

Заметки по машинному обучению и анализу данных. Том 2

Подвойский А.О.

Здесь приводятся заметки по некоторым вопросам, касающимся машинного обучения, анализа данных, программирования на языках Python, R и прочим сопряженным вопросам так или иначе, затрагивающим работу с данными.

Краткое содержание

1	Классические алгоритмы машинного обучения	3
2	Глубокое обучение	9
3	Приемы работы с библиотекой анализа временных рядов ETNA	15
4	Генерация признаков и кодирование категориальных признаков	29
5	Перестановочная важность признаков и важность признаков по Шепли	33
6	Приемы работы с библиотекой Polars	36
7	Приемы работы с библиотекой Catboost	41
8	Приемы работы с решателем SCIP	47
9	Приемы работы с библиотеками Gym и Escole	49
10	Нейронные сети	56
11	Приемы работы с библиотекой marshmallow	56
12	Борьба с переобучением в нейронных сетях	58
13	Графовые нейронные сети	59
14	Отбор признаков с библиотекой BoostARoota	63
15	Классический и байесовский бутстреп	63
16	HDI	63
17	Площадь по ROC-кривой	64
18	Приемы работы с Gurobi	66
	Список иллюстраций	67

Содержание

1 Классические алгоритмы машинного обучения	3
1.1 Линейная регрессия	3
1.2 Логистическая регрессия	4
1.3 Деревья решений	5
1.4 Метод опорных векторов	6
1.5 Метод k ближайших соседей	8
1.6 Многослойный персептрон	9
2 Глубокое обучение	9
2.1 Сверточная нейронная сеть	11
2.2 Рекуррентная нейронная сеть	13
3 Приемы работы с библиотекой анализа временных рядов ETNA	15
3.1 Перекрестная проверка на временных рядах	15
3.2 CatBoost. Базовая модель с конструированием признаков	17
3.3 Пользовательские классы для вычисления скользящих статистик	20
3.4 Работа с несколькими временными рядами	26
4 Генерация признаков и кодирование категориальных признаков	29
4.1 Кодирование одного категориального признака по другому категориальному признаку с помощью сингулярного разложения	30
5 Перестановочная важность признаков и важность признаков по Шепли	33
6 Приемы работы с библиотекой Polars	36
6.1 Установка	36
6.2 Вводные замечания	36
6.3 Polars-выражения	37
6.4 Оконные функции	40
6.5 Универсальные NumPy-функции	40
6.6 Примеры	40
7 Приемы работы с библиотекой Catboost	41
7.1 Установка CatBoost	41
7.2 Ключевые особенности пакета	42
7.3 Параметры	42
7.4 Классификатор CatBoostClassifier	42
7.5 Регрессор CatBoostRegressor	44
7.6 Функции потерь и метрики качества	44
7.6.1 Для классификации	44
7.6.2 Для регрессии	45

8	Приемы работы с решателем SCIP	47
8.1	Общие сведения	48
8.2	Emphasis Settings	48
8.3	Проблемы «using pseudo solution instead»	48
9	Приемы работы с библиотеками Gym и Ecole	49
9.1	Gym	49
9.2	Ecole	49
9.2.1	Общие сведения	50
9.2.2	Observations	50
9.2.3	Анализ примера работы связки Ecole+GNN	51
10	Нейронные сети	56
11	Приемы работы с библиотекой marshmallow	56
12	Борьба с переобучением в нейронных сетях	58
12.1	Нормировка	58
12.2	Инициализация весов	58
12.3	Продвинутая оптимизация	59
13	Графовые нейронные сети	59
14	Отбор признаков с библиотекой BoostARoota	63
15	Классический и байесовский бутстреп	63
16	HDI	63
17	Площадь по ROC-кривой	64
18	Приемы работы с Gurobi	66
	Список иллюстраций	67
	Список литературы	67

1. Классические алгоритмы машинного обучения

1.1. Линейная регрессия

Дано: коллекция размеченных данных $\{\mathbf{x}_i, y_i\}_{i=1}^N$, где N – размер коллекции, \mathbf{x}_i – D -мерный вектор признаков образца $i = 1, \dots, N$, y_i – действительное целевое значение, и каждый признак $x_i^{(j)}, j = 1, \dots, D$ также является действительным числом.

Требуется: сконструировать модель $f_{\mathbf{w}, b}(\mathbf{x})$, являющуюся *линейной комбинацией признаков экземпляра \mathbf{x}*

$$f_{\mathbf{w}, b}(\mathbf{x}) = \mathbf{w}\mathbf{x} + b,$$

где \mathbf{w} – D -мерный вектор параметров, b – действительное число (смещение).

Запись $f_{\mathbf{w},b}$ означает, что модель параметризуется двумя значениями: \mathbf{w} и b .

В линейной регрессии, в отличие от метода опорных векторов, *гиперплоскость* проводится так, чтобы оказаться как можно ближе ко всем *обучающим образцам*.

Чтобы удовлетворить это последнее требование (о прохождении гиперплоскости как можно ближе ко всем обучающим образцам), процедура оптимизации, используемая для поиска оптимальных значений \mathbf{x}^* и b^* , должна *минимизировать* следующее выражение [2, стр. 44]

$$\frac{1}{N} \sum_{i=1}^N (f_{\mathbf{w},b}(\mathbf{x}_i) - y_i)^2.$$

Замечание

Вместо квадратической функции потерь можно было использовать и функцию абсолютного отклонения, но последняя в отличие от квадратической функции потерь *негладкая* (т.е. не имеет непрерывной производной) и потому создает лишние сложности, когда для поиска аналитических решений оптимизационных задач используются методы линейной алгебры.

Аналитические решения для нахождения оптимума функции – это простые алгебраические выражения, и они часто предпочтительнее использования сложных *численных методов оптимизации*, таких как *градиентный спуск*.

Очевидно, что квадраты штрафов выгодны еще и потому, что преувеличивают разность между истинными и прогнозируемыми целевыми значениями, в соответствии с величиной этой разности.

1.2. Логистическая регрессия

Модель логистической регрессии

$$f_{\mathbf{w},b}(\mathbf{x}) \stackrel{\text{def}}{=} \frac{1}{1 + e^{-(\mathbf{w}\mathbf{x}+b)}}$$

То есть другими словами модель логистической регрессии представляет собой линейную комбинацию признаков, обернутую *логистическим сигмоидом* $\sigma(x)$, т.е.

$$f_{\mathbf{w},b}(\mathbf{x}) = \sigma\left(\sum_{i=1}^N w_i x_i + b\right), \quad \sigma(x) = \frac{1}{1 + e^{-x}}.$$

Замечание

В *линейной регрессии* минимизируется средне квадратическая ошибка (MSE), а в *логистической регрессии* максимизируется *логарифм функции правдоподобия*

В статистике функция правдоподобия определяет, насколько правдоподобным выглядит наблюдение (образец) в соответствии с нашей моделью [2, стр. 48].

Критерий оптимизации в логистической регрессии называется максимальным правдоподобием. Вместо того чтобы минимизировать среднеквадратическую ошибку, как в линейной регрес-

сии, мы теперь максимизируем правдоподобие обучающих данных в соответствии с моделью

$$L_{\mathbf{w},b} \stackrel{\text{def}}{=} \prod_{i=1}^N f_{\mathbf{w},b}(\mathbf{x}_i)^{y_i} (1 - f_{\mathbf{w},b}(\mathbf{x}_i))^{(1-y_i)}.$$

Выражение $f_{\mathbf{w},b}(\mathbf{x}_i)^{y_i} (1 - f_{\mathbf{w},b}(\mathbf{x}_i))^{(1-y_i)}$ всего навсего означает, что « $f_{\mathbf{w},b}(\mathbf{x}_i)$, когда $y_i = 1$, и $(1 - f_{\mathbf{w},b}(\mathbf{x}_i))$ иначе».

Таким образом, задача оптимизации в случае логистической регрессии имеет вид

$$\arg \min_{\mathbf{w},b} - \sum_{i=1}^N \left[y_i \ln f_{\mathbf{w},b}(\mathbf{x}) + (1 - y_i) \ln (1 - f_{\mathbf{w},b}(\mathbf{x}_i)) \right].$$

В отличие от линейной регрессии, задача оптимизации выше не имеет аналитического решения. Поэтому в таких случаях обычно используется процедура численной оптимизации – градиентный спуск.

1.3. Деревья решений

Дерево решений – это ациклический граф, который можно использовать для принятия решений. В каждом ветвящемся узле графа исследуется j -ый признак из вектор признаков. Если значение признака ниже определенного порога, то выбирается левая ветвь, а иначе – правая. По достижении листового узла принимается решение о классе, к которому принадлежит образец.

В алгоритме ID3 качество расщипления оценивается с использованием энтропии. Энтропия достигает своего минимума, когда случайная величина может иметь только одно значение. И достигает своего максимума, когда все значения случайной величины равновероятны.

Алгоритм ID3 останавливается на листовом узле в любой из следующих ситуаций:

- Все примеры в листовом узле правильно классифицируются моделью,
- Невозможно найти атрибут для расщипления,
- Расщипление уменьшает энтропию ниже некоторого значения ε ,
- Дерево достигает некоторой максимальной глубины d .

Поскольку в ID3 решение о расщиплении набора данных в каждой итерации является локальным (не зависит от будущих расщиплений), алгоритм не гарантирует оптимального решения. Модель можно улучшить, используя в процессе поиска оптимального дерева решений такие методы, как возврат, хотя и за счет увеличения времени построения модели.

Наиболее широко используемая версия алгоритма обучения дерева решений называется C4.5. Версия алгоритма C4.5 имеет несколько дополнительных особенностей по сравнению с ID3 [2, стр. 54]:

- принимает непрерывные и дискретные признаки,
- поддерживает возможность обработки неполных данных,
- решает проблему переобучения с использованием восходящего метода, известного как «подрезка» (отсечение ветвей)ю

Подрезка заключается в том, чтобы выполнить обратный обход только что созданного дерева и удалить ветви, которые не вносят существенного вклада в уменьшение ошибки, заменив их листовыми узлами.

1.4. Метод опорных векторов

Чтобы с помощью модели метода опорных векторов предсказать, является ли электронное письмо спамом или нет, нужно взять текст письма, преобразовать его в вектор признаков, затем умножить этот вектор на \mathbf{w}^* , вычесть b^* и взять знак результата. Это даст прогноз (+1 означает «спам», а -1 означает «не спам»).

Но как машина находит \mathbf{b}^* и b^* ? Она решает задачу оптимизации. Машины хорошо справляются с оптимизацией функций в условиях ограничений.

Итак, какие ограничения должны удовлетворяться здесь? Прежде всего, модель должна правильно предсказывать метки имеющихся 10 000 данных. Каждый образец задается парой (\mathbf{x}_i, y_i) , где \mathbf{x}_i – вектор признаков i -го образца, а y_i – его метка, которая принимает значение -1 или +1. Ограничения выглядят следующим образом [2]

$$\begin{aligned}\mathbf{w}\mathbf{x}_i + b &\geq +1, \text{ если } y_i = +1, \\ \mathbf{w}\mathbf{x}_i + b &\leq -1, \text{ если } y_i = -1.\end{aligned}$$

Желательно также, чтобы гиперплоскость отделяла положительные данные от отрицательных с максимальным зазором. Зазор – это расстояние между ближайшими образцами двух классов, отделяемых границей принятия решения.

Большой зазор способствует лучшему обобщению, то есть тому, насколько хорошо модель будет классифицировать новые данные.

Для максимизации зазора нужно минимизировать евклидову норму $\|\mathbf{w}\| = \sqrt{\sum_{j=1}^D (w^{(j)})^2}$ [2, стр. 23], так как расстояние между границами определяется как $\frac{2}{\|\mathbf{w}\|}$.

В методе опорных векторов решается следующая задача *оптимизации*: минимизировать евклидову норму $\|\mathbf{w}\|$ с учетом $y_i(\mathbf{w}\mathbf{x}_i - b) \geq 1, i = 1, \dots, N$

Рассмотренная версия алгоритма строит *линейную модель* (граница принятия решения – это прямая линия, плоскость или гиперплоскость). Однако метод опорных векторов также может включать *ядра*, способные сделать границу решения *произвольно нелинейной*.

Замечание

Градиент – обобщение понятия производной на случай скалярной функции векторного аргумента. Другими словами градиент – вектор частных производных

В некоторых случаях невозможно полностью разделить две группы точек из-за шума в данных, ошибок разметки или аномалий (данных, сильно отличающихся от «типичного» образца в наборе данных). Для таких случаев есть версия алгоритма SVM, способная включить гиперпараметр штрафа за неправильную классификацию обучающих данных конкретных классов.

Замечание

Для того чтобы понять связь между ошибкой модели, размером обучающего набора, формой математического уравнения, определяющего модель, и временем построения модели, следует прочитать о *вероятностно-приближительном корректном обучении* (Probably Approximately Correct, PAC). Теория вероятностно-приближительного корректного обучения поможет проанализировать и понять, сможет ли и при каких условиях алгоритм обучения получить приблизительно корректный классификатор

Минимизация $\|w\|$ эквивалентна минимизации $\frac{1}{2}\|w\|^2$. Тогда оптимизационную задачу для метода опорных векторов можно переписать так (алгоритм метода опорных векторов с жестким зазором¹)

$$\min \frac{1}{2}\|w\|^2, \text{ такое, что } y_i(\mathbf{x}_i\mathbf{w} - b) - 1 \geq 0, i = 1, \dots, N. \quad (1)$$

Чтобы распространить SVM на случаи, когда данные невозможно разделить линейно, введем *кусочно-линейную функцию потерь* (hinge loss function): $\max(0; 1 - y_i(\mathbf{w}\mathbf{x}_i - b))$.

Кусочно-линейная функция потерь равна нулю, если прогноз $\mathbf{w}\mathbf{x}_i$ лежит с правильной стороны от границы решения, так как в этом случае правая часть кусочно-линейной функции потерь будет отрицательна. Для данных, лежащих с неправильной стороны, значение функции пропорционально расстоянию от границ решения.

Алгоритм метода опорных векторов, оптимизирующий кусочно-линейную функцию потерь, называют методом опорных векторов с мягким зазором (soft-margin SVM) [2, стр. 55]

$$C\|\mathbf{w}\|^2 + \frac{1}{N} \sum_{i=1}^N \max(0; 1 - y_i(\mathbf{w}\mathbf{x}_i - b)),$$

где C – гиперпараметр, определяющий компромисс между увеличением размера границы решения и гарантией местонахождения каждого \mathbf{x}_i с правильной стороны от границы решения.

Как нетрудно заметить, при достаточно *высоких* значениях C ² (то есть при малой регуляризации решения, когда мы разрешаем модели переобучаться) второй член в кусочно-линейной функции потерь становится пренебрежимо малым, поэтому алгоритм SVM будет пытаться найти наибольший зазор, полностью игнорируя ошибочную классификацию. По мере уменьшения значения C (то есть при увеличении степени регуляризации, когда мы запрещаем модели слишком хорошо подстраиваться под данные) ошибки классификации становятся более дорогостоящими, поэтому алгоритм SVM будет пытаться делать меньше ошибок, жертвуя размером зазора. Как уже говорилось, большой зазор дает лучшее обобщение. Следовательно, C регулирует компромисс между хорошей классификацией обучающих данных (минимальный эмпирический риск) и хорошей классификацией данных в будущем (обобщение) [2, стр. 55].

SVM можно адаптировать для работы с наборами данных, которые нельзя разделять гиперплоскостью в исходном пространстве. Действительно, если удастся преобразовать исходное пространство в пространство более высокой размерности, можно надеяться, что данные станут линейно сепарабельны в этом преобразованном пространстве.

Использование функции для неявного преобразования исходного пространства в пространство более высокой размерности в ходе оптимизации функции стоимости в SVM называется *ядерным трюком* (kernel trick).

Чтобы понять, как работают ядра, прежде нужно посмотреть, как алгоритм оптимизации для SVM находит оптимальные значения для \mathbf{w} и b .

¹hard-margin SVM

²Параметр C в моделях метода опорных векторов и логистической регрессии имеет смысл параметра обратно пропорционального параметру α в модели линейной регрессии

Для решения задачи оптимизации (1) традиционно используется *метод множителей Лагранжа*. Вместо оригинальной задачи проще решить эквивалентную задачу, сформулированную так

$$\max_{\alpha_1, \dots, \alpha_N} \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{k=1}^N y_i \alpha_i (\mathbf{x}_i \mathbf{x}_k) y_k \alpha_k \text{ при условии, что } \sum_{i=1}^N \alpha_i y_i = 0 \text{ и } \alpha_i \geq 0, i = 1, \dots, N,$$

где α_i называются множителями Лагранжа.

В такой формулировке задача оптимизации превращается в выпуклую задачу квадратичной оптимизации, которая эффективно решается применением алгоритмов квадратичного программирования.

Чтобы преобразовать *исходное векторное пространство* в *пространство с большим числом измерений*, нужно преобразовать \mathbf{x}_i в $\varphi(\mathbf{x}_i)$ и \mathbf{x}_k в $\varphi(\mathbf{x}_k)$, а затем перемножить $\varphi(\mathbf{x}_i)$ и $\varphi(\mathbf{x}_k)$. Эти вычисления могут оказаться очень дорогостоящими.

С другой стороны, нас интересует только результат скалярного произведения $\mathbf{x}_i \mathbf{x}_k$, который, как мы знаем, является действительным числом. Нам все равно, как будет получено это число, лишь бы оно было верным. Используя функцию ядра, можно избавиться от дорогостоящего преобразования исходных векторов признаков в векторы с более высокой размерностью и избежать необходимости вычислять их скалярное произведение. Мы заменим эти вычисления простой операцией с исходными векторами признаков, которая даст тот же результат.

