i=1}^l - \text{обучающая выборка}$$

Есть какая-то неизвестная зависимость, которая и дает нам ответы по объектам:

$$y_i = y(x_i),\ y : X \to Y - \text{неизвестная зависимость}$$

В задаче регрессии мы хотим построить некоторую функцию f, которая бы приближала эту неизвестную зависимость.

$$a(x) = f(x, w) - \text{модель зависимости},$$
$$w \in \mathbb{R}^p - \text{вектор параметров модели}.$$

В качестве функционала качества будем использовать **метод наименьших квадратов (МНК):**

$$Q(w, X^l) = \sum_{i=1}^{l} \alpha_i (f(x_i, w) - y_i)^2 \to \min_w$$

K ближайших соседей

Первый осмысленный алгоритм, который был разобран в этом семестре - метод k ближайших соседей.

Суть алгоритма: мы берем функцию расстояния, считаем для нашего объекта, который мы хотим классифицировать, расстояния до всех возможных объектов и упорядочиваем объекты по возрастанию этой величины. Дальше идея заключалась в том, что мы смотрим на ближайших соседей и берем доминирующий среди них класс.

$$\rho(u, x_1) \leq \rho(u, x_2) \leq \cdots \leq \rho(u, x_l)$$

x_i – i-й сосед объекта u
y_i – класс i-го соседа u

В этом методе используется **гипотеза о компактности**. То есть, мы считаем, что похожие объекты должны лежать близко или объекты, которые лежат близко, должны лежать в одном классе.

В этом алгоритме из важных составляющих блоков была метрика расстояния или **мера похожести**. В качестве такой метрики можно использовать, например, евклидово расстояние, манхэттенское расстояние и редакционное расстояние, обобщенное евклидово расстояние.

Так же важно не забывать про **масштаб**. Если наши признаки находятся в разных масштабах, то вклад от одного признака получается сильно больше, чем от других и по сути, мы вообще забываем про то, что другие признаки существуют.

Еще одна идея, которая была достаточно важна, заключалась в том, что **более близкие соседи должны привносить больший вклад** в классификацию. Это мы решали несколькими разными способами. Первый подход был достаточно простой. Возьмем линейно убывающие веса или экспоненциально убывающие веса. Следующий способ был более замысловатый. Это метод окна Парзена, в котором вводится некоторая функция от расстояния до объектов.

Итого, получалось, что алгоритм выглядит следующим образом. Для каждого объекта выборки есть некоторые веса. И мы смотрим на argmax по всем возможным классам, которые присутствуют в выборке.

$$a(u, X^l) = \arg\max_{y \in Y} \sum_{y_i = y} w(i, u)$$

$$w(i, u) = [i \leq k]$$

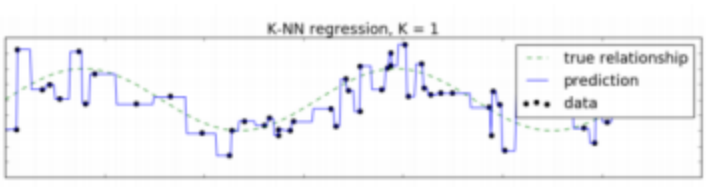
Как алгоритм привязан к задаче классификации и как его можно использовать для задачи регрессии?

К задаче классификации алгоритм привязан тем, что мы, когда ищем свой класс, мы смотрим только на класс соседей. Вот если бы у нас была задача регрессии, где ответы - это непрерывные действительные числа, то что можно было бы сделать? Из достаточно простых идей, чтобы превратить алгоритм для решения задачи регрессии, мы присваиваем не метку класса, а **усредняем характеристики ближайших соседей.**

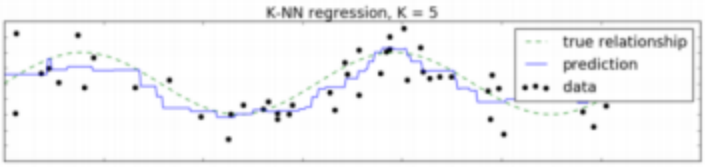
$$a(u, X^l) = \frac{1}{k} \sum_{i=1}^{k} w(i, u) * y_i$$

Посмотрим, как это будет выглядеть. При различных значениях количества ближайших соседей картинка будет слегка различаться.

