

Содержание

I Модуль 1	6
1 Лекция 1. Линейный методы регрессии	6
1.1 Напоминание о том, что такое линейная регрессия	6
1.2 Общий вид линейной регрессии	6
1.3 Как мы обучаем модель	6
1.4 Проблемы возникающие при обучении на данных	6
1.5 One-hot encoding - решение проблемы категориальных признаков	6
1.5.1 Проблемы ОНЕ при обучении	6
1.6 Решение проблемы немонотонных признаков	7
1.6.1 Как разбивать эти переменные?	7
1.7 Метрики	7
1.7.1 Среднеквадратичное отклонение (MSE)	7
1.7.2 RMSE	7
1.7.3 R ²	8
1.7.4 MAE	8
1.7.5 MAPE	8
1.7.6 SMAPE	8
1.7.7 MSLE	9
1.7.8 Квантильная регрессия	9
1.8 Небольшое дополнение по линейной регрессии в skelarn	9
2 Лекция 2 (Анастасия)	9
2.0.1 Дополнительные ссылки для проверки	9
2.1 Работа с кодом	9
2.1.1 Скачивание и обработка данных	9
2.1.2 Обучение модели и оценка качества	10
2.2 Классическая предобработка данных для линейной регрессии	10
2.3 Счётчики	11
3 Лекция 3. Логистическая регрессия. Лекция с ФЭН 2021	12
3.1 Модель бинарной классификации	12
3.1.1	12
3.1.2 Функции потерь	12
3.2 Функция потерь	13
3.3 Предсказание для логистической регрессии	13
3.4 Функция потерь	13
3.5 Вероятностная постановка задачи	13
3.6 Правдоподобие и Log-Loss	14
3.7 Выбор алгоритма предсказания	14
3.8 Смысл (w, x) в логистической регрессии	14
3.9 Логарифмическая функция потерь	14
3.10 Бонус. Алгоритм Персептрона Розенблата	14
4 Лекция 3. Логистическая регрессия. Наша лекция	15
4.1 Кодовая реализация	15
4.1.1 Загрузка данных	15
4.1.2 Функция для предсказания сигмоиды и функции потерь Log-Loss	15
4.1.3 Градиент функции потерь	15
4.1.4 Функция предсказания	16
4.2 Различие гипер-параметров и параметров модели	16

5 Лекция 4. SVM, классификаторы, метрики качества классификации. Лекция с ФЭН	17
5.1 Метод опорных векторов	17
5.1.1 Линейно разделимая выборка	17
5.1.1.1 Постановка задачи нахождения гиперплоскости	17
5.2 Линейно неразделимая выборка	17
5.2.1 Постановка задачи нахождения гиперплоскости	17
5.3 Сравнение с логистической регрессией	18
5.4 Метрики качества классификации	18
5.4.1 Accuracy	18
5.4.1.1 Недостаток	18
5.4.2 Матрица ошибок	18
5.4.3 Precision (Точность)	18
5.4.4 Recall (Полнота)	19
5.4.5 F-мера	19
5.4.6 Регулирование точности и полноты	19
5.4.7 ROC-AUC	19
5.4.8 Индекс Джини	19
5.4.9 Precision-Recall кривая	19
6 Лекция 4. SVM. Наша лекция	20
6.1 Кодирование Категориальных признаков	20
6.1.1 One-Hot Encoding	20
6.1.1.1 Недостаток	20
6.1.2 Счётчики	20
6.1.2.1 Недостаток	20
6.1.2.2 Сглаживание	20
6.1.2.3 Отложенная выборка	20
6.1.2.4 Кросс-валидация	20
6.1.3 Expanding Mean	20
7 Лекция 5. Многоклассовая классификация, отбор признаков и методы снижения размерности. Лекция ФЭН	21
7.1 Подходы многоклассовое классификации	21
7.1.1 One-Vs-All	21
7.1.2 All-Vs-All	21
7.2 Метрики качества	21
7.2.1 Micro-average	21
7.2.2 Macro-average	21
7.3 Многоклассовая логистическая регрессия	21
7.3.0.1 Многоклассовый Log-Loss	22
7.4 Отбор признаков	22
7.4.1 Фильтрационные методы	22
7.4.1.1 Преимущества	22
7.4.1.2 sklearn	22
7.4.1.3 Тест χ^2	22
7.4.1.4 Mutual Information	23
7.4.2 Обёрточные методы	23
7.4.2.1 Recursive Feature Elimination	23
7.5 Встроенные в модель методы	23
7.5.1 L_1 регуляризация	23
7.5.2 L_0 регуляризация	23
7.5.3 Информационный критерий	23
7.5.3.1 Критерий Акаике (AIC)	23
7.5.3.2 Критерий Шварца (BIC)	24
7.5.3.3 Отбор признаков с помощью информационных критериев	24
7.6 Метод главных компонент (PCA)	24
7.7 Manifold Embeddings	25
7.7.1 MultiDimensional Scaling (MDS)	25
7.7.2 ISOMAP	25
7.7.3 TSNE	25
7.7.3.1 Близость объектов в исходном пространстве	25

8 Лекция 6. Нелинейные алгоритмы классификации и регрессии. Лекция ФЭН	27
8.1 Наивный байесовский классификатор	27
8.1.1 Теория Байеса	27
8.1.1.1 В случае нескольких признаков	27
8.1.1.2 Преимущества	27
8.1.1.3 Недостатки	27
8.2 Метод ближайших соседей	27
8.2.0.1 Классификация нового объекта	27
8.3 Решающие деревья	27
8.3.1 Что влияет на построение решающего дерева	28
8.3.2 Критерий информативности	28
8.3.2.1 Функции потерь	28
8.3.2.2 H(R) в задачах мягкой классификации	28
8.3.2.3 Критерий Джини	29
8.3.2.4 Энтропийный критерий	29
8.3.3 Критерий останова	29
8.3.4 Плюсы решающих деревьев	29
8.3.5 Минусы решающих деревьев	29
9 Лекция 6. Решающие деревья. Наша Лекция. Елена	30
9.1 ROC-AUC: интуиция	30
9.1.1 ROC-AUC алгоритм	30
9.2 Решающие деревья	31
9.2.1 Задачи	31
9.2.1.1 Задача 1	31
9.2.1.2 Решение задачи 1	31
9.2.1.3 Задача 3	31
9.2.1.4 Решение задачи 3	32
9.2.1.5 Задача 4	32
9.2.1.6 Решение задачи 4	32
9.3 Работа с кодом	32
9.3.1 Подключение библиотек	32
9.3.2 Генерация выборки для задачи регрессии	32
9.3.3 Функция для обучения классификатора и построения разделяющей прямой	32
9.3.4 Сравнение качества между логистической регрессией и деревом решений	33
9.3.5 Генерация выборки логического ИЛИ	33
9.3.6 Сравнение качества между логистической регрессией и деревом решений	33
9.3.7 Переобучение дерева	33
9.3.7.1 Генерация выборки	33
9.3.7.2 Перебор гиперпараметров и отображение результата	33
9.3.7.3 Дерево с нулевой ошибкой	34
9.3.8 Неустойчивость	34
9.3.9 Анализ деревьев на основе датасета Бостона	34
9.3.9.1 Загрузка датасета	34
9.3.9.2 Построение и отображение дерева с предикатами	34
9.3.9.3 Получение построенных параметров дерева	34
9.3.9.4 Построение зависимости MSE от гиперпараметра max_depth и отбор лучших значений на кросс-валидирующй выборке	35
9.3.9.5 Построение зависимости MSE от гиперпараметра max_samples_leaf и отбор лучших значений на кросс-валидирующй выборке	35
9.3.9.6 Разделение выборки на обучающую и тестовую	35
9.3.9.7 Построение дерева без ограничивающих параметров	35
9.3.9.8 Поиск лучших параметров max_depth и max_features	36
9.3.9.9 Стрижка дерева	36
9.3.9.10 Перебор по гиперпараметру регуляризации	36
9.3.9.11 Применение лучшего параметра регуляризации	37
9.3.10 Решающее дерево своими руками	37
9.4 Калибровка вероятностей	37
9.4.1 Программируем это всё дело	37
9.4.1.1 Генерация выборки	37
9.4.1.2 Обучение моделей	37

9.4.1.3	Отобразим наши вероятности	38
10	Материалы к лекции 6	39
10.1	Статья про разложение ошибки	39
10.2	Уроки 6.4, 6.5 со стекика	40
11	Лекция 7. Композиция алгоритмов, разложение ошибки и лес	41
11.1	Смещение и разброс	41
11.1.1	Формула для разложения ошибки на смещение и разброс	42
11.2	Бэггинг	42
11.2.1	Бутстрэп	42
11.2.2	Описание бэггинга	42
11.2.3	Смещение и разброс у бэггинга	43
11.2.4	Как сделать некоррелированные алгоритмы	43
11.3	Случайный лес (Random Forest)	43
11.3.1	Практические рекомендации	43
11.3.2	Out-Of-Bag ошибка	43
12	Лекция 7. Случайный лес. Наша лекция	44
12.1	Программируем	44
12.1.1	Загружаем датасет	44
12.1.2	Обучим решающее дерево без ограничений	44
12.1.3	Получим примерное смещение и разброс	44
12.1.4	Посмотрим на смещение и разброс у бэггинга деревьев	44
12.1.5	Посмотрим как это повлияло на MSE	44
12.1.6	Построим случайный лес	45
12.1.7	Переобучение случайного леса	45
12.2	Важность признаков	45
12.2.1	Программируем изучение важности признаков	45
13	Лекция 8. Бустинг. Лекция с ФЭН	47
13.1	Случайный лес	47
13.2	Бустинг	47
13.2.1	Частный случай бустинга	47
13.2.2	Алгоритм бустинга для MSE	47
13.3	Выбор базовых алгоритмов	48
13.4	Смещение и разброс бустинга	48
13.5	Стохастический градиентный бустинг	48
13.6	Имплементации градиентного бустинга	48
13.6.1	Xgboost	49
13.6.1.1	Особенности xgboost	49
13.6.2	CatBoost	49
13.6.2.1	Особенности CatBoost	49
13.6.3	LightGBM	50
14	Лекция 8. Бустинги. Наша лекция	51
14.1	Программируем	51
14.1.1	Устанавливаем библиотеки	51
14.1.2	Импорт библиотек	51
14.1.3	Генерация датасета и функция для визуализации решений дерева	51
14.1.4	Отобразим визуализацию дерева для CatBoost и вычислим ROC-AUC	52
14.1.5	Обучаем обычный градиентный бустинг и анализируем качество от количества деревьев	52
14.1.6	Сделаем то же самое для CatBoost	52
14.1.7	Сравнение GradientBoostingClassifier и CatBoost	53
14.1.8	Решение XGBoost	53
14.2	CatBoost для решения задачи + интерпритация признаков	54
14.2.1	Загрузка данных и разбитие для обучения и теста	54
14.2.2	Обучим модель	54
14.2.3	Построим предсказание	54
14.2.4	Переберём параметры	54

14.3	Оценка важности признаков	55
14.3.1	Вспомогательные функции для оценки	55
14.3.2	Визуализация важности признаков	55
14.3.3	Визуализация важности признаков при случайной перестановке	56
14.4	Блендинг и стекинг	56
14.4.1	Стекинг	56
14.5	Блендинг	56
14.6	Программируем блендинг	57
14.6.1	Загружаем данные	57
14.6.2	Проверка качества алгоритмов, если просто обучить на train и проверить на test и блендинг на глазок	57
14.6.3	Простой блендинг	58
14.6.4	Подбор весов	58
14.7	Полноценный блендинг	58
14.7.1	Загрузка данных	58
14.7.2	Использование блендинга	58
14.8	Используем стекинг	59
15	Материалы к лекции 9. Многоклассовая классификация. Уроки со stepik	60
15.1	Многоклассовая и multi-label классификация	60
15.1.1	Сведение к бинарной классификации	60
15.1.1.1	One-Versus-Rest	60
15.1.1.2	One-Vs-One	60
15.2	Метрики качества многоклассовой классификации	61
15.2.1	Accuracy	61
15.2.2	Recall и f1-score	61
15.2.2.1	Макроусреднение (macro-average)	62
15.2.2.2	Взвешенное усреднение (weighted-average)	62
15.2.2.3	Микроусреднение (micro-average)	62
15.2.3	Результат в python	62
15.3	Метод ближайших соседей (KNN)	63
15.3.1	Алгоритм метода	63
15.3.2	Влияние гиперпараметра k	63
15.3.3	Выбор метрики	64
15.3.3.1	Евклидова метрика	64
15.3.3.2	Манхэттенское расстояние	64
15.3.3.3	Расстояние Хемминга	64
15.3.3.4	Мера Жаккара	64
15.3.4	Масштабирование данных для KNN	64
15.3.5	Обобщения KNN	65
15.3.6	KNN в задачах регрессии	66
15.3.7	Преимущества и недостатки KNN	67
15.3.8	Реализация в питоне	67
15.4	Быстрый поиск соседей	67
15.4.1	Хеширование	68
15.4.2	Метод случайных проекций	68
15.4.3	MiniHashing	68
15.4.3.1	Мера Жаккара	69
15.4.3.2	Матрица текстов	69
15.4.3.3	MinHashing	69
15.4.3.4	Locality-sensitive hashing (LSH)	71

Часть I

Модуль 1

1. Лекция 1. Линейный методы регрессии

Базовые вещи знаем из курса математики - как обучать с помощью градиентного спуска и как решать аналитически.

1.1. Напоминание о том, что такое линейная регрессия

Предположим, что мы хотим предсказать стоимость одома y по его площади (x_1) и количеству комнат (x_2).

Линейная модель для предсказания стоимости: $\alpha(x) = w_0 + w_1x_1 + w_2x_2$,

где w_0, w_1, w_2 - параметры модели (веса)

Линейная регрессия означает то, что все веса линейны от признаков (сами признаки могут быть любыми).

1.2. Общий вид линейной регрессии

$$\alpha(x) = w_0 + w_1x_1 + \dots + w_nx_n$$

Сокращённая запись:

$$\alpha(x) = w_0 + \sum_{i=1}^n w_i x_i$$

Запись через скалярное произведение (с добавлением признака $x_0 = 1$):

$$\alpha(x) = (w, x)$$

1.3. Как мы обучаем модель

Мы просто пытаемся минимизировать ошибку предсказаний:

$$Q(\alpha, X) = \frac{1}{l} \sum_{i=1}^l (\alpha(x_i) - y_i)^2 = \frac{1}{l} \sum_{i=1}^l ((w, x_i) - y_i)^2 \rightarrow \min_w$$

где l - кол-во объектов

1.4. Проблемы возникающие при обучении на данных

Допустим к предсказанию стоимость квартиры мы захотели добавить два признака: x_3 - район, в котором находится квартира и x_4 - удалённость от МКАД.

- Что такое район? Это категориальный признак и непонятно как на нём обучать.
- Также, когда мы добавляем признак, мы предполагаем, что он как-то монотонно влияет на ответ. А вот с удалённостью от МКАД так нельзя сказать. Есть районы, которые лежат вне МКАД, но при этом там квартиры дороже, чем внутри.

1.5. One-hot encoding - решение проблемы категориальных признаков

One-hot encoding (OHE) - мы создаём новые числовые столбцы, каждого из которых является индикатором одного из районов.

1.5.1. Проблемы ОНЕ при обучении

Но при ОНЕ мы можем столкнуться с проблемой, что мы один из столбцов ОНЕ можем выразить через остальные, как 1 минус сумма остальных. Столбцы линейно зависимы. Это плохо для линейных моделей:

- При аналитическом решении у нас сломается формула

- При градиентном спуске, если x_1, \dots, x_l линейно зависимы, то $\exists v : (v, x) = 0$, а это означает, что мы можем добавить сколько угодно добавить $(w + \alpha v, x)$ и предсказание не поменяется, а вектор весов получается неоднозначный и получается много решений у задачи. При минимизации функции потерь могут получиться большие веса, а большие веса - переобучение.

В качестве решения мы можем выкинуть одну из колонок ОНЕ (в sklearn за это отвечает параметр `drop_first`). Мы можем так делать, если знаем, что новый район не добавится. Если районы будут добавляться - мы можем не выкидывать, потому что линейная зависимость пропадает.

1.6. Решение проблемы немонотонных признаков

Мы можем разбить признак удалённости от МКАД на отрезки и сделать что-то типа ОНЕ: сделать признаки, что квартира находится в $[0; 10]$ км от МКАД, в $[10; 30]$ км от МКАД. Получаются бинарные признаки

1.6.1. Как разбивать эти переменные?

Можно разбивать по квантилям - универсальное решение, чтобы не думать.

1.7. Метрики

Функцию потерь мы минимизируем. Метрика - другая функция, которую мы считаем, чтобы понять - насколько модель хорошая.

1.7.1. Среднеквадратичное отклонение (MSE)

Mean Squared Error - MSE

$$MSE(\alpha, X) = \frac{1}{l} \sum_{i=1}^l (\alpha(x_i) - y_i)^2$$

где l - количество объектов. Почему чаще всего используют MSE для обучения линейной регрессии? Минимизация этой ошибки - это максимизация правдоподобия, в случае если мы предполагаем, что наши данные имеют нормальное распределение, а это часто так.

Какие плюсы:

- Позволяет сравнивать модели между собой
- Подходит для контроля качества во время обучения

С какими проблемами мы столкнёмся?

- Выбросы
- Плохая интерпретируемость
- Неочевидно, хорошая ошибка или нет, нужно, например, сравнивать со средним значением целевой переменной, чтобы понять, хорошо мы предсказываем или нет. Неограничена сверху.

1.7.2. RMSE

Root Mean Squared Error - RMSE

$$RMSE(\alpha, X) = \sqrt{\frac{1}{l} \sum_{i=1}^l (\alpha(x_i) - y_i)^2}$$

Плюсы:

- Все плюсы MSE
- Сохраняет единицы измерения (в отличие от MSE)

Минусы:

- Тяжело понять, насколько хорошо данная модель решает задачу, так как тоже не ограничена сверху, как и MSE

1.7.3. \mathbf{R}^2

Коэффициент детерминации - R^2

$$R^2(\alpha, X) = 1 - \frac{\sum_{i=1}^l (\alpha(x_i) - y_i)^2}{\sum_{i=1}^l (y_i - \bar{y})^2}$$

где $\bar{y} = \frac{1}{l} \sum_{i=1}^l y_i$

Коэффициент детерминации - это доля дисперсии целевой переменной, объясняемая моделью.

- Чем ближе R^2 к 1, тем лучше модель объясняет данные
- Чем ближе R^2 к 0, тем ближе модель к константному предсказанию
- Отрицательный R^2 говорит о том, что модель плохо решает задачу, и даже хуже, чем константное предсказание.

1.7.4. MAE

Mean Absolute Error - MAE

$$MAE(\alpha, X) = \frac{1}{l} \sum_{i=1}^l |\alpha(x_i) - y_i|$$

1.7.5. MAPE

Mean Absolute Percentage Error - MAPE

$$MAPE(\alpha, X) = \frac{1}{l} \sum_{i=1}^l \frac{|y_i - \alpha(x_i)|}{|y_i|}$$

MAPE измеряет относительную ошибку

Плюсы:

- Ограничена: $0 \leq MAPE \leq 1$
- Хорошо интерпретируема: например, MAPE = 0.16 значит, что ошибка модели в среднем составляет 16% от фактических значений

Минусы:

- По разному относится к недо- и перепрогнозу. Например, если правильный ответ $y = 10$, а прогноз $\alpha(x) = 20$, то ошибка $\frac{|10-20|}{10} = 1$, а если ответ $y = 30$, то ошибка $\frac{|30-20|}{30} = \frac{1}{3} = 0.33$

1.7.6. SMAPE

Symmetric Mean Absolute Percentage Error - SMAPE. Симметричный вариант MAPE:

$$SMAPE(\alpha, X) = \frac{1}{l} \sum_{i=1}^l \frac{|y_i - \alpha(x_i)|}{(|y_i| + |\alpha(x_i)|)/2}$$

SMAPE - попытка сделать симметричным прогноз - то есть дать одинаковую ошибку для недо- и перепрогноза

Проверим: пусть правильный ответ $y = 10$, а прогноз $\alpha(x) = 20$, то ошибка $= \frac{|10-20|}{|10+20|/2} = \frac{2}{3} = 0.67$, а если ответ $y = 30$, то ошибка $\frac{|30-20|}{|30+20|/2} = \frac{2}{5} = 0.4$

Сейчас уже в среде прогнозистов сложилось более-менее устойчивое понимание, что SMAPE не является хорошей ошибкой. Тут дело не только в завышении прогнозов, но и в том, что наличие прогноза в знаменателе позволяет манипулировать результатами оценки.

1.7.7. MSLE

Mean Squared Logarithmic Error - MSLE. Среднеквадратическая логарифмическая ошибка

$$MSLE(\alpha, X) = \frac{1}{l} \sum_{i=1}^l (\log(\alpha(x_i) + 1) - \log(y + 1))^2$$

Особенности:

- Подходит для задач с неотрицательной целевой переменной ($y \geq 0$)
- Штрафует за отклонения в порядке величин
- Штрафует заниженные прогнозы сильнее, чем завышенные

1.7.8. Квантильная регрессия

Квантильная функция потерь:

$$Q(\alpha, X^l) = \sum_{i=1}^l \rho_\tau(y_i - \alpha(x_i))$$

где $\rho_\tau(z) = (\tau - 1)[z < 0]z + \tau[z \geq 0]z = (\tau - \frac{1}{2}) + \frac{1}{2}|z|$

Параметр $\tau \in [0; 1]$. Чем больше τ , тем больше штрафуем за занижение прогноза.