Например, вместо преобразования $(q_1, p_1) \rightarrow (q_1^2, \sqrt{2}q_1p_1, p_1^2)$ и $(q_1, p_2) \rightarrow (q_2^2, \sqrt{2}q_2p_2, p_2^2)$ и последующего вычисления скалярного произведения $(q_1^2, \sqrt{2}q_1p_1, p_1^2)$ и $(q_2^2, \sqrt{2}q_2p_2, p_2^2)$, чтобы получить $(q_1^2q_2^2 + 2q_1q_2p_1p_2 + p_1^2p_2^2)$, можно найти скалярное произведение (q_1, p_1) и (q_2, p_2) , чтобы получить $(q_1q_2 + p_1p_2)$, а затем возвести в квадрат, чтобы получить тот же результат $(q_1^2q_2^2 + 2q_1q_2p_1p_2 + p_1^2p_2^2)$.

Это был пример функции ядра, и мы использовали квадратичное ядро $k(\mathbf{x}_i \mathbf{x}_k) \stackrel{\text{def}}{=} (\mathbf{x}_i \mathbf{x}_k)^2$.

Существует несколько функций ядра, из которых наиболее широко используется *радиальная базисная функция* (гауссово ядро, RBF)

$$k(\mathbf{x}, \mathbf{x}') = \exp \left(- \frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2} \right),$$

где $\|\mathbf{x} - \mathbf{x}'\|^2$ – квадрат евклидова расстояния между двумя векторами признаков.

Евклидово расстояние определяется следующим уравнением

$$d(\mathbf{x}_i, \mathbf{x}_k) \stackrel{\text{def}}{=} \sqrt{\sum_{j=1}^D (x_i^{(j)} - x_k^{(j)})^2}.$$

1.5. Метод k ближайших соседей

Метод k ближайших соседей – это непараметрический алгоритм обучения. В отличие от других алгоритмов обучения, позволяющих отбрасывать обучающие данные после построения модели, метод kNN сохраняет все обучающие экземпляры в памяти. Когда появится новый, ранее не встречавшийся образец, алгоритм kNN находит k обучающих данных, наиболее близких к \mathbf{x} , и возвращает наиболее часто встречающуюся метку в случае классификации или среднее значение метки в случае регрессии.

Близость двух экземпляров данных определяется функцией расстояния. Нередко используется *косинусное сходство*

$$s(\mathbf{x}_i, \mathbf{x}_k) \stackrel{\text{def}}{=} \frac{\sum_{j=1}^D x_i^{(j)} x_k^{(j)}}{\sqrt{\sum_{j=1}^D (x_i^{(j)})^2} \sqrt{\sum_{j=1}^D (x_k^{(j)})^2}},$$

которая является мерой сходства двух векторов.

1.6. Многослойный персептрон

Нейронная сеть, так же как модель регрессии или SVM, – это всего лишь математическая функция $y = f_{NN}(\mathbf{x})$. Функция f_{NN} имеет особую форму: это вложенная функция (вроде матрички). Например, для трехслойной нейронной сети, возвращающей скаляр, f_{NN} выглядит так [2, стр. 91]

$$y = f_{NN}(\mathbf{x}) = f_3(\mathbf{f}_2(\mathbf{f}_1(\mathbf{x}))),$$

где \mathbf{f}_1 и \mathbf{f}_2 в уравнении выше – это векторные функции, которые определяются как

$$\mathbf{f}_l(\mathbf{z}) \stackrel{\text{def}}{=} \mathbf{g}_l(\mathbf{W}_l \mathbf{z} + \mathbf{b}_l),$$

где l – это индекс слоя и может охватывать от 1 до любого количества слоев; \mathbf{g}_l – функция активации.

Параметры \mathbf{W}_l (матрица) и \mathbf{b}_l (вектор) определяются для каждого слоя в ходе обучения, с использованием градиентного спуска для оптимизации.

Важный момент: здесь вместо вектора \mathbf{w}_l используется матрица \mathbf{W}_l . Причина в том, что \mathbf{g}_l – векторная функция. Каждая строка $\mathbf{w}_{l,u}$ (u означает «узел») в матрице \mathbf{W}_l является вектором той же размерности, что и \mathbf{z} .

Пусть $a_{l,u} = \mathbf{w}_{l,u} \mathbf{z} + b_{l,u}$. На выходе $\mathbf{f}_l(z)$ возвращает вектор $\{g_l(a_{l,1}), g_l(a_{l,2}), \dots, g_l(a_{l, \text{size}_l})\}$, где g_l – некоторая скалярная функция (возвращает скаляр, а не вектор), size_l – количество узлов в l -ом слое.

Основная цель *нелинейных* компонентов в функции f_{NN} состоит в том, чтобы позволить нейронной сети аппроксимировать *нелинейные функции*. В отсутствие нелинейных компонентов f_{NN} была бы линейной, независимо от количества слоев. Причина в том, что $\mathbf{W}_l \mathbf{z} + \mathbf{b}$ является линейной функцией, а линейная функция от линейной функции также является линейной.

2. Глубокое обучение

Под глубоким обучением подразумевается обучение нейронных сетей, имеющих больше двух невыходных слоев. В прошлом обучение таких сетей усложнялось все больше с ростом количества слоев. В числе самых больших проблем назывались *взрывной рост градиента* и *затухание градиента*, поскольку для определения параметров сети использовался градиентный спуск.

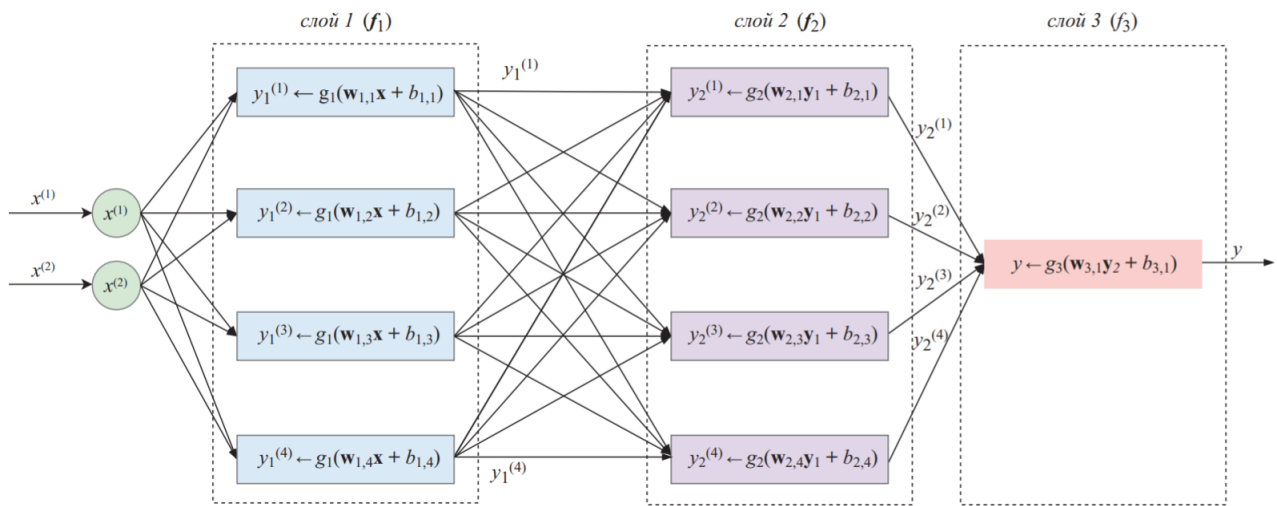


Рис. 1. Многослойный персептрон с двумерным входом, двумя слоями по четыре узла в каждом и выходным слоем с единственным узлом

Если проблема взрывного роста градиента решалась относительно просто, с применением таких простых методов, как *ограничение градиента* и *L1- или L2-регуляризация*, то проблема затухания градиента оставалась неразрешимой в течение десятилетий.

Для обновления значений параметров в нейронных сетях обычно используется алгоритм *обратного распространения*. Обратное распространение – это эффективный алгоритм вычисления градиентов в нейронных сетях с использованием правила дифференцирования сложных функций.

Замечание

В каждой итерации обучения, в процессе градиентного спуска, параметры нейронной сети обновляются пропорционально частной производной функции потерь для текущего параметра

Проблема в том, что иногда *градиент* оказывается *исчезающе малым*, что фактически мешает изменению значений некоторых параметров. В худшем случае это может *полностью остановить обучение* нейронной сети.

Традиционные функции активации, такие функция гиперболического тангенса имеют градиенты в диапазоне $(0, 1)$, при этом градиенты вычисляются на этапе обратного распространения по правилу дифференцирования сложных функций. В результате для вычисления градиентов предыдущих слоев (расположенных *левее*) в n -слойной сети производится перемножение n этих небольших чисел, из-за чего градиент экспоненциально уменьшается с увеличением n . Это приводит к тому, что *более ранние слои* обучаются *намного медленнее*, если вообще обучаются [2, стр. 96].

Однако современные реализации алгоритмов обучения нейронных сетей позволяют эффективно обучать очень глубокие нейронные сети (до нескольких сотен слоев). Это объясняется внедрением целого комплекса усовершенствований, включая ReLU, LSTM (и другие вентильные узлы), а также таких методов, как соединение с пропуском слоя, используемые в остаточных нейронных сетях, а также усовершенствованные версии алгоритма градиентного спуска.

Когда в роли обучающих данных используются изображения, входные данные получаются слишком многомерными. Так как каждый пиксель в изображении – это отдельный признак

2.1. Сверточная нейронная сеть

Сверточная нейронная сеть (CNN) – это особый вид сетей прямого распространения, который значительно сокращает количество параметров в глубокой нейронной сети с большим количеством узлов практически без потери качества модели.

Учитывая, что наиболее важная информация занимает на изображении ограниченную площадь, мы можем разделить изображение на квадратные фрагменты, используя метод скользящего окна. Затем обучить несколько *небольших регрессионных моделей* одновременно, передавая каждой квадратный фрагмент.

Цель каждой небольшой регрессионной модели – научиться обнаруживать определенный шаблон во фрагменте на вход. Например, одна небольшая модель может научиться определять небо, другая – траву, третья – края зданий и т.д.

Небольшие регрессионные модели в CNN напоминают модель многослойного персептрона, но имеют *только по одному слою* [2, стр. 98]. Чтобы обнаружить какой-либо шаблон, модель регрессии должна *определить параметры матрицы* $\mathbf{F}_{p \times p}$. Некоторый фрагмент может выглядеть как следующая матрица \mathbf{P}

$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Этот фрагмент представляет шаблон с изображением креста. Небольшая регрессионная модель, которая будет обнаруживать такие шаблоны (и только их), должна обучить матрицу \mathbf{F} размером 3×3 , в которой параметры в позициях, соответствующих единицам во входном фрагменте, будут положительными числами, а параметры в позициях, соответствующих нулям, будут иметь значения, близкие к нулю. Если вычислить свертку матриц \mathbf{P} и \mathbf{F} , полученное значение будет тем больше, чем больше \mathbf{F} похожа на \mathbf{P} .

Оператор свертки определен только для матриц, имеющих одинаковое количество строк и столбцов (рис. 2).

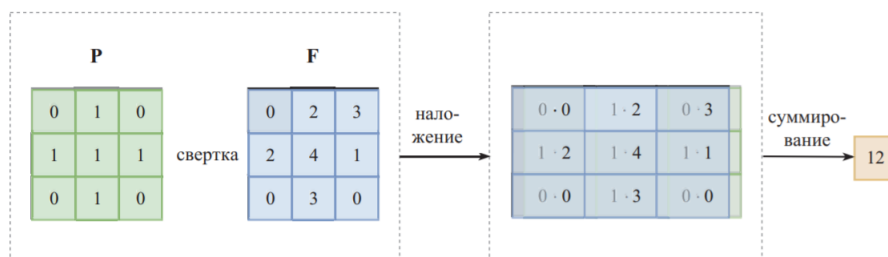


Рис. 2. Свертка двух матриц

Например, если подать на вход фрагмент \mathbf{P} , имеющий L-шаблон

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix},$$

тогда свертка с \mathbf{F} даст в результате меньшее значение: 5.

Замечание

То есть чем больше фрагмент похож на фильтр, тем выше значение операции свертки

Каждый фильтр в первом (самом левом) слое скользит – свертывает – по входному изображению слева направо, сверху вниз, и в каждой итерации вычисляет значение свертки.

Матрица фильтра (по одной для каждого фильтра в каждом слое) и значения смещения являются *обучаемыми параметрами*, которые оптимизируются с использованием градиентного спуска с обратным распространением.

Нелинейность применяется к сумме свертки и смещения, т.е. $\sigma(\mathbf{P} \circ \mathbf{F} + b)$. Как правило, во всех скрытых слоях используется функция активации ReLU. Функция активации в выходном слое зависит от решаемой задачи. Функция активации в выходном слое зависит от решаемой задачи.

Если CNN имеет один сверточный слой, следующий за другим сверточным слоем, то последующий слой $l + 1$ будет обрабатывать выходные данные предыдущего слоя l , как коллекцию *size_l* матриц изображения. Такая коллекция называется *томом*. Размер коллекции называется *глубиной тома*. Каждый фильтр в слое $l + 1$ выполняет свертку всего тома. Свертка фрагмента тома – это просто сумма сверток соответствующих фрагментов отдельных матриц, из которых состоит том.

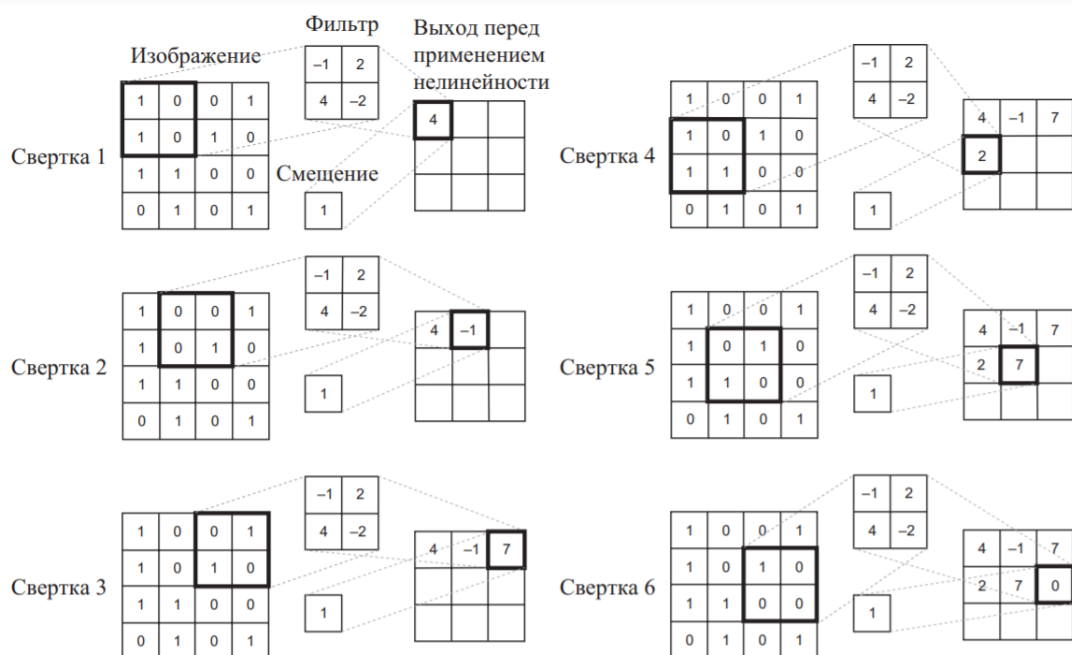


Рис. 3. Фильтр, свертывающий изображение

Свертки имеют два важных свойства – шаг и дополнение. Шаг – это величина одного шага смещения окна. Дополнение позволяет получить увеличенную выходную матрицу. Это ширина

рамки с дополнительными ячейками, которые добавляются вокруг изображения (или тома) перед сверткой с помощью фильтра. Обычно дополнительные ячейки, формирующие дополнение, содержат нули.

Дополнение может пригодиться при использовании более крупных фильтров, позволяя им лучше «сканировать» границы изображения.

Подобно свертке операция подвыборки (пулинга, субдискретизации) имеет гиперпараметры – размер фильтра и шаг. Как правило, слой подвыборки следует за сверточным слоем и получает на входе выходные данные свертки. Когда подвыборка применяется к тому, каждая матрица в этом томе обрабатывается независимо от других. То есть в результате применения подвыборки к тому получается том с той же глубиной.

Замечание

Подвыборка (пулинг, субдискретизация) имеет только гиперпараметры и не имеет обучаемых параметров

На практике обычно используются фильтры с размером 2 или 3 и с шагом 2. Подвыборка с определением максимального значения более популярна, чем с определением среднего, и часто дает лучшие результаты.

2.2. Рекуррентная нейронная сеть

Рекуррентные нейронные сети (RNN) используются для маркировки, классификации или генерации последовательностей. Последовательность – это матрица, каждая строка которой является вектором признаков и в которой порядок строк имеет значение.

Рекуррентная нейронная сеть не является сетью прямого распространения: она содержит циклы. Идея состоит в том, что каждый узел u рекуррентного слоя l имеет *вещественное состояние* $h_{l,u}$. Состояние можно рассматривать как *память узла*. В RNN каждый узел u в каждом слое l имеет два входа: вектор состояний из предыдущего слоя $l - 1$ и вектор состояний из этого же слоя l , но из *предыдущего временного шага*.

Для иллюстрации рассмотрим первый и второй рекуррентные слои в сети RNN. Первый (самый левый) слой получает на входе вектор признаков. Второй слой получает на входе выходные данные из первого слоя (рис. 4).

Каждый обучающий образец представлен матрицей, в которой каждая строка является вектором признаков.

Для обучения моделей RNN используется специальная версия обратного распространения, называемая *обратным распространением во времени*.

Обе функции – \tanh и softmax – страдают проблемой затухания градиентов. Даже если сеть RNN имеет только один или два рекуррентных слоя, из-за последовательного характера входных данных обратное распространение «развертывает» сеть с течением времени. С точки зрения вычисления градиента это означает, что чем длиннее входная последовательность, тем глубже получается развернутая сеть.