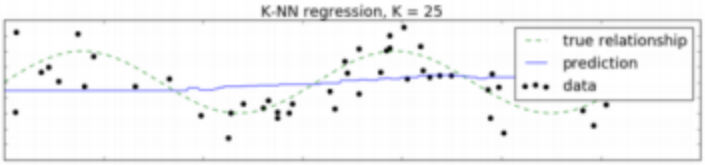
- Если мы возьмем мало ближайших соседей, то мы будем очень хорошо настраиваться на обучающую выборку и у нас будет совсем кусочно-постоянная функция.



- Если мы берем чуть больше ближайших соседей, то получается чуть более сглаженная линия, хотя она тоже кусочная.



- Если мы возьмем слишком много ближайших соседей, то приближение выродится в некоторую линию, которая не сильно соответствует той неизвестной зависимости, которую мы ищем. Потому что мы взяли все объекты и посчитали среднее по ним.



Для настройки можно подбирать количество ближайших соседей и влиять на ту метрику расстояния, которая используется.

Деревья принятия решения

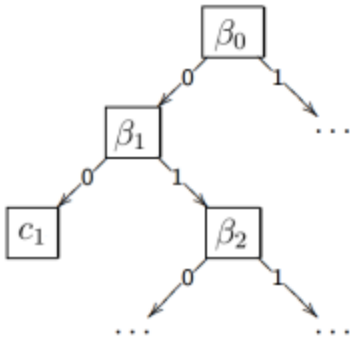
Следующая лекция была посвящена метрическим способам кластеризации. Задачу кластеризации мы не будем пытаться превратить в задачу регрессии по понятным причинам.

А четвертая лекция была посвящена деревьям принятия решения.

Основным логическим блоком для деревьев принятия решений были предикаты. Они, должны обладать двумя свойствами, они должны быть **интерпретируемыми** и **информативными**. Предикат - это некоторая логическая закономерность.
Мы рассматривали следующие варианты предикатов:

- Свойство объекта равно какому-то конкретному значению.
- … лежит в каком-то отрезке.
- Выполнилось хотя бы сколько-то условий-индикаторов.

После обсуждения предикатов первая логичная идея была -- список принятия решений. И он был плох тем, что мы сильно зависим от порядка, в котором были выстроены предикаты и ошибка одного из предикатов является ошибкой всего алгоритма, что достаточно печально.



Поэтому следующей идеей стало дерево принятия решений. Заключалось оно в том, что выстраивалось некоторое бинарное дерево, которое состояло из:

- внутренних узлов, в которых записан некоторый предикат и по которому выборка разделяется на две части, одна из частей шла в левое поддерево, вторая в правое. В левую часть идут объекты, на которых предикат не сработал, в правую часть объекты, на которых предикат сработал.

$$\forall v \in V_{inner} \to \beta_v : X \to \{0,1\},\ \beta \in \mathscr{B}$$

- листьев, в которых записаны метки соответствующих классов.

$$\forall v \in V_{leaf} \to \text{имя класса } c_v \in Y$$

Классификация выглядит таким образом, что мы проходимся по дереву сверху вниз и по очереди применяем каждый из предикатов на нашем объекте и в некоторый момент мы попадаем в один из листьев и класс, который записан в этом листе, мы и назначаем объекту.

Основной алгоритм, который мы рассматривали называется ID3.
Это рекурсивный алгоритм.

2 Сначала мы проверяем, если все объекты принадлежат одному классу, то мы сразу строим лист с соответствующей меткой класса.