Теорема: Пусть в каждой точке $x \in X$ (пространство объектов) задано распределение $p(y|x)$ на ответах для данного объекта. Тогда оптимизация функции потерь $\rho_\tau(z)$ даёт алгоритм $\alpha(x)$, приближающий τ -квантиль распределения ответов в каждой точке $x \in X$

Иными словами: допустим мы предсказываем стоимость квартиры. Если мы используем MSE, то мы получим \bar{y} , если по признакам все параметры совпадают у разных объектов, но целевая переменная однаковая. К примеру $\{10, 20, 90\}$ выдаст $\{40\}$. Мы можем попросить завысить прогноз, поставить квантиль.

- Если мы хотим завысить прогноз, то берём τ ближе к единице.
- Если занизить - берём τ ближе к нулю.

1.8. Небольшое дополнение по линейной регрессии в sklearn

В sklearn класс LinearRegression всегда использует MSE. В то время, как в SGDRegressor, эта же самая линейная регрессия, но мы можем подставить нужную функцию потерь.

2. Лекция 2 (Анастасия)

2.0.1. Дополнительные ссылки для проверки

Будут позже

2.1. Работа с кодом

2.1.1. Скачивание и обработка данных

```
import pandas as pd
X_raw = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-database/autos/import...
print(X_raw.head())

print("nan_is_ ", X_raw[25].isna().sum())
X_raw = X_raw[X_raw.notna()]
y = X_raw[25]
X_raw = X_raw.drop(25, axis=1)
print(X_raw.shape, len(y))

from sklearn import impute

% get cat features
cat_feature_mask = (X_raw.dtypes == "object").values
```

```

X_real = X_raw[X_raw.columns[~cat_feature_mask]]
mis_replacer = impute.SimpleImputer(strategy="mean")
X_no_miss_real = pd.DataFrame(data=mis_replacer.fit_transform(X_real), columns=X_real.columns)

X_cat = X_raw[X_raw.columns[cat_feature_mask]].fillna("")
X_cat = X_cat.reset_index(drop=True)

X_no_miss = pc.concat([X_no_miss_real, X_cat], axis=1)

X_dum = pd.get_dummies(X_no_miss, drop_first=True)
print(X_dum.shape)
print(X_dum.head())

from sklearn import preprocessing

normalizer = preprocessing.MinMaxScaler()
X_real_norm_np = normalizer.fit_transform(X_dum)
X = pd.DataFrame(data=X_real_norm_np)

```

2.1.2. Обучение модели и оценка качества

```

from sklearn.model_selection import train_test_split

Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.25, random_state=123)

from sklear.linear_model import LinearRegression

model = LinearRegression()
model.fit(Xtrain, ytrain)

pred_mse = model.predict(Xtest)

from sklearn.metrics import r2_score

print(r2_score(y_test, pred_mse))
print(r2_score(ytrain, model.predict(Xtrain)))

from sklearn import SGDRegressor

lr_mse = SGDRegressor(loss='squared_loss')
lr_mse.fit(Xtrain, ytrain)

pred_mse = lr_mse.predict(Xtest)

print(r2_score(ytest, pred_mse))

lr_mae = SGDRegressor(loss='squared_loss')
lr_mae.fit(Xtrain, ytrain)

pred_mae = lr_mae.predict(Xtest)

print(r2_score(ytest, pred_mae))

```

2.2. Классическая предобработка данных для линейной регрессии

- pairplot by seaborn

- бинаризировать данные
- Ordinal Encoding
- target encoding - может быть переобучение
- category_encoders

2.3. Счётчики

Counts

$$counts(u, K) = \sum_{(x,y) \in X} [f(x) = u]$$

Successes

$$successes_k(u, X) = \sum_{(x,y) \in X} [f(x) = u][y = k]$$

Замена категориального счётчика на вещественный

$$g_k(x, X) = \frac{successes_k(f(x), X) + c_k}{counts(f(x), X) + \sum_{m=1}^K c_m}$$

3. Лекция 3. Логистическая регрессия. Лекция с ФЭН 2021

3.1. Модель бинарной классификации

Мы можем всё ещё использовать нашу линейную регрессию таким образом:

$$\alpha(x, w) = \text{sign}\left(\sum_{j=1}^l w_j x_j\right)$$

где

- Если $\sum_{j=1}^l w_j x_j > 0$, то $\text{sign}(\sum_{j=1}^l w_j x_j) = +1$, то есть объект отнесён к положительному классу
- Если $\sum_{j=1}^l w_j x_j < 0$, то $\text{sign}(\sum_{j=1}^l w_j x_j) = -1$, то есть объект отнесён к отрицательному классу
- Значит $\sum_{j=1}^l w_j x_j = 0$ - уравнение разделяющей границы между классами. Это уравнение плоскости (или прямой в двумерном случае), поэтому классификатор является линейным

В данном случае мы будем минимизировать такой функционал ошибки:

$$Q(\alpha, X) = \frac{1}{n} \sum_{i=1}^n [\alpha(x_i) \neq y_i] \rightarrow \min$$

3.1.1.

Отступ

Давайте обозначим $M_i = y_i \cdot (w, x_i)$ - отступ на i -м объекте. В терминах отступа мы можем переопределить нашу задачу:

Какой должен быть знак у отступа ($\text{sign}(M_i)$), если $\alpha(x_i) = y_i$?

На самом деле знак положительный. Нужно рассмотреть $\{-, +\}^2$ и понятно откуда это получается.

Тогда задача минимизации ошибки будет $\frac{1}{n} \sum_{i=1}^n [M_i < 0] \rightarrow \min$

Кроме этого величина отступа M обозначает степень уверенности классификатора в ответе - чем ближе M к нулю, тем меньше уверенность в ответе.

Сложность с минимизацией этой функции - то что от неё нельзя взять производную - эта функция разрывна. Давайте заменим её на другую функцию потерь, которая будет решать примерно ту же самую задачу.

Пусть у нас была эта функция $L(\alpha, y) = L(M) = [M < 0]$ - разрывная функция потерь

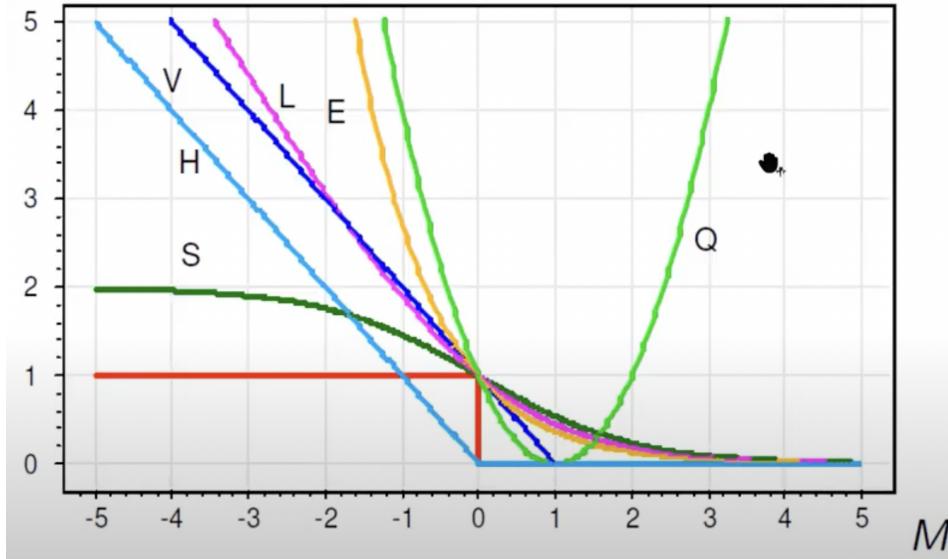
А теперь мы найдём какую-то \tilde{L} , такую, что $L(M) \leq \tilde{L}(M)$, где $\tilde{L}(M)$ - непрерывная или гладкая функция потерь, тогда

$$Q(\alpha, X) = \frac{1}{n} \sum_{i=1}^n L(y_i \cdot (w, x_i)) \leq \frac{1}{n} \sum_{i=1}^n \tilde{L}(y_i \cdot (w, x_i)) \rightarrow \min$$

3.1.2. Функции потерь

Минимизируя различные функции потерь, получаем разные результаты. Поэтому разные функции потерь определяют различные классификаторы.

- $L(M) = \log(1 + e^{-M})$ - логистическая функция потерь
- $V(M) = (1 - M)_+ = \max(0, 1 - M)$ - кусочно-линейная функция потерь (метод опорных векторов PCA)
- $H(M) = (-M)_+ = \max(0, -M)$ - кусочно-линейная функция потерь (персентрон)
- $E(M) = e^{-M}$ - экспоненциальная функция потерь
- $S(M) = \frac{2}{1+e^{-M}}$ - сигмоидная функция потерь
- $[M < 0]$ - пороговая функция потерь



3.2. Функция потерь

Очевидно, что для градиентного спуска теперь мы используем стандартный спуск по формуле

3.3. Предсказание для логистической регрессии

Формула предсказания для логистической регрессии выглядит как $\sigma(w^T x)$, где $\sigma(z) = \frac{1}{1+e^{-z}} \in (0; 1)$

$$w^{(k)} = w^{(k-1)} - \eta \cdot \nabla Q(w^{(k-1)})$$

Где мы предсказываем $y = +1$, если $\alpha(x, w) \geq 0.5$ Где $\alpha(x, w) = \sigma(w^T x) \geq 0.5$, если $w^T x \geq 0$

Получаем, что

- $y = +1$ при $w^T x > 0$
- $y = -1$ при $w^T x < 0$

То есть, $w^T x = 0$ - разделяющая гиперплоскость

3.4. Функция потерь

Если мы будем использовать MSE, то для совсем неправильного класса (вероятность нужного = 0) у нас будет ошибка $(1 - 0)^2 = 1$, это очень мало.

Давайте возьмём логистическую функцию потерь:

$$Q(w) = - \sum_{i=1}^l ([y_i = +1] \cdot \log(\alpha(x_i, w)) + [y_i = -1] \cdot \log(1 - \alpha(x_i, w)))$$

В этом случае:

- если $\alpha(x, w) = 1$ и $y = +1$, то штраф $L(\alpha, y) = 0$
- если $\alpha(x, w) \rightarrow 0$ и $y = +1$, то штраф $L(\alpha, y) \rightarrow +\infty$

3.5. Вероятностная постановка задачи

У нас могут быть разные объекты с одинаковым описанием (просто самого описания может быть недостаточно): на примере задачи "вернёт ли клиент кредит возраст у людей может быть одинаковый, задача одинаковая, профессия одна и та же, но мы можем не знать о жизненной ситуации людей. Поэтому объекты с одним признаком описаниям могут иметь разные значения целевой переменной.

Цель: построить алгоритм $b(x)$ в каждой точке x предсказывающий $p(y = +1|x)$.

Пусть объект x встречается в выборке n раз с ответами $\{y_1, \dots, y_n\}$. Хотим, чтобы алгоритм выдавал вероятность положительного класса:

$$b_*(x) = \arg \min_{b \in \mathbb{R}} \frac{1}{n} \sum_{i=1}^n L() y_i, b \approx p(y = +1|x)$$

По закону больших чисел при $n \rightarrow \infty$ получаем

$$b_*(x) = \arg \min_{b \in \mathbb{R}} E[L(y, b)x]$$

Отсюда получаем условие на функцию потерь

$$\arg \min E[L(y, b)|x] = p(y = +1|x)$$

LogLoss подходит, а вот MAE не подходит.

3.6. Правдоподобие и Log-Loss

Вероятности, которые выдаёт алгоритм $b(x)$, должны согласовываться с выборкой

Вероятность того, что в выборке встретится объект x с классом y :

$$b(x)^{[y=+1]} \cdot (1 - b(x))^{[y=-1]}$$

Правдоподобие выборки:

$$(b, X) = \prod_{i=1}^l b(x_i)^{[y_i=+1]} \cdot (1 - b(x_i))^{[y_i=-1]}$$

Для нахождения оптимальных параметров алгоритма можно воспользоваться методом максимума правдоподобия (ММП):

$$(b, x) = \prod_{i=1}^l b(x_i)^{[y_i=+1]} \cdot (1 - b(x_i))^{[y_i=-1]} \rightarrow \max_b$$

Прологарифмировав и поставив минус в начале - получим аналогичную задачу, да ещё и ровно Log-Loss:

$$-\sum_{i=1}^l ([y_i = +1] \log b(x_i) + [y_i = -1] \log(1 - b(x_i))) \rightarrow \min_b$$

3.7. Выбор алгоритма предсказания

Хотим взять алгоритм $b(x)$ такой, чтобы он возвращал числа из отрезка $[0; 1]$

Можно взять $b(x) = \sigma(w^T x)$, где σ - любая монотонно неубывающая функция с областью значений $[0; 1]$

Возьмём сигмоиду $\sigma(z) = \frac{1}{1+e^{-z}}$

3.8. Смысл (w , x) в логистической регрессии

Логистическая регрессия в каждой точке x предсказывает вероятность того, что x принадлежит к положительному классу $p(y = +1|x)$. То есть $p(y = +1|x) = \frac{1}{1+e^{-w^T x}}$. Отсюда можно выразить $(w, x) = w^T x$:

$$(w, x) = w^T x = \log \left(\frac{p(y = +1|x)}{p(y = -1|x)} \right)$$

И эта величина логарифма отношения называется логарифм отношения шансов (log odds). Из формулы видно: что величина может принимать любое значение.

3.9. Логарифмическая функция потерь

Тогда функция потерь может быть записана в куда более красивом виде:

$$L(b, x) = \sum_{i=1}^l \log(1 + e^{-y_i(w, x)})$$

3.10. Бонус. Алгоритм Персептрана Розенблата

Персептрон - это простейшая модель классификации, при этом являющаяся предшественником нейронных сетей. Он решает задачу классификации с двумя классами и бинарными признаками (каждый признак равен либо 0, либо 1).

Сам алгоритм выглядит как:

$$\alpha(x, w) = [w_1 x_1 + \dots + w_n x_n > 0] = [(w, x) > 0]$$

4. Лекция 3. Логистическая регрессия. Наша лекция

Логистическая регрессия решает задачу классификации с помощью регрессии.

Строится линейная модель. Она умеет выдавать вероятность определения какому либо классу и отлично это делает, если классы хорошо линейно разделяются.

У моделей в питоне есть метод `predict_proba`, который выдаёт вероятность попадания в тот или иной класс.

Как получается эта вероятность? В общем случае - это степень уверенности алгоритма с определённым классом. В случае с Логистической регрессией мы минимизируем `log_loss`, то есть константное предсказание самое лучшее будет $\log_loss(a, y) = -[y \log(a) + (1 - y) \log(1 - a)]$

4.1. Кодовая реализация

4.1.1. Загрузка данных

```
import random
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
%matplotlib inline

df = pd.read_csv("BloodPressure.csv")

print(df.head())

X = df[['SBP', 'DBP']]
y = df['target'].values
```

4.1.2. Функция для предсказания сигмоиды и функции потерь Log-Loss

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def compute_cost(X, y, theta):
    m = len(y)
    h = sigmoid(X @ theta)
    epsilon = 1e-5
    cost = (1/m)*(((-y).T @ np.log(h + epsilon)) - ((1-y).T @ np.log(1-h + epsilon)))
```

4.1.3. Градиент функции потерь

```
def stochastic_gradient_descent(X, y, params, learning_rate, iterations):
    X = np.hstack((np.ones((X.shape[0], 1)), X))
    params = np.random.rand(X.shape[1])

    cost_track = np.zeros((iterations, 1))

    for i in range(iterations):
        ind = random.sample(range(X.shape[0]), 1)

        params = params - learning_rate * (X[ind] * (sigmoid(X[ind] @ params) - y[ind]))
        params = params.ravel()
        cost_track[i] = compute_cost(X, y, params)