Другая проблема, характерная для RNN, заключается в обработке долгосрочных зависимостей. По мере увеличения длины входной последовательности векторы признаков, находящиеся в начале последовательности, постепенно «забываются», потому что состояние всех узлов, которые играют роль памяти сети, в значительной степени зависит от векторов признаков, прочитанных последними.

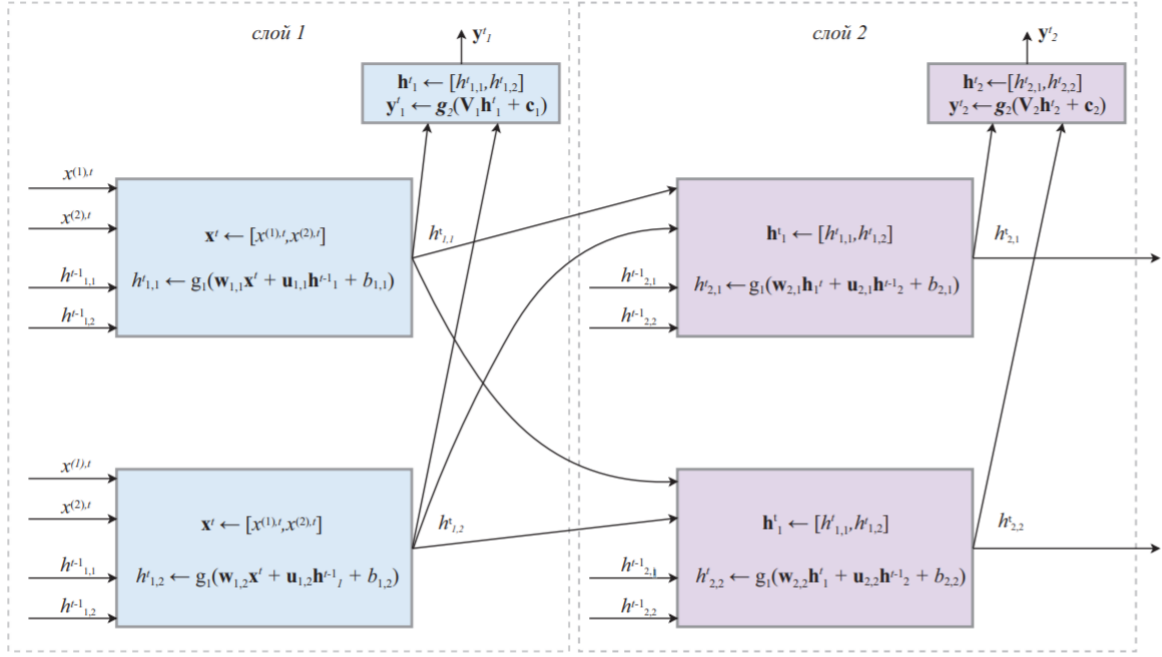


Рис. 4. Первые два слоя в рекуррентной нейронной сети. На вход подается двумерный вектор признаков. Каждый слой имеет два узла

Наиболее эффективными рекуррентными моделями нейронных сетей, используемые на практике, являются вентильные RNN. К ним относятся сети с *долгой краткосрочной памятью* (LSTM) и сети с вентильными рекуррентными узлами (GRU).

В рекуррентных нейронных сетях операции чтения, записи и стирания информации, хранящейся в каждом узле, контролируются функциями активации, которые принимают значения в диапазоне $(0, 1)$. Обученная нейронная сеть может «прочитать» входную последовательность векторов признаков и на некотором раннем временном шаге t решить сохранить конкретную информацию о векторах признаков. Эта информация о более ранних векторах признаков может позже использоваться моделью для обработки векторов признаков в конце входной последовательности.

Решение о том, какую информацию хранить и когда разрешать чтение, запись и удаление, принимают узлы. Эти решения принимаются на основе данных и реализуются через идею вентилей. Есть несколько архитектур управляемых узлов. Простая, но эффективная называется *минимальным вентильным узлом* и состоит из *ячейки памяти* и *вентилья забывания*

$$\begin{aligned}\tilde{h}_{l,u}^t &\leftarrow \tanh(\mathbf{w}_{l,u}\mathbf{x}^t + \mathbf{u}_{l,u}\mathbf{h}_l^{t-1} + b_{l,u}), \\ \Gamma_{l,u}^t &\leftarrow \sigma(\mathbf{m}_{l,u}\mathbf{x}^t + \mathbf{o}_{l,u}\mathbf{h}_l^{t-1} + a_{l,u}), \\ h_{l,u}^t &\leftarrow \Gamma_{l,u}^t \tilde{h}_l^t + (1 - \Gamma_{l,u}^t) h_l^{t-1}, \\ \mathbf{h}_l^t &\leftarrow [h_{l,1}^t, \dots, h_{l,size_l}^t], \\ \mathbf{y}_l^t &\leftarrow \mathbf{g}(\mathbf{V}_l \mathbf{h}_l^t + \mathbf{c}_{l,u}),\end{aligned}$$

где \mathbf{g} – функция softmax.

Если вентиль $\Gamma_{l,u}^t$ близок к 0, тогда ячейка памяти сохраняет значение, полученное на предыдущем временном шаге h_l^{t-1} . Если вентиль $\Gamma_{l,u}^t$ близок к 1, значение ячейки памяти затирается новым значением $\tilde{h}_{l,u}^t$.

3. Приемы работы с библиотекой анализа временных рядов ETNA

3.1. Перекрестная проверка на временных рядах

Перекрестную проверку с расширяющимся окном (или на скользящем окне) в библиотеке ETNA можно выполнить с помощью метода `.backtest()`. Этот метод возвращает три кадра данных: кадр данных с метриками по каждой тестовой выборке перекрестной проверки, кадр данных с прогнозами и кадр данных с временными метками обучающего и тестового поднаборов данных.

В перекрестной проверке расширяющимся окном количество наблюдений, использованных для обучения в каждой итерации, растет с числом итераций, предоставляет все больший объем данных для обучения.



Рис. 5. Перекрестная проверка на временном ряду *расширяющимся* окном

Для тестирования мы каждый раз берем совершенно новые более поздние наблюдения. Обучающая выборка прирастает на количество наблюдений, равное горизонту прогнозирования.

При необходимости обучение модели в каждом разбиении можно сделать последовательным, используя в каждой итерации для обучения фиксированное количество наиболее свежих (поздних) наблюдений, предшествующих точке разбиения. Таким образом, в каждой новой итерации мы будем обучаться на более свежих данных, обучающая выборка каждый раз сдвигается вперед по временной оси (обычно на горизонт прогнозирования) и такой способ проверки называют перекрестной проверкой скользящим окном (sliding/rolling window).



Рис. 6. Перекрестная проверка *на скользящем* окне

С каждой итерацией обучающая выборка использует все более свежие наблюдения, при этом для тестирования мы каждый раз берем совершенно новые более поздние наблюдения. Размер обучающей выборки остается неизменным, поэтому в ETNA этот вид проверки назван **constant**.

NB При выполнении перекрестной проверки для временных рядов полезно помнить ряд правил:

- Размер тестовой выборки, как правило, определяется горизонтом прогнозирования, а тот в свою очередь определяется бизнес-требования. Если вы предсказываете на 14 дней вперед, то и тестовая выборка должна включать 14 более поздних наблюдений.

- Размер тестовой выборки остается постоянным. Это значит, что метрики качества, полученные в результате вычислений прогнозов каждой обученной модели по тестовому набору, будут последовательны и их можно объединять и сравнивать.
- Размер обучающей выборки не может быть меньше тестовой выборки.
- Если данные содержат сезонность, обучающая выборка должна содержать не менее двух полных сезонных циклов (правило $2L$, где L – количество периодов в полном сезонном цикле, необходимое для инициализации параметров некоторых моделей, например, для вычисления исходного значения тренда в модели тройного экспоненциального сглаживания), учитывая уменьшение длины ряда при выполнении процедур обычного и сезонного дифференцирования.
- Если применяются переменные – лаги, разности на лагах, скользящие статистики, то каждый раз для получения значений в тестовой выборке используются только данных обучающей выборки.

Перекрытую проверку расширяющимся окном можно модифицировать так, чтобы обучающая выборка прирастала на количество наблюдений меньше горизонта прогнозирования и тогда в тестовую выборку попадут наблюдения, уже попадавшие в тестовую выборку на предыдущей итерации. Это позволяет управлять скоростью обновления модели, лучше выявлять аномальные, нетипичные наблюдения, которые плохо предсказываются, точнее определить момент ухудшения качества модели.



Рис. 7. Модифицированная перекрытая проверка расширяющимся окном

Перекрытую проверку скользящим окном тоже можно модифицировать так, чтобы обучающая выборка сдвигалась вперед не на весь горизонт прогнозирования, а на половину или на треть, и тогда в тестовую выборку попадут наблюдения, уже попадавшие в тестовую выборку на предыдущей итерации. Это позволяет управлять скоростью обновления модели, лучше выявлять аномальные, нетипичные наблюдения, которые плохо предсказываются, точнее определять момент ухудшения качества модели.

Однако, в библиотеке ETNA и библиотеке scikit-learn с помощью класса TimeSeriesSplit нельзя корректно реализовать вышеописанные модификации.

При использовании перекрытой проверки расширяющимся окном модель в большей степени нацелена на обнаружение глобальных паттернов и менее склонна к изменениям, т.е. более консервативна. При использовании перекрытой проверки скользящим окном используется меньше данных, модель быстрее меняет поведение, т.е. менее консервативна. В ситуации, когда вы уверены, что процесс, генерирующий данные, изменился или неоднократно менялся в течение периода, охватывающего исторические данные, используйте перекрытую проверку скользящим окном.



Рис. 8. Модифицированная перекрестная проверка скользящим окном

Для рынков товаров с низкой вовлеченностью (товаров повседневного спроса), в ситуации, когда вы уверены или у вас есть доказательства, что процесс, генерирующий данные, остается неизменным или претерпевает несущественные изменения, перекрестная проверка расширяющимся окном может быть более полезна.

Замечание

Важно помнить, что во временных рядах *перекрестная проверка*, которую вы применяете, является *прообразом* вашей *производственной системы*. Если вы применяли для валидации перекрестную проверку расширяющимся окном, то и в производстве вы должны обучать модель на обучающей выборке возрастающего объема и обновлять в том же темпе, что обновляли в ходе перекрестной проверки (на весь горизонт прогнозирования, на половину горизонта и т.д.)

Наконец, поскольку в рамках перекрестной проверки расширяющимся окном мы на каждой итерации обучаем модель на выборке все большего объема, при использовании моделей на основе градиентного бустинга это может потребовать коррекции темпа обучения, количества деревьев и максимальной глубины.

3.2. CatBoost. Базовая модель с конструированием признаков

В ETNA есть два класса-обертки над классом `CatboostRegressor`: `CatBoostModePerSegment` и `CatBoostModelMultiSegment`. Разница заключается в том, что класс `CatBoostModelPerSegment` обучает отдельную модель для каждого сегмента, а класс `CatBoostModelMultiSegment` – одну модель для всех сегментов.

Для создания признаков можно использовать классы-трансформеры:

- `LagTransform` для генерации лагов,
- `MeanTransform` для вычисления скользящего среднего по заданному окну.

Замечание

Ширину окна w для скользящих статистик *рекомендуется* задавать равной или превышающей горизонт прогнозирования h , т.е. $w \geq h$. То же относится и к лагам. Порядок лага lag должен быть равен или превышать горизонт прогнозирования, т.е. $lag \geq h$. В противном случае признаки тестового поднабора данных (построенные на лагах с порядком меньшим горизонта прогнозирования), будут использовать значения целевой переменной из тестового поднабора данных (утечка)

С помощью параметра `in_column` класса-трансформера задаем переменную, которую нужно преобразовать или на основе которой нужно создать признаки (по умолчанию этой переменной

будет переменная `target`). С помощью параметра `out_column` (этот параметр есть у всех классов-трансформеров, создающих признаки) можно задать имена генерируемых переменных.

Для более надежной оценки качества модели следует воспользоваться *перекрестной проверкой расширяющимся окном* с помощью класса `Pipeline`.

Создадим список преобразований. В данном случае он включать формирование лагов и скользящего среднего на каждой итерации перекрестной проверки.

```
lags = LagTransform(in_column="target", lags=list(range(8, 24, 1)), out_column="lag")
mean8 = MeanTransform(in_column="target", window=8, out_column="mean8")
transforms = [lags, mean8]
```

Теперь создаем конвейер для выполнения перекрестной проверки расширяющимся окном, передав в него модель, список процедур формирования признаков (лагов и скользящего среднего) и горизонт прогнозирования

```
model = CatBoostModelMultiSegment()
model.fit(train_ts)

pipeline = Pipeline(
    model=model,
    transforms=transforms,
    horizon=HORIZON,
)
# перекрестная проверка расширяющимся окном
metrics_df, _, _ = pipeline.backtest(
    ts=ts,
    mode="expand",
    metrics=[smape],
)
```

Класс `Pipeline` можно использовать для перекрестной проверки сразу нескольких моделей

```
# задаем конвейер преобразований для модели наивного прогноза
naive_pipeline = Pipeline(
    model=NaiveModel(lag=12), transforms=[], horizon=HORIZON)
# задаем конвейер преобразований для Prophet
prophet_pipeline = Pipeline(
    model=ProphetModel(), transforms=[], horizon=HORIZON)
# задаем конвейер преобразований для CatBoost
catboost_pipeline = Pipeline(
    model=CatBoostModelMultiSegment(),
    transforms=[LagTransform(lags=[8, 9, 10, 11, 12],
                             in_column='target')],
    horizon=HORIZON)
# задаем список имен конвейеров
pipeline_names = ['naive', 'prophet', 'catboost']
# задаем список конвейеров
pipelines = [naive_pipeline, prophet_pipeline, catboost_pipeline]
# задаем пустой список метрик
metrics = []
# записываем метрики в список
for pipeline in pipelines:
    metrics.append(
        pipeline.backtest(
            ts=ts, metrics=[MAE(), MSE(), SMAPE(), MAPE()],
            n_folds=3, aggregate_metrics=True
        )[0].iloc[:, 1:]
    )
```

```
)

# конкатенируем метрики
metrics = pd.concat(metrics)
# в качестве индекса используем список имен конвейеров
metrics.index = pipeline_names
```

С помощью класса `VotingEnsemble` можно выполнить обучение и перекрестную проверку ансамбля моделей. Веса моделей можно задавать с помощью параметра `weights`

```
# создаем экземпляр класса VotingEnsemble
voting_ensemble = VotingEnsemble(pipelines=pipelines,
weights=[1, 2, 4],
n_jobs=4)
# получаем метрики
voting_ensemble_metrics = voting_ensemble.backtest(
    ts=ts,
    metrics=[MAE(), MSE(), SMAPE(), MAPE()],
    n_folds=3,
    aggregate_metrics=True,
    n_jobs=2
)[0].iloc[:, 1:]
voting_ensemble_metrics.index = ['voting ensemble']
```

С помощью класса `StackingEnsemble` можно выполнить *стекинг*. Мы прогнозируем будущее, используя метамодель (линейную регрессию по умолчанию) для объединения прогнозов моделей в списке конвейеров. С помощью параметра `final_model` можно задать метамодель. С помощью `features_to_use` можно задавать признаки для метамодели

- `None`: метамодель в качестве признаков может использовать прогнозы моделей конвейеров,
- `List`: прогнозы моделей конвейеров плюс признаки из списка (в виде строковых значений),
- `"all"`: все доступные признаки.

С помощью параметра `cv` задаем количество тестовых выборок перекрестной выборки (используем не для оценки моделей, а для получения прогнозов, которые станут у нас потом признаками).

Под капотом происходит примерно следующее. Допустим, запустили перекрестную проверку расширяющимся окном, получили 5 тестовых выборок, прогнозы каждой из модели конвейера в 5 тестовых выбоках стали признаками. Затем снова запускаем проверку расширяющимся окном, по этим признакам строим метамодель – линейную регрессию, берем прогнозы в 3 тестовых выборках и усредняем

```
# создаем экземпляр класса StackingEnsemble,
# признаки - прогнозы конвейеров
stacking_ensemble_unfeatured = StackingEnsemble(
    features_to_use='None', pipelines=pipelines,
    n_folds=10, n_jobs=4)
# выполняем стекинг
stacking_ensemble_metrics = stacking_ensemble_unfeatured.backtest(
    ts=ts, metrics=[MAE(), MSE(), SMAPE(), MAPE()], n_folds=3,
    aggregate_metrics=True, n_jobs=2)[0].iloc[:, 1:]
stacking_ensemble_metrics.index = ['stacking ensemble']
stacking_ensemble_metrics
```

3.3. Пользовательские классы для вычисления скользящих статистик

Можно писать свои собственные классы для вычисления скользящих статистик и обучения моделей. Допустим, мы хотим использовать не только скользящие средние, но и скользящие средние абсолютные отклонения

```
# пишем класс MadTransform, вычисляющий скользящие
# средние абсолютные отклонения
class MadTransform(WindowStatisticsTransform):
    """
    MadTransform вычисляет среднее абсолютное отклонение
    (mean absolute deviation - mad) для заданного окна.
    """
    def __init__(
        self,
        in_column: str,
        window: int,
        seasonality: int = 1,
        min_periods: int = 1,
        fillna: float = 0,
        out_column: Optional[str] = None
    ):
        """
        Параметры
        -----
        in_column: str
        имя обрабатываемого столбца
        window: int
        ширина окна для агрегирования
        out_column: str, optional
        имя результирующего столбца. Если не задано,
        используем __repr__()
        seasonality: int
        коэффициент сезонности
        min_periods: int
        Минимальное количество наблюдений в окне
        для агрегирования
        fillna: float
        значение для заполнения значений NaN
        """
        self.in_column = in_column
        self.window = window
        self.seasonality = seasonality
        self.min_periods = min_periods
        self.fillna = fillna
        self.out_column = out_column
        super().__init__(
            window=window,
            in_column=in_column,
            seasonality=seasonality,
            min_periods=min_periods,
            out_column=self.out_column
        )
        if self.out_column is not None
        else self.__repr__(),
        fillna=fillna,
    )
    def _aggregate_window(
        self, series: pd.Series
    ) -> float:
        """Вычисляет mad для серии."""
```

```
tmp_series = self._get_required_lags(series)
return tmp_series.mad(**self.kwargs)
```

Теперь предположим, мы хотим использовать LightGBM вместо CatBoost. Нам понадобится класс LGBRegressor и базовые классы библиотеки ETNA Model и PerSegmentModel.