4 Дальше мы ищем наиболее информативный предикат, записываем его в узел дерева и по этому предикату разделяем множество объектов, которое у нас есть, на два подмножества: множество, на котором предикат сработал, и множество, на котором предикат не сработал.

```
1  function LearnID3(U, 𝓑)
2      if все объекты из U лежат в одном классе c ∈ Y then
3          return новый лист v, c_v = c
4      β* = max_{β∈𝓑} I(β, U)
5      U_left = {x ∈ U : β*(x) = 0}
6      U_right = {x ∈ U : β*(x) = 1}
7      if U_left = ∅ или U_right = ∅ then
8          return v, c_v = Majority(U)
9      Создать новую внутреннюю вершину v: β_v = β*
10     L_v = LearnID3 (U_left, 𝓑)
11     R_v = LearnID3 (U_right, 𝓑)
12     return v
```

7 Если одно из этих подмножеств оказалось пустым, то мы снова возвращаем лист, в котором мы записываем метку класса, который преобладает. Если же это условие не сработало, то значит мы находимся где-то в середине построения дерева и мы продолжаем строить его рекурсивно.

9 Строим узел дерева, в который записываем предикат, для того, чтобы потом мы могли классифицировать. И рекурсивно строим левое и правое поддеревья.

Как алгоритм привязан к задаче классификации и как его можно использовать для задачи регрессии?

С классификацией связь очевидная. В каждом листе записана метка класса. Если объект спустился к листу, значит ему должны назначить эту метку класса.

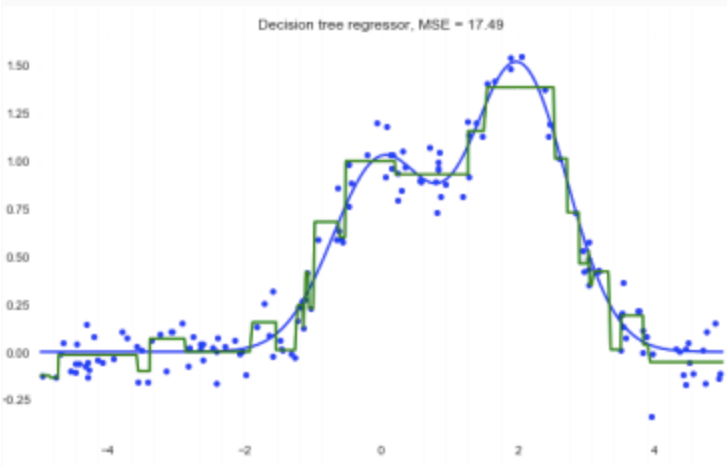
Деревья принятия решения достаточно легко адаптируются. Как значения функции можем взять **среднее из обучающей выборки по тем, кто попал в этот лист**.

Достаточно важной частью алгоритма было то, что на каждом шаге мы выбирали предикат, по которому мы будем разделять нашу подвыборку. Для текущей задачи не очень интересно смотреть на энтропию или на информационный критерий Джини. Сейчас бы хотелось чтобы в каждом узле выборка разбивалась примерно таким образом, чтобы **значения в каждом из поддеревьев были примерно равны**.

И выглядит этот примерно таким образом. Справа стоит усреднение по всем объектам, которые попали в одно из поддеревьев. И мы смотрим отклонение нашего значения от такого среднего значения в этом поддереве.

$$I(\beta, X^l) = \frac{1}{l}\sum_{i=1}^{l}(y_i - \frac{1}{l}\sum_{j=1}^{l} y_j)^2$$

И снова, если смотреть на применение деревьев решений для задачи регрессии, то мы получаем некоторую кусочно-линейную функцию, которая, естественно, не вполне точно восстанавливает зависимость, однако довольно часто они работают хорошо.



Как модифицировались деревья принятий решений для того, чтобы они были одним из самых мощных алгоритмов из тех, которые сейчас используются? Мы использовали **ансамбли деревьев**. Об этом будет подробнее рассказано на следующей лекции. Суть алгоритма в следующем: мы берем достаточно слабый алгоритм дерева, у него очень большой разброс, очень сильно зависим от выборки, по которой мы настраиваем деревья. Однако если мы возьмем много деревьев и настроим по разным подвыборкам, окажется, что мы очень хорошо умеем предсказывать значения и это очень хороший алгоритм, который много где используется.

Случайные леса работают и для задачи регрессии, ничем не хуже чем для задачи классификации.

