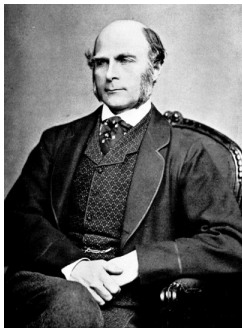


Классификация: ансамбли моделей

Случайный лес. Градиентный бустинг

Лекция 11

Скрытая мудрость толпы?



- ▶ В 1906 сэр Фрэнсис Гальтон посетил рынок скота.
- ▶ Фермеры были приглашены, чтобы отгадать вес призового быка.
- ▶ Около 800 человек дали свои оценки, но ни одна из них не была достаточно близка к точному значению: 1198 фунтов.
- ▶ Однако, на удивление, среднее этих 800 попыток было очень близко к точному значению. Оно было 1197 фунтов.

Приемы улучшения классификаторов

- ▶ Stacking
- ▶ Бэггинг (bagging)
- ▶ Бустинг (boosting)

Идея: берем много слабых классификаторов и пытаемся сделать из них один сильный.

Слабый классификатор — алгоритм, позволяющий классифицировать объекты с вероятностью ошибки меньшей, чем простое угадывание (0.5 для бинарной классификации).

Сильный классификатор — алгоритм классификации, позволяющий добиться произвольно малой ошибки обучения.

Stacking

Укладка в штабель?

Идея: получаем вектор прогнозов и добавляем его к исходным данным.

Прогнозы получаем одним методом классификации, а пополненные исходные данные используем в другом методе.

Бэггинг (bagging = **b**ootstrap+**a**ggregating)

Метод “Случайный лес” (random forest) — пример использования бэггинга.

Предложен Лео Брейманом (Leo Breiman) и Адель Катлер (Adele Cutler) в 1994 г.

Обозначения

- ▶ Обучающая выборка состоит из N примеров (строк).
- ▶ Размерность пространства признаков равна M .

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1M} \\ x_{21} & x_{22} & \dots & x_{2M} \\ \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & \dots & x_{NM} \end{bmatrix}, \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_N \end{bmatrix}$$

Принцип действия

- ▶ Строится `ntree` деревьев.
- ▶ Голосование: побеждает класс, за который проголосовало наибольшее число деревьев.

Все деревья разные, благодаря бэггингу

Прием бэггинга заключается в то, что:

- ▶ Каждое дерево обучается на своей подвыборке исходных данных.
- ▶ Подвыборку составляет `samplesize` наблюдений и `mtry` переменных.

Брать все наблюдения и переменные не годится. В этом случае все деревья будут одинаковыми, а результат голосования тривиальным.

Вместо деревьев можно использовать другие классификаторы. Но: деревья быстро строятся, быстро обучаются, а потом и быстро работают.

Ключевые параметры модели

- ▶ `ntree` — число деревьев. *Тактика*: сделай много, потом сокращай! (можно не обучать заново)
- ▶ `mtry` — число переменных в подвыборке. *Тактика* (Breiman): $\sqrt{M}$
- ▶ `sampsize` — число наблюдений в подвыборке. *Тактика*: $(2/3)N$
- ▶ `nodesize` — минимальное число наблюдений в узле. *Тактика*: 10 или больше.
- ▶ `replace` — подвыборка с возвращением или нет. *Тактика* (Breiman): с возвращением.

Подсказка: смотрите на значения параметров, принятые по умолчанию.

Комментарии

1. Отбирая переменные добиваемся декорреляции результатов разных деревьев.
2. Леса могут использоваться в задачах регрессии. Если у вас нелинейная модель, то случайные леса — хороший кандидат для работы с ней.
3. Ошибку можно измерять используя out-of-bag error.

out-of-bag — наблюдения, которые не попали в подвыборку (сумку?).

Применим дерево, построенное на подвыборке, к тем наблюдениям, которые в эту подвыборку не попали. Ошибка классификации на таких наблюдениях и будет out-of-bag error. Напоминает ошибку на тестовом множестве.

Информативность переменных

Интересная информация, которую может сообщить нам случайный лес: информативность (importance) переменных — насколько переменная важна, чтобы включать ее в анализ.

Оценка информативности

У нас есть лес. Он обучен. Мы берем один из столбцов (переменную) и случайным образом переставляем в нем значения. Если эта переменная важна для получения результатов — если она информативна, — то качество распознавания сильно уменьшится.

Информативность переменной вычисляется как разность двух out-of-bag ошибок: до и после перестановки значений в столбце. Чем больше эта разность, тем более информативна переменная.

Проведем эту процедуру по всем столбцам и получим информативность каждой переменной.

Пример: классификация вин

```
library(randomForest)

wine <- read.table("wine1.txt", header=T, sep="", dec=".")
```

Создание обучающей и тестовой выборки

```
# Формируем случайную подвыборку
set.seed(1234)
test.num <- sample(1:nrow(wine), 60, replace = FALSE)
# Тестовая выборка
test <- wine[test.num, -ncol(wine)]
# Код класса для тестовой выборки
y.test <- wine[test.num, ncol(wine)]
# Обучающая выборка
train <- wine[-test.num, ]
# Делаем колонку меток класса фактором
train$Wine_type <- as.factor(train$Wine_type)
# Предикторы
x <- train[, -ncol(wine)]
# Отклик.
# Должен быть фактором, иначе выполняется регрессия
y <- train$Wine_type
```

Сбалансированность классов

```
# Значения результирующей переменной сбалансированы?  
table(y)
```

```
## y  
## 0  1  2  
## 40 32 46
```

Для несбалансированных классов алгоритмы классификации сбоят!