    return cost_track, params
```

4.1.4. Функция предсказания

```
def predict(X, params):  
    x = np.hstack((np.ones((X.shape[0], 1)), X))  
    return np.round(sigmoid(X @ params))
```

4.2. Различие гипер-параметров и параметров модели

- Гипер-параметры - это те параметры, которые объявляются ещё до обучения модели (в kNN - k будет гиперпараметром)
- Параметры модели - те параметры, которые получаются в процессе обучения

Тогда мы понимаем, что у kNN нет вообще параметров - только гиперпараметры. Это одна из аргументаций почему kNN слабее обычных моделей - она не умеет подстраиваться под данные.

Написать конспект со сравнением kNN и LogReg

5. Лекция 4. SVM, классификаторы, метрики качества классификации. Лекция с ФЭН

5.1. Метод опорных векторов

5.1.1. Линейно разделимая выборка

Выборка линейно разделима, если существует такой вектор параметров w^* , что соответствующий классификатор $\alpha(x)$ не допускает ошибок на этой выборке.

Цель метода опорных векторов (Support Vector Machine): максимизировать ширину разделяющей полосы.

Наш алгоритм предсказания $\alpha(x) = \text{sign}((w, x))$

Дальше нам нужно отнормировать параметры w так, чтобы

$$\min_{x \in X} |(w, x)| = 1$$

Тогда мы можем посчитать расстояние от точки x_0 до разделяющей гиперплоскости, задаваемой классификатором

$$\rho(x_0, \alpha) = \frac{|(w, x_0)|}{\|w\|}$$

Минимальное расстояние до разделяющей прямой будет $\frac{1}{\|w\|}$. Ширина разделяющей полосы получается $\frac{2}{\|w\|}$. Мы хотим найти такой вектор параметров, чтобы максимизировать эту полосу.

Если мы перевернём дробь, то у нас как раз будет задача минимизации $\frac{\|w\|}{2}$, и для удобства взятия производных возьмём и возведём в квадрат $\frac{1}{2}\|w\|^2 \rightarrow \min_w$:

5.1.1.1. Постановка задачи нахождения гиперплоскости

$$\begin{cases} \frac{1}{2}\|w\|^2 \rightarrow \min_w \\ y_i \cdot (w, x_i) \geq 1, i = 1, \dots, l \end{cases}$$

5.2. Линейно неразделимая выборка

Так как выборка линейно не разделима, существует хотя бы один объект $x \in X$ такой, что $y_i \cdot (w, x_i) < 1$.

Для того, чтобы находить эту гиперплоскость в линейно неразделяемой выборке, давайте штрафовать объекты штрафом $\xi_i \geq 0$ за нахождение внутри разделяющей гиперплоскости:

$$y_i \cdot (w, x_i) \geq 1 - \xi_i, i = 1, \dots, l$$

5.2.1. Постановка задачи нахождения гиперплоскости

Хотим:

- Минимизировать штрафы $\sum_{i=1}^l \xi_i$
- Максимизировать отступ $\frac{1}{\|w\|}$

Несмотря на то, что эти задачи противоположные по смыслу, мы можем перевернуть дробь в максимизации и начать её минимизировать

Тогда задача оптимизации будет выглядеть как

$$\begin{cases} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^l \xi_i \rightarrow \min_{w, w_0, \xi_i} \\ y_i \cdot (w, x_i) \geq 1 - \xi_i, i = 1, \dots, l \\ \xi_i \geq 0, i = 1, \dots, l \end{cases}$$

Где C - константа регуляризации:

- Чем больше C - тем больше обращаем внимание на штрафы

- Чем меньше C - тем меньше обращаем внимание на штрафы

Задача является выпуклой и имеет единственное решение.

Мы можем переписать второе и третье условие в более красивый вид, потому что $y_i \cdot (w, x_i) \geq 1 - \xi_i \rightarrow \xi_i \geq 1 - y_i \cdot (w, x_i) = 1 - M_i$, тогда $\xi_i = \max(0, 1 - M_i)$, и тогда задача принимает вид

$$\frac{1}{2} \|w\|^2 + C \sum_{i=1}^l \max(0, 1 - M_i) \rightarrow \min_w$$

Если мы поделим на C , то сама задача не изменится.

Тогда на задачу оптимизации SVM можно смотреть, как на оптимизацию функции потерь $L(M) = \max(0, 1 - M) = (1 - M)_+$ с регуляризацией

$$Q(\alpha, X) = \sum_{i=1}^l (1 - M_i)_+ + \frac{1}{2C} \|w\|^2 \min_w$$

Где $(1 - M_i)_+ = \max(0, 1 - M_i)$

5.3. Сравнение с логистической регрессией

На первый взгляд может показаться, что SVM крайне похож на LogReg, но давайте подумаем об этих пунктах:

- LogReg даёт нам вероятность попадения в класс - уверенность алгоритма в ответе, SVM ничего такого не говорит
- В SVM мы просто хотим сделать как можно большую ширину разделяющей полосы (гиперплоскости)

5.4. Метрики качества классификации

5.4.1. Accuracy

Accuracy - доля правильных ответов

$$\text{accuracy}(\alpha, X) = \frac{1}{n} \sum_{i=1}^n [\alpha(x_i) = y_i]$$

5.4.1.1. Недостаток В случаях сильно несбалансированной выборки не отражает качество работы алгоритма

5.4.2. Матрица ошибок

		Actual Value	
		positives	negatives
Predicted Value	positives	TP True Positive	FN False Negative
	positives	FP False Positive	TN True Negative

5.4.3. Precision (Точность)

$$\text{Precision}(\alpha, X) = \frac{TP}{TP + FP}$$

Показывает, насколько можно доверять классификатору при $\alpha(x) = +1$

5.4.4. Recall (Полнота)

$$Recall(\alpha, X) = \frac{TP}{TP + FN}$$

Показывает, как много объектов положительного класса находит классификатор

5.4.5. F-мера

F-мера - это метрика качества, учитывающая и точность, и полноту

$$F(\alpha, X) = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

5.4.6. Регулирование точности и полноты

В моделях бинарной классификации с вероятностями попадения в класс - мы можем сами руками изменять в случае какой вероятности в какой класс попадёт объект. Обычно если $p(x) > 0.5$, то мы относим объект к одному классу, а $p(x) < 0.5$ наоборот. Но мы можем сами исправить и поставить другую константу. Точность и полнота при этом изменятся.

При $p(x) > 0 \rightarrow +1$:

- Точность будет $\frac{TP}{n}$, где n - количество классов
- Полнота будет равна единичке, потому что мы все объекты предсказали как $+1$, и получается у нас нет ни одного FN.

При увеличении порога t :

- Точность возрастает
- Полнота уменьшается

Хочется, чтобы нам не нужно было выбирать порог, а потом считать качество

5.4.7. ROC-AUC

Для каждого порога t вычислим

- False Positive Rate: $FPR = \frac{FP}{FP+TN}$
- True Positive Rate: $TPR = \frac{TP}{TP+FN}$

Если мы захотим перебрать все возможные пороги, то нам достаточно перебрать $n + 1$ порог, где n - количество объектов в выборке.

Давайте построим такой график, где осями будут FPR и TPR. Получится ломанная.

Идеальный алгоритм нам должен выдать прямой угол в вершине TPR 1.

Чтобы посчитать качество - нужно посчитать качество под кривой. Для идеального классификатора площадь будет 1.

5.4.8. Индекс Джини

$$Gini = 2 \cdot AUC - 1$$

Удвоенная площадь между диагональю и ROC кривой.

5.4.9. Precision-Recall кривая

В случае малого количества объектов положительного класса $AUC - ROC$ может давать неадекватно хороший результат: оси $Recall(x)$ и $Precision(y)$. И называется это PR-кривой

6. Лекция 4. SVM. Наша лекция

6.1. Кодирование Категориальных признаков

6.1.1. One-Hot Encoding

Предположим, категориальный признак $f_j(x)$ принимает m различных значений: C_1, C_2, \dots, C_m

Заменим категориальный признак на m бинарных признаков $b_i(x) = [f_j(x) = C_i]$ (индикатор события)

Естественно, нам нужно не забыть выкинуть одну колонку из данных, чтобы между столбцами не было линейной зависимости.

6.1.1.1. Недостаток При большом кол-ве значений (m) у нас будет большая размерность - много столбцов.

6.1.2. Счётчики

Счётчики (mean target encoding) - это вероятность получить значение целевой переменной для данного значения категориального признака.

Естественно, нам нужно не забыть выкинуть одну колонку из данных, чтобы между столбцами не было линейной зависимости.

6.1.2.1. Недостаток Если есть очень редкие категории - то модель может переобучаться.

На примере кредитного скоринга по городам - если из какого-то города было всего два человека и оба не вернули кредит - мы можем переобучиться и на любого человека из этого города говорить, что они кредиты не вернут.

6.1.2.2. Сглаживание Для исправления предыдущего недостатка можно использовать сглаживание - учитывать как в среднем по всем категориям у нас признак соотносится:

$$\frac{\text{mean}(\text{target}) \cdot n_{\text{rows}} + \text{global mean} \cdot \alpha}{n_{\text{rows}} + \alpha}$$

где

- n_{rows} - количество объектов с категорией, у которых целевая переменная равна target
- α - гипер-параметр регуляризации

6.1.2.3. Отложенная выборка Нам придётся выкинуть часть данных, но для того, чтобы не допустить переобучения: Разделим данные на две части, по одной мы подсчитаем счётчики, а во вторую вставим полученные счётчики и будем обучаться на второй части данных.

Модель не смотрела на целевую переменную второй части данных и не переобучивается на этом.

6.1.2.4. Кросс-валидация Похоже на отложенную выборку, но мы сможем закодировать все объекты:

1. Разбиваем данные на m выборок
2. Для каждой выборки i для подсчёта счётчика используем все выборки, кроме выборки i

6.1.3. Expanding Mean

Отсортируем в определённом порядке датасет и для подсчёта значения на m -й строке будем использовать предыдущие с 0 по $m - 1$ строки.

Дотехать секцию с кодом

7. Лекция 5. Многоклассовая классификация, отбор признаков и методы снижения размерности. Лекция ФЭН

7.1. Подходы многоклассовой классификации

7.1.1. One-Vs-All

Решаем задачу классификации на k классов.

Обучим k бинарных классификаторов $b_1(x), \dots, b_k(x)$, каждый из которых решает задачу: принадлежит ли объект x классу k_i или не принадлежит?

Линейный классификатор будет выглядеть как $b_k(x) = \text{sign}((w_k, x))$

Понятно как принимать решение, но возможна такая ситуация, когда несколько классификаторов сказали, что объект x принадлежит их классу.

Как сделать выбор?

Возьмём объект самого уверенного классификатора.

Тогда возьмём самый уверенный классификатор: $\alpha(x) = \arg \max_{k \in [1, \dots, K]} ((w_k, x))$

Но с этим тоже может быть проблема: предсказания классификаторов могут быть в разных масштабах, поэтому сравнивать их некорректно.

7.1.2. All-Vs-All

Для каждой пары классов i и j обучим бинарный классификатор, который предсказывает класс i или класс j . Получается C_K^2 классификаторов

Как здесь принимать итоговое решение? Выберем просто самый популярный класс.

Что здесь плохого: много классификаторов

7.2. Метрики качества

7.2.1. Micro-average

Если у нас k классификаторов, то для каждого находим TP_i, FP_i, FN_i, TN_i и итоговые считаем как $TP = \frac{1}{k} \sum_{i=1}^k TP_i$

Тогда все наши изученные ранее для бинарной классификации метрики мы считаем по средним показателям

7.2.2. Macro-average

Макро усреднение отличается только тем, что такие метрики как precision, recall итд мы считаем для каждого классификатора, а потом усредняем их:

$$\text{precision}(\alpha, X) = \frac{1}{k} \sum_{i=1}^k \text{precision}_i(\alpha, X)$$

7.3. Многоклассовая логистическая регрессия

Напомним, что бинарная логистическая регрессия предсказывает вероятность класса:

$$(w, x) \rightarrow \alpha(x) = \frac{1}{1 + e^{-(w, x)}} = \frac{e^{(w, x)}}{1 + e^{(w, x)}}$$

Предположим у нас есть k моделей, каждая из которых даёт оценку принадлежности выбранному классу: $b_k(x) = (w_k, x)$

Преобразуем вектор предсказаний в вектор вероятностей (softmax-преобразование):

$$\text{softmax}(b_1, \dots, b_k) = \left(\frac{\exp(b_1)}{\sum_{i=1}^k \exp(b_i)}, \frac{\exp(b_2)}{\sum_{i=1}^k \exp(b_i)}, \dots, \frac{\exp(b_k)}{\sum_{i=1}^k \exp(b_i)} \right)$$

Тогда вероятность класса k :

$$P(y = k|x, w) = \frac{\exp((w_k, x))}{\sum_{i=1}^k \exp((w_i, x))}$$

В иной форме, предсказание для j -го класса:

$$a_j(x) = P(y = j|x, w) = \frac{\exp(b_j(x))}{\sum_{i=1}^k \exp(b_i(x))}$$

Обучение будет происходить по методу максимального правдоподобия (аналогичной бинарной классификации):

$$\prod_{i=1}^n a_1(x_i)^{[y_i=1]} \cdot a_2(x_i)^{[y_i=2]} \cdots \cdot a_k(x_i)^{[y_i=k]} = \prod_{i=1}^n \prod_{j=1}^k a_j(x_i)^{[y_i=j]}$$

7.3.0.1. Многоклассовый Log-Loss

$$-\prod_{i=1}^n \prod_{j=1}^k [y_i = j] \log(P(y = j|x_i, w)) \rightarrow \min_{w_1, \dots, w_k}$$

7.4. Отбор признаков

На каком основании можно удалить признаки?

- Маленькая дисперсия - признак почти константный и не информативный
- Для линейных моделей можем посчитать его корреляцию с целевой переменной и выкинуть признаки, которые имеют по модулю маленькую корреляцию.
- Filtration methods - фильтрационные методы
- Wrapping methods - обёрточные методы
- Model selection - встроенные в модель методы отбора признаков

7.4.1. Фильтрационные методы

Фильтрационные методы - это отбор признаков по различным статистическим тестам. Идея метода состоит в вычислении влияния каждого признака в отдельности на целевую переменную (с помощью вычисления некоторой статистики)

7.4.1.1. Преимущества Скорость: линейна от количества признаков

7.4.1.2. sklearn

- SelectKBest - оставляет k признаков с наибольшим значением выбранной статистики
- SelectPercentile - оставляет признаки со значениями выбранной статистики, попавшие в заданный пользователем квантиль

7.4.1.3. Тест χ^2

Тест χ^2 используется в статистике для проверки независимости двух событий. поскольку χ^2 проверяет степень независимости между двумя переменными, а мы хотим сохранить только признаки, наиболее зависимые от метки, то будем вычислять χ^2 между каждым признаком и целевой переменной, сохраняя только признаки с наибольшими значениями

Критерий χ^2 можно применять только для бинарных или порядковых признаков

Статистика χ^2 вычисляется по формуле

$$\chi^2(X; Y) = \sum_{i,j} \frac{O_{ij} - E_{ij}}{E_{ij}}$$

где O_{ij} - наблюдаемая частота, E_{ij} - ожидаемая частота

Чем больше значение этой статистики - тем больше влияние этого признака на ответ (это эмпирический факт).

7.4.1.4. Mutual Information Для векторов X и Y статистика вычисляется по формуле

$$I(X, Y) = \sum_{y \in Y} \sum_{x \in X} p(x, y) \log \left(\frac{p(x, y)}{p(x)p(y)} \right)$$

Где $p(x, y)$ - вероятность того, что данный набор значений появился в нашей выборке (частоты)

Недостатки подхода статистического отбора: может быть так быть, что несколько признаков в целом мало влияют на целевую переменную, а в совокупности они влияют очень сильно.

7.4.2. Обёрточные методы

Обёрточные методы используют жадный отбор признаков, то есть последовательно выкидывают наименее подходящие по мнению методов признаки

7.4.2.1. Recursive Feature Elimination В sklearn есть обёрточный метод Recursive Feature Elimination (RFE)

Параметры метода:

- Алгоритм, используемый для отбора признаков
- Число признаков, которые мы хотим оставить

Недостаток: долгое время работы.

7.5. Встроенные в модель методы

7.5.1. L_1 регуляризация

Например, L_1 регуляризация умеет отбирать признаки - некоторые веса зануляются, поэтому по сути это и есть отбор

$$Q(w) + \alpha \sum_{i=1}^d |w_i| \rightarrow \min_w$$

7.5.2. L_0 регуляризация

Это регуляризация, которая делает ограничение на кол-во признаков в модели:

$$Q(w) + \alpha \sum_{i=1}^d [w_i \neq 0] \rightarrow \min_w$$

7.5.3. Информационный критерий

Информационный критерий - мера качества модели, учитывающая степень "подгонки" модели под данные с корректировкой (штрафом) на используемое количество параметров

Информационные критерии основаны на компромиссе между точностью и сложностью модели

7.5.3.1. Критерий Акаике (AIC) Критерий Акаике (AIC - Akaike Information Criterion)

Мы дополнительно предполагаем, что модель α - линейна, тогда

$$AIC(\alpha, X) = Q(\alpha, X) + \frac{2\hat{\sigma}^2}{l}n \rightarrow \min$$

где:

- Q - функционал ошибки
- $\hat{\sigma}^2$ - оценка дисперсии ошибки $\mathbb{D}(y_i - \alpha(x_i))$
- n - количество используемых признаков
- l - число объектов

Если Q - среднеуadraticная ошибка для линейной регрессии и шумы нормально распределены, то

$$AIC = -\ln(\prod) + n$$

7.5.3.2. Критерий Шварца (BIC) Bayesian Information Criterion

$$BIC(\alpha X) = \frac{l}{\hat{\sigma}^2}(Q(\alpha, X) + \frac{\hat{\sigma}^2 \ln(l)}{l}n) \rightarrow \min$$

Если Q - среднеквадратичная ошибка для линейной регрессии и шумы нормально распределены, то

$$BIC = -\ln(\prod) + \frac{n}{2} \ln(l)$$

7.5.3.3. Отбор признаков с помощью информационных критериев

Если в модели k признаков (регрессоров), то существует 2^k всевозможных моделей

В идеале необходимо построить все 2^k моделей, для каждой посчитать значение критерия качества (AIC, BIC) и выбрать модель, лучшую по этому критерию

При большом количестве регрессоров используют метод включений-исключений

Дотехать сложную часть с подсчётом

7.6. Метод главных компонент (PCA)

Principal Component Analysis - PCA

Предыдущие методы отбирали из исходных признаков некоторое подмножество признаков.

Теперь мы хотим придумать новые признаки, каким-то образом выражаются через старые, причём новых признаков хочется получить меньше, чем старых. В этой лекции рассмотрены только случаи, когда новые признаки линейно выражаются через старые.

Ссылка на МАД 1-го модуля

Лекция 5. Отбор признаков и визуализация. Наша лекция Будем обрабатывать данные про покемонов
Вставить работу с кодом

7.7. Manifold Embeddings

Задача визуализации состоит в отображении объектов в 2D или 3D пространство с сохранением отношений между ними. Наша цель - просто посмотреть на данные

7.7.1. MultiDimensional Scaling (MDS)

Идея метода - минимизация квадратов отклонений между исходными и новыми попарными расстояниями

$$\sum_{i \neq j}^l (\rho(x_i, x_j) - \rho(z_i, z_j))^2 \rightarrow \min$$

где ρ - расстояние между объектами (метрика), x - старые объекты, z - новые объекты

7.7.2. ISOMAP

Для начала скажем, что

- Евклидово расстояние в 3D пространстве - как мы обычно привыкли мерить расстояние между предметами
- Геодезическое расстояние - расстояние между точками через ближайших соседей

Как найти Геодезическое расстояние?

1. Найдём n ближайших соседей
2. Соединим их и получим граф
3. Вычислим кратчайший путь по рёбрам между двумя точками

Для того, чтобы применить алгоритм ISOMAP, осталось только применить MDS для построения проекции в низкоразмерное пространство.

7.7.3. TSNE

t-SNE - t-distributed stochastic neighbor embedding

Идея на поверхности: при проекции нам важно не сохранение расстояний между объектами, а сохранение пропорций:

$$(x_1, x_2) = \alpha \rho(x_1, x_3) \rightarrow \rho(z_1, z_2) = \alpha \rho(z_1, z_3)$$

7.7.3.1. Близость объектов в исходном пространстве Если есть объекты i и j , то

$$\rho(i, j) = \frac{\exp(-\|x_i - x_j\|^2 / 2\sigma_j^2)}{\sum_{k \neq j} \exp(-\|x_k - x_j\|^2 / 2\sigma_j^2)}$$

(затем симметризуем $\rho(i, j)$)

Объекты из окрестности x_j приближаются нормальным распределением. Чем кучнее объекты из этой окрестности - тем меньше берётся значение σ_j^2 .

Поскольку при уменьшении пространства, точек станет меньше - какие-то станут дальше.

Тогда будем измерять сходство объектов в новом пространстве с помощью распределения Коши, так как оно не так сильно штрафует за увеличение расстояний между объектами:

$$q_{ij} = \frac{(1 + \|z_i - z_j\|^2)^{-1}}{\sum_{k \neq j} (1 + \|z_k - z_j\|^2)^{-1}}$$

Для построения проекций z_i объектов x_i , будем минимизировать расстояние между исходным и полученным распределениями (минимизируем дивергенцию Кульбака-Лейблера)

$$KL(p, q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}} \rightarrow \min_{z_1, \dots, z_l}$$

где p и q - распределение в старом и новом пространстве.

Законспектить код питоновский

8. Лекция 6. Нелинейные алгоритмы классификации и регрессии. Лекция ФЭН

8.1. Наивный байесовский классификатор

Наивный байесовский классификатор - это алгоритм классификации, основанный на теореме Байеса с допущением о независимости признаков

Как пример: фрукт может считаться яблоком, если он:

- Красный
- Круглый
- Его диаметр составляет порядка 8 см

и мы предполагаем, что признаки вносят независимый вклад в вероятность того, что фрукт является яблоком: обычно на основании того, что фрукт круглый и красный - мы уже можем сказать, что это скорее всего яблоко, а в модели наивного байесовского классификатора - не можем.

8.1.1. Теорема Байеса

$$\mathbb{P}(c|x) = \frac{\mathbb{P}(x|c) \cdot \mathbb{P}(c)}{\mathbb{P}(x)}$$

$$8.1.1.1. \text{ В случае нескольких признаков } \mathbb{P}(y|x_1, \dots, x_n) = \frac{\mathbb{P}(y) \cdot \prod_{i=1}^n \mathbb{P}(x_i|y)}{\prod_{i=1}^n \mathbb{P}(x_i)}$$

Вероятности вычисляются с помощью частотных таблиц, как и в одномерном случае.

8.1.1.2. Преимущества

- Классификация быстрая и простая
- В случае, если выполняется предположение о независимости, классификатор показывает очень высокое качество

8.1.1.3. Недостатки

- Если в тестовых данных присутствует категория, не встречавшаяся в данных для обучения, модель присвоит ей нулевую вероятность

8.2. Метод ближайших соседей

Идея: схожие объекты находятся близко к друг другу в пространстве признаков.

8.2.0.1. Классификация нового объекта

1. Вычисляем расстояние до каждого из объектов обучающей выборки
2. Выбираем k объектов, расстояние до которых минимально
3. Классифицируем этот объект, как самый часто встречающийся класс среди k ближайших соседей

И при этом, в качестве расстояния обычно берут Евклидову метрику, поэтому данные нужно масштабировать.

8.3. Решающие деревья

Решающее дерево - это бинарное дерево, в котором

- В каждой вершине v приписана функция (предикат) $\beta_v : X \rightarrow \{0, 1\}$ для ровно одного признака
- В каждой листовой вершине v приписан прогноз $c_v \in Y$ (для классификации - класс или вероятность класса, для регрессии - действительное значение целевой переменной)

8.3.1. Что влияет на построение решающего дерева

- Вид предикатов в вершинах
- Функционал качества $Q(X, j, t)$
- Критерий останова

8.3.2. Критерий информативности

В каждой вершине оптимизируется функционал $Q(X, j, t)$

Пусть R - множество объектов, попадающих в вершину на данном шаге, а R_l и R_r - объекты, попадающие в левую и правую ветви после разбиения

Цель: хотим, чтобы после разбиения объектов на две группы, внутри каждой группы как можно больше объектов было одного класса

$H(R)$ - критерий информативности - это мера неоднородности (разнообразности) целевой переменной. Чем меньше разнообразие целевой переменной внутри группы - тем меньше значение $H(R)$. То есть хотим решить задачу

$$H(R_l) \rightarrow \min, H(R_r) \rightarrow \min$$

Но это всё даёт сложность, потому что вместо одной задачи оптимизации - мы получаем две задачи оптимизации. Тогда можно определить функционал как

$$Q(R, j, t) = H(R) - \frac{|R_l|}{|R|} H(R_l) - \frac{|R_r|}{|R|} H(R_r) \rightarrow \max_{j,t}$$

где

- R - объекты попавшие в вершину
- j - номер признака
- t - порог признака

В каждом листе дерево выдаёт константу c - вещественное число в регрессии? класс или вероятность класса в классификации.

8.3.2.1. Функции потерь

Если в качестве функции потерь мы берём квадратичную ошибку, то

$$H(R) = \min_{c \in \mathbb{R}} \frac{1}{R} \sum_{(x_i, y_i) \in R} (y_i - c)^2$$

Её минимум достигается при

$$C = \frac{1}{|R|} \sum_{(x_i, y_i) \in R} y_i$$

Информативность $H(R)$ в вершине дерева - это дисперсия целевой переменной (для объектов, попавших в вершину)

8.3.2.2. H(R) в задачах мягкой классификации

Будем в каждой вершине в качестве ответа выдавать не класс, а распределение вероятностей классов:

$$c = (c_1, \dots, c_k), \sum_{i=1}^k c_i = 1$$

Качество распределения можно измерить с помощью критерия Бриера, измеряющего точность вероятностных прогнозов:

$$H(R) = \min_c \frac{1}{|R|} \sum_{(x_i, y_i) \in R} \sum_{j=1}^k (c_j - [y_i = j])^2$$

8.3.2.3. Критерий Джини

1. Минимальное значение функционала $H(R)$ достигается на векторе, состоящем из долей классов:
 $c_* = (p_1, \dots, p_k)$
2. На векторе c_* функционал (*) переписывается в виде

$$H(R) = \sum_{k=1}^K p_k(1 - p_k)$$

8.3.2.4. Энтропийный критерий

Запишем логарифм правдоподобия:

$$H(R) = \min_c \left(-\frac{1}{|R|} \sum_{(x_t, y_t \in R)} \sum_{k=1}^K [y_i = k] \log(c_k) \right) (*)$$

На векторе $c_* = (p_1, \dots, p_k)$ функционал (*) записывается в виде

$$H(R) = -\sum_{k=1}^K p_k \log(p_k) \text{ (энтропия)}$$

8.3.3. Критерий останова

- Ограничение максимальной глубины дерева (max_depth)
- Ограничение минимального числа объектов в листьях (min_samples_leaf)
- Ограничение максимального числа листьев в дереве
- Остановка в случае, если все объекты в листе из одного класса
- Требование, чтобы функционал качества при дроблении увеличивался как минимум на $s\%$

8.3.4. Плюсы решающих деревьев

- Чёткие правила классификации (интерпретируемые предикаты, например, "возраст > 25")
- Деревья решений легко визуализируются, то есть хорошо интерпретируются
- Быстро обучаются и выдают прогноз
- Малое число параметров

8.3.5. Минусы решающих деревьев

- Очень чувствительны к шумам в данных, модель сильно меняется при небольшом изменении обучающей выборки
- Разделяющая граница имеет свои ограничения (состоит из гиперплоскостей)
- Необходимость борьбы с переобучением (стрижка или какой-либо из критериев останова)
- Проблема поиска оптимального дерева (NP-полнная задача, поэтому на практике используется жадное построение дерева)

9. Лекция 6. Решающие деревья. Наша Лекция. Елена

9.1. ROC-AUC: интуиция

Пусть есть предсказание модели и правильный класс. Отсортируем по убыванию вероятности:

p	класс	p	класс
0.5	0	0.6	1
0.1	0	0.5	0
0.25	0	0.3	1
0.6	1	0.25	0
0.2	1	0.2	1
0.3	1	0.1	0
0.0	0	0.0	0

Небольшое дополнение: порог не обязательно ставить 0.5, можно хоть 0.3

Как у хорошей модели будет выглядеть столбец с классами после сортировки по вероятности? По хорошему должны стоять сначала подряд все единицы, а потом все нули.

Если модель чередует единицы и нолики - это плохая модель. В нашем случае как раз видим такое.

9.1.1. ROC-AUC алгоритм

1. Нарисуем квадрат 1×1
 2. Горизонтальную сторону (x) разобьём на равные отрезки, число которых равно количеству объектов с классом 0
 3. Вертикальную сторону разобьём на равные отрезки, число которых равно количеству объектов с классом 1
 4. Стартуем из точки $(0, 0)$
 5. Идём сверху вниз нашей таблицы:
 - если попалась единица - шагаем в клетку $(n, m + 1)$
 - если попался ноль - шагаем в клетку $(n + 1, m)$
- ($+1$ естественно по размеченной сетке, а не в прямом смысле $+1$)

В идеальном случае мы должны сначала достигнуть угла $(1, 0)$ нашего квадрата, а потом уйти в $(1, 1)$. Сама кривая называется ROC-кривая, а метрика ROC-AUC считается как площадь под нашей описанной кривой.

У идеальной модели площадь будет равна 1.

У случайной модели (при одинаковом кол-ве классов) будет зигзаг около главной диагонали, и площадь будет примерно 0.5.

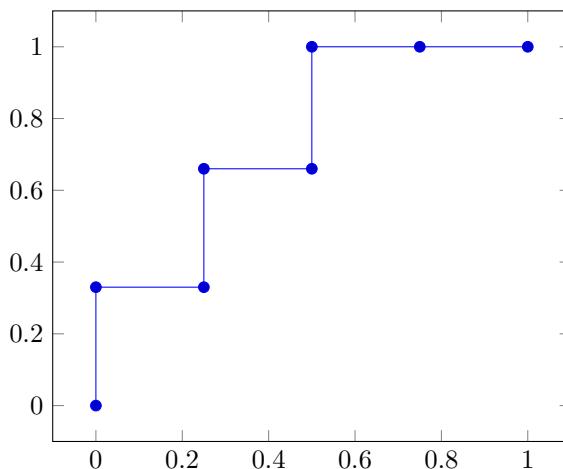


Рис. 1: ROC-AUC нашей задачи

И ROC-AUC метрика у нас будет примерно 0.75.

9.2. Решающие деревья

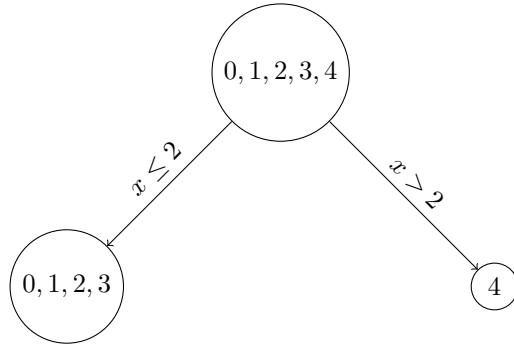
9.2.1. Задачи

9.2.1.1. Задача 1 Постройте регрессионное дерево для прогнозирования y с помощью x на обучающей выборке

x_i	0	1	2	3
y_i	5	6	4	100

Критерий деления узла на два - минимизация RSS (MSE). Дерево строится до трёх терминальных узлов.

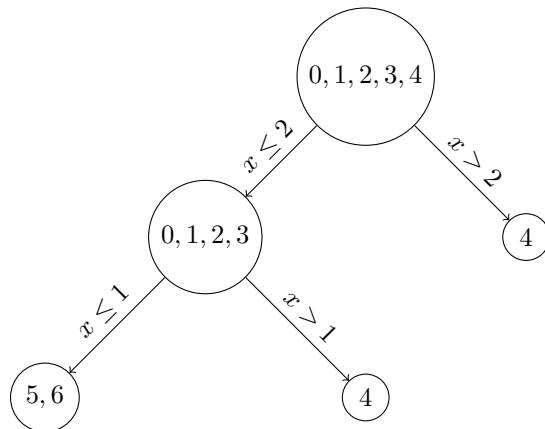
9.2.1.2. Решение задачи 1 На первом шаге можно взять разделяющим критерием " $x > 2$ ".



Теперь у нас есть вариант сделать разделяющую поверхность $x > 1$ или $x > 0$. Проверим как получится метрика тогда:

- Если мы разбиваем по $x > 1$, то в левом поддереве уйдут вершины с $y_i = \{5, 6\}$, в правую $y_i = \{4\}$. В левом поддереве ответ получится 5.5, а в правом 4. Для левого поддерева MSE получится $\frac{(5-5.5)^2+(6-5.5)^2}{2} = 0.25$. Для правого поддерева $MSE = 0$.
- Если бьём по $x > 0$, то в левом поддереве уйдёт вершина $y_i = \{5\}$, а в правом поддереве $y_i = \{6, 4\}$. Значит будет предсказано в левом поддереве ответ 5 и в другом 5. Для левого поддерева $MSE = 0$, а для правого $\frac{(6-5)^2+(4-5)^2}{2} = 1$.

MSE получится лучше, если разделять по " $x > 1$ ":



9.2.1.3. Задача 3 Дон-Жуан предпочитает брюнеток. Перед Новым Годом он посчитал, что в записной книжке у него 20 блондинок, 40 брюнеток, две рыжих и восемь шатенок. С Нового Года Дон-Жуан решил перенести все сведения в две записные книжки. В одну - брюнеток, во вторую - остальных.

Как изменился индекс Джини и энтропия в результате такого разбиения?

9.2.1.4. Решение задачи 3 До разделения на две книжки:

$$H = - \sum_{i=1}^n p_i \cdot \log(p_i) = -\left(\frac{20}{70} \log\left(\frac{20}{70}\right) + \frac{40}{70} \log\left(\frac{40}{70}\right) + \frac{2}{70} \log\left(\frac{2}{70}\right) + \frac{8}{70} \log\left(\frac{8}{70}\right)\right) = 1.02$$

После разбиения на две книжки:

Книжка с брюнетками:

$$H(L) = -(1 \cdot \log(1)) = 0$$

Книжка с остальными:

$$H(R) = -\left(\frac{20}{30} \log\left(\frac{20}{30}\right) + \frac{2}{30} \log\left(\frac{2}{30}\right) + \frac{8}{30} \log\left(\frac{8}{30}\right)\right) = 0.8$$

9.2.1.5. Задача 4 Привидите набор данных, для которых индекс Джини равен 0, 0.5 и 0.999

9.2.1.6. Решение задачи 4 Индекс Джини: $G = \sum_{i=1}^K p_i(1 - p_i)$

- Для индекса Джини 0 достаточно, чтобы все данные принадлежали одному классу: $1 \cdot (1 - 1) = 0$
- Для индекса Джини 0.5 достаточно, чтобы было два класса и объектов было поровну: $0.5 \cdot (1 - 0.5) + 0.5 \cdot (1 - 0.5) = 0.25 + 0.25 = 0.5$
- Для индекса Джини 0.999 достаточно, чтобы было 1000 классов и каждого было по одному объекту:

$$\overbrace{\frac{1}{1000} \cdot (1 - \frac{1}{1000}) + \dots + \frac{1}{1000} \cdot (1 - \frac{1}{1000})}^{1000 \text{ раз}} = 1000 \cdot \frac{1}{1000} \cdot (1 - \frac{1}{1000}) = 1 - \frac{1}{1000} = 0.999$$

9.3. Работа с кодом

9.3.1. Подключение библиотек