Сначала надо написать ядро – внутренний класс _LGBMModel, в котором используется LGBRegressor. Символ нижнего подчеркивания указывает, что данный класс будет использоваться внутри других классов. У класса _LGBMModel будут два метода `fit()` и `predict()`.

```
# пишем ядро - внутренний класс _LGBMModel,
# внутри - класс LGBMRegressor
class _LGBMModel:
    def __init__(
        self,
        boosting_type='gbdt',
        num_leaves=31,
        max_depth=-1,
        learning_rate=0.1,
        n_estimators=100,
        **kwargs
    ):
        self.model=LGBMRegressor(
            boosting_type=boosting_type,
            num_leaves=num_leaves,
            max_depth=max_depth,
            learning_rate=learning_rate,
            n_estimators=n_estimators,
            **kwargs
        )
    def fit(self, df: pd.DataFrame):
        features = df.drop(columns=['timestamp', 'target'])
        target = df['target']
        self.model.fit(X=features, y=target)
        return self

    def predict(self, df: pd.DataFrame):
        features = df.drop(columns=['timestamp', 'target'])
        pred = self.model.predict(features)
        return pred
```

Вспомним, что мы можем строить отдельную модель для каждого сегмента и одну модель для всего набора (т.е. всех сегментов). Значит мы можем написать два класса. Начнем с класса, который будет строить отдельную модель для каждого сегмента. Назовем его LGBModelPerSegment. Для этого воспользуемся наследованием, нам понадобится базовый класс PerSegmentModel

```
# пишем класс LGBModelPerSegment, который строит
# отдельную модель LGBM для каждого сегмента
class LGBModelPerSegment(PerSegmentModel):
    def __init__(
        self,
        boosting_type='gbdt',
        num_leaves=31,
        max_depth=-1,
        learning_rate=0.1,
        n_estimators=100,
        **kwargs
    ):
        self.kwargs = kwargs
```

```

model = _LGBMModel(
    boosting_type=boosting_type,
    num_leaves=num_leaves,
    max_depth=max_depth,
    learning_rate=learning_rate,
    n_estimators=n_estimators,
    **kwargs
)
super(LGBMModelPerSegment, self).__init__(
    base_model=model)

```

Теперь напишем класс, который будет строить одну модель для всех сегментов. Назовем его `LGBMModelMultiSegment`. Для этого вновь воспользуемся наследованием, нам понадобится базовый класс `Model`

```

# пишем класс LGBMModelMultiSegment, который строит
# одну модель LGBM для всех сегментов
class LGBMModelMultiSegment(Model):
    def __init__(
        self,
        boosting_type='gbdt',
        num_leaves=31,
        max_depth=-1,
        learning_rate=0.1,
        n_estimators=100,
        **kwargs
    ):
        self.kwargs = kwargs
        super(LGBMModelMultiSegment, self).__init__(
            self._base_model=_LGBMModel(
                boosting_type=boosting_type,
                num_leaves=num_leaves,
                max_depth=max_depth,
                learning_rate=learning_rate,
                n_estimators=n_estimators,
                **kwargs
            )
        )

    def fit(self, ts: TSDataset):
        # превращаем TSDataset в датафрейм pandas
        # с плоским индексом
        df = ts.to_pandas(flatten=True)
        df = df.dropna()
        df = df.drop(columns='segment')
        self._base_model.fit(df=df)
        return self

    def forecast(self, ts: TSDataset):
        result_list = list()
        # собираем новый датафрейм с помощью self._forecast_segment
        # из базового класса
        for segment in ts.segments:
            segment_predict = self._forecast_segment(
                self._base_model, segment, ts)
            result_list.append(segment_predict)

        result_df = pd.concat(result_list, ignore_index=True)
        result_df = result_df.set_index(['timestamp', 'segment'])

        df = ts.to_pandas(flatten=True)

```

```

df = df.set_index(['timestamp', 'segment'])
# заменяем пропуски прогнозами
df = df.combine_first(result_df).reset_index()
df = TSDataset.to_dataset(df)
ts.df = df
# выполняем обратные преобразования
ts.inverse_transform()

return ts

```

Аналогично можно реализовать XGBoost в ETNA. Пишем класс `_XGBModel`

```

class _XGBModel:
    def __init__(
        self,
        booster="gbtree",
        max_depth=3,
        learning_rate=0.1,
        n_estimators=100,
        **kwargs,
    ):
        self.model=XGBRegressor(
            booster=booster,
            max_depth=max_depth,
            learning_rate=learning_rate,
            n_estimators=n_estimators,
            **kwargs,
        )

    def fit(
        self,
        df: pd.DataFrame,
    ):
        features = df.drop(columns=["timestamp", "target"])
        for col in features.columns.tolist():
            features[col] = features[col].astype("category").cat.codes
        target = df["target"]
        self.model.fit(X=features, y=target)
        return self

    def predict(
        self,
        df: pd.DataFrame,
    ):
        features = df.drop(columns=["timestamp", "target"])
        for col in features.columns.tolist():
            features[col] = features[col].astype("category").cat.codes
        pred = self.model.predict(features)
        return pred

```

Пишем классы `XGBModelPerSegment` и `XGBModelMultiSegment`

```

# пишем класс XGBModelPerSegment, который строит
# отдельную модель XGB для каждого сегмента
class XGBModelPerSegment(PerSegmentModel):
    def __init__(
        self,
        booster='gbtree',
        max_depth=3,
        learning_rate=0.1,
        n_estimators=200,
    ):

```

```

        **kwargs
    ):
        self.kwargs = kwargs
        model = _XGBModel(
            booster=booster,
            max_depth=max_depth,
            learning_rate=learning_rate,
            n_estimators=n_estimators,
            **kwargs
        )
        super(XGBModelPerSegment, self).__init__(
            base_model=model)

# пишем класс XGBModelMultiSegment, который строит
# одну модель XGB для всех сегментов
class XGBModelMultiSegment(Model):
    def __init__(
        self,
        booster='gbtree',
        max_depth=3,
        learning_rate=0.1,
        n_estimators=100,
        **kwargs
    ):
        self.kwargs = kwargs
        super(XGBModelMultiSegment, self).__init__()
        self._base_model=_XGBModel(
            booster=booster,
            max_depth=max_depth,
            learning_rate=learning_rate,
            n_estimators=n_estimators,
            **kwargs
        )

    def fit(self, ts: TSDataset):
        # превращаем TSDataset в датафрейм pandas
        # с плоским индексом
        df = ts.to_pandas(flatten=True)
        df = df.dropna()
        df = df.drop(columns='segment')
        self._base_model.fit(df=df)
        return self

    def forecast(self, ts: TSDataset):
        result_list = list()
        # собираем новый датафрейм с помощью
        # self._forecast_segment
        # из базового класса
        for segment in ts.segments:
            segment_predict = self._forecast_segment(
                self._base_model, segment, ts)
            result_list.append(segment_predict)

        result_df = pd.concat(result_list, ignore_index=True)
        result_df = result_df.set_index(['timestamp', 'segment'])

        df = ts.to_pandas(flatten=True)
        df = df.set_index(['timestamp', 'segment'])
        # заменяем пропуски прогнозами
        df = df.combine_first(result_df).reset_index()

```



```

df = TSDataset.to_dataset(df)
ts.df = df
# выполняем обратные преобразования
ts.inverse_transform()

return ts

```

Замечание

Порядок лагов не должен быть меньше длины горизонта! Потому как в противном случае, признаки тестового поднабора данных, построенные на лагах, будут использовать информацию из целевой переменной тестового поднабора данных (утечка!)

Таким образом, необходимо создавать лаговые переменные так, чтобы они не проникали в тестовый набор. Лаги вида L_{t-k} лучше создавать так, чтобы k был равен или превышал горизонт прогнозирования (рис. 9). Впрочем, допускается создание лагов, у которых порядок будет меньше длины горизонта прогнозирования, но тогда значения зависимой переменной в тестовой выборке нужно заменить на значение NaN. Если лаг и залезет в тест, ему ничего не останется, как использовать значение NaN, таким образом, в тесте появится значение NaN. В таком случае, чем больше горизонт прогнозирования будет превышать порядок лага, тем больше пропусков будет в тесте.

На практике для избежания утечки данных при вычислении лагов (а также скользящих и расширяющихся статистик) поступают двумя способами:

- о значения зависимой переменной в наблюдениях исходного набора, которые будут соответствовать будущей тестовой выборке (набору новых данных), заменяют значениями NaN,
- о берем обучающую выборку и удлиняем ее на длину горизонта прогнозирования, зависимая переменная в наблюдениях, соответствующих новым временным меткам (т.е. в тестовой выборке/наборе новых данных) получает значения NaN.

В обоих случаях мы формируем защиту от утечки при вычислении лагов в тестовой выборке / наборе новых данных.

Теперь создадим лаги, скользящее среднее, скользящее среднее абсолютное отклонение, обучим модель `LGBMModelMultiSegment`, получим прогнозы и визуализируем их

```

# создаем экземпляр класса LagTransform для генерации лагов,
# с помощью in_column задаем переменную, на основе которой
# генерируем лаги, мы будем генерировать лаги порядка от 8 до 23,
# порядок лагов не должен быть меньше длины горизонта
lags = LagTransform(in_column='target',
                    lags=list(range(8, 24, 1)),
                    out_column='lag')
# создаем экземпляр класса MeanTransform для вычисления
# скользящего среднего по заданному окну
mean8 = MeanTransform(in_column='target',
                      window=8,
                      out_column='mean8')
# создаем экземпляр класса MadTransform для вычисления
# среднего абсолютного отклонения по заданному окну
mad8 = MadTransform(in_column='target',
                    window=8,
                    out_column='mad8')
# добавляем лаги, mean8, mad8 в обучающую выборку
train_ts.fit_transform([lags, mean8, mad8])
# создаем экземпляр класса LGBMModelPerSegment

```

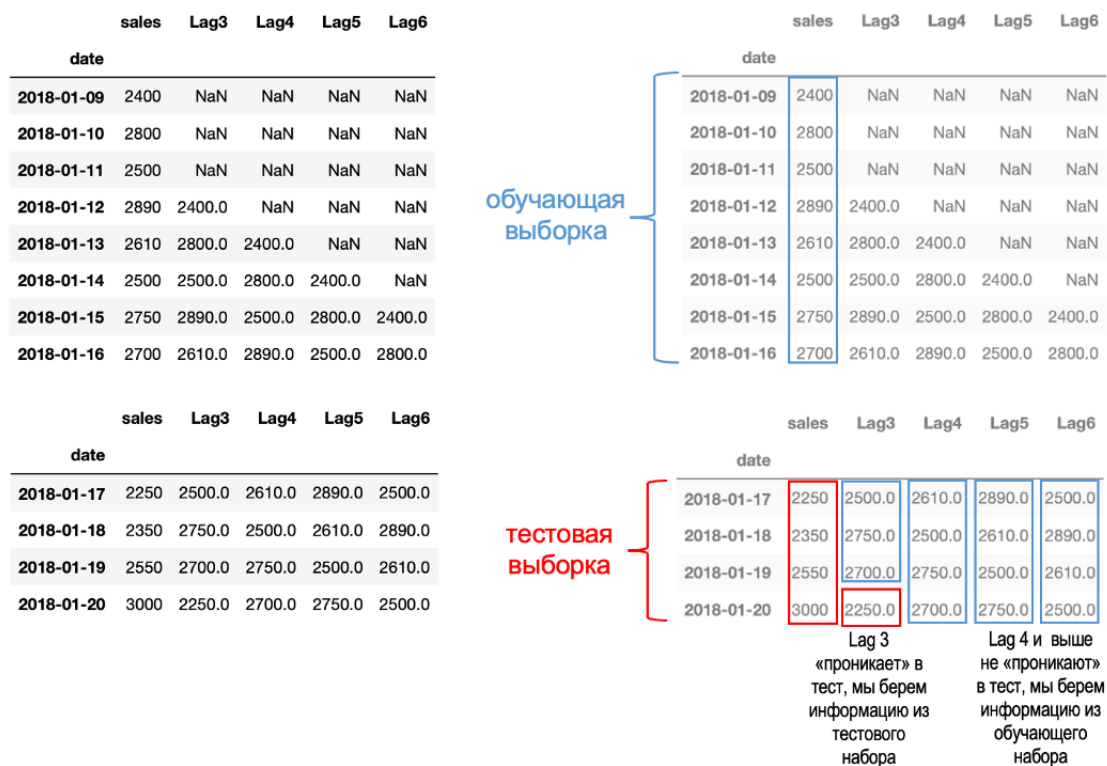


Рис. 9. Лаги, у которых порядок равен горизонту прогнозирования или превышает его, не используют тестовую выборку

```
model = LGBMModelPerSegment()
# обучаем модель
model.fit(train_ts)
# формируем тестовый набор
future_ts = train_ts.make_future(HORIZON)
# получаем прогнозы
forecast_ts = model.forecast(future_ts)
# оцениваем качество прогнозов
smape(y_true=test_ts, y_pred=forecast_ts)
```

3.4. Работа с несколькими временными рядами

Прогнозирование нескольких временных рядов. Загрузим набор, в котором каждому сегменту соответствует свой временной ряд

```
original_df = pd.read_csv("data.csv")
original_df.head()

df = TSDataset.to_dataset(original_df)
```

Вновь воспользуемся моделью CatBoost с помощью класса CatBoostModelMultiSegment. Перед построением модели выполним некоторые преобразования и создадим новые признаки для наших рядов.

Нам понадобятся следующие классы-трансформеры:

- класс `LogTransform` для логарифмирования и экспонирования перерменной (логарифмирование позволяет сгладить негативное влияние выбросов объективной природы, помогает выделить тренд),

- `LinearTrendTransform` для прогнозирования тренда, удаления тренда из данных и добавления тренда к прогнозам (это необходимо для деревьев решений и для ансамблей деревьев решений, не умеющих экстраполировать),
- `LagTransform` для генерации лагов,
- `DateFlagsTransform` для генерации признаков на основе дат – порядковый номер дня недели, порядковый номер дня месяца, порядковый номер недели в месяце и пр.,
- `MeanTransform` для вычисления скользящего среднего по заданному окну.

Сначала выполним логарифмирование зависимой переменной, а затем вычтем из нее тренд. Потом на основе пролагрифмированной зависимой переменной с удаленным трендом мы создадим лаги и скользящие средние, добавим календарные признаки.

```
# создаем экземпляр класса LogTransform для логарифмирования
# и экспоненцирования зависимой переменной
log = LogTransform(in_column='target')

# создаем экземпляр класса LinearTrendTransform
# для прогнозирования тренда, удаления тренда из
# данных и добавления тренда к прогнозам
trend = LinearTrendTransform(in_column='target')

# создаем экземпляр класса SegmentEncoderTransform
# для кодирования меток сегментов целочисленными
# значениями в лексикографическом порядке (LabelEncoding):
# сегменты a, b, c, d получают значения 0, 1, 2, 3
seg = SegmentEncoderTransform()

# создаем экземпляр класса LagTransform
# для генерации лагов (с лага 31 по лаг 95)
lags = LagTransform(in_column='target',
                    lags=list(range(31, 96, 1)),
                    out_column='lag')

# создаем экземпляр класса DateFlagsTransform для
# генерации признаков на основе дат - порядковый
# номер дня недели, порядковый номер дня месяца,
# порядковый номер недели в месяце, порядковый
# номер недели в году, порядковый номер месяца
# в году, индикатор выходных дней
d_flags = DateFlagsTransform(day_number_in_week=True,
                             day_number_in_month=True,
                             week_number_in_month=True,
                             week_number_in_year=True,
                             month_number_in_year=True,
                             special_days_in_week=[5, 6],
                             out_column='datetime')

# создаем экземпляр класса MeanTransform для вычисления
# скользящего среднего по заданному окну
mean30 = MeanTransform(in_column='target',
                       window=30,
                       out_column='mean30')
```

Разбиваем набор (наш объект `TSDataset`) на обучающую и тестовую выборки с учетом временной структуры. Здесь горизонт прогнозирования составит 31 день

```
# разбиваем набор на обучающую и тестовую выборки
# с учетом временной структуры
```

```

train_ts, test_ts = ts.train_test_split(
    train_start="2019-01-01",
    train_end="2019-11-30",
    test_start="2019-12-01",
    test_end="2019-12-31",
)

# выполняем преобразования набора
train_ts.fit_transform([
    log, # логарифмируем
    trend, # удаляем тренд
    lags, # вычисляем лаги
    d_flags, # вычисляем признаки на основе дат
    seg, # кодируем метки сегментов
    mean30 # вычисляем скользящее среднее
])

```

Задаем явно горизонт в 31 день, обучаем модель CatBoost, оцениваем качество прогнозов и визуализируем прогнозы. Кроме того, не забываем выполнить обратные преобразования (добавление тренда, экспоненцирование зависимой переменной) с помощью метода `.inverse_transform()` для обучающего набора для правильной визуализации значений зависимой переменной в обучающей выборке.

```

# задаем горизонт прогнозирования
HORIZON = 31
# создаем экземпляр класса CatBoostModelMultiSegment
model = CatBoostModelMultiSegment()
# обучаем модель CatBoost
model.fit(train_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_ts = train_ts.make_future(HORIZON)

# получаем прогнозы
forecast_ts = model.forecast(future_ts)

# оцениваем качество прогнозов
smape(y_true=test_ts, y_pred=forecast_ts)

```

Выполняем обратное преобразование для обратной выборки (добавляем тренд, делаем экспоненцирование переменной `target`)

```

train_ts.inverse_transform()
plot_forecast(forecast_ts, test_ts, train_ts, n_train_sample=20)

```

Для более надежной оценки качества модели CatBoost воспользуемся *перекрестной проверкой расширяющимся окном*

```

pipe = Pipeline(
    model=model,
    transform=[
        log,
        trend,
        seg,
        lags,
        d_flags,
        mean30,
    ],
    horizon=HORIZON,
)

```

```
)
metrics, forecast, info = pipe.backtest(ts, [smape], aggregate_metrics=True)
```

Ансамбль бустингов

```
transforms = [log, trend, seg, lags, d_flags, mean30]
catboost_pipeline = Pipeline(
    model=CatBoostModelMultiSegment(),
    transforms=transforms,
    horizon=HORIZON
)

lightgbm_pipeline = Pipeline(
    model=LGBMModelMultiSegment(),
    transforms=transforms,
    horizon=HORIZON
)

xgboost_pipeline = Pipeline(
    model=XGBModelMultiSegment(learning_rate=0.2, n_estimators=500, max_depth=1),
    transforms=transforms,
    horizon=HORIZON
)

pipeline_names = ["catboost", "lightgbm", "xgboost"]
pipelines = [catboost_pipeline, lightgbm_pipeline, xgboost_pipeline]

voting_ensemble = VotingEnsemble(
    pipelines=pipelines,
    weight=[1, 1, 2],
    n_jobs=1
)

metrics, forecast, _ = voting_ensemble.backtest(
    ts=ts, metrics=[SMAPE()], n_folds=3, aggregate_metrics=True, n_jobs=1
)
```

Заметим, что скользящее среднее используется не только для конструирования признаков, но и в качестве прогнозной модели (когда прогноз – скользящее среднее n последних наблюдений), а также для сглаживания выбросов, краткосрочных колебаний и более четкого выделения долгосрочных тенденций в ряде данных.