Основная причина по которой начали использовать **деревья вместо k ближайших соседей**, есть параметры, по которым нельзя так просто посчитать расстояния, плюс деревья очень хорошо интерпретируются. Человеку даже далекому от области можно выдать дерево и он поймет, что происходит внутри алгоритма. Алгоритм хорошо работает с разнотипными данными и почти всегда их не нужно масштабировать. Потому что чаще всего мы разбиваем по какому-нибудь одному признаку.

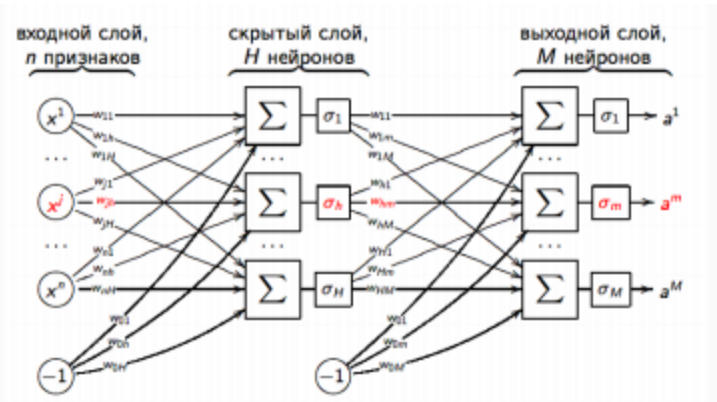
Еще один важный момент по поводу деревьев для задачи регрессии. В задачи классификации достаточно редко **ограничивается глубина дерева**. В задачах регрессии это один из наиболее мощных инструментов, чтобы получить функцию более похожую на искомую зависимость. Мы ограничиваем глубину дерева, которую хотим строить и с некоторого момента обрезаем и говорим, что мы добавляем в дерево только листья. Таким образом мы можем получить различные картинки для различного ограничения максимальной глубины. Так же как и в методе ближайших соседей у нас получится либо более сглаженная картина, либо более хорошо настроенная на разные выбросы.

Следующая тема, которую мы проходили, это байесовская классификация. Его мы не будем превращать, потому что эти классификаторы на практике не применяются для решения задачи регрессии.

Линейная классификация очень легко превратилась в задачу регрессии, как мы уже убедились на прошлых лекциях, в виде линейной регрессии.

Нейронные сети

Следующий алгоритм, который мы использовали - это нейронные сети. Идея заключается в том, что мы берем много небольших линейных классификаторов и соединяем их в большую сеть.



У нас есть входной слой, на котором расположены признаки объектов. Эти объекты подаются на нейроны скрытого слоя. Скрытых слоев может быть больше одного. И в нейроне скрытого слоя происходит операция суммирования, а дальше применяется некоторое нелинейное преобразование и выход каждого нейрона уже подается на вход следующего нейронного слоя и нейронам выходного слоя в качестве признаков.

И алгоритм, который мы применяли - это **алгоритм обратного распространения ошибки.** И на самом деле алгоритм обратного распространения ошибки это модифицированный градиентный спуск. Тем не менее обычный градиентный спуск по некоторым причинам не работает для нейронных сетей и приходится изобретать что-то более замысловатое.

Этим замысловатым и является алгоритм backpropagation. И заключается в том, что мы перемешиваем объекты и, пока функционал не стабилизируется, мы делаем несколько условных проходов по нашей нейронной сети.

```
1 function BACKPROPAGATION(X^l, H, α, η)
2     ...
3     repeat[пока Q не стабилизируются]
4         Взять x_i из X^l
```

5

$$\begin{cases} u_i^h = \sigma_h(\sum\limits_{j=0}^{J} w_{jh} x_i^j), \quad h = 1, \dots, H \\ a_i^m = \sigma_m(\sum\limits_{h=0}^{H} w_{hm} u_i^h), \quad \varepsilon_i^m = a_i^m - y_i^m, \quad m = 1, \dots, M \\ \mathcal{L}_i = \frac{1}{2} \sum\limits_{m=1}^{M} (\varepsilon_i^m)^2 \end{cases}$$

Первый проход называется прямой ход, в ходе которого мы вычисляем значения для нашего объекта, который мы хотим классифицировать в каждом узле сети, а так же мы вычисляем производную на выходном слое.