```
import matplotlib.pyplot as plt
%matplotlib inline
from mlxtend.plotting import plot_decision_regions
import numpy as np
from sklearn.datasets import load_boston
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor, plot_tree
import pandas as pd

plt.rcParams["figure.figsize"] = (11, 6.5)
```

9.3.2. Генерация выборки для задачи регрессии

```
n = 400
np.random.seed(1)
X = np.zeros((n, 2))
X[:, 0] = np.linspace(-5, 5, n)
X[:, 1] = X[:, 0] + 0.5 * np.random.normal(size=n)
y = (X[:, 1] > X[:, 0]).astype(int)

plt.scatter(X[:, 0], X[:, 1], s=100, c=y, cmap='bwr')
plt.show()
```

9.3.3. Функция для обучения классификатора и построения разделяющей прямой

```
def train_model(model=LogisticRegression()):
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=1)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

```

plot_decision_regions(X_test, y_test, model)
plt.show()

print(f"Accuracy: {accuracy_score(y_pred, y_test):.2f}")

```

9.3.4. Сравнение качества между логистической регрессией и деревом решений

```

train_model(LogisticRegression())
train_model(DecisionTreeClassifier(random_state=13))

```

9.3.5. Генерация выборки логического ИЛИ

```

X = np.random.randn(n, 2)
y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0).astype(int)
plt.scatter(X[:, 0], X[:, 1], s=100, c=y, cmap="bwr")
plt.show()

```

9.3.6. Сравнение качества между логистической регрессией и деревом решений

```

train_model(LogisticRegression())
train_model(DecisionTreeClassifier(random_state=13))

```

9.3.7. Переобучение дерева

Решающие деревья могут переобучаться под любую выборку, если их не регуляризовать: при большом количестве листьев для каждого объекта может выделиться своя область в признаковом пространстве. Дерево просто выучивает обучающую выборку, но не выделяет закономерности в данных. Давайте убедимся в этом на практике.

```

np.random.seed(13)
n = 100
X = np.random.normal(size=(n, 2))
X[:50, :] += 0.25
X[50:, :] -= 0.25
y = np.array([1] * 50 + [0] * 50)
plt.scatter(X[:, 0], X[:, 1], s=100, c=y, cmap="bwr")
plt.show()

```

```

fig, ax = plt.subplots(nrows=3, ncols=3, figsize=(15, 12))

for i, max_depth in enumerate([3, 5, None]):
    for j, min_samples_leaf in enumerate([1, 5, 15]):
        dt = DecisionTreeClassifier(max_depth=max_depth, min_samples_leaf=min_samples_leaf)
        dt.fit(X, y)
        ax[i][j].set_title("max_depth={}|min_samples_leaf={}".format(max_depth, min_samples_leaf))
        ax[i][j].axis("off")
        plot_decision_regions(X, y, dt, ax=ax[i][j])

plt.show()

```

9.3.7.3. Дерево с нулевой ошибкой При этом нужно сказать, что если никак не ограничивать модель, то она может выдавать нулевую ошибку, но при этом сильно переобучиться.

```
model = DecisionTreeClassifier(max_depth=None, min_samples_leaf=1, random_state=13)
model.fit(X, y)

print(f"Accuracy: {accuracy_score(y, dt.predict(X)):.2f}")

plot_decision_regions(X, y, model)
plt.show()
```

9.3.8. Неустойчивость

Дерево обычно очень сильно меняется от изменения выборки. Давайте попробуем давать дереву разные 90% от объектов.

```
fig, ax = plt.subplots(nrows=3, ncols=3, figsize=(15, 12))

for i in range(3):
    for j in range(3):
        seed_idx = 3 * i + j
        np.random.seed(seed_idx)
        dt = DecisionTreeClassifier(random_state=13)
        idx_part = np.random.choice(len(X), replace=False, size=int(0.9 * len(X)))
        X_part, y_part = X[idx_part, :], y[idx_part]
        dt.fit(X_part, y_part)
        ax[i][j].set_title(f"sample#{j}.format(seed_idx))")
        ax[i][j].axis("off")
        plot_decision_regions(X_part, y_part, dt, ax=ax[i][j])

plt.show()
```

Мы бы хотели, чтобы от одинакового по размеру датасета и примерно со схожими данными наша модель не менялась, но у деревьев, как мы видим так не происходит.

9.3.9. Анализ деревьев на основе датасета Бостона

9.3.9.1. Загрузка датасета .

```
boston = load_boston()
print(boston["DESCR"])

X = pd.DataFrame(data=boston["data"], columns=boston["feature_names"])
y = boston["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=13)

print(f"Shape:{X.shape}")
X.head()
```

9.3.9.2. Построение и отображение дерева с предикатами .

```
tree = DecisionTreeRegressor(max_depth=3, random_state=13)
tree.fit(X_train, y_train)

plot_tree(tree, feature_names=X.columns, filled=True, rounded=True)
plt.show()
```

9.3.9.3. Получение построенных параметров дерева .

```
n_nodes = tree.tree_.node_count
children_left = tree.tree_.children_left
children_right = tree.tree_.children_right
feature = tree.tree_.feature
threshold = tree.tree_.threshold
```

9.3.9.4. Построение зависимости MSE от гиперпараметра max_depth и отбор лучших значений на кросс-валидирующей выборке .

```
from sklearn.model_selection import cross_val_score

max_depth_array = range(2, 20)
mse_array = []

for max_depth in max_depth_array:
    tree = DecisionTreeRegressor(max_depth=max_depth, random_state=13)
#    tree.fit(X_train, y_train)
#    mse_array.append(mean_squared_error(y_test, tree.predict(X_test)))
    mse = -cross_val_score(tree, X, y, cv=3, scoring='neg_mean_squared_error').mean()
    mse_array.append(mse)

plt.plot(max_depth_array, mse_array)
plt.title("Dependence_of_MSE_on_max_depth")
plt.xlabel("max_depth")
plt.ylabel("MSE")
plt.show()

pd.DataFrame({"max_depth": max_depth_array, "MSE": mse_array}).sort_values(by="MSE").reset_index()
```

9.3.9.5. Построение зависимости MSE от гиперпараметра max_samples_leaf и отбор лучших значений на кросс-валидирующей выборке .

```
min_samples_leaf_array = range(1, 20)
mse_array = []

for min_samples_leaf in min_samples_leaf_array:
    dt = DecisionTreeRegressor(max_depth=4, min_samples_leaf=min_samples_leaf, random_state=13)
    res = -cross_val_score(dt, X, y, cv=3, scoring='neg_mean_squared_error').mean()
    mse_array.append(res)

plt.plot(min_samples_leaf_array, mse_array)
plt.title("Dependence_of_MSE_on_min_samples_leaf")
plt.xlabel("min_samples_leaf")
plt.ylabel("MSE")
plt.show()

pd.DataFrame({"min_samples_leaf": min_samples_leaf_array, "MSE": mse_array}).sort_values(by="MSE").reset_index()
```

9.3.9.6. Разделение выборки на обучающую и тестовую .

```
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.2)
```

9.3.9.7. Построение дерева без ограничивающих параметров .

```
tree = DecisionTreeRegressor()

tree.fit(Xtrain, ytrain)

pred_train = tree.predict(Xtrain)
pred_test = tree.predict(Xtest)

from sklearn.metrics import r2_score

r2_score(ytrain, pred_train), r2_score(ytest, pred_test)
```

```

from sklearn.model_selection import GridSearchCV

params = { 'max_depth' : np.arange(2, 12),
           'max_features' : [ "auto", "sqrt", "log2"] }

gs = GridSearchCV(DecisionTreeRegressor(), params, cv=3, scoring='r2')

gs.fit(X, y)

print(gs.best_estimator_)

```

9.3.9.9. Стрижка дерева Мы можем как обычно делать при переобучении: добавлять регуляризатор, и выглядит он как

$$Q_{now} = Q + \alpha|T|$$

где T - число вершин в дереве, а α - наш параметр

```

path = tree.cost_complexity_pruning_path(Xtrain, ytrain)
alphas = path['ccp_alphas']

```

в $alphas$ лежат все значения параметра регуляризации, которые нам нужно рассмотреть

9.3.9.10. Перебор по гиперпараметру регуляризации .

```

import seaborn as sns
from sklearn.metrics import r2_score

accuracy_train, accuracy_test = [], []
MaxR2 = -1
Alpha = 0

for i in alphas:
    tree = DecisionTreeRegressor(ccp_alpha = i)

    tree.fit(Xtrain, ytrain)
    y_train_pred = tree.predict(Xtrain)
    y_test_pred = tree.predict(Xtest)

    accuracy_train.append(r2_score(ytrain, y_train_pred))
    accuracy_test.append(r2_score(ytest, y_test_pred))

    R2 = r2_score(ytest, y_test_pred)
    if R2 > MaxR2:
        MaxR2 = R2
        Alpha = i

sns.set()
plt.figure(figsize=(14,7))
sns.lineplot(y = accuracy_train, x = alphas, label = "Train_r2")
sns.lineplot(y = accuracy_test, x = alphas, label = "Test_r2")
plt.xticks(ticks = np.arange(0.00, 0.25, 0.01))
plt.show()

print('Best_alpha:', Alpha)

```

```

tree3 = DecisionTreeRegressor(ccp_alpha = Alpha)

tree3.fit(Xtrain, ytrain)

pred_train3 = tree3.predict(Xtrain)
pred_test3 = tree3.predict(Xtest)

r2_score(ytrain, pred_train3), r2_score(ytest, pred_test3)

```

9.3.10. Решающее дерево своими руками

9.3.9.11. Применение лучшего параметра регуляризации

9.4. Калибровка вероятностей

Почти у всех моделей, которые мы используем, есть две опции:

- predict - возвращает предсказание класса
- predict_proba - возвращает вероятность класса, насколько классификатор уверен в своём предсказании

Допустим у нас есть 10 одинаковых объекта, но целевая переменная у 6 из них относится к одному классу, а у 4-ёх относится к другому. Тогда наша модель будет уверена на 0.6 в том, что такой же объект будет отнесён к первому классу.

Для большинства моделей эти числа никак не связаны с реальными вероятностями. Но эти результаты можно откалибровать. Чтобы возвращаемое значение predict_proba правда выдавало что-то похожее на реальные вероятности.

9.4.1. Программируем это всё дело

9.4.1.1. Генерация выборки .

```

from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

X, y = make_classification(
    n_samples=100_000, n_features=20, n_informative=2, n_redundant=10, random_state=42
)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.99, random_state=42
)

```

9.4.1.2. Обучение моделей Давайте обучим модели и для дерева используем CalibratedClassifierCV, который как раз и калибрует вероятности.

```

import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

from sklearn.calibration import CalibratedClassifierCV, CalibrationDisplay
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier

lr = LogisticRegression(C=1.0)
dt = DecisionTreeClassifier(max_depth = 6)
dt_isotonic = CalibratedClassifierCV(dt, cv=3, method="isotonic")
dt_sigmoid = CalibratedClassifierCV(dt, cv=3, method="sigmoid")

clf_list = [
    (lr, "Logistic"),
    (dt, "Decision_Tree"),
]

```

```

(dt_isotonic, "Decision Tree + Isotonic"),
(dt_sigmoid, "Decision Tree + Sigmoid"),
]

```

9.4.1.3. Отобразим наши вероятности .

```

fig = plt.figure(figsize=(10, 10))
gs = GridSpec(4, 2)
colors = plt.cm.get_cmap("Dark2")

ax_calibration_curve = fig.add_subplot(gs[:2, :2])
calibration_displays = {}
for i, (clf, name) in enumerate(clf_list):
    clf.fit(X_train, y_train)
    display = CalibrationDisplay.from_estimator(
        clf,
        X_test,
        y_test,
        n_bins=10,
        name=name,
        ax=ax_calibration_curve,
        color=colors(i),
    )
    calibration_displays[name] = display

ax_calibration_curve.grid()
ax_calibration_curve.set_title("Calibration plots (Naive Bayes)")

# Add histogram
grid_positions = [(2, 0), (2, 1), (3, 0), (3, 1)]
for i, (_, name) in enumerate(clf_list):
    row, col = grid_positions[i]
    ax = fig.add_subplot(gs[row, col])

    ax.hist(
        calibration_displays[name].y_prob,
        range=(0, 1),
        bins=10,
        label=name,
        color=colors(i),
    )
    ax.set(title=name, xlabel="Mean predicted probability", ylabel="Count")

plt.tight_layout()
plt.show()

```

При этом на качество и метрики эта калибровка никак не влияет. Это влияет только на вероятности, выдаваемые моделью.