4. Генерация признаков и кодирование категориальных признаков

Процесс создания признакового пространства зависит от модели, которую будем использовать:

- ONE-кодирование предпочтительнее для линейных моделей,
- умное кодирование категорий – для деревьев,
- выбросы можно не удалять для робастной модели.

Если в тестовом наборе данных присутствуют категории, которых не было в обучающем наборе данных, то нужно принять решение о том, как их кодировать. Например, категорию из нового набора данных можно отнести к самой опасной категории из тех, категорий, которые присутствуют в обучающем наборе данных.

Еще категории можно кодировать по разным признакам.

Можно кодировать признаки по мощности (Count Encoding): сколько раз каждая уникальная категория встречалась в категориальном признаке. Проблема в том, что некоторые категории могут встречаться одинаковое количество раз (коллизия). Чтобы различать такие категории можно добавить шум, т.е. $count + \varepsilon$. Мелкие и новые категории объединяют в одну.

Кодирование по мощности можно использовать, если требуется быстро решить задачу.

4.1. Кодирование одного категориального признака по другому категориальному признаку с помощью сингулярного разложения

Если матрица признакового описания объекта состоит только из категориальных признаков, то можно кодировать один признак на основе другого

```
from numpy.linalg import svd

def code_factor(data, cat_feature, cat_feature2):
    """
    Кодирование признака на основе другого признака
    """
    ct = pd.crosstab(data[cat_feature], data[cat_feature2])
    u, _, _ = svd(ct.values)
    coder = dict(zip(ct.index, u[:, 0])) # берем только первый сингулярный вектор

    return data[cat_feature].map(coder)
```

Сингулярное разложение (Singular Value Decomposition, SVD) – декомпозиция вещественной матрицы с целью ее приведения к каноническому виду. Сингулярное разложение является удобным методом при работе с матрицами. Оно показывает геометрическую структуру матрицы и позволяет наглядно представить имеющиеся данные. В числе прочего SVD позволяет вычислять обратные и псевдообратные матрицы большого размера, что делает его полезным инструментом при решении задач регрессионного анализа.

Замечание

В числе прочего с помощью сингулярного разложения можно решать задачи обращения или псевдо-обращения матриц большого размера

Для любой вещественной $(n \times n)$ -матрицы A существуют две вещественные ортогональные $(n \times n)$ -матрицы U и V такие, что

$$\Lambda = U^T A V,$$

где Λ – диагональная матрица.

Матрицы U и V выбираются так, чтобы диагональные элементы матрицы Λ имели вид

$$\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_r > \lambda_{r+1} = \lambda_n = 0,$$

где r – ранг матрицы A .

В частности, если A невырождена (то есть существует обратная матрица A^{-1} , $\det A \neq 0$), то

$$\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n > 0.$$

Столбцы матриц U и V называются соответственно *левыми* и *правыми сингулярными векторами*, а значения диагонали матрицы Λ – *сингулярными числами*.

Эквивалентная запись сингулярного разложения

$$A = U\Lambda V^T$$

Например, матрица

$$A = \begin{pmatrix} 0.96 & 1.72 \\ 2.28 & 0.96 \end{pmatrix}$$

имеет сингулярное разложение

$$A = U\Lambda V^T = \begin{pmatrix} 0.6 & 0.8 \\ 0.8 & -0.6 \end{pmatrix} \begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0.8 & -0.6 \\ 0.6 & 0.8 \end{pmatrix}^T$$

Легко увидеть, что матрицы U и V ортогональны,

$$U^T U = U U^T = I, \quad V^T V = V V^T = I,$$

и сумма квадратов значений их столбцов равна единице.

Для *прямоугольных* матриц существует так называемое экономное представление сингулярного разложения

$$A_{(m \times n)} = U_{(m \times r)} \Lambda_{(r \times r)} V_{(r \times n)}^T,$$

где $r = \min(m, n)$.

Сингулярное разложение и собственные числа матрицы

Сингулярное разложение обладает свойством, которое связывает задачу отыскания сингулярного разложения и задачу отыскания собственных векторов. Собственный вектор x матрицы A – такой вектор, при котором выполняется условие $Ax = \lambda x$, где λ – собственное число.

Так как матрицы U и V ортогональные, то

$$\begin{aligned} AA^T &= U\Lambda \underbrace{V^T V}_{=I} \Lambda U^T = U\Lambda^2 U^T, \\ A^T A &= V\Lambda \underbrace{U^T U}_{=I} \Lambda V^T = V\Lambda^2 V^T. \end{aligned}$$

Умножая оба выражения справа соответственно на U и V , получаем

$$\begin{aligned} AA^T U &= U\Lambda^2, \\ A^T A V &= V\Lambda^2. \end{aligned}$$

Из этого следует, что столбцы матрицы U являются собственными векторами матрицы AA^T , а квадраты сингулярных чисел $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_r)$ – ее собственным числам. Также столбцы матрицы V являются собственными векторами матрицы AA^T , а квадраты сингулярных чисел являются ее собственными числами.

SVD и норма матриц

Евклидова норма

$$|A|_E = \max_{|x|=1} \frac{|Ax|}{|x|}.$$

Норма Фробениуса

$$|A|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n a_{ij}^2}.$$

Если известно сингулярное разложение, то обе эти нормы легко вычислить. Пусть $\lambda_1, \dots, \lambda_r$ – сингулярные числа матрицы A , отличные от нуля.

$$\text{Тогда } |A|_E = \lambda_1 \text{ и } |A|_F = \sqrt{\sum_{k=1}^r \lambda_k^2}.$$

Нахождение псевдообратной матрицы с помощью SVD

Если $(m \times n)$ -матрица A является вырожденной или прямоугольной, то обратной матрицы A^{-1} для нее не существует.

Однако, для A может быть найдена псевдообратная матрица A^+ – такая матрица, для которой выполняются условия

$$\begin{aligned} A^+ A &= I_n, \\ A A^+ &= I_m, \\ A^+ A A^+ &= A^+, \\ A A^+ A &= A. \end{aligned}$$

Пусть найдено разложение матрицы A вида

$$A = U \Lambda V^T,$$

где $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_r)$, $r = \min(m, n)$ и $U^T U = I_m$, $V V^T = I_n$.

Тогда матрица

$$A^+ = V^T \Lambda^{-1} U$$

является для матрицы A псевдообратной.

Усеченное SVD при обращении матриц

Для получения обращения, устойчивого к малым изменениям значений матрицы A , используется усеченное SVD. Пусть матрица A представлена в виде $A = U \Lambda V^T$.

Тогда усеченная псевдообратная матрица A_s^+

$$A_s^+ = V \Lambda_s^{-1} U^T,$$

где $\Lambda_s^{-1} = \text{diag}(\lambda_1^{-1}, \dots, \lambda_s^{-1}, 0, \dots, 0)$ – $(n \times n)$ -диагональная матрица, s – первые s сингулярных чисел, $s \leq \text{rang} A$.

Есть еще хэш-кодирование (`sklearn.feature_extraction.FeatureHasher`). Для быстрого анализа пойдет, но применяется редко.

Можно кодировать категории по целевой переменной (Target Encoding). Есть варианты целевого кодирования по среднему (Mean Target Encoding), по стандартному отклонению (Std Target Enconing) и т.д. Подход для *любого* алгоритма. Кодирование по значению целевой переменной «логично».

Главная проблема: неадекватная кодировка мелких категорий + слияние этих категорий. Нельзя допустить утечки значений целевой переменной!

Теоретически можно кодировать категориальные признаки на обучающем поднаборе, а обучать алгоритм на отложенной выборке, но это не очень здорово, так как теряем значительную часть данных на этапе кодирования.

Кодирование по *предыдущим* объектам (CatBoost). Одна категория в обучении кодируется по-разному, а на контроле фиксировано

```
gb = data.groupby(name)
data[name + "_cb"] = (gb["target"].cumsum() - data["target"]) / gb.cumcount()
```

Получаются более менее адекватные значения, но без подглядывания. В самом начале кодирования (в первых строках) пока статистика не наберется значения будут неадекватные. Можно тасовать матрицу признакового описания объекта, а затем усреднять результаты кодировки.

На практике хорошо работает смесь подходов!

5. Перестановочная важность признаков и важность признаков по Шепли

Полезный ресурс https://scikit-learn.org/stable/modules/permutation_importance.html

Статья Дьяконова [про интерпретацию черных ящиков](#)

Важность признаков – числовые оценки, насколько каждый признак *важен* для решения поставленной задачи.

Плохой метод – чем чаще выбирался признак, тем лучше. Дело в том, что признак может действительно часто выбираться, но на более низких уровнях дерева (дальше от корня). Другими словами, признак выбирается часто, но используется для построения небольших «уточняющих» разбиений (рис. 10). Как правило, все наоборот. Если признак выбирается часто, значит модель не может по каким-то причинам сразу получить от него нужную информацию.

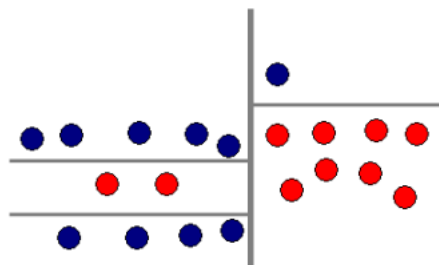


Рис. 10. К вопросу о важности признака по частоте его выбора. Признак по оси x выбирается только один раз, а признак по оси y выбирается три раза, но очевидно, что первый признак лучше справляется с задачей

Нельзя отбрасывать признаки по порогу.

Перестановочная важность признаков и важность признаков по Шепли обладают свойством *согласованности* (если модель изменить так, что она более существенно начинает зависеть от какого-то признака, то его важность не убывает).

Подход вычисления **перестановочной важности признаков** (Permutation Feature Importance):

- (+) не меняет распределение по конкретному признаку (так как рассматриваемый признак просто перемешивается),
- (+) не требует обучать модель заново – обученную модель тестируют на отложенной выборке с испорченным признаком,
- (+) можно применять на любых алгоритмах,
- (+) самый надежный метод,
- (+) в бутстрепе можно использовать ООВ-контроль (строить дерево и на экземплярах, не попавших в дерево, вычислять перестановочную важность),
- (-) очень медленный.

Замечание

Вместо метрик качества для вычисления перестановочной важности можно использовать что-то другое. Например, долю верно классифицирующих деревьев

Идея перестановочной важности признаков: признак важный, если его перетасовка снижает качество. Можно вычислять перестановочную важность признаков на *обучающем поднаборе данных* (вроде как бы можно, но лучше использовать отложенную выборку PFI-holdout), на *отложенном контроле* (тестовый поднабор данных PFI-holdout, рис. 11) и на любой схеме *валидации* (надежнее использовать валидацию).

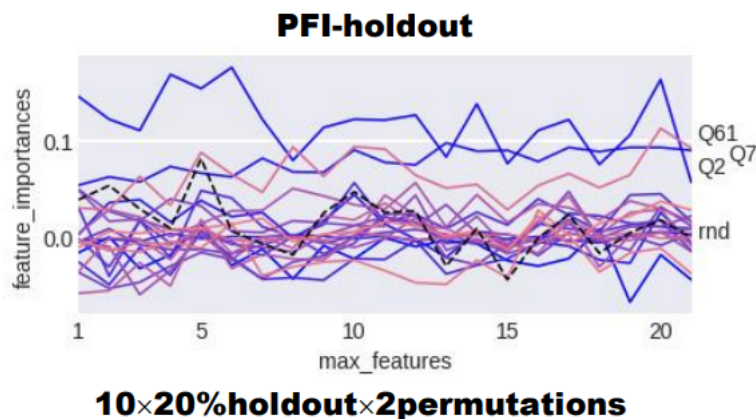


Рис. 11. Перестановочная важность признаков, вычисленная на отложенной выборке

Перестановочная важность скоррелированных признаков может размазываться между ними.

Можно удалять признаки (Drop-column importance), но тогда каждый раз нужно будет заново обучать модель. Однако при этом результат более однозначный. Используется редко!

Если два признака коррелируют друг с другом, то перестановка одного из них не будет значимо сказываться на эффективности модели, потому что она может извлечь требуемую информацию из второго коррелирующего признака. Одним из способов работы с мультиколлинеарными признаками является иерархическая кластеризация на базе ранговых корреляций Спирмена, выбор порога отсечения и сохранение одного признака из каждого кластера.

Замечание

Важности признаков, полученные с помощью `feature_importances_` (важность по неоднородности), встроенного в алгоритмы построения ансамблей деревьев – это НЕ важности признаков для решения задачи, а лишь для настройки конкретной модели. Этот подход не обладает свойством согласованности!!!

В разделе 4.2.2 Relation to impurity-based importance in trees документации sklearn говорится, что impurity-based importance (*важность признаков по неоднородности*; еще называют Gini importance) сильно смещена и отдает предпочтение высококардинальным признакам (обычно вещественным) по сравнению с низкокардинальными признаками, такими как бинарные или категориальные признаки с небольшим числом категорий. Кроме того, важность по неоднородности годится только для деревьев и их ансамблей.

Важность по Шепли i -ого признака вычисляется следующим образом

$$\varphi_i = \sum_{S \subseteq \{1, 2, \dots, n\} \setminus \{i\}} \frac{|S|!(n - |S| - 1)!}{n!} (f(S \cup \{i\}) - f(S)),$$

где $f(S)$ – ответ модели, обученной на подмножестве S множества n признаков (на конкретном объекте – вся формула записывается для конкретного объекта).

Вычисление требует переобучения модели на всевозможных подмножествах признаков, поэтому на практике применяются приближения формулы, например, с помощью метода Монте-Карло.

Замечания по методам оценки важности признаков:

- нет идеального алгоритма оценки важности признаков (для любого можно подобрать пример, когда он плохо работает),
- если много похожих признаков (например, сильно коррелированных), то важность может «делиться между ними», поэтому не рекомендуется отбрасывать признаки по порогу важности,
- есть старая рекомендация (впрочем, без теоретического обоснования): модель для решения задачи и оценки важности должны основываться на разных парадигмах (например, оценивать важность с помощью случайного леса и потом настраивать его же на важных признаках не рекомендуется). **То есть не рекомендуется оценивать важность и решать ML-задачу одним и тем же алгоритмом!!!**

Советы Дьяконова:

- можно выбрать некоторое подмножество признаков и потом то добавить k признаков, то отнять l ,
- оценивать важность не обязательно с помощью лучшей модели (то есть не нужно строить супермодель для того, чтобы оценить важность признаков!)
- перестановочная важность самая естественная (но есть нюансы: коррелированность признаков, стабильность оценки и т.п.),
- есть и другие подходы и удобные библиотеки (SHAP, например).

6. Приемы работы с библиотекой Polars

6.1. Установка

Установить библиотеку Polars <https://www.pola.rs/> можно с помощью менеджера пакетов pip

```
pip install polars
```

С библиотекой Polars удобнее работать, используя оболочку ptpython.

6.2. Вводные замечания

Polars поддерживает режимы *жадных* (eager)³ и *отложенных* (lazy) вычислений.

Пример

```
import polars as pl

df = pl.read_csv("https://j.mp/iris.csv")
print(
    df.filter(pl.col("sepal_length") > 5)
      .groupby("species")
      .agg(pl.all().sum())
)
```

Или в «отложенном» режиме

```
import polars as pl

print(
    pl.read_csv("https://j.mp/iris.csv")
      .lazy() # набор данных сразу не загружается!
      .filter(pl.col("sepal_length") > 5)
      .groupby("species")
      .agg(pl.all().sum())
      .collect()
)
```

На Pandas этот запрос выглядел бы так

```
import numpy as np
import pandas as pd

df = pd.read_csv("https://j.mp/iris.csv")
df.loc[df.loc[:, "sepal_length"] > 5].groupby("species").agg(np.sum)
```

Если набор данных храниться локально, то можно воспользоваться семейством функций scan_**

```
pl.scan_csv("./iris.csv").\
    select(["sepal_width", "petal_width"]).\
    filter(pl.col("sepal_width") > 3.5).\
    collect()
```

Lazy API создает план выполнения запроса. Никакой шаг пайплайна не выполняется пока мы явно попросим Polars выполнить запрос (с помощью LazyFrame.collect() или LazyFrame.fetch()). Этот прием дает возможность Polars проанализировать весь запрос и выбрать наиболее подходящий по контексту алгоритм вычислений.