Следующий шаг - это шаг обратного хода. То преимущество, которым мы воспользовались, чтобы изобрести алгоритм обратного хода, заключается в том, что мы смогли выразить производную или градиент на скрытом слое через производную на выходном слое. И соотвенно на обратном ходе алгоритма мы вычисляем уже производные на скрытом слое используя значения, которые мы вычислили в момент прямого прохода для выходного слоя.

$$6 \quad \left\{ \quad \varepsilon_i^h = \sum_{m=1}^{M} \varepsilon_i^m \sigma_m' w_{hm}, \quad h = 1, \ldots, H \right.$$

И последняя часть алгоритма заключается в том, что мы обновляем все веса, которые у нас есть в нейросети и модифицируем их таким образом, чтобы веса изменились в том направлении, которое позволит нам наилучшим образом классифицировать объекты.

$$7 \quad \left\{ \begin{array}{l} w_{hm} = w_{hm} - \alpha \varepsilon_i^m \sigma_m' u_i^h, \quad h = 0, \ldots, H, \quad m = 1, \ldots, M \\ w_{jh} = w_{jh} - \alpha \varepsilon_i^h \sigma_h' x_i^j, \quad j = 0, \ldots, n, \quad h = 1, \ldots, H \\ Q = (1 - \eta) Q + \eta \mathcal{L}_i \end{array} \right.$$

Как алгоритм привязан к задаче классификации и как его можно использовать для задачи регрессии?

Обычно для каждого класса, который у нас есть, мы заводили по одному нейрону на выходном слое. Таким образом относили наш объект к тому классу, на котором соответсвующий нейрон выдал наибольшее значение.

Вопрос: как можно было избежать этого и как можно изменить нашу нейросеть так, чтобы регрессию тоже можно было бы подсунуть нейросети и она выдавала нам хорошие значения.

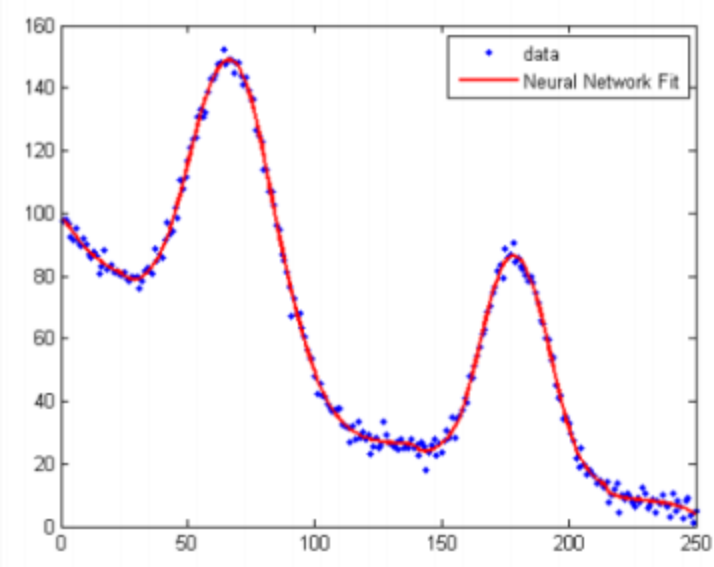
Предыдущие алгоритмы мы модифицировали таким образом, чтобы брать какое-то усредненное значение целевой функции. Логично, первая идея заключается в том, чтобы выбрать какое-то количество отрезков, в которые мы хотим попасть и завести соответствующее количество нейронов выходного слоя и записать туда усредненное значение функции. Однако, хотя это и самый простой способ модифицировать нейросеть по аналогии с предыдущими алгоритмами, но такой способ никто не использует, потому что так мы не добьемся ничего принципиально лучшего по сравнению с деревьями принятия решения для задачи регрессии.

Идея, которая используется в нейросетях, когда мы хотим использовать их для задачи восстановления регрессии, заключается в том, чтобы использовать не ступенчатую функцию активации, которую мы использовали до этого в задачах классификации, а

использовать **непрерывную функцию активации** и на выходном слое мы таким образом получим всего **один нейрон,** который будет нам выдавать искомую функцию.