Если число кликнувших по баннеру (класс 1) составляет 0.1% от числа некликнувших (класс 0), то классификатор отнесет всех к классу 0, причем с весьма малой ошибкой.

Что делать с несбалансированностью?

1. Штрафовать за ошибки определенного вида (за ошибку в распознавании класса 1 как класса 0).
2. Пополнять “малые” классы (копиями строк-наблюдений).
3. Объединять несколько “малых” классов в один.

Классы	1	2	3	4	5
Число элементов	700	900	62	4	18

Объединяем классы 3—5

Классы	1	2	3*
Число элементов	700	900	84

Задаем параметры леса

```
# При bagging'е используем датчик случайных чисел!  
set.seed(3217)  
# Число деревьев в лесу  
ntree.1 <- 500 # так по умолчанию  
# Трассировка: число шагов через которые выдается  
# промежуточный результат  
# do.trace = ntree.1/10  
# Минимальное число наблюдений в узле  
nodesize.1 <- 1  
# Сохранить обученный лес  
keep.forest.1 <- TRUE
```

Запускаем функцию классификации

```
rf.wine <- randomForest(x, y,  
                        ntree = ntree.1,  
                        mtry = floor(sqrt(ncol(wine))),  
                        replace = FALSE,  
                        nodesize = nodesize.1,  
                        importance = TRUE,  
                        do.trace = ntree.1/10,  
                        keep.forest = keep.forest.1)
```

Промежуточные результаты помогают определить нужное число деревьев

# ntree	OOB	1	2	3
# 50:	1.69%	0.00%	0.00%	4.35%
# 100:	0.00%	0.00%	0.00%	0.00%
# 150:	0.00%	0.00%	0.00%	0.00%
# 200:	0.00%	0.00%	0.00%	0.00%
# 250:	0.00%	0.00%	0.00%	0.00%
# 300:	0.85%	2.50%	0.00%	0.00%
# 350:	0.85%	2.50%	0.00%	0.00%
# 400:	0.85%	2.50%	0.00%	0.00%
# 450:	0.85%	2.50%	0.00%	0.00%
# 500:	0.85%	2.50%	0.00%	0.00%

Показывают, какой процент ошибок (OOB — out-of-bag error) допущен при распознавании каждого из сортов вина. И самое важное: при каком количестве деревьев.

Хороший результат получается уже при 100—250 деревьях.

Размечаем тестовую выборку

```
predict.wine <- predict(rf.wine, newdata = test)
# Оценим качество результата
table(y.test, predict.wine)
```

```
##          predict.wine
## y.test  0  1  2
##       0 19  0  0
##       1  0 16  0
##       2  1  2 22
```

Попробуем теперь при `ntree = 250`

```
# При bagging'е используем датчик случайных чисел!  
set.seed(3217)  
# Число деревьев в лесе  
ntree.1 <- 250  
...  
# Повторяем операции...
```

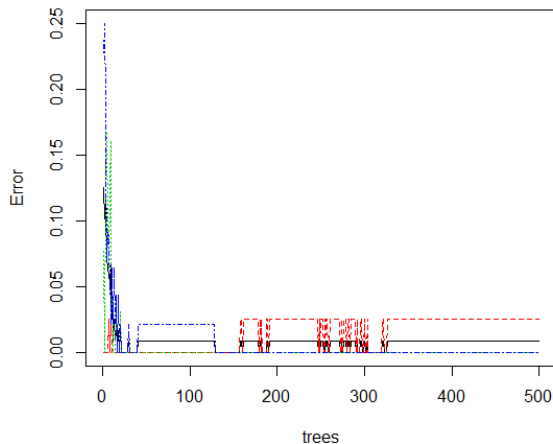
```
##      predict.wine  
## y.test  0  1  2  
##      0 19  0  0  
##      1  0 16  0  
##      2  1  1 23
```

Лучше!

График ошибки классификации

```
plot(randomForest(Wine_type ~ ., train, keep.forest=FALSE,  
                 ntree=500))
```

andomForest(Wine_type ~ ., train, keep.forest = FALSE, ntree =



Как проходило голосование?

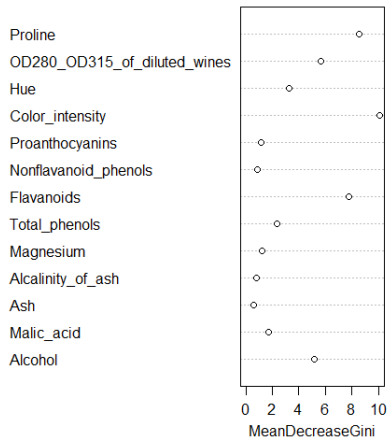
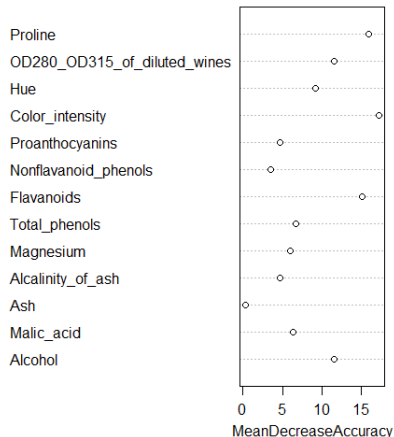
```
predict.wine.prob <- predict(rf.wine, newdata = test,  
                             type = "prob")
```

	0	1	2
21	0.936	0.012	0.052
111	0.060	0.044	0.896
108	0.008	0.056	0.936
110	0.096	0.008	0.896
150	0.040	0.880	0.080
177	0.020	0.968	0.012
2	0.956	0.008	0.036
40	0.916	0.048	0.036
114	0.000	0.004	0.996
87	0.000	0.000	1.000
117	0.004	0.000	0.996
92	0.020	0.044	0.936
...			