³В этом режиме работа с данными выглядит как в pandas

Чтобы перейти от жадного режима к ленивому достаточно просто начать запрос с `.lazy()`, а закончить `.collect()`.

6.3. Polars-выражения

Метод `filter` отбирает строки, а метод `select` – столбцы.

Polars поддерживает очень мощную концепцию выражений. Рассмотрим несколько примеров

```
out = df.select([
    pl.col("name").n_unique().alias("unique_names_1"),
    pl.col("name").unique().count().alias("unique_names_2")
])
```

Можно вызывать различные агрегаторы на объекте солбца

```
df.select(pl.col("name").std())
df.select(pl.col("name").max())
```

Можно выбирать столбцы по регулярному выражению

```
df.select(r"~Type\s{1}\d{1}$") # выберет столбцы 'Type 1', 'Type 2' etc.
```

Можно использовать фильтры и условия. В фильтр передается булева маска

```
df.select([
    pl.col("species").filter(
        pl.col("species").str.contains(r"*.osa$")
    ).count() # count() -- это метод объекта столбца
]) # 50
```

На Pandas этот запрос выглядел бы так

```
df.loc[
    df.loc[:, "species"].str.contains(r"*.osa$") # вернет булеву маску
][ "species"].count() # 50

# или так
df.loc[
    df.loc[:, "species"].str.contains(r"*.osa$") # вернет булеву маску
].shape[0] # 50
```

Можно использовать конструкцию `when → then → otherwise`

```
df.select([
    pl.when(pl.col("petal_length") > 3)
    .then(100)
    .otherwise(pl.col("sepal_width") * 10) * pl.col("sepal_length").sum()
])
```

На Pandas запрос выглядел бы так

```
np.where(
    df.loc[:, "petal_length"] > 3,
    100,
    df.loc[:, "sepal_width"] * 10
) * df.loc[:, "sepal_length"].sum()
```

Еще пример

```
# Polars
(
```

```

df
.lazy()
.groupby("species")
.agg(pl.all().sum())
.select(["species", "petal_length"])
.collect()
)

# Pandas
(
    df
    .groupby("species")
    .agg(np.sum)[["species", "petal_length"]]
)

```

Пример группировки с последующим построением агрегата

```

df.groupby("species").agg([
    pl.col("sepal_width").mean(),
    pl.col("petal_length").max(),
]).sort("species")

```

На Pandas

```

df.groupby("species").agg({
    "sepal_width": np.mean,
    "petal_length": np.max,
}).sort_index()

```

Если нужны склеить поэлементно строковые столбцы, то лучше воспользоваться функцией `pl.concat_str(...)`, а не `fold`-выражением, так как из-за материализации промежуточных столбцов у `fold`-выражений будет квадратичная временная сложность

```

df.select([
    pl.concat_str(["a", "b"]) # "a" и "b" это имена столбцов
])

```

Polars-выражения нельзя использовать везде. Вот допустимые *контексты*:

- `df.select(...)`
- `df.groupby(...).agg(...)`
- `df.with_columns(...)`

Даже в тех случаях, когда мы явно работаем в «жадном» режиме, на самом деле используется режим отложенных вычислений

```

df.groupby("foo").agg([pl.col("bar").sum()])
# на самом деле
(df.lazy().groupby("foo").agg([pl.col("bar").sum()])).collect()

```

Добавить столбец (или несколько столбцов) можно так

```

df.with_columns([
    (pl.col("sepal_length").sum() + pl.col("petal_width")).alias("new_col_1"),
    pl.col("sepal_length").mean().alias("new_col_2")
])

```

Создать производный столбец с кастомной функцией можно так

```

df.with_columns([
    pl.col("Speed").map(f=lambda elem: elem ** 2).alias("SquaredSpeed")
])

```

Если псевдоним не задать, то метод `with_columns` перезапишет существующий столбец "Speed". Но `with_columns` возвращает новый объект.

Замечание

Пользовательские Python-функции убивают распараллеливание!

Можно также фильтровать группы. Пусть требуется вычислить среднее по группе, но не для всех элементов группы. Для ясности можно использовать Python-функции. Они ничего не стоят, потому как применяются только в Polars-выражении, а на шаге выполнения запроса

```
from datetime import date

import polars as pl

from .dataset import dataset

def compute_age() -> pl.Expr:
    return date(2021, 1, 1).year - pl.col("birthday").dt.year()

# здесь мы вычисляем временную дельту для столбца 'birthday',
# а фильтруем по ассоциированным строкам столбца 'gender'
def avg_birthday(gender: str) -> pl.Expr:
    return compute_age().filter(pl.col("gender") == gender).mean().alias(f"avg {gender} birthday")

q = (
    dataset.lazy()
    .groupby(["state"])
    .agg(
        [
            avg_birthday("M"),
            avg_birthday("F"),
            (pl.col("gender") == "M").sum().alias("# male"),
            (pl.col("gender") == "F").sum().alias("# female"),
        ]
    )
    .limit(5)
)

df = q.collect()
```

С помощью метода `filter` мы отбираем строки в столбце 'gender', а затем по оставшимся строкам столбца 'gender' вычисляем среднее и связываем результат с псевдонимом.

Еще вместо группировки можно использовать фильтр по одному столбцу и агрегацию по другому столбцу

```
df.select([
    pl.col("sepal_width").filter(pl.col("species") == "setosa").mean()
]) # 3.418
```

На Pandas было бы так

```
df.loc[df.loc[:, "species"] == "setosa", "sepal_width"].mean()
```

6.4. Оконные функции

Оконную функцию из обычной можно сделать, вызвав метод `.over()`.

Пример

```
df.select(["Type 1", "Type 2", "Attack", "Defense"]).select([
    "Type 1",
    "Type 2",
    pl.col("Attack").mean().over("Type 1").alias("avg_attack_by_type"),
    pl.col("Defense").mean().over(["Type 1", "Type 2"]).alias("avg_defense_by_type_comb"),
])
```

Добавить столбец "WinCumsum" с кумулятивной суммой по группе

```
df.select([
    "Type 1",
    "Speed",
    pl.col("Speed").cumsum().over("Type 1").alias("WinCumsum"),
]).sort("Type 1")
```

Еще пример оконной функции. NB: столбцы, которые используются в `.over(...)` должны быть отсортированы

```
df.sort("Type 1").select([
    pl.col("Type 1").head(3).list().over("Type 1").flatten(),
    pl.col("Name").sort_by("Attack").head(3).list().over("Type 1").flatten().alias("fastest/group"),
    pl.col("Name").sort_by(pl.col("Speed")).head(3).list().over("Type 1").flatten().alias("strongest/group"),
])
```

6.5. Универсальные NumPy-функции

Polars поддерживает универсальные numpy-функции `ufuncs` <https://numpy.org/doc/stable/reference/ufuncs.html#available-ufuncs>.

Пример

```
df.select([
    np.log(pl.all()).suffix("_log") # pl.all() -- применяется ко всем столбцам
])
```

6.6. Примеры

Столбцы можно выбирать по регулярному выражению

```
df.select([
    pl.col(r"^Type\s{1}\d{1}$") # Polars-выражение
])

# или так
df.select(r"^Type\s{1}\d{1}$")

df.select([
    pl.col(r"^Sp.*$").sum()
])
```

С помощью конструкции `pl.all()` можно выбирать все столбцы в таблице (работает также как `*` в SQL-запросах)


```
df.select([
    pl.all().reverse().suffix("_rev") # выбрать все столбцы в обратном порядке
])

df.select([
    pl.all().sum().suffix("_sum") # найти сумму по всем столбцам
])
```

Polars-выражения можно связывать с переменными

```
mask = pl.col("Type 1").str.contains(r"~Gr.*$")

df.select([
    mask.sum(), # найти количество элементов, удовлетворяющих шаблону
])

df.select([
    pl.col("Speed").filter(mask).sum(),
])
```

Пример с when-then-otherwise

```
df.select([
    "Type 1",
    pl.when(pl.col("Type 1") == "Grass")
    .then(pl.col("Type 2").str.to_uppercase())
    .otherwise(pl.col("Name").str.to_lowercase())
])
```

Пример с оконными функциями

```
df.select([
    "fruits",
    "cars",
    "B",
    pl.col("B").sum().over("fruits").alias("B_sum_by_fruits"),
    pl.col("B").sum().over("cars").alias("B_sum_by_cars"),
])
```

Сумма вычисляется по группе и транслируется на все элементы группы.

7. Приемы работы с библиотекой Catboost

Онлайн документация пакета https://catboost.ai/en/docs/concepts/python-reference_catboostregre

7.1. Установка CatBoost

Установить пакет можно с помощью менеджера conda (или с помощью pip)

```
$ conda config --show channels
# если канала conda-forge нет в списке, то следует его добавить
$ conda config --add channels conda-forge
$ conda install catboost
$ pip install catboost
```

7.2. Ключевые особенности пакета

7.3. Параметры

Замечание

Помимо `iterations` и `learning_rate` у CatBoost 5 важнейших гиперпараметров: `max_depth`, `l2_leaf_reg`, `border_count`, `random_strength` и `bagging_temperature`

Применительно к `random_strength` замечено, что часто значение, близкое к 0 (примерно, 0.15), дает лучшее качество. Если переменных много, можно попробовать настраивать `rsm`.

В случае дисбаланса классов будет полезным настраивать

- либо гиперпараметр `auto_class_weights`,
- либо гиперпараметры `class_weights` и `scale_pos_weight`

при этом не нужно настраивать эти параметры одновременно.

Можно сначала при *небольшом* числе итераций найти оптимальные значения гиперпараметров, *меньше* всего зависящие от количества итераций (речь прежде всего идет об `auto_class_weights`, `max_depth`; при этом `learning_rate` и `rsm` зависят от количества итераций, поэтому нам для небольшого количества итераций придется увеличить темп обучения, а варьирование `rsm` сделать минимальным). Затем можно построить модель с найденными оптимальными значениями гиперпараметров `auto_class_weights`, `max_depth` и `rsm`, но уже с большим количеством деревьев, при этом, разумеется, помня о двух вещах:

- при большом количестве итераций нужно уменьшить темп обучения,
- выбирая меньшее значение `rsm`, нужно задавать больше итераций.

Ознакомится с описанием параметров можно здесь <https://catboost.ai/en/docs/references/training-parameters/>

Общие параметры:

- `loss_function` (objective) – функция потерь, которая используется на шаге обучения модели.
- `iterations` – максимальное число деревьев в ансамбле,
- `learning_rate` – темп обучения,
- `l2_leaf_reg` – коэффициент при члене L_2 -регуляризации,
- `bagging_temperature` – задает настройки Байесовского бутстрапа

7.4. Классификатор CatBoostClassifier

Класс CatBoostClassifier

```
class CatBoostClassifier(  
    iterations=None,  
    learning_rate=None,  
    depth=None,  
    l2_leaf_reg=None,  
    model_size_reg=None,  
    rsm=None,  
    loss_function=None,  
    border_count=None,  
    feature_border_type=None,  
    per_float_feature_quantization=None,  
    input_borders=None,  
    output_borders=None,  
    fold_permutation_block=None,
```

```
od_pval=None,
od_wait=None,
od_type=None,
nan_mode=None,
counter_calc_method=None,
leaf_estimation_iterations=None,
leaf_estimation_method=None,
thread_count=None,
random_seed=None,
use_best_model=None,
verbose=None,
logging_level=None,
metric_period=None,
ctr_leaf_count_limit=None,
store_all_simple_ctr=None,
max_ctr_complexity=None,
has_time=None,
allow_const_label=None,
classes_count=None,
class_weights=None,
one_hot_max_size=None,
random_strength=None,
name=None,
ignored_features=None,
train_dir=None,
custom_loss=None,
custom_metric=None,
eval_metric=None,
bagging_temperature=None,
save_snapshot=None,
snapshot_file=None,
snapshot_interval=None,
fold_len_multiplier=None,
used_ram_limit=None,
gpu_ram_part=None,
allow_writing_files=None,
final_ctr_computation_mode=None,
approx_on_full_history=None,
boosting_type=None,
simple_ctr=None,
combinations_ctr=None,
per_feature_ctr=None,
task_type=None,
device_config=None,
devices=None,
bootstrap_type=None,
subsample=None,
sampling_unit=None,
dev_score_calc_obj_block_size=None,
max_depth=None,
n_estimators=None,
num_boost_round=None,
num_trees=None,
colsample_bylevel=None,
random_state=None,
reg_lambda=None,
objective=None,
eta=None,
max_bin=None,
scale_pos_weight=None,
```

```

gpu_cat_features_storage=None,
data_partition=None
metadata=None,
early_stopping_rounds=None,
cat_features=None,
grow_policy=None,
min_data_in_leaf=None,
min_child_samples=None,
max_leaves=None,
num_leaves=None,
score_function=None,
leaf_estimation_backtracking=None,
ctr_history_unit=None,
monotone_constraints=None,
feature_weights=None,
penalties_coefficient=None,
first_feature_use_penalties=None,
model_shrink_rate=None,
model_shrink_mode=None,
langevin=None,
diffusion_temperature=None,
posterior_sampling=None,
boost_from_average=None,
text_features=None,
tokenizers=None,
dictionaries=None,
feature_calcers=None,
text_processing=None
)

```

LogLoss применяется для задач бинарной классификации (когда целевой вектор содержит только два уникальных значения или когда параметр `target_border is not None`).

MultiClass используется в задачах мультиклассовой классификации (когда целевой вектор содержит более 2 уникальных значений или параметр `border_count is None`)

7.5. Перепеппер CatBoostRegressor

Помимо `iterations` и `learning_rate` у CatBoost 5 важнейших гиперпараметров:

- `max_depth`: макс,
- `l2_leaf_reg`,
- `border_count`,
- `random_strength`,
- `bagging_temperature`.

7.6. Функции потерь и метрики качества

7.6.1. Для классификации

Для мультиклассификации <https://catboost.ai/en/docs/concepts/loss-functions-multiclassificat>
Функции потерь

- LogLoss

$$-\frac{\sum_{i=1}^N w_i (c_i \log p_i + (1 - c_i) \log(1 - p_i))}{\sum_{i=1}^N w_i},$$

- CrossEntropy

$$-\frac{\sum_{i=1}^N w_i (t_i \log p_i + (1 - t_i) \log(1 - p_i))}{\sum_{i=1}^N w_i},$$

Метрики качества

- Precision (точность),
- Recall (полнота),
- F1 (гармоническое среднее),
- BalancedAccuracy

$$\frac{1}{2} \left(\frac{TP}{T} + \frac{TN}{N} \right),$$

- BalancedErrorRate

$$\frac{1}{2} \left(\frac{FP}{TN + FP} + \frac{FN}{FN + TP} \right),$$

- AUC,
- BrierScore,
- HingeLoss,
- HammingLoss

$$\frac{\sum_{i=1}^N w_i [[p_i > 0.5] == t_i]}{\sum_{i=1}^N w_i},$$

- Кappa

$$RAccuracy = \frac{(TN + FP)(TN + FN) + (FN + TP)(FP + TP)}{\left(\sum_{i=1}^N w_i \right)^2} \left(1 - \frac{1 - Accuracy}{1 - RAccuracy} \right),$$

- LogLikelihoodOfPrediction.

7.6.2. Для регрессии

Метрики качества, которые могут играть роль функции потерь

- MultiRMSE (в случае мультирегрессии)

$$\left(\frac{\sum_{i=1}^N \sum_{d=1}^{dim} (a_{i,d} - t_{i,d})^2 w_i}{\sum_{i=1}^N w_i} \right)^{1/2}$$

- MAE

$$\frac{\sum_{i=1}^N w_i |a_i - t_i|}{\sum_{i=1}^N w_i},$$

- MAPE

$$\frac{\sum_{i=1}^N w_i \frac{|a_i - t_i|}{\max(1, |t_i|)}}{\sum_{i=1}^N w_i}$$

- Poisson

$$\frac{\sum_{i=1}^N w_i (e^{a_i} - a_i t_i)}{\sum_{i=1}^N w_i},$$

- Quantile (большие значения α сильнее штрафуют за заниженные прогнозы)

$$\frac{\sum_{i=1}^N \left(\alpha - 1[t_i \leq a_i] \right) (t_i - a_i) w_i}{\sum_{i=1}^N w_i},$$

- RMSE

$$\left(\frac{\sum_{i=1}^N (a_i - t_i)^2 w_i}{\sum_{i=1}^N w_i} \right)^{1/2}$$

- LogLinQuantile,
- Lq

$$\frac{\sum_{i=1}^N |a_i - t_i|^q w_i}{\sum_{i=1}^N w_i}$$

- Huber

$$L(t, a) = \sum_{i=0}^N l(t_i, a_i) \cdot w_i, \quad l(t, a) = \begin{cases} \frac{1}{2}(t - a)^2, & |t - a| \leq \delta, \\ \delta|t - a| - \frac{1}{2}\delta^2, & |t - a| > \delta. \end{cases}$$

- Expectile

$$\frac{\sum_{i=1}^N |\alpha - 1[t_i \leq a_i]|(t_i - a_i)^2 w_i}{\sum_{i=1}^N w_i}$$

- Tweedie

$$\frac{\sum_{i=1}^N \left(\frac{e^{a_i(2-\lambda)}}{2-\lambda} - t_i \frac{e^{a_i(1-\lambda)}}{1-\lambda} \right) w_i}{\sum_{i=1}^N w_i},$$

где λ – значение параметра дисперсии мощности,

Метрики качества

- SMAPE

$$\frac{100 \sum_{i=1}^N \frac{w_i |a_i - t_i|}{(|t_i| + |a_i|)/2}}{\sum_{i=1}^N w_i}$$

- R2 (коэффициент детерминации)

$$1 - \frac{\sum_{i=1}^N w_i (a_i - t_i)^2}{\sum_{i=1}^N w_i (\bar{t} - t_i)^2}.$$

- MSLE (среднеквадратическая логарифмическая ошибка)

$$\frac{\sum_{i=1}^N w_i (\ln(1 + t_i) - \ln(1 + a_i))^2}{\sum_{i=1}^N w_i}$$

- MedianAbsoluteError

$$\text{median}(|t_1 - a_1|, \dots, |t_N - a_N|)$$

8. Приемы работы с решателем SCIP

Полезный ресурс https://www.gams.com/37/docs/S SCIP.html#INDEX SCIP_2d__21_solver

Пример конфигурационного файла настроек решателя (если файл `scip.set` лежит в директории, из-под которой запускается сеанс SCIP, то этот файл будет использован для настройки сеанса)

`scip.set`

```
propagating/probing/maxprerounds = 0
separating/maxrounds = 0
separating/maxroundsroot = 0
```

8.1. Общие сведения

Задачи линейного программирования в частично-целочисленной линейной постановке относятся к классу NP-трудных. Это означает, что мы знаем ни одного алгоритма, способного решать задачи такого типа за время, которое масштабируется как полином от длины входа (от размера задачи).