10. Материалы к лекции 6

10.1. Статья про разложение ошибки

Ссылка на статью - <https://habr.com/ru/company/ods/blog/323890/>

Допустим у нас есть функция предсказания $f(x)$. Реальное значение целевой переменной выглядит как $y = f(x) + \epsilon$, где $\epsilon \sim N(0, \sigma^2)$, а $y \sim N(f(x), \sigma^2)$. Пусть также есть точечная оценка для f - \hat{f}

Тогда ошибка в точке x раскладывается как

$$Err(x) = \mathbb{E}[(y - \hat{f}(x))^2] = \mathbb{E}(y^2) + \mathbb{E}(\hat{f}(x)^2) - 2\mathbb{E}(y \cdot \hat{f}(x)) =$$

Мы помним, что $Var(x) = \mathbb{E}(x^2) - \mathbb{E}(x)^2$, тогда

$$\begin{aligned}\mathbb{E}(y^2) &= Var(y) + \mathbb{E}(y)^2 \\ \mathbb{E}(f(x)^2) &= Var(f(x)) + E(f(x))^2\end{aligned}$$

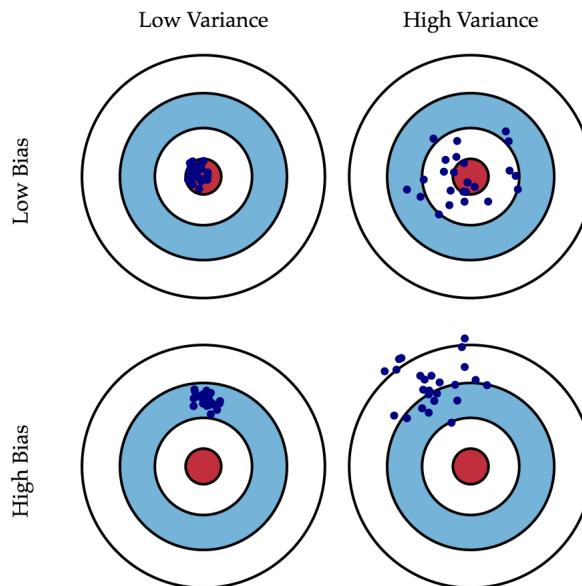
и ещё

$$\mathbb{E}(y \cdot \hat{f}) = \mathbb{E}((f + \epsilon) \cdot \hat{f}) = \mathbb{E}(f\hat{f}) + \mathbb{E}(\epsilon\hat{f}) = f \cdot \mathbb{E}(\hat{f}) + \mathbb{E}(\epsilon) \cdot \mathbb{E}(\hat{f}) = f\mathbb{E}(\hat{f})$$

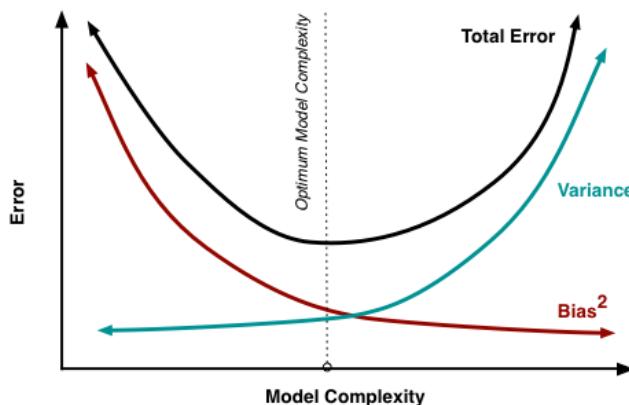
Собрав всё вместе - считаем:

$$Err(x) = \mathbb{E}[(y - \hat{f}(x))^2] = \sigma^2 + f^2 + Var(\hat{f}) + \mathbb{E}(\hat{f})^2 - 2f\mathbb{E}(\hat{f}) = (f - \mathbb{E}(\hat{f}))^2 + Var(\hat{f}) + \sigma^2 = Bias(\hat{f})^2 + Var(\hat{f}) + \sigma^2$$

С последним слагаемым мы ничего сделать не можем, а вот первые два мы вполне можем попытаться минимизировать. Если делать ничего не будем, то нас ждёт примерно такой результат:



Как правило, при увеличении сложности модели (например, при увеличении количества свободных параметров) увеличивается дисперсия (разброс) оценки, но уменьшается смещение. Из-за того что тренировочный набор данных полностью запоминается вместо обобщения, небольшие изменения приводят к неожиданным результатам (переобучение). Если же модель слабая, то она не в состоянии выучить закономерность, в результате выучивается что-то другое, смещенное относительно правильного решения.



10.2. Уроки 6.4, 6.5 со степика

11. Лекция 7. Композиция алгоритмов, разложение ошибки и лес

Ссылка на видео - https://www.youtube.com/watch?v=X4arg_OLxUk&list=PLEwK9wdS5g0qi14fXKFnFzruUDg3n16db&index=34

11.1. Смещение и разброс

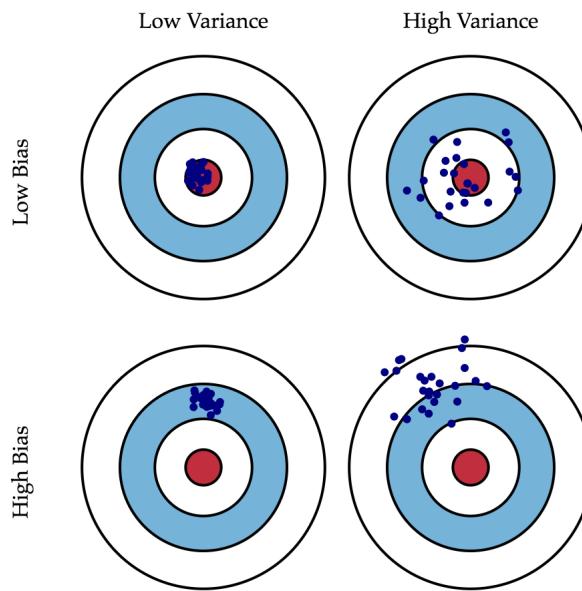
Зачастую для улучшения качества модели необходимо понять из-за чего возникает ошибка в предсказаниях. Ошибку можно представить в виде

$$Err(x) = Bias^2(\alpha(x)) + Var(\alpha(x)) + \sigma^2$$

где

- $Bias(\alpha(x))$ - средняя ошибка по всем возможным наборам данных - смещение. Смещение показывает насколько в среднем модель хорошо предсказывает целевую переменную. Маленькое смещение - хорошее предсказание, большое смещение - плохое предсказание
- $Var(\alpha(x))$ - дисперсия ошибки, то есть как сильно различается ошибка при обучении на различных наборах данных - разброс. Большой разброс означает, что ошибка очень чувствительна к изменению обучающей выборки. Большой разброс - сильно переобученная модель
- σ^2 - неустранимая ошибка, шум в данных

И их влияние можно увидеть на картинке



Что означают синие точки на этой картинке?

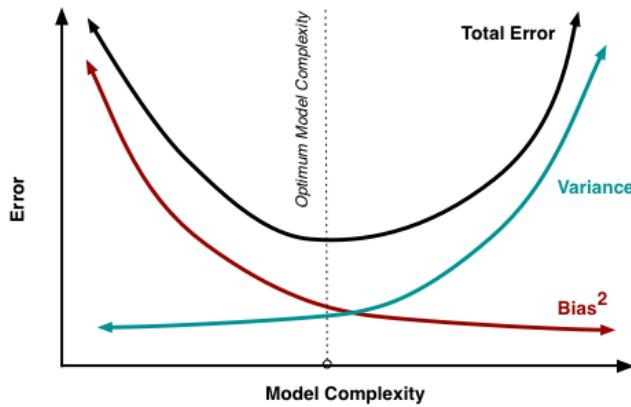
Мы берём модель и одна синяя точка отвечает просто за один набор обучающих данных, а другая за другой. То есть на картинке изображено предсказание одного и того же объекта, но при каждом предсказании обучающая выборка разная для модели, с помощью которой предсказывали.

Что мы понимаем под сложностью модели? Под сложностью модели мы понимаем число её параметров, количество весов, которые мы настраиваем в процессе обучения. Не гипер-параметры. Например у линейной регрессии мало параметров. У метода ближайших соседей вообще нет параметров. А если брать решающие деревья или случайные леса - там много параметров.

Оказывается, модели, у которых мало параметров - могут плохо решить задачу, у них большое смещение. Но из-за того, что параметров мало - они не подогнаны под наши данные. У моделей получается большое смещение, но маленький разброс.

Чем сложнее модель, тем лучше она решает задачу и у неё уменьшается смещение, но увеличивается разброс.

Эти тезисы изображены на рисунке



11.1.1. Формула для разложения ошибки на смещение и разброс

Пусть есть какая-то истинная зависимость: $y = f(x) + \epsilon$.

Пусть $\epsilon \sim N(0, \sigma^2)$

Мы строим модель: $x \rightarrow \alpha(x)$. Измерять ошибку будем метрикой MSE.

Мы хотим найти $\mathbb{E}((y - \alpha)^2)$.

Среднюю ошибку мы вычисляем по всем одинаковым объектам: по всем одинаковым x . x бывают очень часто одинаковые: для предсказания стоимости квартиры у нас могут совпасть все данные, которые у нас имеются.

Давайте раскроем нашу ошибку:

$$\mathbb{E}((y - \alpha)^2) = \mathbb{E}(y^2 + \alpha^2 - 2y\alpha) = E(y^2) + \mathbb{E}(\alpha^2) - \mathbb{E}(2y\alpha)$$

Произведём трюк: добавим и вычтем одно и те же слагаемые:

$$E(y^2) - \mathbb{E}(y^2) + \mathbb{E}(y^2) + \mathbb{E}(\alpha^2) - \mathbb{E}(\alpha^2) + \mathbb{E}(\alpha^2) - \mathbb{E}(2y\alpha) = Var(y) + Var(\alpha) + (\mathbb{E}(y)^2 - \mathbb{E}(2y\alpha) + \mathbb{E}(\alpha)^2) = Var(y) + Var(\alpha) + (\mathbb{E}(y) - \mathbb{E}(\alpha))^2$$

При этом у нас $\mathbb{E}(y) = \mathbb{E}(f(x) + \epsilon) = \mathbb{E}(f(x)) = f(x)$ потому что ϵ имеет нулевое матожидание, а $f(x)$ - константа. Продолжим вычисления:

$$Var(y) + Var(\alpha) + (\mathbb{E}(y) - \mathbb{E}(\alpha))^2 = Var(y) + Var(\alpha) + (f - \mathbb{E}(\alpha))^2$$

При этом $Var(y)$ - это шум в данных. К примеру, когда некоторые объекты одинаковые, а целевая переменная у них разная.

11.2. Бэггинг

Для того, чтобы понять что такое бэггинг, нужно дать другое определение

11.2.1. Бутстрэп

Бутстрэп - метод генерации данных.

Мы из нашей исходной выборки X сгенерируем n выборок X_1, \dots, X_n . Каждая из выборок будет сгенерирована взятием объектов с возвращением (то есть объекты могут повторяться внутри новой выборки).

11.2.2. Описание бэггинга

Далее для каждой выборки будем обучать алгоритм из одного и того же семейства. Получится n моделей: $b_1(x), \dots, b_n(x)$.

Построим новую функцию регрессии:

$$\alpha(x) = \frac{1}{n} \sum_{j=1}^n b_j(x)$$

Это и называется бэггинг.

В случае классификации это может быть голосование.

Такое усреднение будет снижать переобучение, которое было бы, если бы мы обучали только одну модель (в нашем случае дерево).

11.2.3. Смещение и разброс у бэггинга

- Бэггинг не ухудшает смещенность модели: смещение $\alpha_n(x)$ равно смещению базового алгоритма
- Если базовые алгоритмы некоррелированы, то дисперсия бэггинга $\alpha_n(x)$ в n раз меньше дисперсии отдельных базовых алгоритмов.

11.2.4. Как сделать некоррелированные алгоритмы

Пусть наши базовые алгоритмы $\mu_1(x), \dots, \mu_n(x)$, а наша композиция $\alpha(x) = \frac{1}{n} \sum_{i=1}^n \mu_i(x)$

Что мы хотим доказать? Что смещение у бэггинга такое же, как у одного дерева, а разброс меньше.

- Смещение: $\mathbb{E}(\alpha(x) - f(x))^2 = \mathbb{E}\left(\frac{1}{n} \sum_{i=1}^n \mu_i(x) - f\right)^2 = \left(\frac{1}{n} \sum_{i=1}^n \mu_i(x) - f\right)^2 = (\mathbb{E}(\mu_i(x)) - f)^2$
- Разброс: $Var(\alpha) = \mathbb{E}(\alpha - \mathbb{E}(\alpha))^2 = \mathbb{E} \left[\frac{1}{n} \sum_{i=1}^n \mu_i(x) - \mathbb{E} \left(\frac{1}{n} \sum_{i=1}^n \mu_i(x) \right) \right]^2 = \frac{1}{n^2} \mathbb{E} \left[\sum_{i=1}^n \mu_i(x) - \mathbb{E} \left(\sum_{i=1}^n \mu_i(x) \right) \right]^2 =$

$$\stackrel{\text{из-за некоррелированности}}{=} \frac{1}{n^2} \sum_{i=1}^n \mathbb{E}[\mu_i(x) - \mathbb{E}\mu_i(x)] + \frac{1}{n^2} \sum_{i_1 \neq i_2} \overbrace{\mathbb{E}[(\mu_{i_1}(x) - \mathbb{E}\mu_{i_1}(x))(\mu_{i_2}(x) - \mathbb{E}\mu_{i_2}(x))]} = \frac{1}{n^2} \sum_{i=1}^n \mathbb{E}[\mu_i(x) - \mathbb{E}\mu_i(x)] =$$

$$\frac{1}{n} Var(\mu(x)).$$
 Уменьшили разброс.

11.3. Случайный лес (Random Forest)

Возьмём в качестве базовых алгоритмов для бэггинга решающие деревья, то есть каждое случайное дерево $b_i(x)$ построено по своей предвыборке X_i

В каждой вершине дерева будем искать разбиение не по всем признакам, а по подмножеству признаков.

Дерево строится до тех пор, пока в листе не окажется n_{min} объектов

11.3.1. Практические рекомендации

Если p - количество признаков, то при классификации обычно берут $m = \lceil \sqrt{p} \rceil$, а при регрессии $m = \lceil \frac{p}{3} \rceil$ признаков.

При классификации обычно дерево строится, пока в листе не окажется $n_{min} = 1$ объект, а при регрессии $n_{min} = 5$ объектов.

11.3.2. Out-Of-Bag ошибка

Для каждого дерева посчитаем Err_i на тех объектах, на которых оно не обучалось. Итоговая ошибка рассчитывается как

$$Err_{oob} = \frac{1}{B} \sum_{i=1}^B Err_i$$

Это некий аналог cross-валидации.

При большом количестве деревьев, метрика Out-Of-Bag даёт очень хороший результат. По ООВ можно подобрать оптимальное кол-во деревьев.

12. Лекция 7. Случайный лес. Наша лекция

Вспомним, что математическое ожидание среднеквадратичной ошибки для различных вариаций тренировочной выборки можно разложить как

$$\mathbb{E}(y - \hat{y})^2 = bias^2 + Var + noise$$

12.1. Программируем

12.1.1. Загружаем датасет

```
from sklearn.model_selection import train_test_split
from mlxtend.data import boston_housing_data

X, y = boston_housing_data()
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size = 0.3,
                                                    random_state = 123,
                                                    shuffle = True)
```

12.1.2. Обучим решающее дерево без ограничений

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

dt = DecisionTreeRegressor(random_state = 123)
dt.fit(X_train, y_train)
print("MSE_train:", mean_squared_error(y_train, dt.predict(X_train)))
print("MSE_test:", mean_squared_error(y_test, dt.predict(X_test)))
```

12.1.3. Получим примерное смещение и разброс

```
from mlxtend.evaluate import bias_variance_decomp

_, avg_bias, avg_var = bias_variance_decomp(dt, X_train, y_train, X_test, y_test,
                                             loss = 'mse', random_seed = 123)
```

12.1.4. Посмотрим на смещение и разброс у бэггинга деревьев

```
from sklearn.ensemble import BaggingRegressor

base_tree = DecisionTreeRegressor(random_state = 123)

bagging = BaggingRegressor(base_estimator = base_tree, n_estimators = 20, random_state = 123)
_, avg_bias, avg_var = bias_variance_decomp(bagging, X_train, y_train, X_test, y_test,
                                             loss = 'mse', random_seed = 123)
```

12.1.5. Посмотрим как это повлияло на MSE

```
print("MSE_train:", mean_squared_error(y_train, bagging.predict(X_train)))
print("MSE_test:", mean_squared_error(y_test, bagging.predict(X_test)))
```

12.1.6. Построим случайный лес

```
from sklearn.ensemble import RandomForestRegressor  
  
rf = RandomForestRegressor(n_estimators = 20, random_state = 123)  
  
, avg_bias, avg_var = bias_variance_decomp(rf, X_train, y_train, X_test, y_test,  
                                             loss = 'mse', random_seed = 123)  
  
print("MSE_train:", mean_squared_error(y_train, rf.predict(X_train)))  
print("MSE_test:", mean_squared_error(y_test, rf.predict(X_test)))
```

12.1.7. Переобучение случайного леса

Давайте посмотрим на ошибку случайного леса в зависимости от количества деревьев

```
n_trees = 100  
train_loss = []  
test_loss = []  
  
for i in range(1, n_trees):  
    rf = RandomForestRegressor(n_estimators = i, random_state = 123)  
    rf.fit(X_train, y_train)  
    train_loss.append(mean_squared_error(y_train, rf.predict(X_train)))  
    test_loss.append(mean_squared_error(y_test, rf.predict(X_test)))  
  
plt.figure(figsize = (10, 7))  
plt.grid()  
plt.plot(train_loss, label = 'MSE_train')  
plt.plot(test_loss, label = 'MSE_test')  
plt.ylabel('MSE')  
plt.xlabel('#trees')  
plt.legend()
```

12.2. Важность признаков

Relative importance - важность признака в относительных величинах. Допустим у нас в корне дерева лежит 1000 объектов. В какой-то вершине мы делим по какому-то признаку 200 объектов и ещё в одной вершине по этому же признаку делим 50 объектов. Тогда у этого признака будет важность $\frac{200}{1000} + \frac{50}{1000} = 0.2 + 0.05 = 0.25$.

Impurity-based importance - по каждому разбиению по признаку мы ещё можем посчитать $Q = H(R) - (H(R_l) + H(R_r))$, а потом просуммируем $Q_1 + Q_2 + \dots + Q_n$ для одного признака.

Это не всегда хорошо работает, потому что этот подход зависит от количества уникальных значений признаков.

Как получше померить важность признака? Давайте возьмём тот признак, важность которого хотим померить, и перемешаем значения этого признака для всех объектов и обучим модель на таких данных. Если качество модели особо не уменьшилось - значит модель не использовала этот признак.

12.2.1. Программируем изучение важности признаков

```
from sklearn.inspection import permutation_importance  
r = permutation_importance(rf, X_test, y_test,  
                           n_repeats=30,  
                           random_state=0)  
  
for i in r.importances_mean.argsort()[:-1]:  
    if r.importances_mean[i] - 2 * r.importances_std[i] > 0:  
        print(f"data.feature_names[{i}]: {r.importances_mean[i]:.3f}"  
              f"\u00b1{r.importances_std[i]:.3f}")
```