Влиятельность (Importance)

```
varImpPlot(rf.wine, sort=F)
```

rf.wine



Оценки влиятельности в табличном виде

```
import.wine <- importance(rf.wine, type=NULL, class=1,  
                          scale=TRUE)
```

	0	1	2	MeanDecreaseAccuracy	MeanDecreaseGini
Alcohol	10.192476	4.98783192	9.52856819	11.5434061	5.1646356
Malic_acid	3.197782	5.79975822	3.02820584	6.3713094	1.7345567
Ash	1.969636	0.48695893	-1.76646303	0.3419617	0.6270103
Alcalinity_of_ash	4.143065	3.66439226	1.53317405	4.6458485	0.7980622
Magnesium	5.678863	-0.05730446	2.89107793	6.0513014	1.2509343
Total_phenols	6.025867	5.81603472	0.20249351	6.6903191	2.3573289
Flavanoids	10.763586	13.94289433	5.92747315	15.1096236	7.7696922
Nonflavanoid_phenols	2.547385	3.92867364	-0.65181020	3.5658697	0.8766331
Proanthocyanins	3.687746	3.59092067	0.02309565	4.7132219	1.1524802
Color_intensity	11.046514	13.61242707	14.55063771	17.2055407	10.0616296
Hue	6.873915	7.70815118	3.92986905	9.2033176	3.2437448
OD280_OD315_of_diluted_wines	9.672069	9.84268530	3.73182573	11.5374625	5.6796674
Proline	14.371782	6.78917660	10.95980135	15.8717116	8.5502914

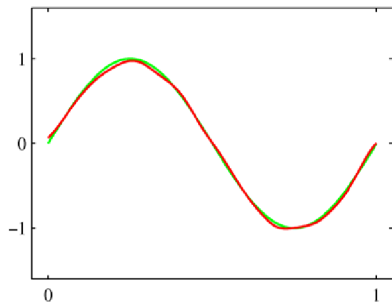
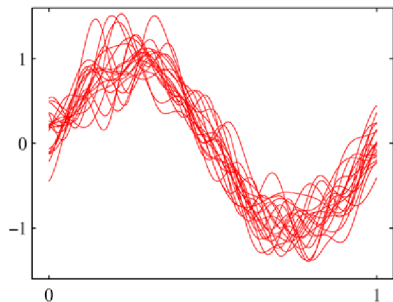
Эффективность бэггинга

объясняется следующими обстоятельствами:

- ▶ Объекты-выбросы могут не попадать в некоторые обучающие подвыборки.
- ▶ Если каждый классификатор имеет высокую дисперсию (нестабильность), то комбинированный классификатор имеет меньшую дисперсию по сравнению с отдельными классификаторами.
- ▶ Благодаря различности базовых алгоритмов, их ошибки взаимно компенсируются при голосовании.

Случайные леса переобучают, но реже чем другие методы.

Как работает бэггинг при регрессии



Бустинг (boosting)

boosting — повышение, усиление, форсирование.

- ▶ Строим слабый классификатор.
- ▶ Смотрим, где он ошибается.
- ▶ Строим следующий классификатор на ошибках предыдущего.
- ▶ И т. д.

Бустинг напоминает разложение в ряд Тейлора

Дано: значение функции $f(x)$ и ее производных в точке x_0 .

Найти: значение функции в некоторой близкой к x_0 точке x .

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \dots$$

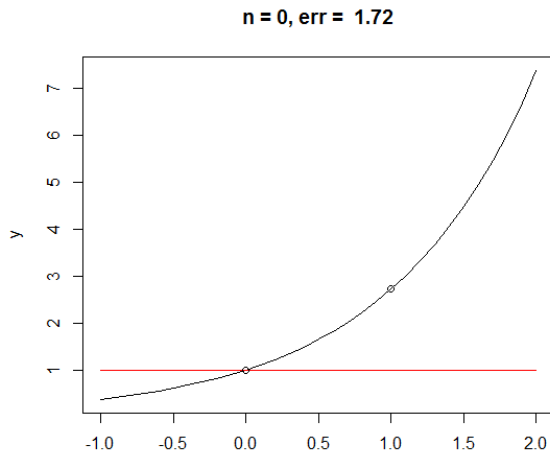
$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \dots$$

...

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \dots$$
$$+ \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + \dots$$

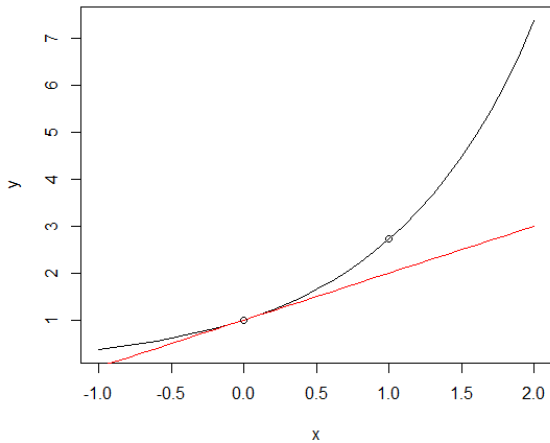
Приближение e^x рядом Тейлора в окрестности $x = 0$

```
x <- seq(-1,2,.1)
y <- exp(x)
y0 <- rep(1,length(x))
```