Наша нейросеть для реальной задач выдает более интересные ответ. Она умеет восстанавливать какую-то не линейную зависимость и восстанавливать достаточно хорошо.

На картинке снова синенькими точками обозначены данные, которые у нас есть. И красная функция - это функция, которую восстановила нейросеть.



Метод опорных векторов

Смысл алгоритма в том, что мы разделяем данные полосой и максимизируем ее ширину. Мы считали сначала, что наши **данные линейно-разделимы** и поэтому мы можем провести между этими данными гиперплоскость. И далее максимизируем ширину зазора, который есть между двумя гиперплоскостями, при этом считаем, что все отступы должны быть больше или равны 1, потому как мы отнормировали отступы так, что бы минимальный из них был равен 1.

$$\begin{cases} \frac{1}{2}\|\mathbf{w}\|^2 \to \min\limits_{\mathbf{w}} \\ M_i(\mathbf{w}, w_0) \geq 1 \qquad i = 1, \ldots, l \end{cases}$$

Когда мы задумались над случаем, когда данные у нас **линейно-неразделимы**, мы ввели некоторые ослабления условий, которые были записаны для линейно разделимой выборки. А именно мы позволили отступам быть меньше 1, но при этом мы добавили дополнительное условие на новый параметр ξ, который показывает насколько объекты заступают за полосу. Мы бы хотели, чтобы сумма всех заступов была тоже

$$\begin{cases} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum\limits_{i=1}^{l} \xi_i \to \min\limits_{\mathbf{w},\xi} \\ M_i(\mathbf{w}, w_0) \geq 1 - \xi_i \qquad i = 1, \ldots, l \\ \xi_i \geq 0 \qquad i = 1, \ldots, l \end{cases}$$

минимальна. Поэтому в наш функционал, который мы минимизируем добавилось второе слагаемое.

Следующим шагом мы выписали **двойственную задачу** для решения нашей исходной задачи и смогли выразить решение исходной задачи через решение двойственной задачи. Таким образом получив некоторые интересные свойства решения.

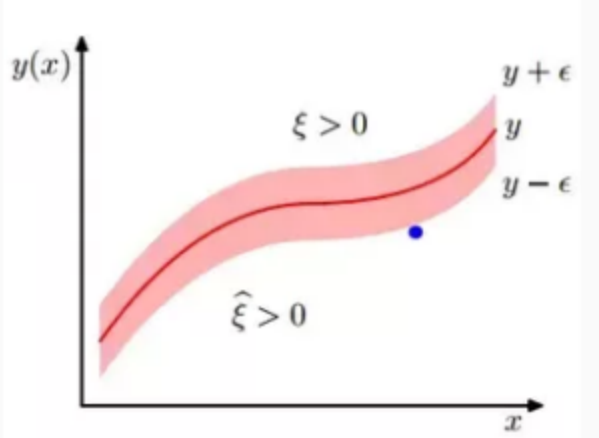
$$\begin{cases} \mathbf{w} = \sum\limits_{i=1}^{l} \alpha_i y_i \mathbf{x_i} \\ w_0 = \langle \mathbf{w}, \mathbf{x_i} \rangle - y_i \end{cases}$$

$$a(\mathbf{x}) = \text{sign}(\sum_{i=1}^{l} \alpha_i y_i \langle \mathbf{x_i}, \mathbf{x} \rangle - w_0)$$

Оказалось, что достаточно многие из объектов в принципе не участвуют в классификации, потому что коэффициент при них обнуляется и они достаточно глубоко находятся в своих классах от разделяющей поверхности.

Кроме этого есть объекты, которые называются **опорными объектами** или опорными векторами. И это те объекты, с помощью которых и строится алгоритм. Их довольно часто оказывается сравнительно немного и этим алгоритм хорош тем, что нам не приходится хранить всю выборку и бесконечное количество весов, которые есть в нейросети. Мы можем хранить только небольшое количество опорных векторов, с помощью которых мы собственно и строим наш алгоритм.