```
plt.figure(figsize = (10, 7))
plt.bar(data.feature_names, r.importances_mean)
```

13. Лекция 8. Бустинг. Лекция с ФЭН

13.1. Случайный лес

Возьмём в качестве базовых алгоритмом для бэггинга решающие деревья. Отличие от бэггинга будет в том, что каждой вершине дерева будем искать разбиение не по всем признакам, а по подмножеству признаков. Это нужно для некоррелированности моделей. Это даст нам уменьшение разброса.

Итоговая композиция имеет вид $\alpha(x) = \frac{1}{n} \sum_{i=1}^n b_i(x)$.

13.2. Бустинг

В среднем бустинг даёт результаты лучше, чем случайный лес и считается одной из самой лучшей моделью для табличных данных.

Тут деревья не обучаются одновременно, как в случайном лесе. Идея бустинга в том, что каждый алгоритм исправляет ошибку предыдущего.

13.2.1. Частный случай бустинга

Давайте решать задачу минимизации MSE:

$$\frac{1}{l} \sum_{i=1}^l (\alpha(x_i) - y_i)^2 \rightarrow \min_{\alpha}$$

Ищем алгоритм $\alpha(x)$ в виде суммы n базовых алгоритмов:

$$\alpha(x) = \sum_{i=1}^n b_i(x)$$

где базовые алгоритмы $b_i(x)$ принадлежат некоторому семейству A .

13.2.2. Алгоритм бустинга для MSE

1. Ищем алгоритм $b_1(x)$, минимизирующий ошибку:

$$b_1(x) = \arg \min_{b \in A} \frac{1}{l} \sum_{i=1}^l (b(x_i) - y_i)^2$$

Ошибка на объекте x : $s = y - b_1(x)$

Следующий алгоритм должен настраиваться на эту ошибку, то есть целевая переменная для следующего алгоритма - это вектор ошибок s (а не исходный вектор y).

2. Ищем алгоритм $b_2(x)$, настраивающийся на ошибки s первого алгоритма:

$$b_2(x) = \arg \min_{b \in A} \frac{1}{l} (b(x_i) - s_i)^2$$

3. Алгоритм $b_3(x)$ будем выбирать так, чтобы он минимизировал ошибку предыдущей композиции (то есть $b_1(x) + b_2(x)$)

4. Ошибка $s_i^{(n)} = y_i - \sum_{j=1}^n b_j(x_i) = y_i - \alpha_{n-1}(x_i)$. Ищем алгоритм $b_n(x)$:

$$b_n(x) = \arg \min_{b \in A} \frac{1}{l} \sum_{i=1}^l (b(x_i) - s_i^{(n)})^2$$

Такой алгоритм называется "Градиентный бустинг". Ошибка на n -м шаге - это антиградиент функции потерь по ответу модели, вычисленный в точке ответа уже построенной композиции:

$$s_i^{(n)} = y_i - \alpha_{n-1}(x_i) = -\frac{\partial}{\partial z} \frac{1}{l} (z - y_i)^2 \Big|_{z=\alpha_{n-1}(x)}$$

Пусть $L(y, z)$ - произвольная дифференцируемая функция потерь. Строим алгоритм $\alpha_n(x)$ вида

$$\alpha_L(x) = \sum_{i=1}^L \gamma_i b_i(x)$$

где на n -м шаге

$$b_n(x) = \arg \min_{b \in A} \sum_{i=1}^l (b(x_i) - s_i^{(n)})^2$$

Тогда $s_i^{(n)} = -\frac{\partial L}{\partial z}$

Коэффициент γ_n должен минимизировать ошибку:

$$\gamma_n = \min_{\gamma_n \in \mathbb{R}} \sum_{i=1}^l L(y_i, \alpha_{n-1}(x_i) + \gamma_n b_n(x_i))$$

13.3. Выбор базовых алгоритмов

Что произойдёт с предсказанием бустинга, если базовые алгоритмы слишком простые? Если взять слишком простые базовые алгоритмы, то они не могут найти зависимость в данных и ничего не получится. Бустинг будет что-то типа случайного блуждания.

Что будет, если базовые алгоритмы слишком простые? Если взять слишком сложные модели, по типу решающих деревьев с глубиной 10 это тоже плохо. Алгоритм слишком быстро переобучится. Одно дерево слишком быстро переобучается, а несколько тем более быстрее будут.

Чаще всего в качестве базовых алгоритмов используют решающие деревья.

В таком случае решающие деревья не должны быть очень маленькими, а также очень глубокими: оптимальная глубина 3-6 (зависит от задачи). Оптимальную глубину можно подбирать с помощью GridSearch'a.

13.4. Смещение и разброс бустинга

Смещение у бустинга будет меньше, чем у одной модели, потому что мы каждым следующим алгоритмом исправляем ошибки предыдущего.

Разброс может получиться большим, гарантий нет. Мы на каждом алгоритме всё больше и больше подстраиваемся под данные и можем переобучиться.

Для этого ещё нужно смотреть оптимальное количество операций бустинга: если в случайном лесе у нас большое кол-во деревьев не увеличивало ошибку, то здесь спокойно может.

13.5. Стохастический градиентный бустинг

Будем обучать базовый алгоритм b_n не по всей выборке X , а по случайному взятой подвыборке $X^k \subset X$. Благодаря этому:

- Снижается уровень шума в данных
- Вычисления становятся быстрее

Обычно берут $|X^k| = \frac{1}{2}X$.

13.6. Имплементации градиентного бустинга

- Xgboost
- CatBoost
- LightGBM

13.6.1. Xgboost

На каждом шаге градиентного бустинга решается задача

$$\sum_{i=1}^l (b(x_i) - s_i)^2 \rightarrow \min \iff \sum_{i=1}^l \left(-s_i b(x_i) + \frac{1}{2} b^2(x_i) \right)^2 \rightarrow \min_b$$

На каждом шаге xgboost решается задача

$$\sum_{i=1}^l \left(-s_i b(x_i) + \frac{1}{2} h_i b^2(x_i) \right) + \gamma J + \frac{\lambda}{2} \sum_{j=1}^J b_j^2 \rightarrow \min_b \quad (*)$$

где $h_i = \frac{\partial^2 L}{\partial z^2} \Big|_{a_{n-1}(x_i)}$, J - количество листов, второе слагаемое - регуляризатор по кол-ву листьев, а третье слагаемое - регуляризатор по значению (норма коэффициентов).

13.6.1.1. Особенности xgboost

- Базовый алгоритм приближает направление, посчитанное с учетом второй производной функции потерь
- Функционал регуляризуется - добавляются штрафы за количество листьев и за норму коэффициентов
- При построении дерева используется критерий информативности, зависящий от оптимального вектора сдвига
- критерий останова при обучении дерева также зависит от оптимального сдвига

13.6.2. CatBoost

CatBoost - алгоритм, разработанный в Яндексе. Он является оптимизацией xgboost и в отличие от xgboost умеет обрабатывать категориальные признаки.

13.6.2.1. Особенности CatBoost

- Используются симметричные деревья решений. Полные деревья. Если дерево не ограничивать по структуре - оно будет сильно меняться при изменении выборки. Но здесь мы зафиксировали структуру - это всегда будет полное дерево, где на одном уровне дерева будут одни и те же предикаты. Это позволяет дереву быть более устойчивым к переобучению. Выразительная способность у таких деревьев куда хуже - надо брать деревья глубины 12 и больше.
- Для кодирования категориальных признаков используется набор методов (one-hot encoding, счётчики, комбинации признаков и другие). На каждой итерации обучения алгоритмов выбирается наилучшее кодирование.
- Динамический бустинг (ordering boosting). На этапе построения предсказаний в листьях происходит следующие действия:
 - Упорядочиваются объекты в листе
 - Для предсказания y_i у нас ответ будет $\frac{1}{i} \sum_{j=0}^i y_j$.
- Поддержка пропусков в данных
- Обучается быстрее, чем xgboost
- Показывает хороший результат даже без подбора параметров
- Удобные методы: проверка на переобученность, вычисление значений метрик, удобная кросс-валидация и другие

13.6.3. LightGBM

- В других реализациях бустинга деревья строятся по уровням (level-wise tree growth). В LightGBM деревья строятся полиствено (leaf-wise tree growth). То есть на каждом шаге будет добавляться лист, который улучшает точность модели.
- LightGBM разбивает значения категориального признака на два подмножества в каждой вершине дерева, находя при этом лучшее разбиение. Если категориальный признак имеет k различных значений, то возможных разбиений $2^{k-1} - 1$. В LightGBM реализован способ поиска оптимального разбиения за $O(k \log(k))$ операций.
- Ускорение построения деревьев за счёт бинаризации признаков: например из признаков $(2,3,5,9,11,12,16)$ сделать признаки $(1,1,1,1,2,2,2)$ - похожие значения собираем в одно.

14. Лекция 8. Бустинги. Наша лекция

В классическом бустинге используется квадратичная функция потерь, потому что это удобно: функция выпуклая, с одним минимумом и дифференцируемая.

14.1. Программируем

14.1.1. Устанавливаем библиотеки

```
!pip install catboost==1.0.3
!pip install lightgbm==3.2.1
!pip install cmake==3.22.0 # without it xgboost will not import
!pip install xgboost==1.5.0
```

14.1.2. Импорт библиотек

```
import warnings
warnings.filterwarnings('ignore')

import catboost
import lightgbm
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
%matplotlib inline
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
import xgboost

plt.rcParams["figure.figsize"] = (8, 5)
```

14.1.3. Генерация датасета и функция для визуализации решений дерева

```
def plot_surface(X, y, clf):
    h = 0.2
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)
    cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    #
    cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())

X, y = make_classification(n_samples=500, n_features=2, n_informative=2,
                           n_redundant=0, n_repeated=0,
                           n_classes=2, n_clusters_per_class=2,
                           flip_y=0.05, class_sep=0.8, random_state=241)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=241)
```

14.1.4. Отобразим визуализацию дерева для CatBoost и вычислим ROC-AUC

Параметр `n_estimators` у CatBoost - сколько мы хотим деревьев.

```
from catboost import CatBoostClassifier

catboost = CatBoostClassifier(n_estimators=300, logging_level='Silent')
catboost.fit(X_train, y_train)
plot_surface(X_test, y_test, catboost)

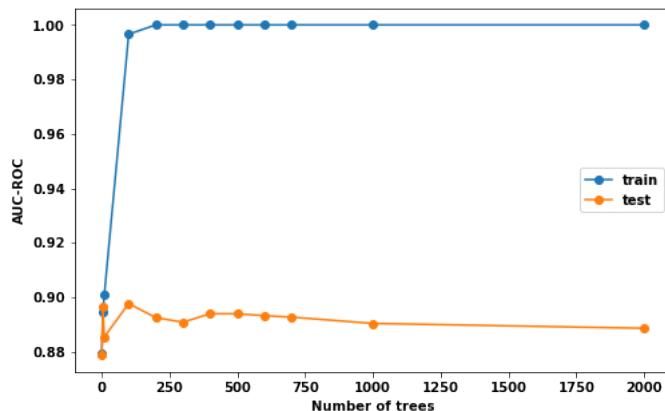
print(roc_auc_score(y_test, catboost.predict_proba(X_test)[:, 1]))
```

14.1.5. Обучаем обычный градиентный бустинг и анализируем качество от количества деревьев

```
from sklearn.ensemble import GradientBoostingClassifier

n_trees = [1, 5, 10, 100, 200, 300, 400, 500, 600, 700, 1000, 2000]
quals_train = []
quals_test = []
for n in n_trees:
    gbc = GradientBoostingClassifier(n_estimators=n)
    gbc.fit(X_train, y_train)
    q_train = roc_auc_score(y_train, gbc.predict_proba(X_train)[:, 1])
    q_test = roc_auc_score(y_test, gbc.predict_proba(X_test)[:, 1])
    quals_train.append(q_train)
    quals_test.append(q_test)

plt.plot(n_trees, quals_train, marker='o', label='train')
plt.plot(n_trees, quals_test, marker='o', label='test')
plt.xlabel('Number of trees')
plt.ylabel('AUC-ROC')
plt.legend()
plt.show()
```



14.1.6. Сделаем то же самое для CatBoost

В алгоритме сделаны улучшения и выбор разных опций для борьбы с переобучением, подсчету среднего таргета на отложенной выборке, подсчету статистик по категориальным фичам, бинаризацией фичей, рандомизации скора сплита, разные типы бутстрепирования.

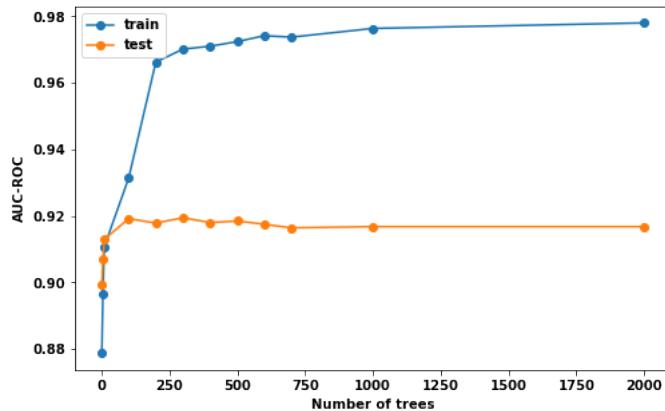
```
n_trees = [1, 5, 10, 100, 200, 300, 400, 500, 600, 700, 1000, 2000]
quals_train = []
quals_test = []
for n in n_trees:
    catboost = CatBoostClassifier(iterations=n, logging_level='Silent')
    catboost.fit(X_train, y_train)
```

```

q_train = roc_auc_score(y_train, catboost.predict_proba(X_train)[:, 1])
q_test = roc_auc_score(y_test, catboost.predict_proba(X_test)[:, 1])
quals_train.append(q_train)
quals_test.append(q_test)

plt.plot(n_trees, quals_train, marker='o', label='train')
plt.plot(n_trees, quals_test, marker='o', label='test')
plt.xlabel('Number_of_trees')
plt.ylabel('AUC-ROC')
plt.legend()
plt.show()

```



14.1.7. Сравнение GradientBoostingClassifier и CatBoost

Разница между графиками

- У CatBoost график выходит на плато и нет переобучения при увеличении числа деревьев.
- Качество у CatBoost на train данных не доходит до единицы - мы не подстраиваемся под данные.
- Метрика ROC-AUC в целом выше.

14.1.8. Решение XGBoost

- Базовый алгоритм приближает направление, посчитанное с учетом второй производной функции потерь
- Функционал регуляризуется – добавляются штрафы за количество листьев и за норму коэффициентов
- При построении дерева используется критерий информативности, зависящий от оптимального вектора сдвига
- Критерий останова при обучении дерева также зависит от оптимального сдвига

Ссылка на источник: <https://github.com/esokolov/ml-course-hse/blob/master/2021-fall/lecture-notes/lecture11-ensembles.pdf>

```

n_trees = [1, 5, 10, 100, 200, 300, 400, 500, 600, 700, 1000, 2000]
quals_train = []
quals_test = []
for n in n_trees:
    xgboost = XGBClassifier(n_estimators=n, verbosity=0)
    xgboost.fit(X_train, y_train)
    q_train = roc_auc_score(y_train, xgboost.predict_proba(X_train)[:, 1])
    q_test = roc_auc_score(y_test, xgboost.predict_proba(X_test)[:, 1])
    quals_train.append(q_train)
    quals_test.append(q_test)

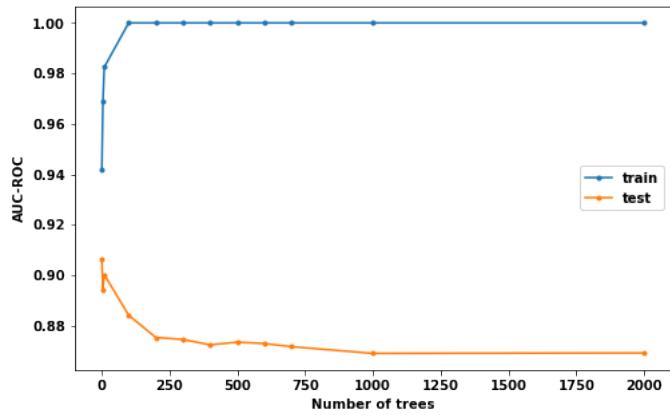
```

```

plt.figure(figsize=(8, 5))
plt.plot(n_trees, quals_train, marker='.', label='train')
plt.plot(n_trees, quals_test, marker='.', label='test')
plt.xlabel('Number of trees')
plt.ylabel('AUC-ROC')
plt.legend()

plt.show()

```



Видно, что на тесте качество от количества деревьев только ухудшается, идёт жёсткое переобучение. Если параметры не настраивать - XGBoost переобучился даже больше, чем обычный бустинг.

14.2. CatBoost для решения задачи + интерпритация признаков

14.2.1. Загрузка данных и разбитие для обучения и теста

```

data = pd.read_csv("bike_buyers_clean.csv")

X = data.drop(['ID', 'Purchased_Bike'], axis=1)
y = data['Purchased_Bike']

from sklearn.model_selection import train_test_split

Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.5, random_state=42)

```

14.2.2. Обучим модель

```

from catboost import CatBoostClassifier

model1 = CatBoostClassifier(cat_features = [0, 1, 4, 5, 6, 8, 9]), # one_hot_max_size=10
model1.fit(Xtrain, ytrain)

```

14.2.3. Построим предсказание

```

from sklearn.metrics import accuracy_score

pred = model1.predict(Xtest)

accuracy_score(ytest, pred)

```

14.2.4. Переберём параметры

- max_depth - глубина деревьев
- learning_rate - с каким коэффициентом прибавляются модели

- ohe_hot_max_size - если модель решит, что ОНЕ лучше кодировать, модель может сделать очень много столбцов и переобучить модель или сделать дольше обучение. Этим параметром мы говорим, что мы не хотим получать больше сколько-то столбцов.

```
from sklearn.model_selection import GridSearchCV

params = { 'max_depth' : np.arange(10,20,3),
           'learning_rate' : [0.01, 0.05, 0.1],
           'one_hot_max_size' : [10, 50, 100]}

gs = GridSearchCV(CatBoostClassifier(n_estimators=100, cat_features = [0, 1, 4, 5, 6, 8, 9]), params)
gs.fit(Xtrain, ytrain)

print(gs.best_score_, gs.best_params_)
```

Вообще перебирать параметры так долго и в дальнейшем будет лекция по auto-ml, где будет говориться о том, как перебирать лучше.

14.3. Оценка важности признаков

Статья - <https://towardsdatascience.com/deep-dive-into-catboost-functionalities-for-model-interpretation>

14.3.1. Вспомогательные функции для оценки

model1 - модель CatBoost

```
from sklearn.metrics import log_loss

def logloss(model, X, y):
    return log_loss(y, model.predict_proba(X)[:,1])

def permutation_importances(model, X, y, metric):
    baseline = metric(model, X, y)
    imp = []
    for col in X.columns:
        save = X[col].copy()
        X[col] = np.random.permutation(X[col])
        m = metric(model, X, y)
        X[col] = save
        imp.append(m-baseline)
    return np.array(imp)
```

14.3.2. Визуализация важности признаков

```
fi_1 = model1.feature_importances_

feature_score = pd.DataFrame(list(zip(Xtest.dtypes.index, fi_1)),
                               columns=['Feature', 'Score'])

feature_score = feature_score.sort_values(by='Score', ascending=False, inplace=False, kind='quicksort')

plt.rcParams["figure.figsize"] = (12,7)
ax = feature_score.plot('Feature', 'Score', kind='bar', color='c')
ax.set_title("Feature_Importance_using_{}".format('Standard_Importances'), fontsize = 14)
ax.set_xlabel("features")
plt.show()
```

14.3.3. Визуализация важности признаков при случайной перестановке

```
fi_2 = permutation_importances(model1, Xtest, ytest, logloss)

feature_score = pd.DataFrame(list(zip(Xtest.dtypes.index, fi_2)),
                             columns=['Feature', 'Score'])

feature_score = feature_score.sort_values(by='Score', ascending=False, inplace=False, kind='quicksort')

plt.rcParams["figure.figsize"] = (12,7)
ax = feature_score.plot('Feature', 'Score', kind='bar', color='c')
ax.set_title("Feature_Importance_using_{}".format('Permutation_Importances'), fontsize=14)
ax.set_xlabel("features")
plt.show()
```

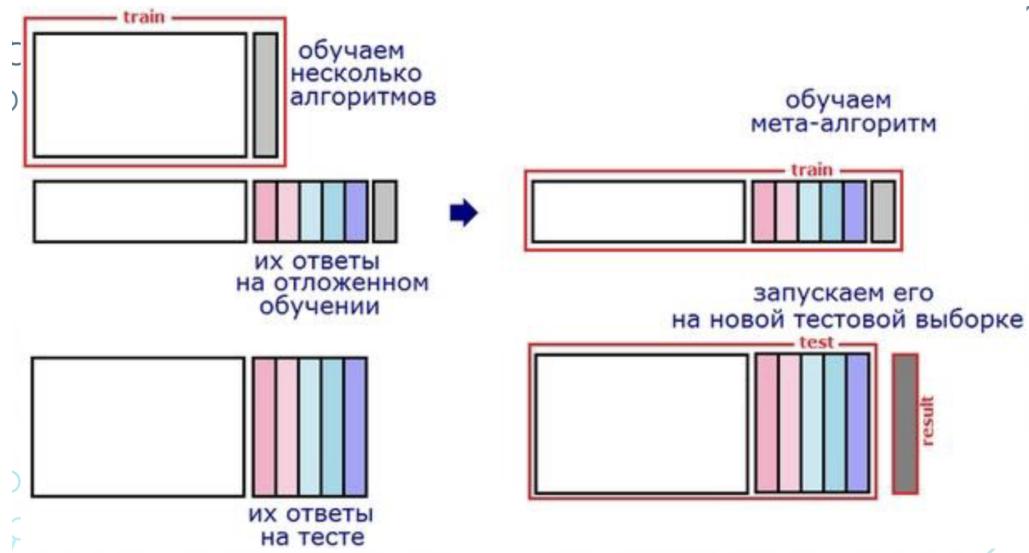
14.4. Блендинг и стекинг

14.4.1. Стекинг

Идея: обучаем несколько **разных** алгоритмов и передаём их результаты на вход последнему, который принимает итоговые решения. Нужно чтобы все базовые алгоритмы были разными, потому что в другом случае стекинг прироста в качестве не даст.

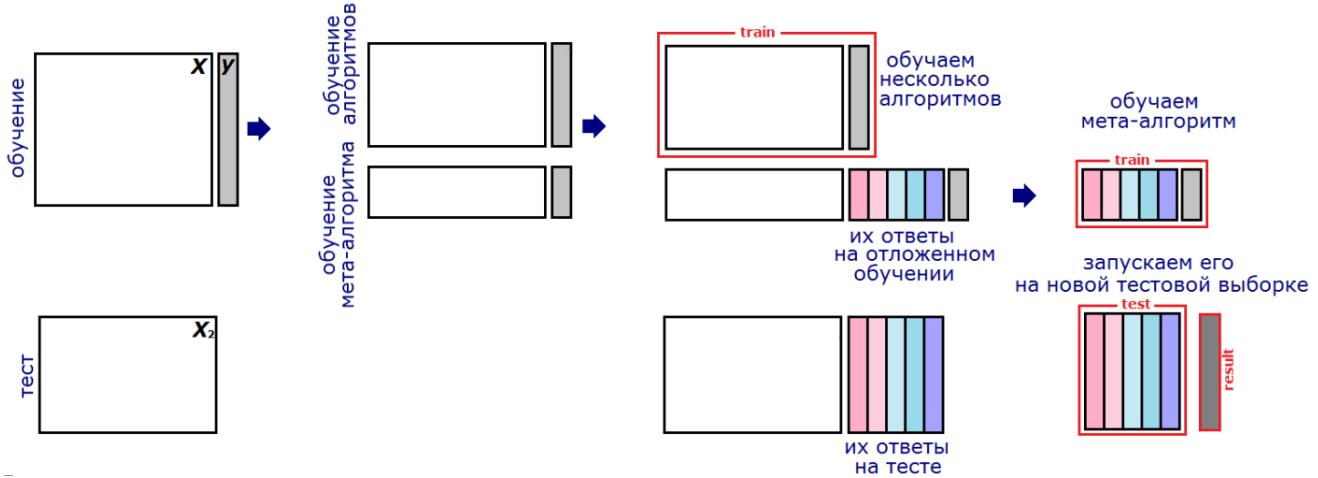
Итоговый алгоритм, который из пресказаний других делает одно, называется мета-алгоритм.

Особенностью стекинга является то, что нужно обучать базовые алгоритмы и мета-алгоритм на разных фолдах.



14.5. Блендинг

Блендинг - это частный случай стекинга, в котором мета-алгоритм линеен:



Но при этом, если мы делаем стекинг или блендинг, то у нас страдает интерпретация модели. Если мы преследуем цель просто поднять качество - то вполне рабочий вариант.

14.6. Программируем блендинг

14.6.1. Загружаем данные

```
from sklearn.datasets import load_boston

data = load_boston()
X_init = pd.DataFrame(data.data, columns=data.feature_names)
y_init = data.target

X, X_test, y, y_test = train_test_split(X_init, y_init, test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.25, random_state=42)

assert X_init.shape[0] == X_train.shape[0] + X_val.shape[0] + X_test.shape[0]

def rmse(y_true, y_pred):
    error = (y_true - y_pred) ** 2
    return np.sqrt(np.mean(error))
```

14.6.2. Проверка качества алгоритмов, если просто обучить на train и проверить на test и блендинг на глазок

```
from catboost import CatBoostRegressor
from sklearn.linear_model import LinearRegression

cbm = CatBoostRegressor(iterations=100, max_depth=4, learning_rate=0.01, loss_function='RMSE')
cbm.fit(X_train, y_train)
test_pred_cbm = cbm.predict(X_test)

lr = LinearRegression()
lr.fit(X_train, y_train)
test_pred_lr = lr.predict(X_test)

print("Test_RMSE_Linear_Regression=% .3f" % rmse(y_test, test_pred_lr))
print("Test_RMSE_Catboost=% .3f" % rmse(y_test, test_pred_cbm))

pred = 0.7 * test_pred_lr + 0.3 * test_pred_cbm

print("Test_RMSE_mix=% .3f" % rmse(y_test, pred))
```

14.6.3. Простой блендинг

Но тут у нас происходит обучение по тесту, это плохо.

14.6.4. Подбор весов

Представим новый алгоритм $\alpha(x)$ как взвешенную сумму из базовых алгоритмов:

$$\alpha(x) = \sum_{i=1}^n w_i \cdot b_i(x)$$

Вот эти самые w_i нам и нужно подобрать.

Сначала рассмотрим более простой случай подбора весов, методом перебора w_1 и получения $w_2 = 1 - w_1$ (так как у нас всего два алгоритма).

Будем подбирать веса на валидации, а проверять качество на тесте.