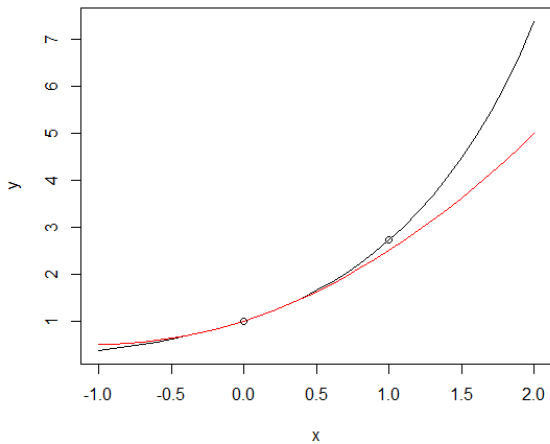
```
y1 <- 1 + x
```

n = 1, err = 0.72



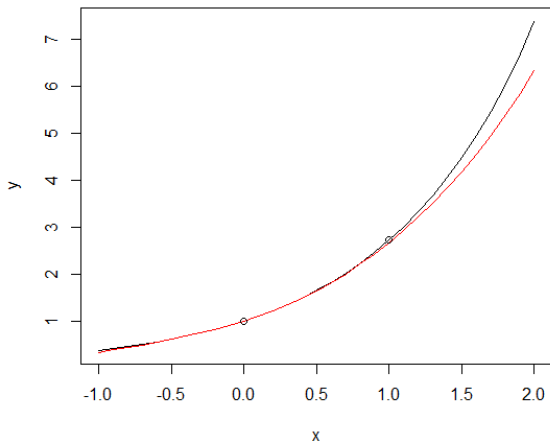
```
y2 <- 1 + x + x^2/2
```

n = 2, err = 0.22



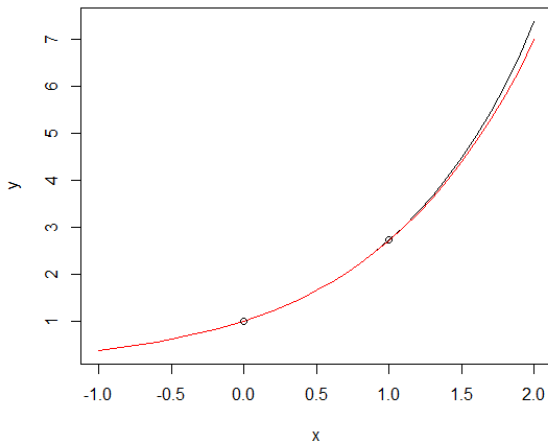
```
y3 <- 1 + x + x^2/2 + x^3/6
```

n = 3, err = 0.05



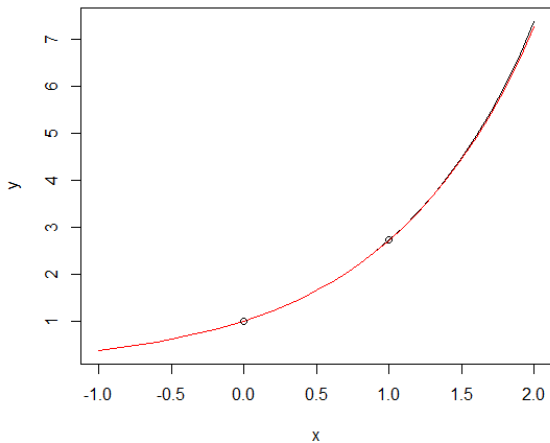
```
y4 <- 1 + x + x^2/2 + x^3/6 + x^4/24
```

n = 4, err = 0.01



```
y5 <- 1 + x + x^2/2 + x^3/6 + x^4/24 + x^5/120
```

n = 5, err = 0



Отличие бустинга от бэггинга

- ▶ Бэггинг — примеры выбираются так, что каждый пример имеет одинаковые шансы попасть в обучающую подвыборку.
- ▶ Бустинг — обучающая выборка на каждой итерации определяется, исходя из ошибок классификации на предыдущих итерациях.

Gradient boosting machine, GBM

Предложил Джером Фридман (Jerome H. Friedman) в 1999 г.

Основные статьи:

- ▶ J.H. Friedman (2001). *Greedy Function Approximation: A Gradient Boosting Machine*, Annals of Statistics, 29 (5): 1189-1232.
- ▶ J.H. Friedman (2002). *Stochastic Gradient Boosting*, Computational Statistics and Data Analysis, 38 (4): 367-378.

Происхождение названия:

- ▶ Gradient — есть элементы метода скорейшего спуска.
- ▶ Boosting — последовательные улучшения.
- ▶ Machine — потому что является одной из тех machine, которых обучают (learning).

Коммерческие названия: MART, TreeNet, MatrixNet

GBM всегда решает задачу регрессии

Дано: $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$.

Найти: функцию $F(\mathbf{x})$, аппроксимирующую данные или минимизирующую функцию потерь.

Функция потерь: разность квадратов

$$L = (1/2) \sum_{i=1}^n (y_i - F(\mathbf{x}_i))^2 \text{ (Gaussian regression)}.$$

GBM выдает числа, которые интерпретируются как вероятность принадлежности к классу.