Следующее довольно важное наблюдение в методе опорных векторов, что наш алгоритм зависит только от скалярного произведения от двух объектов и никаким образом не зависит непосредственно от признаков объектов. Таким образом родилась идея **вместо скалярного произведения использовать некоторую другую функцию**, похожее на скалярное произведение по некоторым свойствам, однако не совсем. Важный момент, что этот переход нам нужен не для того, чтобы классифицировать линейно неразделимые объекты. Для линейно неразделимых данных мы уже ввели трюк, когда мы позволяем отступам быть меньше единицы. Ядро вводится для того чтобы мы могли получать более интересные поверхности, и это был переход, можно сказать, в некоторое другое пространство. Когда мы переводили объекты, которые находятся в n мерном пространстве в пространство большей размерности.



Как алгоритм привязан к задаче классификации и как его можно использовать для задачи регрессии?

Проблема в том, что мы классифицируем объекты на два класса. Мы находимся либо с одной стороны от поверхности, либо с другой стороны от поверхности.

Для задачи регрессии можно смотреть насколько близко мы находимся к разделяющей поверхности, а на самом деле даже забыть, что это разделяющая поверхность, точно так же, как когда мы взяли многомерную линейную регрессию, мы в целом забыли, что это разделяющая поверхность, и считали, что это и есть наша искомая зависимость. В задаче SVR можно считать, что этот **классификатор и есть наша искомая зависимость.** Чтобы провести аналогию с SVM будем считать, что если объект на какое-то небольшое значение отклоняется от этой зависимости, то не будем считать это ошибкой. Таким образом мы опять получим некоторую зависимость и область, в которую объекты могут попасть. Раньше мы хотели, чтобы была полоса как можно шире, в которую объекты не попадают, сейчас мы хотим, чтобы все объекты были внутри полосы.

И тогда мы можем модифицировать нашу задачу таким образом, чтобы учитывать, что мы считаем те объекты, которые отклоняются от нашей зависимости **не более чем на эпсилон, тоже правильно восстанавливающими зависимость**. Тогда, если мы введем две новые переменные, которые как раз таки отражают два новых ограничения, которые у нас есть смещение в одну и другую сторону от нашей зависимости. Функционал выглядит очень похожим на то, что у нас было для задачи SVM за исключением того, что мы хотим минимизировать. Мы вводим не одну величину, которая ответственна за то, что наш отступ может быть меньше 1, а мы вводим две величины, которые будут отвечать за положение объекта с обеих сторон от зависимости.

$$\begin{cases} \min \frac{1}{2}\|\mathbf{w}\|^2 \\ y_i - \langle \mathbf{w}, \mathbf{x_i} \rangle + w_0 \leq \varepsilon \\ \langle \mathbf{w}, \mathbf{x_i} \rangle - w_0 - y_i \leq \varepsilon \end{cases}$$

$$\xi_i = \langle \mathbf{w}, \mathbf{x_i} \rangle - w_0 - y_i - \varepsilon$$
$$\hat{\xi}_i = -\langle \mathbf{w}, \mathbf{x_i} \rangle + w_0 + y_i - \varepsilon$$

$$\begin{cases} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}(\xi_i + \hat{\xi}_i) \to \min\limits_{\mathbf{w},\xi} \\ y_i - \varepsilon - \hat{\xi}_i \leq \langle \mathbf{w}, \mathbf{x_i} \rangle - w_0 \leq y_i + \varepsilon + \xi_i \qquad i = 1, \ldots, l \\ \xi_i, \hat{\xi}_i \geq 0 \qquad i = 1, \ldots, l \end{cases}$$

$$a(\mathbf{x}) = \sum_{i=1}^{l}(\alpha_i - \hat{\alpha}_i)\langle \mathbf{x_i}, \mathbf{x} \rangle - w_0$$

Интересный факт заключается в том, что решение этой задачи точно так же сводится к решению задачи через двойственные переменные и точно таким же образом мы получаем

решение минимизации исходного функционала в виде решения через двойственные переменные и снова оказывается, что достаточно немного объектов формируют нашу исходную зависимость и при этом метод может восстановить не менее сложную зависимость, чем нейронные сети, однако плюсом является то, что здесь достаточно ограниченное число объектов используются для того, чтобы построить саму эту зависимость.