```
def select_weights(y_true, y_pred_1, y_pred_2):
    grid = np.linspace(0, 1, 1000)
    metric = []
    for w_0 in grid:
        w_1 = 1 - w_0
        y_a = w_0 * y_pred_1 + w_1 * y_pred_2
        metric.append([rmse(y_true, y_a), w_0, w_1])
    return metric

val_pred_cbm = cbm.predict(X_val)
val_pred_lr = lr.predict(X_val)

rmse_blending_train, w_0, w_1 = min(select_weights(y_val, val_pred_cbm, val_pred_lr),
                                     key=lambda x: x[0])
rmse_blending_train, w_0, w_1
```

Получилось получше, чем без блендинга, но давайте проведём его полноценно

14.7. Полноценный блендинг

14.7.1. Загрузка данных

```
data = load_boston()
X_init = pd.DataFrame(data.data, columns=data.feature_names)
y_init = data.target

X, X_test, y, y_test = train_test_split(X_init, y_init, test_size=0.3, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3, random_state=42)

assert X_init.shape[0] == X_train.shape[0] + X_val.shape[0] + X_test.shape[0]
```

14.7.2. Использование блендинга

```
from lightgbm import LGBMRegressor

gb = CatBoostRegressor(iterations=100, max_depth=4, learning_rate=0.01, loss_function='RMSE')
gb.fit(X_train, y_train)

lr = LinearRegression()
lr.fit(X_train, y_train)

meta_train_df = pd.DataFrame()
meta_train_df['gb_preds'] = gb.predict(X_val)
meta_train_df['lr_preds'] = lr.predict(X_val)
```

```

meta_algo = LGBMRegressor()
meta_algo.fit(meta_train_df, y_val)

meta_pred_df = pd.DataFrame()
meta_pred_df[ 'gb_preds' ] = gb.predict(X_test)
meta_pred_df[ 'lr_preds' ] = lr.predict(X_test)
test_preds_meta = meta_algo.predict(meta_pred_df)

rmse(y_test, test_preds_meta)

```

Получается, что при блендинге базовые алгоритмы и мета-алгоритм не используют весь объем выборки обучения, что является недостатком. Для повышения качества нужно усреднять несколько блендигов.

14.8. Используем стекинг

Попробуем реализовать стэкинг. Выборку разбивают на два фолда, последовательно перебирая фолды, обучаю базовые алгоритмы на всех фолдах, кроме одного, а на оставшемся получают ответы базовых алгоритмов и используют их как значения соответствующих признаков на этом фолде. Для получения мета-признаков объектов тестовой выборки базовые алгоритмы обучаю на всей обучающей выборке и берут их ответы на тестовой.

В estimators лежат наши базовые модели

```

from sklearn.ensemble import StackingRegressor, RandomForestRegressor
from sklearn.neighbors import KNeighborsClassifier

estimators = [ ('rf', RandomForestRegressor(n_estimators=200, random_state=42)),
               ('lr', LinearRegression()),
               ('knn', KNeighborsClassifier(n_neighbors=10))]

reg = StackingRegressor(estimators=estimators,
                        cv=10,
                        final_estimator=CatBoostRegressor(iterations=700, max_depth=5, learning_function='RMSE', logging_level='Silent'))

reg.fit(X_train, y_train).score(X_test, y_test)
reg_preds = reg.predict(X_test)
round(rmse(y_test, reg_preds), 3)

```

Получилось качество, куда лучшее предыдущих наших попыток.

15. Материалы к лекции 9. Многоклассовая классификация. Уроки со stepik

15.1. Многоклассовая и multi-label классификация

Важное замечание, что многоклассовая классификация на английский переводится как multiclass classification.

Примеров таких задач множество:

- задача определения класса риска клиентов в скоринге
- задача определения объекта на изображении
- задача определения тональности отзыва и многие другие

Отметим, что в английском языке есть ещё один термин для многоклассовой классификации: это multilabel классификация. Multiclass и multilabel классификации - это задачи разных типов.

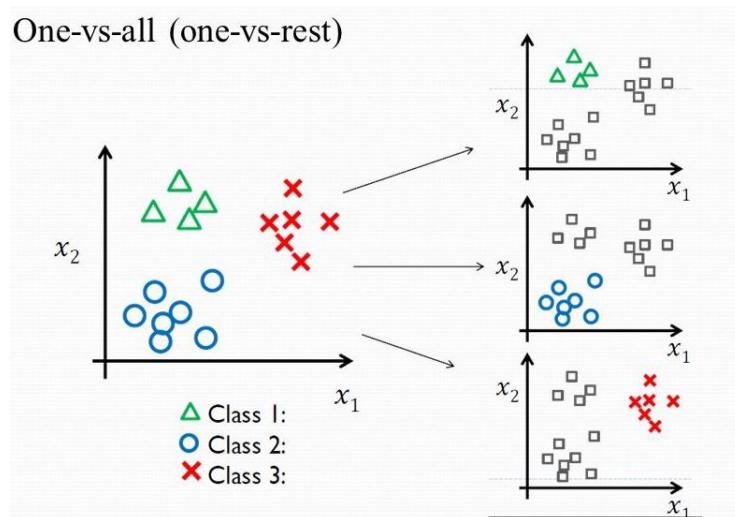
- Multiclass-классификация: задача, в которой целевая переменная - это один из нескольких классов (где классов может быть больше, чем два)
- Multilabel-классификация: задача, в которой для каждого объекта предсказывается несколько меток (например, для фильма одновременно можно предсказать его жанр, год выпуска, язык фильма и так далее). Или же на картинке может быть изображен не один объект, а несколько - и наша задача определить классы всех объектов.

Multilabel-задачу можно свести к серии multiclass-задач.

15.1.1. Сведение к бинарной классификации

Задачи многоклассовой (multiclass) классификации можно свести к серии бинарных задач. Два основных подхода называются one-versus-one (OVO) и one-versus-rest (OVR).

15.1.1.1. One-Versus-Rest Этот подход сводит задачу к серии бинарных классификаторов, где i -й классификатор определяет, относится объект к классу i или не относится (два варианта, то есть бинарный классификатор).



В этом подходе обучается столько бинарных классификаторов, сколько классов в исходной multiclass-задаче.

15.1.1.2. One-Vs-One В этом подходе каждый бинарный классификатор пытается отличить, принадлежит объект классу i или классу j . Каждый такой классификатор обучается только на объектах i -го и j -го классов.

Сравнительная картина подходов выглядит так:

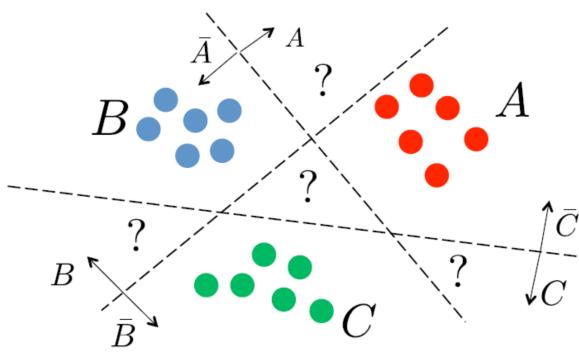


Рис. 2: One-Versus-Rest

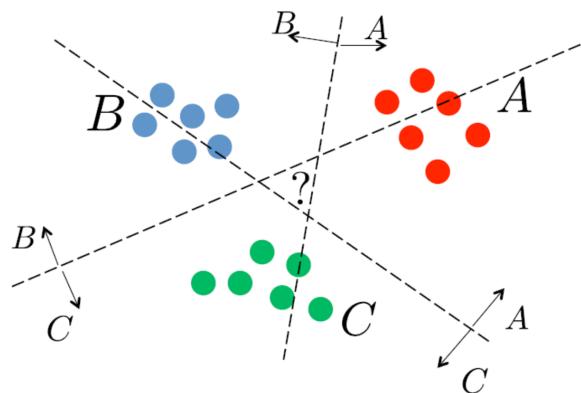


Рис. 3: One-Versus-One

Нельзя в общем случае сказать, какой из подходов лучше. Однако у каждого из подходов есть особенности, связанные со скоростью обучения:

- Если мы выбираем подход one-versus-all, то при большом количестве объектов в данных обучение будет долгим, ведь нужно обучить несколько бинарных классификаторов на большом датасете. В случае one-versus-one каждый бинарный классификатор (с номером ij) обучается только на части датасета, состоящей из классов i и j .
- Если же классов в задаче много, то подход one-versus-one будет работать долго, так как он сводится к обучению бинарных классификаторов на каждой паре классов, то есть, если в задаче N классов, то необходимо обучить $\frac{N \cdot (N - 1)}{2}$ бинарных классификаторов.

15.2. Метрики качества многоклассовой классификации

Для бинарной классификации мы использовали такие метрики как accuracy, precision, recall, f1-score, confusion matrix, roc-auc, pr-auc. Многие из них обобщаются на многоклассовый случай. Давайте поговорим как.

15.2.1. Accuracy

Метрика accuracy - это доля правильных ответов модели, она без изменений в формуле может применяться для любого количества классов.

15.2.2. Recall и f1-score

Теперь матрица ошибок будет не 2×2 , а $N \times N$, где N - количество классов. Пример - матрица классификации животных:

		True/Actual		
		Cat (🐱)	Fish (🐠)	Hen (🐔)
Predicted	Cat (🐱)	4	6	3
	Fish (🐠)	1	2	0
	Hen (🐔)	1	2	6

Для вычисления точности и полноты в этом случае существует несколько подходов:

- Микроусреднение (micro-average)
- Макроусреднение (macro-average)
- Взвешенное усреднение (weighted-average)

15.2.2.1. Макроусреднение (macro-average) В этом подходе мы вычисляем значение выбранной метрики для каждой бинарной ситуации (кошка/не кошка, рыба/не рыба, курица/не курица), а затем усредняем полученные числа.

Например, посчитаем точность и полноту для ситуации кошка/не кошка:

		True/Actual		
		Cat (img alt="cat icon" data-bbox="445 165 475 185"})	Fish (img alt="fish icon" data-bbox="515 165 545 185")>	Hen (img alt="hen icon" data-bbox="585 165 615 185")>
Predicted	Cat (img alt="cat icon" data-bbox="305 190 335 210")	4	6	3
	Fish (img alt="fish icon" data-bbox="305 215 335 235")	1	2	0
	Hen (img alt="hen icon" data-bbox="305 240 335 260")	1	2	6

$$\text{Тогда } precision(cat) = \frac{TP}{TP+FP} = \frac{4}{4+6+3} = \frac{4}{13}$$

то есть false positive - это все объекты, которые модель ошибочно назвала кошкой (их $6 + 3$).

$$recall(cat) = \frac{TP}{TP+FN} = \frac{4}{4+1+1} = \frac{4}{6}$$

Здесь false negative - это все кошки, которых модель не нашла (кошки, названные моделью не кошками).

Тогда macro-average считается как

$$precision = \frac{precision(cat) + precision(fish) + precision(hen)}{3}$$

Аналогично вычисляется macro-average recall.

15.2.2.2. Взвешенное усреднение (weighted-average) В этом подходе мы усредняем посчитанные для каждого класса метрики с весами, пропорциональными количеству объектов класса.

То есть weighted average

$$precision = \frac{6}{25} \cdot precision(cat) + \frac{10}{25} \cdot precision(fish) + \frac{9}{25} \cdot precision(hen)$$

15.2.2.3. Микроусреднение (micro-average) В этом подходе мы вычисляем значения TP, TN, FP, FN по всей матрице ошибок сразу, исходя из их определения. Затем по полученным числам вычисляем выбранные метрики.

TP - это количество верно угаданных объектов положительного класса. В нашем случае $TP = 4+2+6 = 12$

FP - это суммарное количество false positive-предсказаний. Например, если cat предсказана как fish, то это false positive для fish. Таким образом, FP - это сумма всех неверных предсказаний, то есть $FP = 6 + 3 + 1 + 0 + 1 + 2 = 13$

Получаем micro-average

$$precision = \frac{TP}{TP+FP} = \frac{12}{12+13} = \frac{12}{25}$$

Но для recall FN - это сумма false negative-предсказаний. Например, если cat предсказана как fish, то это false negative для cat. Таким образом, FN - это опять же сумма всех неверных предсказаний, то есть $FN = 6 + 3 + 1 + 0 + 1 + 2 = 13$

Получается, что в случае микро-усреднения $precision = recall$.

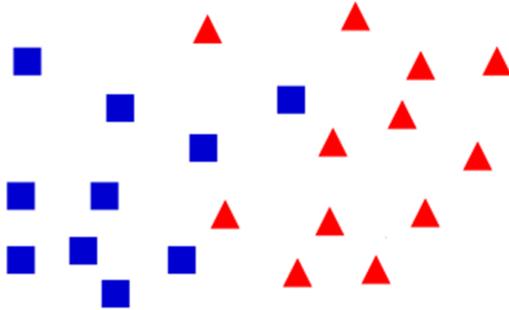
И так как f1-score - это среднее гармоническое точности и полноты, то при микроусреднении $precision = recall = f1 - score$

15.2.3. Результат в python

В python можно получать все эти метрики одной функцией, она называется `sklearn.metrics.classification_report`.

15.3. Метод ближайших соседей (KNN)

Идея метода очень простая, она называется **гипотезой компактности**: схожие объекты находятся близко друг к другу в пространстве признаков.



На рисунке изображены объекты двух классов. Видно, что все квадратики находятся относительно близко друг к другу и далеко от треугольников. Треугольники в свою очередь в основном находятся близко друг к другу.

15.3.1. Алгоритм метода

У метода есть гиперпараметр k - число соседей.

Чтобы определить, к какому классу относится объект, нужно:

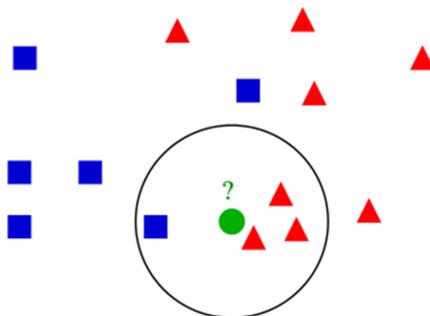
- вычислить расстояние от объекта до каждого объекта выборки
- выбрать k объектов выборки с наименьшим расстоянием (k ближайших соседей)
- класс искомого объекта - это наиболее часто встречающийся класс среди k ближайших соседей

Формально последний пункт записывается так: если Y - множество всех возможных классов в задаче, а y_i - класс i -го объекта из найденных k ближайших объектов, то предсказание модели на объекте q

$$\alpha(q) = \arg \max_{y \in Y} \sum_{i=1}^k I[y_i = y]$$

Сумма берется по k ближайшим к объекту q соседям.

Например, при $k = 4$ для следующей картинки

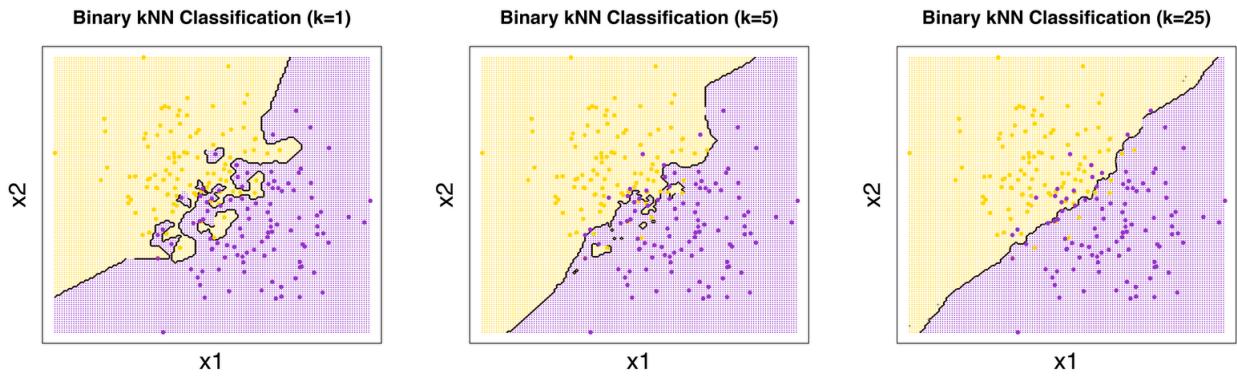


искомый объект будет отнесен к классу треугольников.

15.3.2. Влияние гиперпараметра k

Алгоритм определения классов очень простой. На самом деле у метода нет фазы обучения, потому что нет параметров, которые подбираются в процессе обучения по выборке. В этом смысле метод очень простой.

Несмотря на свою простоту, метод легко переобучается. Например, если взять число соседей очень маленьким ($k = 2$ или $k = 3$), метод будет делать предсказания только по двум или трем самыми близкими точкам, и поэтому сильно подгонится под данные. В этом можно убедиться, посмотрев на разделяющую поверхность метода при разных значениях k :



Разделяющая поверхность при маленьких значениях k очень сложная, и это означает, что модель подстраивается под данные, что ведёт к сильному переобучению.

15.3.3. Выбор метрики

В методе ближайших соседей мы вычисляем расстояния между объектами. Способов вычислить расстояние очень много, каждый из них задается своей формулой (метрикой). Каждая метрика больше подходит для своего типа задач.

Наиболее используемые метрики:

15.3.3.1. Евклидова метрика Евклидова метрика - классический способ измерить расстояние между объектами. Пусть объекты a и b имеют координаты $a = (x_1, y_1)$, $b = (x_2, y_2)$. Тогда евклидово расстояние между ними

$$\rho(a, b) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

15.3.3.2. Манхэттенское расстояние Манхэттенское расстояние - другой способ посчитать расстояние между двумя точками:

$$\rho(a, b) = |x_1 - x_2| + |y_1 - y_2|$$

15.3.3.3. Расстояние Хемминга Расстояние Хемминга - это число различных позиций в координатах двух векторов

Пусть есть вектор $A = \{0, 1\}^n$ и $B = \{0, 1\}^n$, то расстояние хэмминга считается как

$$\rho(a, b) = \sum_{i=1}^n [a_i \neq b_i]$$

15.3.3.4. Мера Жаккара Мера Жаккара - ещё один способ измерить расстояние (часто используется для измерения похожести текстов). Пусть a и b некоторые множества, тогда мера Жаккара - это

$$\rho(A, B) = \frac{A \cap B}{A \cup B}$$

то есть отношение числа совпадающих элементов множеств к общему числу элементов.

Существует множество других метрик. Выбор метрики зависит от задачи и свойств объектов в задаче.

15.3.4. Масштабирование данных для KNN

При использовании KNN в ситуации, когда объекты описываются вектором из числовых признаков необходимо масштабировать данные. Почему так?

Пусть мы ищем ближайшие объекты к объекту $a = (40, 1, 100000)$, где 40 - возраст человека, 1 - пол (1 - мужчина, 0 - женщина), 100000 - месячная зарплата.

Какой объект будет ближайшим к объекту a по евклидовой метрике?

- $b = (80, 0, 90000)$
- $c = (38, 0, 50000)$
- $d = (25, 1, 10000000)$

Если посчитать расстояния между объектами a и b , затем между a и c и, наконец, между a и d , то наименьшим будет расстояние между a и b :

$$\rho(a, b) = \sqrt{(80 - 40)^2 + (0 - 1)^2 + (90000 - 100000)^2} = \sqrt{1600 + 1 + 100000000} \sim 10000$$

Мы видим, что наибольший вклад в ответ вносит зарплата человека, а остальные признаки по большому счету не важны. А это не всегда так, ведь это зависит от задачи и от многих других факторов.

Получилось, что мужчина 40 лет больше всего похож на женщину 80 лет (из предложенных вариантов) просто потому, что у них похожие зарплаты. Так быть не должно!

Поэтому перед применением KNN необходимо привести данные к одному масштабу!

15.3.5. Обобщения KNN

У классического KNN есть один большой недостаток - он никак не учитывает расстояния до ближайших объектов. В то же время понятно, что объекты, находящиеся ближе к объекту запроса, должны вносить больший вклад в ответ. Также учёт расстояний будет очень важен, когда мы будем применять KNN для решения задач регрессии.

Как можно учесть расстояния?

Напомним, что классический KNN определяет класс объекта q как самый популярный класс среди k его ближайших соседей:

$$\alpha(q) = \arg \max_{y \in Y} \sum_{i=1}^k I[y_i = y]$$

то есть все ближайшие соседи входят в формулу с весом 1. Можно изменить вес в зависимости от объекта.

можно упорядочить соседей по увеличению расстояния от объекта запроса qq и давать им вес, обратно пропорциональный номеру соседа: $w_k = \frac{1}{k}$, то есть

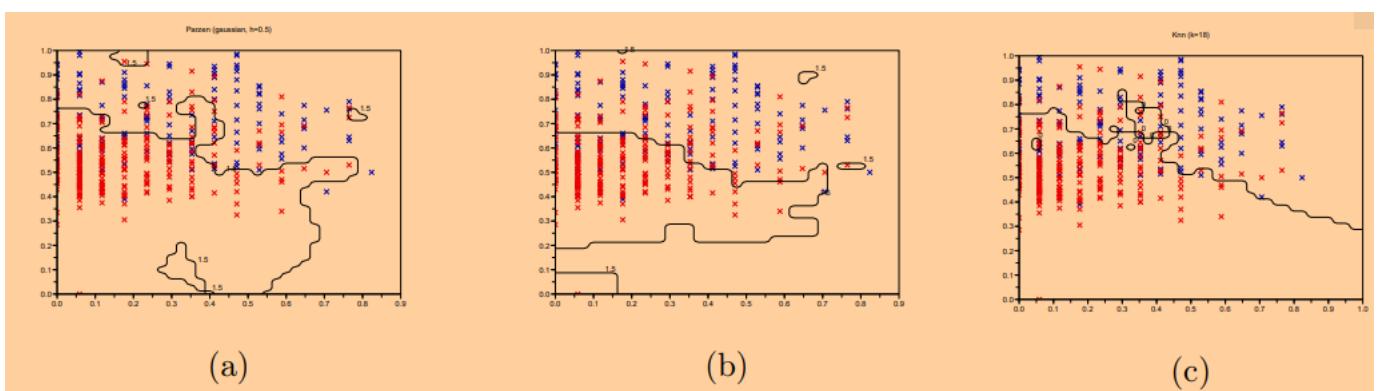
$$\alpha(q) = \arg \max_{y \in Y} \sum_{i=1}^k \frac{I[y_i = y]}{k}$$

Такой подход всё ещё не учитывает реальные расстояния от объекта запроса до ближайших объектов. Способ учесть расстояния называется методом Парзеновского окна:

$$\alpha(q) = \arg \max_{y \in Y} \sum_{i=1}^k K\left(\frac{\rho(q, x_i)}{h}\right) I[y_i = y]$$

где

- q - вектор признаков запроса
- x_i - вектор признаков i -го ближайшего соседа
- h - ширина окна
- Функция $K(x)$, называемая ядром. Существует множество различных ядер, например:
 - $K(x) = \frac{1}{2}I[|x| \leq 1]$ (прямоугольное ядро)
 - $K(x) = (1 - |x|) \cdot I[|X| \leq 1]$ (треугольное ядро)
 - $K(x) = \frac{1}{\sqrt{2\pi}}e^{-2x^2}$ (гауссовское ядро)



На рисунке изображены результаты использования KNN для решения задачи бинарной классификации:

- (a) гауссово ядро, $h = 0.5$
- (b) кубическое ядро, $h = 0.2$
- (c) классический KNN с $k = 18$ соседями

Видно, что использование ядер сильно влияет на результат классификации.

На практике, однако, чаще всего используется прямоугольное ядро для простоты. То есть, например, для окна ширины $h = 1$ веса объектов просто равны $w_i = \frac{1}{2}$, если расстояние между объектами q и x_i не превосходит 1, и 0 иначе.

15.3.6. KNN в задачах регрессии

С помощью KNN можно решать не только задачи классификации, но и задачи регрессии. Существует как простой, так и более сложные подходы - все они полностью аналогичны подходам в задачах классификации:

Простой вариант:

$$\alpha(q) = \frac{1}{k} \sum_{i=1}^k y_i$$

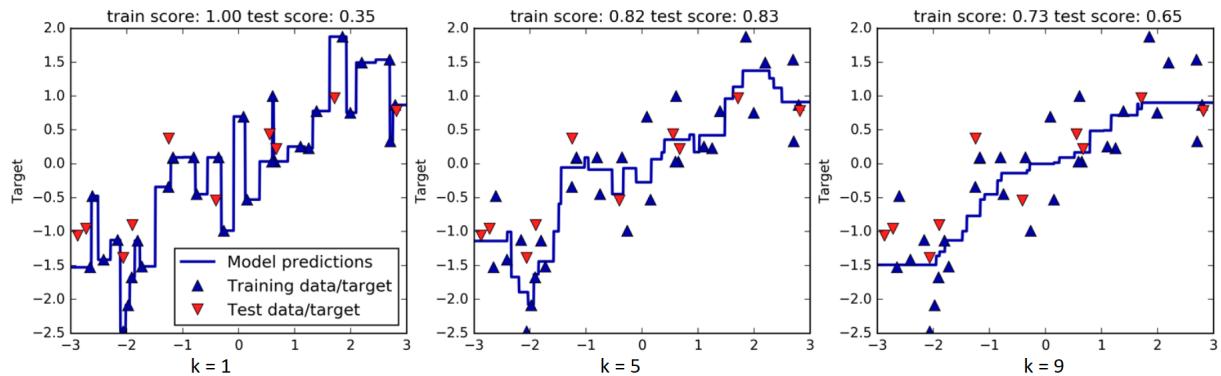
То есть предсказанный ответ это просто среднее арифметическое целевых переменных k соседей.

Взвешенный вариант (формула Надарай-Батсона):

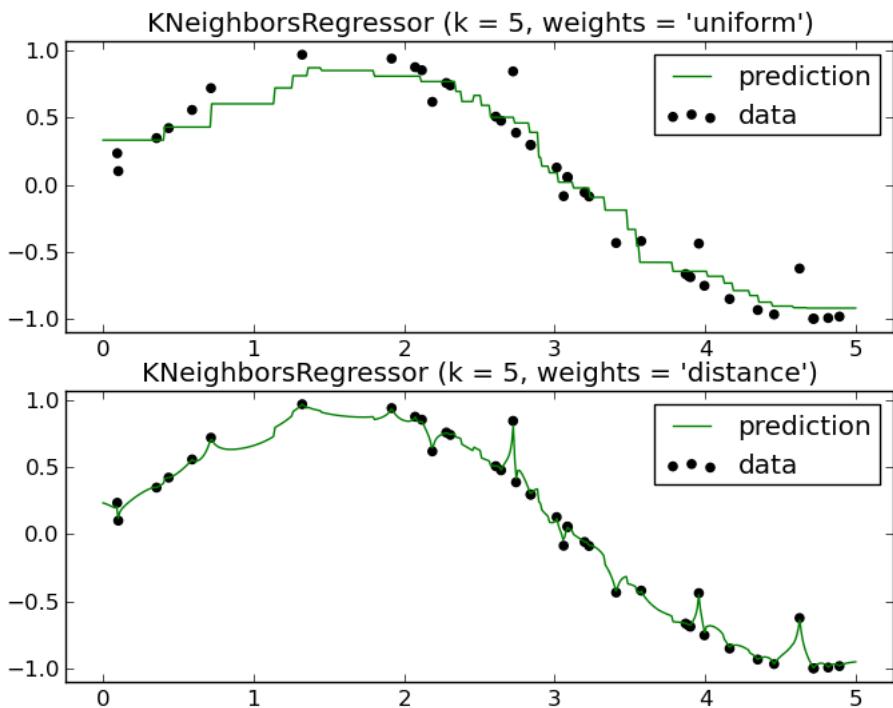
$$\alpha(q) = \frac{\sum_{i=1}^k K\left(\frac{\rho(q, x_i)}{h}\right) \cdot y_i}{\sum_{i=1}^k K\left(\frac{\rho(q, x_i)}{h}\right)}$$

На результат предсказания, как и в задаче классификации, влияет и число соседей, и выбор формулы для предсказания.

Влияние числа соседей изображено на этой картинке:



Пример разных способов учесть расстояния в задаче регрессии изображен здесь:



15.3.7. Преимущества и недостатки KNN

Преимущества:

- Простой алгоритм (для объяснения и для интерпретации)
- Метод не делает никаких предположений о данных (об их линейной разделимости, о распределении данных)
- В некоторых задачах достаточно хорошо работает
- Применяется и для классификации, и для регрессии

Недостатки:

- Требует больших ресурсов по памяти, так как хранит всю выборку
- Требует больших ресурсов по времени, так как вычисляет расстояния до всех объектов выборки
- Чувствителен к масштабу данных
- Зависит от выбранной метрики, которая в свою очередь должна отражать реальное сходство объектов. Найти такую метрику не всегда просто или даже невозможно

15.3.8. Реализация в питоне

Заполнить KNN можно с помощью