Идея градиентного бустинга

- ▶ Предположим, что мы угадали (но не точно): $F(x_1) = 0.8$, когда $y_1 = 0.9$, затем $F(x_2) = 1.4$, когда $y_2 = 1.3$, ...
- ▶ Как улучшить модель, если нельзя изменить параметры $F(x)$?
- ▶ **Идея:** Можно добавить дополнительную модель h к F так, что новая функция будет: $F(x) + h(x)$

$$\begin{array}{rclclcl}
F(x_1) + h(x_1) & = & y_1 & & h(x_1) & = & y_1 - F(x_1) \\
F(x_2) + h(x_2) & = & y_2 & & h(x_2) & = & y_2 - F(x_2) \\
& & \dots & & \text{или} & & \dots \\
F(x_n) + h(x_n) & = & y_n & & h(x_n) & = & y_n - F(x_n)
\end{array}$$

Построим регрессионную модель для $h(x)$, т.е. модель для новой обучающей выборки

$$\{(x_1, y_1 - F(x_1)), (x_2, y_2 - F(x_2)), \dots, (x_n, y_n - F(x_n))\}$$

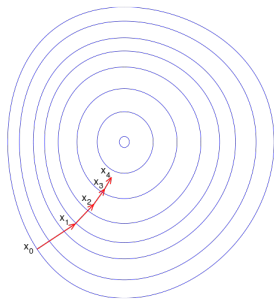
- ▶ Построим регрессионную модель для $h(x)$, т.е. модель для новой обучающей выборки

$$\{(x_1, y_1 - F(x_1)), (x_2, y_2 - F(x_2)), \dots, (x_n, y_n - F(x_n))\}$$

- ▶ Задача $h(x)$ — компенсировать недостаток существующей модели $F(x)$.
- ▶ Что делать, если $F(x) + h(x)$ снова нас не удовлетворяет?
- ▶ **Добавим новую модель $h_2(x)$!**

Как это все соотносится с понятием градиентного спуска?

Градиентный спуск: минимизация функции, двигаясь в направлении, противоположном градиенту



$$\theta_i \leftarrow \theta_i - \rho \frac{\partial J}{\partial \theta_i}$$

Если функция потерь (loss function) $L = (y - F(x))^2/2$, то для поиска $F(x_1), \dots, F(x_n)$ нужно минимизировать

$$J = \sum_{j=1}^n L(y_j, F(x_j))$$

Рассмотрим число $F(x_i)$ как параметр и возьмем производную

$$\frac{\partial J}{\partial F(x_i)} = \frac{\partial \sum_{j=1}^n L(y_j, F(x_j))}{\partial F(x_i)} = \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} = F(x_i) - y_i$$

Отклонение $y_i - F(x_i) = -\partial J / \partial F(x_i)$ — отрицательный градиент.

$$\begin{aligned}
 F(x_i) &\leftarrow F(x_i) + h(x_i) \\
 F(x_i) &\leftarrow F(x_i) + y_i - F(x_i) \\
 F(x_i) &\leftarrow F(x_i) - 1 \frac{\partial J}{\partial F(x_i)} \\
 \theta_i &\leftarrow \theta_i - \rho \frac{\partial J}{\partial \theta_i}
 \end{aligned}$$

- ▶ Модификация F на основе отклонений в нашем случае равносильна модификации F на основе отрицательного градиента.
- ▶ Модифицируем модель, используя градиентный спуск.
- ▶ Оказывается, что градиенты — более общее и полезное понятие, чем отклонения.

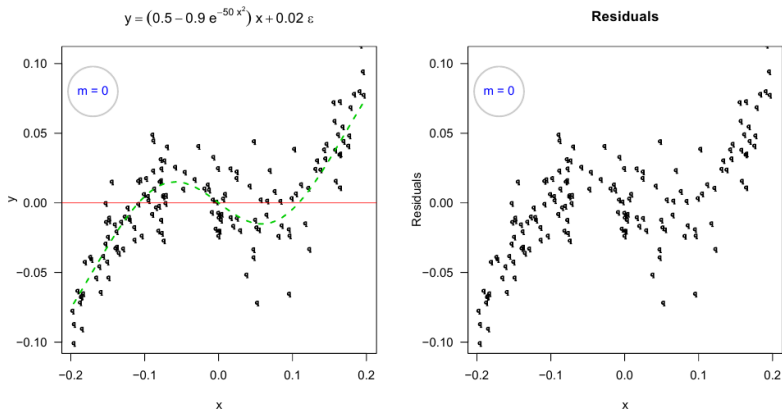
Алгоритм градиентного бустинга для регрессии с квадратичными потерями

$$-g(x_i) = -\frac{\partial J}{\partial F(x_i)} = y_i - F(x_i)$$

1. Начинаем с исходной модели, например, $F(x) = \sum_{j=1}^n y_j/n$.
2. Цикл по $t = 1, \dots, T$:
 - ▶ вычислить $-g_t(x_i)$
 - ▶ построить регрессию h_t по $-g_t(x_i)$
 - ▶ $F(x_i) \leftarrow F(x_i) + \rho_t \cdot h_t(x_i)$
3. Конец цикла по t

Преимущество формулировки алгоритма, использующей градиент, в том, что можно рассматривать другие функции потерь.