Выбор правила выбора переменной играет фундаментальную роль в разработке алгоритмов и оказывает значительное влияние на время решения.

8.2. Emphasis Settings

SCIP поддерживает следующие настройки выразительности (emphasis):

- `set emphasis easyscip`: для простых проблем,
- `set emphasis feasibility`: для поиска осуществимого решения,
- `set emphasis hardlp`: для решения LP-трудных задач,
- `set emphasis optimality`: для доказательства оптимальности,
- `set emphasis numerics`: для повышения численной устойчивости,
- `set heuristics emphasis aggressive`: для агрессивного применения первичных эвристик,
- `set heuristics emphasis fast`: будут использованы только быстрые эвристики,
- `set heuristics emphasis off`: отключает все первичные эвристики,
- `set presolving emphasis aggressive`: для агрессивной предобработки,
- `set presolving emphasis fast`: будут использованы только быстрые шаги предобработки,
- `set presolving emphasis off`: отключает предобработку,
- `set separaing emphasis aggressive`: для агрессивного использования разделителей,
- `set separating emphasis fast`: будут использованы только быстрые разделители,
- `set separating emphasis off`: отключает все разделители.

8.3. Проблемы «using pseudo solution instead»

<https://www.scipopt.org/doc/html/CONS.php>

Обратный вызов `CONSENFOPS` аналогичен обратному вызову `CONSENFOLP`, но имеет дело с пседорешениями вместо LP-решений. Если LP-задача не была решена в текущей подзадаче (либо потому, что пользователь не захотел ее решать, либо потому, что возникли трудности в процессе решения LP-задачи), LP-решение не доступно. В этом случае используется пседо-решение. В этом решении переменные устанавливаются на локальную границу, которая является наилучшей по отношению к целевой функции. О пседо-решении можно думать как о релаксированном решении со всеми ограничениями за исключением удаляемых границ.

9. Приемы работы с библиотеками Gym и Ecole

9.1. Gym

Функция окружения (environment) `step` возвращает четыре значения:

- **observation** (object): это объект, специфичный для окружающей среды и представляющий результат наблюдения за этой средой (например, состояние доски в настольной игре),
- **reward** (float): вознаграждение, полученное за предыдущее действие. Масштаб варьируется в зависимости от среды, но цель всегда в том, чтобы сделать суммарное вознаграждение как можно больше,
- **done** (boolean): флаг завершения эпизода. Многие (но не все) задачи разделены на четко определенные эпизоды, и `done = True` указывает на то, что эпизод завершился (например, мы потеряли последнюю жизнь в игре),
- **info** (dict): диагностическая информация, полезная для отладки.

Это просто реализация классического цикла «агент – среда». На каждом шаге агент совершает то или иное действие и среда возвращает наблюдения (observation) и вознаграждение (reward).

Процесс запускается вызовом функции `reset()`, которая возвращает первое приближение observation.

```
import gym
env = gym.make('CartPole-v0')
for i_episode in range(20):
    observation = env.reset()
    for t in range(100):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
env.close()
```

В этом примере мы отбирали случайные действия из пространства действий среды. Каждая среда поставляется с атрибутами `action_space` и `observation_space`. Эти атрибуты имеют тип `Space` и описывают формат допустимых действий и наблюдений

```
import gym

env = gym.make("CartPole-v0")
print(env.action_space) # Discrete(2)

print(env.observation_space) # Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38], (4,), float32)
```

Пространство `Discrete` описывает фиксированный диапазон неотрицательных чисел, так что в данном случае допустимыми действиями будет 0 или 1. Пространство `Box` представляет n -мерный ящик, так что в данном случае допустимыми наблюдениями будут 4-мерные массивы.

9.2. Ecole

Полезный ресурс о специальных приемах работы с задачами линейного программирования в частично-целочисленной постановке https://www.gams.com/37/docs/UG_LanguageFeatures.html?search=sos1

9.2.1. Общие сведения

Обучение с подкреплением это уникальная парадигма машинного обучения в силу особенностей формулировки цели на базе системы вознаграждения.

Гипотеза о вознаграждении: цель агента – максимизировать накопленную сумму вознаграждений.

На каждом шаге агент выбирает действие, базируясь на политике $\pi(s)$. Эта политика может быть детерминированной и в этом случае она становится просто отображением состояний на действия.

Однако, как правило политика стохастическая, что означает, что каждому состоянию назначается функция распределения вероятностей по пространству действий

$$\pi(a|s) := \mathbb{P}(A_t = a | S_t = s).$$

Цель агента состоит в том, чтобы максимизировать накопленное вознаграждение

$$G_t := \sum_{k=t+1}^T \gamma^{k-t-1} R_k,$$

где $\gamma \in [0, 1]$ – коэффициент дисконтирования.

Таким образом, γ определяет компромисс между немедленным и будущим вознаграждением. Это формализует идею о том, что действия влияют на последующие состояния и, следовательно, имеют последствия, выходящие за рамки мгновенного вознаграждения.

Линейное программирование в частично-целочисленной постановке (Mixed Integer Linear Program, MILP) это проблема линейного программирования, когда и ограничения и целевая функция линейные, а некоторые переменные могут принимать целочисленные значения.

Задачи MILP относятся к классу NP-трудных. На практике это означает, что на текущий момент не известно алгоритмов, способных решить задачу за полиномиальное время.

Секующие плоскости были одним из первых предложенных решений для борьбы с MILP. Ключевая идея состоит в том, чтобы начать с естественной линейной релаксации допустимых множеств и постепенно ужесточать эту релаксацию, отбрасывая области, которые не являются частью выпуклой оболочки. Этот подход сводит MILP к последовательному решению задач линейного программирования.

9.2.2. Observations

Класс `ecole.observation.NodeBipartiteObs`: двудольный граф наблюдений для узлов branch-and-bound дерева. Оптимизационная задача представляется в виде гетерогенного двудольного графа. Между переменной и ограничением будет существовать ребро, если переменная присутствует в ограничении с ненулевым коэффициентом.

Метод `reset()` в `Ecole` принимает в качестве аргумента экземпляр проблемы.

9.2.3. Анализ примера работы связки Ecole+GNN

В рассматриваемом примере проводится анализ упрощенной реализации решения Gasse et al. (2019). Мы будем использовать генератор экземпляров, предоставленный Ecole

```
instances = ecole.instance.SetCoverGenerator(n_rows=500, n_cols=1000, density=0.05)
```

В схеме «исследование, а затем сильное ветвление», чтобы разнообразить состояния, в которых мы собираем примеры сильного ветвления, мы в основном следуем за слабым, но дешевым экспертом (pseudocost branching) и лишь изредка вызываем сильного эксперта (strong branching). Это также гарантирует, что выборки будут ближе к тому, чтобы быть независимыми и одинаково распределенными.

Это может быть реализовано в Ecole путем создания пользовательской функции наблюдения (observation function), которая будет случайным образом вычислять и возвращать оценки псевдооценки (дешево) или оценки сильного ветвления (дорого)

```
class ExploreThenStrongBranch:
    """
    This custom observation function class will randomly return either strong branching scores (
    expensive expert)
    or pseudocost scores (weak expert for exploration) when called at every node.
    """

    def __init__(self, expert_probability):
        self.expert_probability = expert_probability
        self.pseudocosts_function = ecole.observation.Pseudocosts()
        self.strong_branching_function = ecole.observation.StrongBranchingScores()

    def before_reset(self, model):
        """
        This function will be called at initialization of the environment (before dynamics are reset
        ).
        """
        self.pseudocosts_function.before_reset(model)
        self.strong_branching_function.before_reset(model)

    def extract(self, model, done):
        """
        Should we return strong branching or pseudocost scores at time node?
        """
        probabilities = [1 - self.expert_probability, self.expert_probability]
        expert_chosen = bool(np.random.choice(np.arange(2), p=probabilities))
        if expert_chosen:
            return (self.strong_branching_function.extract(model, done), True)
        else:
            return (self.pseudocosts_function.extract(model, done), False)
```

```
# We can pass custom SCIP parameters easily
scip_parameters = {
    "separating/maxrounds": 0,
    "presolving/maxrestarts": 0,
    "limits/time": 3600,
}

# Note how we can tuple observation functions to return complex state information
env = ecole.environment.Branching(
    observation_function=(
        ExploreThenStrongBranch(expert_probability=0.05),
```

```

    ecole.observation.NodeBipartite(),
),
scip_params=scip_parameters,
)

# This will seed the environment for reproducibility
env.seed(0)

```

```

episode_counter, sample_counter = 0, 0
Path("samples/").mkdir(exist_ok=True)

# We will solve problems (run episodes) until we have saved enough samples
while sample_counter < DATA_MAX_SAMPLES:
    episode_counter += 1

    observation, action_set, _, done, _ = env.reset(next.instances))
    while not done:
        (scores, scores_are_expert), node_observation = observation
        action = action_set[scores[action_set].argmax()]

        # Only save samples if they are coming from the expert (strong branching)
        if scores_are_expert and (sample_counter < DATA_MAX_SAMPLES):
            sample_counter += 1
            data = [node_observation, action, action_set, scores]
            filename = f"samples/sample_{sample_counter}.pkl"

            with gzip.open(filename, "wb") as f:
                pickle.dump(data, f)

        observation, action_set, _, done, _ = env.step(action)

    print(f"Episode {episode_counter}, {sample_counter} samples collected so far")

```

Обучение графовой нейронной сети

```

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

```

```

class BipartiteNodeData(torch_geometric.data.Data):
    """
    This class encode a node bipartite graph observation as returned by the 'ecole.observation.
    NodeBipartite'
    observation function in a format understood by the pytorch geometric data handlers.
    """

    def __init__(
        self,
        constraint_features,
        edge_indices,
        edge_features,
        variable_features,
        candidates,
        nb_candidates,
        candidate_choice,
        candidate_scores,
    ):
        super().__init__()
        self.constraint_features = constraint_features
        self.edge_index = edge_indices
        self.edge_attr = edge_features
        self.variable_features = variable_features

```

```

self.candidates = candidates
self.nb_candidates = nb_candidates
self.candidate_choices = candidate_choice
self.candidate_scores = candidate_scores

def __inc__(self, key, value, store, *args, **kwargs):
    """
    We overload the pytorch geometric method that tells how to increment indices when
    concatenating graphs
    for those entries (edge index, candidates) for which this is not obvious.
    """
    if key == "edge_index":
        return torch.tensor(
            [[self.constraint_features.size(0)], [self.variable_features.size(0)]]
        )
    elif key == "candidates":
        return self.variable_features.size(0)
    else:
        return super().__inc__(key, value, *args, **kwargs)

class GraphDataset(torch_geometric.data.Dataset):
    """
    This class encodes a collection of graphs, as well as a method to load such graphs from the
    disk.
    It can be used in turn by the data loaders provided by pytorch geometric.
    """

    def __init__(self, sample_files):
        super().__init__(root=None, transform=None, pre_transform=None)
        self.sample_files = sample_files

    def len(self):
        return len(self.sample_files)

    def get(self, index):
        """
        This method loads a node bipartite graph observation as saved on the disk during data
        collection.
        """
        with gzip.open(self.sample_files[index], "rb") as f:
            sample = pickle.load(f)

        sample_observation, sample_action, sample_action_set, sample_scores = sample

        constraint_features = sample_observation.row_features
        edge_indices = sample_observation.edge_features.indices.astype(np.int32)
        edge_features = np.expand_dims(sample_observation.edge_features.values, axis=-1)
        variable_features = sample_observation.column_features

        # We note on which variables we were allowed to branch, the scores as well as the choice
        # taken by strong branching (relative to the candidates)
        candidates = np.array(sample_action_set, dtype=np.int32)
        candidate_scores = np.array([sample_scores[j] for j in candidates])
        candidate_choice = np.where(candidates == sample_action)[0][0]

        graph = BipartiteNodeData(
            torch.FloatTensor(constraint_features),
            torch.LongTensor(edge_indices),
            torch.FloatTensor(edge_features),

```

```

        torch.FloatTensor(variable_features),
        torch.LongTensor(candidates),
        len(candidates),
        torch.LongTensor([candidate_choice]),
        torch.FloatTensor(candidate_scores)
    )

    # We must tell pytorch geometric how many nodes there are, for indexing purposes
    graph.num_nodes = constraint_features.shape[0] + variable_features.shape[0]

    return graph

```

Архитектура нейронной сети

```

class GNNPolicy(torch.nn.Module):
    def __init__(self):
        super().__init__()
        emb_size = 64
        cons_nfeats = 5
        edge_nfeats = 1
        var_nfeats = 19

        # CONSTRAINT EMBEDDING
        self.cons_embedding = torch.nn.Sequential(
            torch.nn.LayerNorm(cons_nfeats),
            torch.nn.Linear(cons_nfeats, emb_size),
            torch.nn.ReLU(),
            torch.nn.Linear(emb_size, emb_size),
            torch.nn.ReLU(),
        )

        # EDGE EMBEDDING
        self.edge_embedding = torch.nn.Sequential(
            torch.nn.LayerNorm(edge_nfeats),
        )

        # VARIABLE EMBEDDING
        self.var_embedding = torch.nn.Sequential(
            torch.nn.LayerNorm(var_nfeats),
            torch.nn.Linear(var_nfeats, emb_size),
            torch.nn.ReLU(),
            torch.nn.Linear(emb_size, emb_size),
            torch.nn.ReLU(),
        )

        self.conv_v_to_c = BipartiteGraphConvolution()
        self.conv_c_to_v = BipartiteGraphConvolution()

        self.output_module = torch.nn.Sequential(
            torch.nn.Linear(emb_size, emb_size),
            torch.nn.ReLU(),
            torch.nn.Linear(emb_size, 1, bias=False),
        )

    def forward(
        self, constraint_features, edge_indices, edge_features, variable_features
    ):
        reversed_edge_indices = torch.stack([edge_indices[1], edge_indices[0]], dim=0)

        # First step: linear embedding layers to a common dimension (64)

```

```

    constraint_features = self.cons_embedding(constraint_features)
    edge_features = self.edge_embedding(edge_features)
    variable_features = self.var_embedding(variable_features)

    # Two half convolutions
    constraint_features = self.conv_v_to_c(
        variable_features, reversed_edge_indices, edge_features, constraint_features
    )
    variable_features = self.conv_c_to_v(
        constraint_features, edge_indices, edge_features, variable_features
    )

    # A final MLP on the variable features
    output = self.output_module(variable_features).squeeze(-1)
    return output

class BipartiteGraphConvolution(torch_geometric.nn.MessagePassing):
    """
    The bipartite graph convolution is already provided by pytorch geometric and we merely need
    to provide the exact form of the messages being passed.
    """

    def __init__(self):
        super().__init__("add")
        emb_size = 64

        self.feature_module_left = torch.nn.Sequential(
            torch.nn.Linear(emb_size, emb_size)
        )
        self.feature_module_edge = torch.nn.Sequential(
            torch.nn.Linear(1, emb_size, bias=False)
        )
        self.feature_module_right = torch.nn.Sequential(
            torch.nn.Linear(emb_size, emb_size, bias=False)
        )
        self.feature_module_final = torch.nn.Sequential(
            torch.nn.LayerNorm(emb_size),
            torch.nn.ReLU(),
            torch.nn.Linear(emb_size, emb_size),
        )

        self.post_conv_module = torch.nn.Sequential(torch.nn.LayerNorm(emb_size))

        # output_layers
        self.output_module = torch.nn.Sequential(
            torch.nn.Linear(2 * emb_size, emb_size),
            torch.nn.ReLU(),
            torch.nn.Linear(emb_size, emb_size),
        )

    def forward(self, left_features, edge_indices, edge_features, right_features):
        """
        This method sends the messages, computed in the message method.
        """
        output = self.propagate(
            edge_indices,
            size=(left_features.shape[0], right_features.shape[0]),
            node_features=(left_features, right_features),
            edge_features=edge_features,

```

```

    )
    return self.output_module(
        torch.cat([self.post_conv_module(output), right_features], dim=-1)
    )

def message(self, node_features_i, node_features_j, edge_features):
    output = self.feature_module_final(
        self.feature_module_left(node_features_i)
        + self.feature_module_edge(edge_features)
        + self.feature_module_right(node_features_j)
    )
    return output

policy = GNNPolicy().to(DEVICE)

```

10. Нейронные сети

Теорема об универсальной аппроксимации (Hornik, 1991)

Любую непрерывную функцию можно с любой точностью приблизить нейросетью глубины 2 с сигмоидной функцией активации на скрытом слое и линейной функцией на выходном слое.