```
from sklearn.neighbors import KNeighborsClassifier
```

- Параметр `n_neighbors` - гиперпараметр числа соседей.
- Параметр `weights` принимает способы учесть расстояние до соседей (`uniform`, `distance`).

15.4. Быстрый поиск соседей

В алгоритме KNN для определения ближайших соседей необходимо сделать $O(ld)$ операций, где l - число объектов, d - число признаков. Это очень большая величина, поэтому если данных много - алгоритм будет работать очень долго.

В то же время задача поиска близких объектов очень актуальна. В частности, из множества текстов часто требуется находить похожие. Поэтому нужны алгоритмы, которые работают быстрее, чем KNN и могут определить ближайших соседей с высокой точностью.

15.4.1. Хеширование

Идея хеширования состоит в том, чтобы разбить объекты на корзины (buckets) с помощью некоторой функции (хеш-функции). Процедуру разбиения на бакеты необходимо произвести несколько раз.

Тогда интуитивно понятно, что похожие объекты часто будут попадать в одинаковые бакеты (если хеш-функция для задачи выбрана разумно), а непохожие почти всегда будут попадать в разные бакеты.

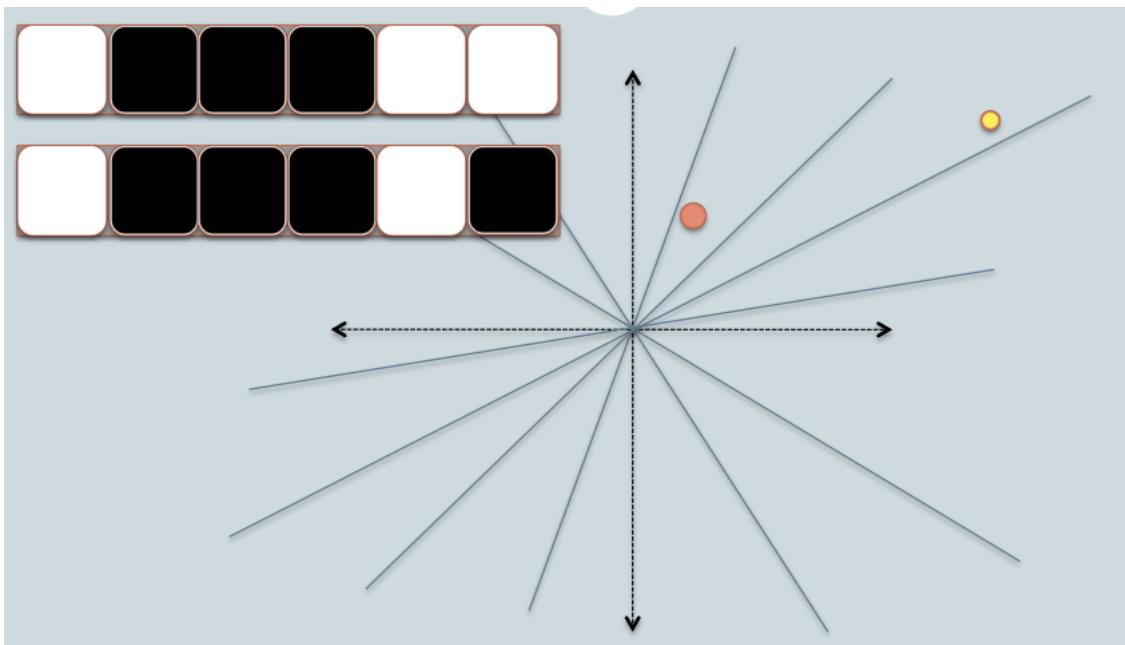
Существует два наиболее используемых подхода к хешированию:

- Метод случайных проекций
- MinHashing

Метод случайных проекций больше используется для табличных задач, где каждый объект - это точка в пространстве признаков. Второй метод больше подходит для поиска похожих текстов.

15.4.2. Метод случайных проекций

Пусть у нас есть произвольные объекты (это точки в d -мерном пространстве, где d - число признаков объекта). Как измерить близость этих объектов?



Алгоритм:

1. Проведем несколько случайных гиперплоскостей, проходящих через начало координат (в нашем примере их 6).
2. Поставим 1 для объекта, если он находится выше гиперплоскости, и 0 если ниже.
3. Таким образом, каждый объект закодирован вектором из 0 и 1 длины 6: первая точка = [0, 1, 1, 1, 0, 0], вторая точка = [0, 1, 1, 1, 0, 1]
4. Схожесть объектов можно определять по количеству совпадающих в кодировке позиций. Это так, потому что похожие объекты находятся близко друг к другу, поэтому часто будут попадать с одной стороны от гиперплоскости. В данном случае похожесть равна $\frac{5}{6}$

Такой способ вычислять схожесть связан с расстоянием Хемминга (Hamming distance). Расстояние Хемминга - это число различных позиций в кодировке. В нашем примере расстояние Хемминга равно 1.

15.4.3. MiniHashing

Второй подход более хитрый, чем первый. Для того, чтобы в нем разобраться, необходимо узнать по шагам, что такое:

- Мера Жаккара и матрица текстов
- MinHashing
- Locality-sensitive hashing

15.4.3.1. Мера Жаккара Часто стоит задача искать именно похожие тексты. Давайте научимся измерять похожесть текстов.

Часто делают так: разбивают текст на буквенные N-граммы (кусочки из N подряд идущих букв), а затем считают долю совпадающих в двух текстах N-грамм среди общего числа различных N-грамм.

Пример: вычислим похожесть имен Маша и Саша.

Разобъем слова на биграммы ($N = 2$)

- Маша → [ма, аш, ша]
- Саша → [са, аш, ша]

Всего различных биграмм 4, а совпадающих 2. Поэтому мера похожести этих слов равна $\frac{2}{4}$.

Эта метрика называется мерой Жаккара (другое название - IoU, Intersection over Union).

Общая формула для меры Жаккара для вычисления похожести A и B :

$$J(A, B) = \frac{A \cap B}{A \cup B}$$

15.4.3.2. Матрица текстов Итак, теперь мы умеем вычислять близость текстов. Надо еще научиться кодировать тексты, используя разбиение на N-граммы.

1. Пронумеруем все N-граммы, встречающиеся в нашем наборе текстов (эти N-граммы иногда называются shingles)
2. Закодируем тексты матрицей:

Documents				
Shingles	0	1	0	1
0	1	0	1	
1	0	1	0	
1	0	0	1	
1	0	1	0	
1	0	1	0	
0	1	0	1	
0	1	0	1	

Здесь по строкам стоят N-граммы, а по столбцам - тексты. Например, если взять второй столбец, то в нём написано, что во втором тексте присутствуют N-граммы 1, 2, 3 и 6, и отсутствуют N-граммы 4, 5 и 7.

Теперь мы умеем кодировать тексты N-граммами и получать из них матрицу. Дальше мы могли бы оценивать схожесть текстов по мере Жаккара, но есть одна довольно **большая проблема**:

В реальных задачах текстов очень много (десятки или сотни тысяч), и различных N-грамм тоже очень много (их могут быть сотни тысяч)! Поэтому полученная матрица будет огромной и при этом разреженной (большинство значений в ней - это нули). Для хранения этой матрицы нужно много ресурсов, кроме того анализ будет занимать большое количество времени.

Поэтому работать с исходной матрицей невозможно, необходимо преобразовать ее так, чтобы по полученной новой матрице меньшего размера аналогичной логикой можно было находить похожие тексты.

15.4.3.3. MinHashing Нам необходимо закодировать каждый текст вектором одной длины так, чтобы избежать очень больших размерностей. При этом кодировка должна сохранять свойство похожести текстов. Для этого используется хеширование с MinHash-функцией.

Объясним как работает MinHash. Пронумеруем строки полученной в предыдущем шаге матрицы текстов и перемешаем их случайным образом:

Permutation π Input matrix (Shingles x Documents)

2	1	0	1	0
3	1	0	0	1
7	0	1	0	1
6	0	1	0	1
1	0	1	0	1
5	1	0	1	0
4	1	0	1	0

MinHash от каждого предложения - это минимальный из индексов, в котором в соответствующем столбце стоит 1.

Для перестановки, показанной на картинке, MinHash получится следующим:

C ₁	C ₂	C ₃	C ₄
----------------	----------------	----------------	----------------

Permutation π Input matrix (Shingles x Documents)

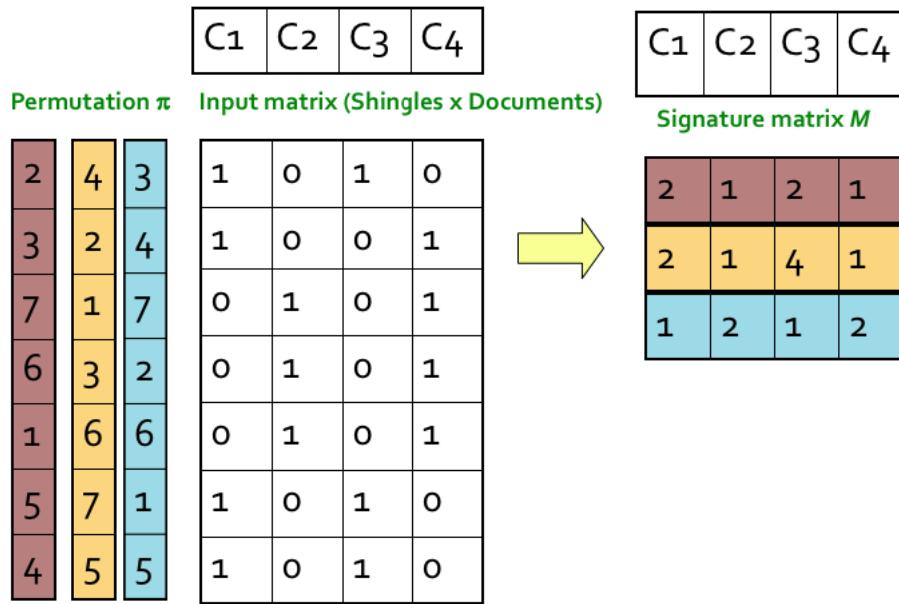
2	1	0	1	0
3	1	0	0	1
7	0	1	0	1
6	0	1	0	1
1	0	1	0	1
5	1	0	1	0
4	1	0	1	0

Signature matrix M



2	1	2	1
---	---	---	---

Одной случайной перестановки недостаточно, поэтому эта процедура (случайная перестановка и вычисление MinHash) происходит несколько раз. Например, при трех случайных перестановках может получиться следующее:



Полученная справа матрица называется матрицей сигнатур (а каждый её столбец - сигнатурой соответствующего текста).

Зачем всё это?

Оказывается, что схожесть текстов, вычисленная по матрице сигнатур, приблизительно равна мере Жаккара, посчитанной по исходным N-граммам! (это не очевидный факт, а теорема с непростым доказательством)

То есть, например, схожесть текстов C_1 и C_3 равна $\frac{2}{3}$ (так как в столбцах C_1 и C_3 совпадают два числа из трёх). Тогда мера Жаккара тоже равна примерно $\frac{2}{3}$

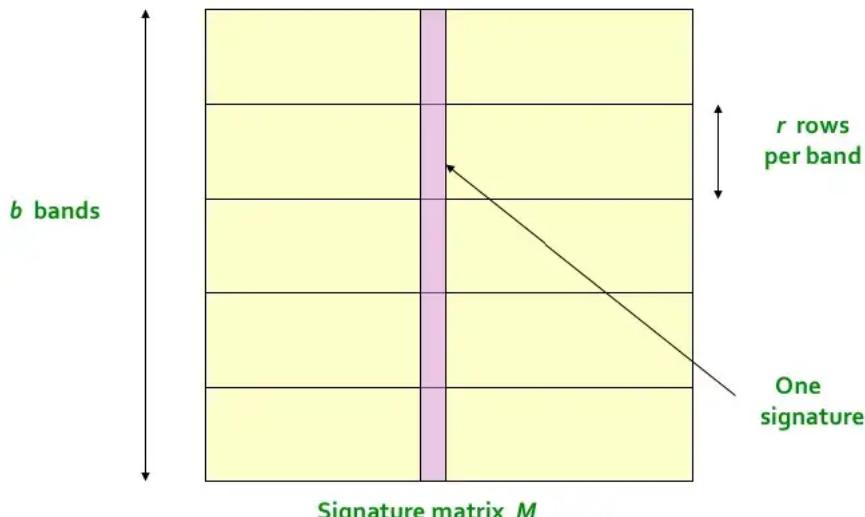
Значит, по матрице сигнатур можно быстро и довольно точно вычислять схожесть текстов!

Конечно, чем больше перестановок используется, тем точнее считается схожесть.

15.4.3.4. Locality-sensitive hashing (LSH) LSH - это алгоритм, который позволяет найти похожие документы с некоторой фиксированной вероятностью. То есть, мы можем зафиксировать порог вероятности, скажем, 80%, и с помощью алгоритма LSH находить документы, похожие друг на друга с этой вероятностью.

Алгоритм LSH заключается в следующем:

- Мы разбиваем матрицу сигнатур на несколько равных полос по строкам (они называются bands):



- Затем для каждой полосы делаем следующее: разбиваем все столбцы на корзины (buckets) таким образом, что если два столбика совпали, то они попадают в одну корзину, а различные столбики попадают в разные корзины.