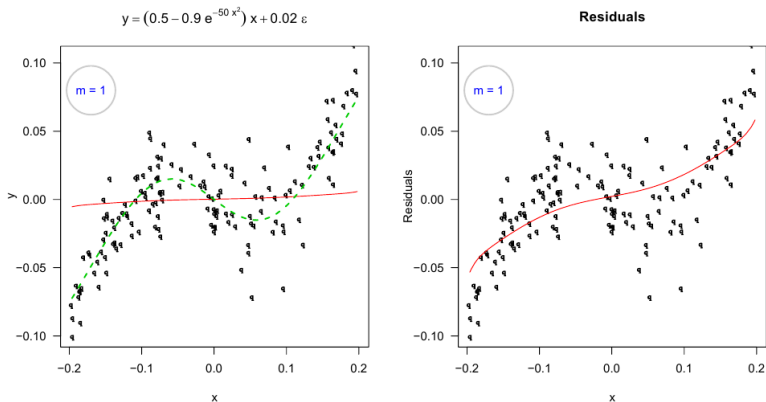
Пример градиентного бустинга



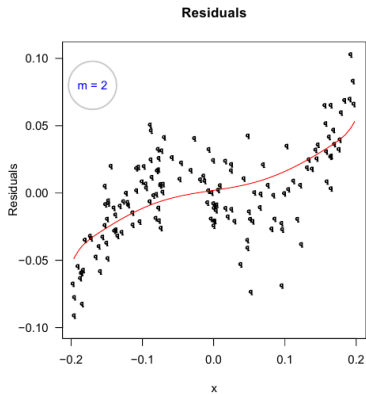
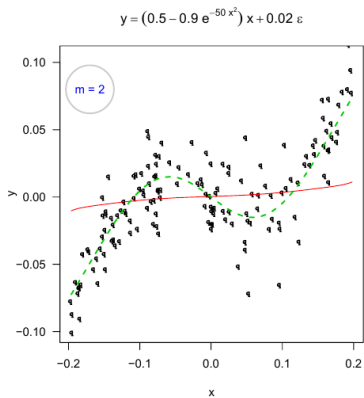
Источник: *Matthias Schmid. Model-based boosting in R. Introduction to Gradient Boosting*

Шаг 1

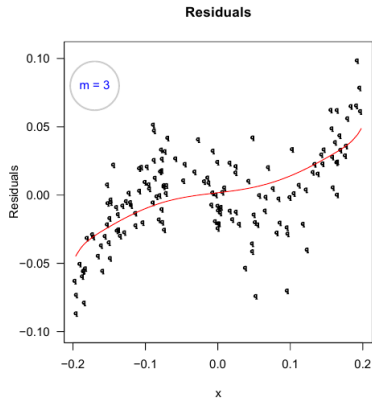
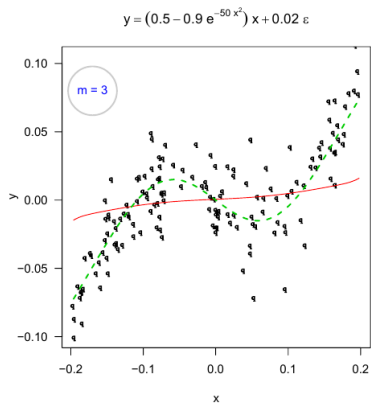
Когда функция потерь является разностью квадратов, градиентный бустинг представляет собой постепенную подгонку остатков регрессионной модели.



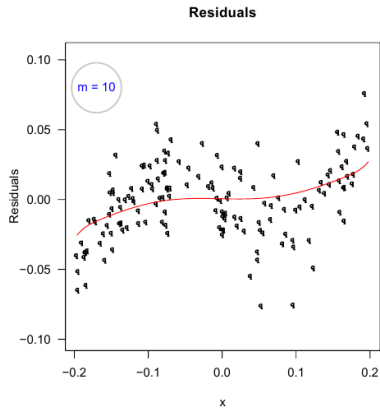
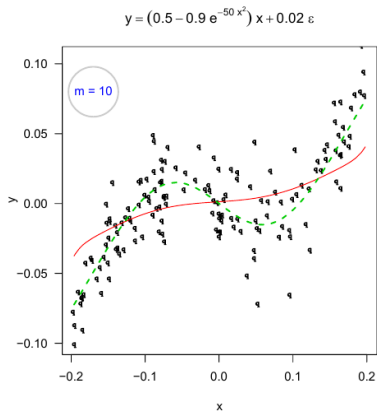
War 2



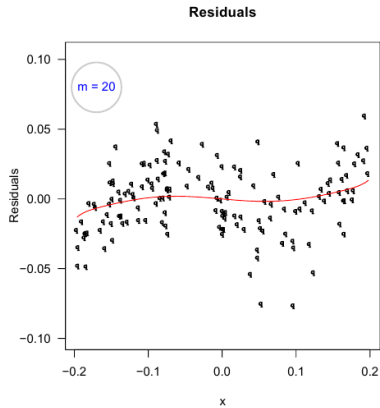
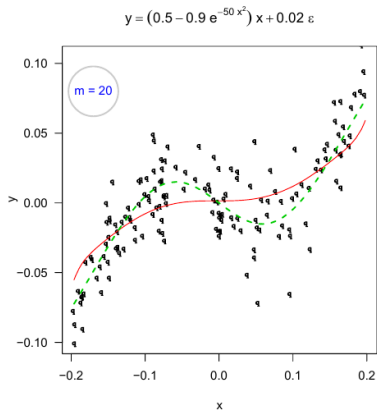
Var 3



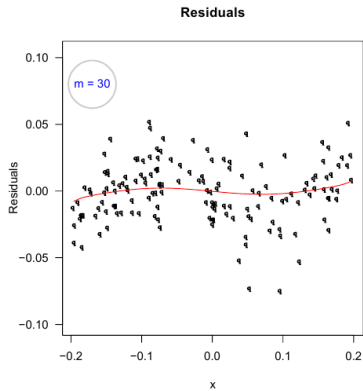
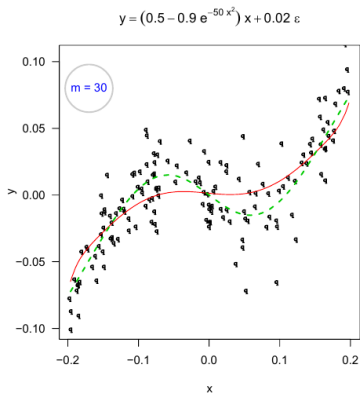
War 10



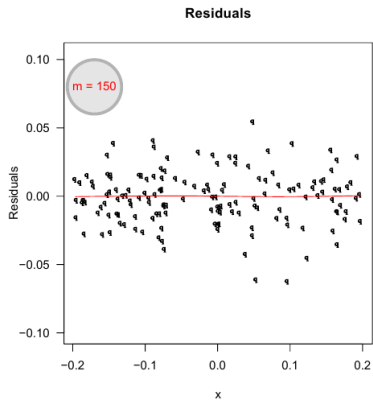
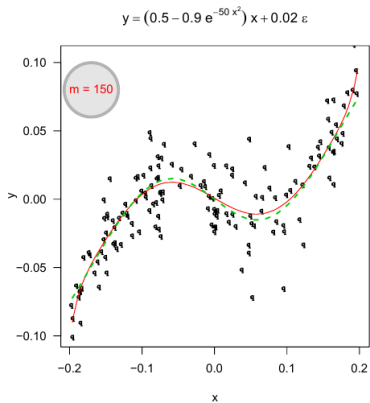
War 20



War 30



War 150



Особенности квадратичной функции потерь в градиентном бустинге

Достаточно простая функция с математической точки зрения.

Не является робастной к выбросам, которые сильно влияют, поскольку величина ошибки берется в квадрате

y_i	0.5	1.2	2	5
$F(x_i)$	0.6	1.4	1.5	1.7
$L = (y - F)^2/2$	0.005	0.02	0.125	5.445

Робастный — устойчивый к помехам, неидеальным условиям работы.

Другие функции потерь (более робастные к выбросам)

- ▶ Абсолютные потери $L(y, F) = |y - F|$
- ▶ Функция потерь Хьюбера (Huber)

$$L(y, F) = \begin{cases} (y - F)^2/2, & \text{если } |y - F| \leq \delta, \\ \delta(|y - F|) - \delta/2, & \text{если } |y - F| > \delta. \end{cases}$$

y_i	0.5	1.2	2	5
$F(x_i)$	0.6	1.4	1.5	1.7
Квадратичн. потери	0.005	0.02	0.125	5.445
Абсолютные потери	0.1	0.2	0.5	3.3
Хьюбера потери ($\delta = 0.5$)	0.005	0.02	0.125	1.525

Градиентный бустинг для регрессии в общем виде

Используем любую дифференцируемую функцию потерь L .

1. Начинаем с исходной модели, например, $F(x) = \sum_{j=1}^n y_j/n$.
2. Цикл по $t = 1, \dots, T$:
 - ▶ вычислить $-g_t(x_i) = -\frac{\partial J}{\partial F(x_i)}$
 - ▶ построить регрессию h_t по $-g_t(x_i)$
 - ▶ $F(x_i) \leftarrow F(x_i) + \rho_t \cdot h_t(x_i)$
3. Конец цикла по t
4. Результат: $F(x) = \sum_{t=1}^T \rho_t h_t(x)$

В общем случае отрицательные градиенты не являются остатками.

Мы используем отрицательные градиенты потому что они менее чувствительны к выбросам по сравнению с остатками.

Итоги

1. Подгоняем аддитивную модель $F(x) = \sum_{t=1}^T \rho_t h_t(x)$, последовательно уточняя ее на каждом шаге.
2. На каждом шаге вводим новое дерево регрессии h , чтобы компенсировать “дефект” модели.
3. Эти “дефекты” описываются отрицательными градиентами.
4. Алгоритм градиентного бустинга можно построить для произвольной функции потерь.
5. Абсолютные потери и функция потерь Хьюбера более робастны к выбросам, чем квадратичная функция потерь.

Достоинства градиентного бустинга

- ▶ Наиболее общий из всех бустингов.
- ▶ Подходит для регрессии, классификации и ранжирования.

Ранжирование

Ранжирование (machine-learned ranking) — задача сортировки набора элементов исходя из их релевантности.

Релевантность понимается по отношению к некому объекту.

1. В задаче информационного поиска объект — это запрос, элементы — всевозможные документы (ссылки на них), а релевантность — соответствие документа запросу
2. В задаче рекомендаций объект — это пользователь, элементы — тот или иной рекомендуемый контент (товары, видео, музыка), а релевантность — вероятность того, что пользователь воспользуется (купит/лайкнет/просмотрит) данным контентом.

Программная реализация в R

- ▶ <https://cran.r-project.org/web/views/MachineLearning.html>
- ▶ Пакет **gbm**, реализует стандартный AdaBoost и алгоритм градиентного бустинга Фридмана.
- ▶ Пакет **mboost**, реализует алгоритмы градиентного бустинга.
- ▶ Пакет **xgboost**, реализует алгоритмы градиентного бустинга.
- ▶ Пакет **ada**, реализует стохастический градиентный бустинг (SGB).

Пакет gbm

Пакет gbm реализует алгоритмы:

- ▶ AdaBoost (Freund and Schapire)
- ▶ gradient boosting machine (Friedman).

Пакет gbm в качестве базовых моделей использует деревья решений. Для применения других функций, могут быть использована библиотека mboost.

По настройке функций: берите за основу значения параметров, принятые по умолчанию.

Пример: классификация вин

```
gbm.wine <- gbm(Wine_type~. , data=wine,  
  distribution="gaussian",      # "laplace"  
  n.trees=ntree.1,             # 500  
  shrinkage=0.05,              # 0.01  
  interaction.depth=5,          # 3-7  
  bag.fraction = 0.66,          # 0.5  
  n.minobsinnode = nodesize.1, # 1  
  cv.folds = 0,                 # 0  
  keep.data=TRUE,  
  verbose=TRUE)
```

Комментарии к `gbm()`

- ▶ `distribution` — строка, содержащая название штрафной функции (критерия качества). “gaussian” — квадратичный штраф; “laplace” — штраф по сумме модулей ошибок. Это самые популярные варианты.
- ▶ `shrinkage` — эмпирическая поправка ν в алгоритме

$$F(x_i) \leftarrow F(x_i) + \nu \cdot \rho_t \cdot h_t(x_i)$$

Часто используют значения: $\nu = 0.01$ или $\nu = 0.05$.

Рекомендовал Фридман, и с этим параметром действительно лучше.

- ▶ `interaction.depth` Дерево решений является функцией тех переменных, которые участвуют в формировании разбиения в каждом не листовом узле. Данный параметр определяет максимальное число переменных, от которых может зависеть каждое дерево решений (тем самым ограничивая его высоту) в модели градиентного бустинга.
Рекомендация: не увлекаться.

Комментарии к `gbm()`

- ▶ `bag.fraction` — вещественное число от 0 до 1, определяющее долю объектов обучающей выборки, используемых на каждой итерации алгоритма. Если объем обучающей выборки равен N , то для обучения каждого одиночного дерева решений будет случайным образом выбрано $[N * \text{bag.fraction}]$ объектов. Таким образом, вместе с бустингом используется также идея бэггинга: каждый следующий классификатор строится на случайной подвыборке из ошибки.
- ▶ `cv.folds` — кратность кросс-валидации (по умолчанию кросс-валидация не проводится).
- ▶ `keep.data` — определяет, будет ли построенная модель хранить данные, использованные при обучении. Целесообразно устанавливать его в `TRUE` при отладке модели и в `FALSE` — в готовом варианте модели.
- ▶ `verbose` — логическая переменная, отвечающая за вывод дополнительной информации о ходе обучения модели.

Что печатает verbose

Iter	TrainDeviance	ValidDeviance	StepSize	Improve
1	0.6697	nan	0.0500	0.0487
2	0.6128	nan	0.0500	0.0558
3	0.5631	nan	0.0500	0.0470
4	0.5169	nan	0.0500	0.0466
5	0.4792	nan	0.0500	0.0270
...				

- ▶ Improve — улучшения критерия качества, достигнутые после добавления очередного классификатора. Показывает, на каком количестве деревьев можно остановиться.
- ▶ TrainDeviance — значение критерия качества на обучающей выборке (Improve — это его убыль за шаг)

В функции predict

```
wine.predict <- predict(gbm.wine, newdata = test,  
                        n.trees = ntree.1)
```

Для случайного леса после выбора оптимального количества деревьев, модель приходилось переобучать. В GBM этого делать не нужно. Достаточно взять часть деревьев. Управляется опцией `n.trees` непосредственно в функции `predict` (в примере мы берем все деревья).

Результаты

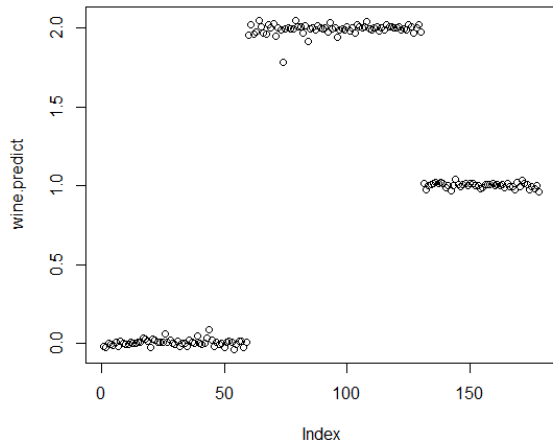
```
wine.predict  
[1] -0.0217419257 -0.0241294775  0.0013745005 -0.0053353134 -0  
[6]  0.0049109084 -0.0175513291  0.0116509357 -0.0009759225 -0  
[11] -0.0031641041  0.0056492618  0.0016705903  0.0031428970  0
```

Что это?

Регрессия.

Результат обучения модели на всем наборе данных wine

```
plot(wine.predict)
```



Как получить результаты классификации?

Нужно выбрать пороговые значения и перекодировать `wine.predict` так, чтобы они указывали на принадлежность к классу: 0, 1 или 2.

```
# Вводим пороговые значения 0.5 и 1.5 и перекодируем  
# вектор результатов регрессии wine.predict в вектор  
# результатов классификации wine.cl  
wine.cl <- rep(0, length(wine.predict))  
wine.cl[(wine.predict > 0.5) & (wine.predict < 1.5)] <- 1  
wine.cl[wine.predict > 1.5] <- 2
```

Пороговые значения можно менять, подстраивая для улучшения качества прогноза.

Проверка на тестовых данных

```
# Оцениваем качество  
table(y.test, wine.cl)
```

```
#      wine.cl  
# y.test  0  1  2  
#      0 19  0  0  
#      1  0 16  0  
#      2  0  0 25
```

Еще о выборе числа деревьев (итераций)

Есть специальная функция, подсказывающая разными методами

```
best.iter <- gbm.perf(gbm.wine, method="OOB")
```

но результаты она дает не лучше, чем просмотр сообщений от `gbm()`.