Нейросеть глубины два с фиксированной функцией активации в первом слое и линейной функцией активации во втором слое может равномерно аппроксимировать (может быть при увеличении числа нейронов на первом слое) любую непрерывную функцию на компактном множестве тогда и только тогда, когда функция активации *неполиномиальная*.

Что плохого в сигмоиде:

- о «убивает» градиенты,
- о выходы не отцентрированы (легко устранить с помощью \tanh),
- о вычисление экспоненты все-таки накладно

Замечание

Обратное распространение = SGD + дифференцирование сложных функций (рис. 12)

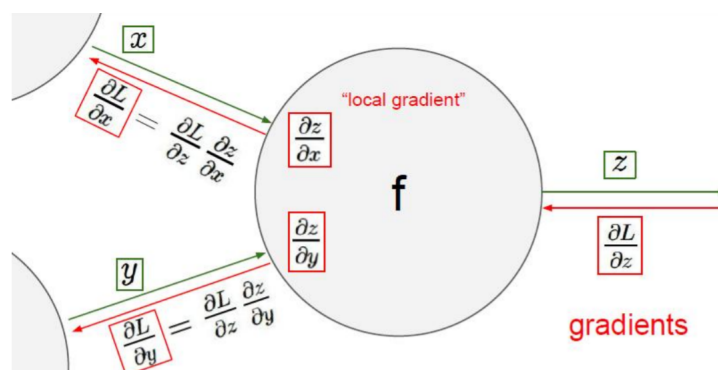


Рис. 12. Прямое распространение сигнала и обратное распространение градиента

11. Приемы работы с библиотекой marshmallow

Сериализация (dumping) – это процедура преобразования объекта или дерева объектов в какой-либо формат, по которому потом эти объекты можно восстановить. Используется, напри-

мер, для сохранения состояния программы (то есть некоторых ее объектов) между запусками. Или для передачи данных по сети.

Основная идея заключается в том, что сериализованный формат – это *последовательность байт* или *строка*.

Обратная процедура называется *десериализацией* (loading).

Например, для сериализации с помощью модуля `json` можно поступить так

```
import json

d = {"key1": 10, "key2": 20}

# для преобразования дерева объектов в последовательность байтов
with open("./make_json.json", mode="w") as f:
    json.dump(d, fp=f) # словарь -> файл

# для преобразования дерева объектов в строку
json.dumps(d) # вернет строку '{"key1": 10, "key2": 20}'
```

Объявим класс данных

```
from marshmallow import Schema, fields, validate, ValidationError

# объявляем структуру данных
class PersonSchema(Schema):
    name = fields.String(
        required=True,
        validate=validate.Regexp("^[a-z].*$")
    )
    age = fields.Integer(
        required=True,
        validate=validate.Range(min=18, max=45)
    )
    job = fields.String(
        required=False,
        validate=validate.Length(min=3)
    )
    email = fields.Email(required=False)
    sex = fields.String(load_default="male") # это значение по умолчанию будет использоваться н
а шаге десериализации

# проверка на согласованность
person_leor = PersonSchema().load({
    "name": "Leor",
    "age": 35,
    "job": "Data Scientist",
    "email": "leor.finkelberg@yandex.ru",
})

type(person_leor) # dict
```

Если переданный словарь отвечает структуре данных, то метод `load()` класса `PersonSchema` этот же словарь и вернет. Но если хотя бы одно значение нарушает требования поля, то будет возбуждено исключение `ValidationError`. Поэтому строки вызова метода `load` следует оборачивать с помощью `try-except`

```
schema = PersonSchema()
leor = {"name": "Leor", ...}
try:
```

```

# метод load прогоняет словарь через структуру данных
# и если все хорошо, то этот же словарь и возвращает
person_leor: dict = PersonSchema().load(leor)
except ValidationError as err:
    print(err.messages, err.valid_data)

```

12. Борьба с переобучением в нейронных сетях

12.1. Нормировка

Пусть даны признаки $X = \{X_1, \dots, X_m\}$.

Тогда

среднее признака

$$\mu = \frac{1}{m} \sum_{i=1}^m X_i,$$

дисперсия признака

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^m (X_i - \mu)^2$$

нормировка

$$X = \frac{X - \mu}{\sqrt{\sigma^2}}$$

12.2. Инициализация весов

Инициализация весов:

- нарушить симметричность (чтобы нейроны были разные),
- недопустить насыщение нейрона (почти всегда близок к 0 или 1),
- ключевая идея – входы на все слои должны иметь одинаковую дисперсию (для избегания «насыщения» нейронов).

Инициализация по Ксавье [Glorot & Bengio, 2010]

$$w_{ij}^{(k)} \sim U \left[-\sqrt{\frac{6}{n_{in}^{(k)} + n_{out}^{(k)}}}, +\sqrt{\frac{6}{n_{in}^{(k)} + n_{out}^{(k)}}} \right].$$

Дисперсия весов

$$D[w_{ij}^{(k)}] = \frac{2}{n_{in}^{(k)} + n_{out}^{(k)}}.$$

Формула выведена в предположении, что нет нелинейностей, т.е. $z^{(k+1)} = f(W^{(k)}z^{(k)}) \equiv W^{(k)}z^{(k)}$.

Смотрим ошибку на отложенной выборке! Выбираем итерацию, на которой ошибка наименьшая (рис. 13).

Увеличение размера пакета – тот же эффект, что и уменьшение темпа обучения.

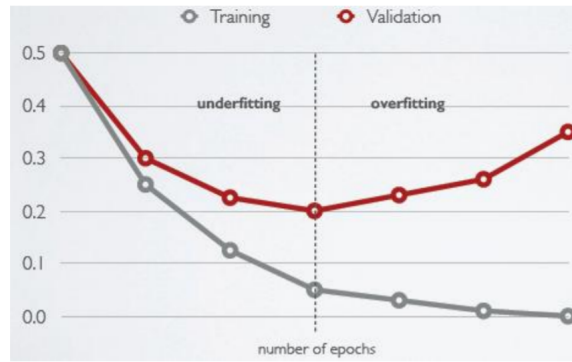


Рис. 13. Настройка темпа обучения

12.3. Продвинутая оптимизация

Стохастический градиент (надо случайно перемешивать данные перед каждой эпохой)

$$w^{(t+1)} = w^{(t)} - \eta \nabla L^{(t)}(w^{(t)}).$$

Стохастический градиент с моментом (Momentum)

$$m^{(t+1)} = \rho m^{(t)} + \nabla L^{(t)}(w^{(t)}),$$

$$w^{(t+1)} = w^{(t)} - \eta m^{(t+1)} = \underbrace{w^{(t)} - \eta \nabla L^{(t)}(w^{(t)})}_{\text{стохастический градиент}} + \underbrace{-\eta \rho m^{(t)}}_{\text{добавление инерции}}.$$

Метод Нестерова

$$m^{(t+1)} = \rho m^{(t)} + \nabla L^{(t)}(w^{(t)} - \eta m^{(t)}),$$

$$w^{(t+1)} = w^{(t)} - \eta m^{(t+1)} = w^{(t)} - \eta \nabla L^{(t)}(w^{(t)} + \underbrace{-\eta m^{(t)}}_{\text{смещение}}) + \underbrace{-\eta \rho m^{(t)}}_{\text{добавление инерции}}.$$

13. Графовые нейронные сети

Полезные ресурсы Distill

- <https://distill.pub/2021/understanding-gnns/>,
- <https://distill.pub/2021/gnn-intro/>.

Графовые нейронные сети (GNN) вычисляют представления вершин в итеративном процессе, разные виды GNN по-разному, каждая итерация соответствует слою сети. Самая простая концепция такого вычисления – неронное распространение (Neural Message Passing). Вообще, распространение сообщений довольно известный прием в анализе графов, заключается в том, что каждая вершина имеет некоторое состояние, которое за одну итерацию уточняется по следующей формуле

$$h_v^{(k)} = \text{UPDATE}^{(k)}\left(h_v^{(k-1)}, \text{AGG}^{(k)}(\{h_u^{(k-1)}\}_{u \in N(v)})\right),$$

где $N(v)$ – окрестной вершины v , AGG – функция агрегации (по смыслу она собирает информацию о соседях, например, суммируя состояния), UPDATE – функция обновления состояния вершины (с учетом собранной информации о соседях).

Единственное требование, которое накладывается на последние две функции – дифференцируемость, чтобы использовать их в вычислительном графе и вычислять параметры сети методом обратного распространения.

Для графовых сверточных нейронных сетей

$$h_v^{(0)} = x_v, \forall v \in V,$$

где $h_v^{(0)}$ – начальное представление узла v , x_v – оригинальные признаки узла v .

И теперь для каждого шага $k = 1, 2, \dots, K$ [Distill]

$$h_v^{(k)} = f^{(k)}\left(W^{(k)} \cdot \frac{\sum_{u \in N(v)} h_u^{(k-1)}}{|N(v)|} + B^{(k)} \cdot h_v^{(k-1)}\right), \forall v \in V,$$

где $h_v^{(k)}$ – представление узла v на шаге k , $h_v^{(k-1)}$ – представление узла v на шаге $k - 1$.

Замечание

Веса $W^{(k)}$ и $B^{(k)}$ разделяются между всеми узлами графа

Выражение справа от коэффициента $W^{(k)}$ – среднее представлений соседей вершины v на шаге $k - 1$.

Построить прогноз на каждом узле можно с помощью последнего вычисленного представления

$$\hat{y}_v = \text{PREDICT}(h_v^{(K)}),$$

где PREDICT – как правило, другая нейронная сеть, обученная вместе с моделью GCN.

Замечание

GCN хорошо масштабируется, поскольку количество параметров в модели не привязано к размеру графа

Пример (рис. 14). Для вершины A на шаге 1 представление можно вычислить следующим образом, опросив соседей вершины

$$\begin{aligned} h_A^{(1)} &= f\left(W^{(1)} \times \frac{h_E^{(0)} + h_F^{(0)} + h_G^{(0)}}{3} + B^{(1)} \times h_A^{(0)}\right) \\ &= f\left(1 \times \frac{2 + (-2) + 0}{3} + 1 \times -4\right) \\ &= f(0 + (-4)) \\ &= f(-4) \\ &= \text{ReLU}(-4) = 0. \end{aligned}$$

На практике каждая описанная выше итерация обычно рассматривается как один «слой нейронной сети». Этой идеологии придерживаются многие популярные библиотеки графовых нейронных сетей (PyTorch Geometric, StellarGraph), позволяющие создавать различные типы сверток графа в одной и той же модели.

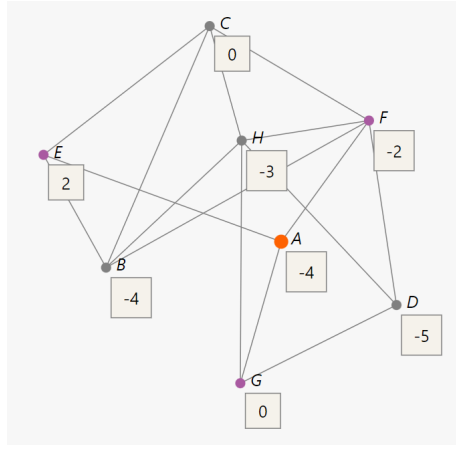


Рис. 14. Пример вычисления обновленного представления узла A на шаге 1 в графовой сверточной нейронной сети

Методы, которые мы рассматривали до сих пор, выполняют «локальные» свертки: признак каждого узла обновляется с использованием информации о признаках его локальных соседей. Однако можно построить и «глобальную» свертку.

После выбора произвольного порядка узлов мы можем собрать все признаки в вектор $x \in \mathbb{R}$. После нормализации x как $\sum_{i=1}^n x_i^2 = 1$

$$R_L(x) = \frac{x^T L x}{x^T x} = \frac{\sum_{(i,j) \in E} (x_i - x_j)^2}{\sum_i x_i^2} = \sum_{(i,j) \in E} (x_i - x_j)^2.$$

Множество собственных чисел лапласиана L называют *спектром*. Спектральное разложение

$$L = U \Lambda U^T, \quad \Lambda = \text{diag}(\lambda_1, \dots, \lambda_n), \quad U = \{u_1 \dots u_n\}, \quad U^T U = I,$$

где Λ – диагональная матрица отсортированных собственных чисел, U – обозначает матрицу собственных векторов, отсортированных по возрастанию собственных чисел.

Каждый вектор признаков x может быть представлен в виде линейной комбинации собственных векторов

$$x = \sum_{i=1}^n \hat{x}_i u_i = U \hat{x},$$

где \hat{x} – это вектор коэффициентов $[x_0, \dots, x_n]$. Будем называть \hat{x} *спектральным представлением* вектора признаков x .

Замечание

Свертку в спектральной области графа можно рассматривать как обобщение свертки в частотной области изображений

Теория спектральных сверток математически обоснована, но есть несколько нюансов:

- Нам требуется вычислить матрицу собственных векторов U_m . Для больших графов это неосуществимо,
- Даже если мы сможем вычислить U_m , сами глобальные свертки неэффективны из-за повторяющегося умножения,

- Изученные фильтры специфичны для графов, поскольку они представлены в терминах спектрального разложения входного графа Лапласиана. Это означает, что они плохо переносятся на новые графы, которые имеют существенно иную структуру (и, следовательно, существенно разные собственные значения).

Хотя спектральные свертки в значительной степени были вытеснены «локальными» свертками по рассмотренным выше причинам, все еще имеет смысл изучать идеи, лежащие в их основе.

Функции потерь для различных задач на графах:

- Классификация узлов:

$$\mathcal{L}(y_v, \hat{y}_v) = - \sum_c y_{vc} \log \hat{y}_{vc},$$

где \hat{y}_{vc} – предсказанная вероятность того что узел v принадлежит классу c . GNNs адаптированы и для обучения на частично-размеченных данных, когда только некоторые узлы имеют разметку. В этом случае функцию потерь можно так

$$\mathcal{L}_G = \frac{\sum_{v \in \text{Lab}(G)} \mathcal{L}(y_v, \hat{y}_v)}{|\text{Lab}(G)|},$$

где потери можно вычислить только на размеченных узлах $\text{Lab}(G)$.

- Классификация графа: собрав информацию о представлении узлов графа, можно составить векторное представление графа.
- Предсказание вероятности появления связи: опираясь на пары смежных и несмежных узлов, можно использовать их векторные представления в качестве входных данных для прогнозирования наличия / отсутствия связи

$$\begin{aligned} \mathcal{L}(y_v, y_u, e_{vu}) &= -e_{vu} \log(p_{vu}) - (1 - e_{vu}) \log(1 - p_{vu}), \\ p_{vu} &= \sigma(y_v^T y_u), \end{aligned}$$

где σ – логистический сигмоид, и $e_{vu} = 1$, если между узлами v и u есть связь, и $e_{vu} = 0$ в противном случае.

- Кластеризация узлов: простая кластеризация представлений узлов.

Основная проблема при использовании описываемого нейронного распространения, т.н. *чрезмерное сглаживание* (over-smoothing): после нескольких итераций пересчета состояний вершин представления соседних вершин становятся похожими, поскольку у них похожие окрестности. Для борьбы с этим делают

- меньше слоев агрегации и больше для «обработки признаков»,
- прокидывание слоев или конкатенацию состояний с предыдущих слоев,
- используют архитектуры, в которых есть эффект памяти,
- приемы с номерками,
- используют аугментацию, например, DropEdge,
- используют noise regularization.

Сводка по графовым нейронным сетям

- На вход сети подается граф, каждая вершина которого имеет признаковое описание. Это описание можно считать начальным состоянием вершины,

- Могут быть слои, которые независимо модифицируют представления (для каждой вершины его представление пропускается через небольшую нейронку),
- Могут быть слои, которые модифицируют представления всех вершин, учитывая представления вершин-соседей,
- Могут быть слои, упрощающие граф (например, уменьшающие число вершин),
- Могут быть слои, получающие представление графа (вектор фиксированной длины) по текущему графу с представлениями вершин.

14. Отбор признаков с библиотекой BoostARoota

BoostARoota <https://github.com/chasedehan/BoostARoota> – алгоритм отбора признаков на базе экстремального градиентного бустинга в реализации XGBoost. Алгоритм требует гораздо меньших затрат времени на выполнение. Перед применением необходимо выполнить дамми-кодирование, поскольку базовая модель работает только с количественными признаками.

Отбор признаков выполняется на обучающем поднаборе данных, поэтому предполагается, что массив меток и массив признаков *обучающие*, а для проверки качества модели отбора признаков есть независимая, *тестовая* выборка. Кроме того, если необходимо выбрать оптимальные значения гиперпараметров модели отбора признаков (например, значения гиперпараметров *cutoff*, *iters* и *delta*), то понадобится еще *проверочная* выборка.

15. Классический и байесовский бутстреп

Бутстреп является универсальным инструментом для оценки статистической точности.

Байесовский бутстреп это байесовский аналог классического бутстрапа. Вместо моделирования распределения выборки для статистики, оценивающей параметр, байесовский бутстреп моделирует *апостериорное распределение параметра*.

Основная идея состоит в том, чтобы случайным образом извлекать наборы данных с возвращением из обучающих данных так, чтобы каждая выборка имела тот же размер, что и исходное обучающее множество. Это делается B раз (скажем, $B = 100$), создавая B множеств бутстрепа. Затем мы заново аппроксимируем модель для каждого из множеств бутстрепа и исследуем поведение аппроксимаций на B выборках.

По выборке бутстрепа мы можем оценить любой аспект распределения $S(\mathbf{Z})$ (это любая величина, вычисленная по данным \mathbf{Z}), например, его дисперсию

$$\widehat{Var}[S(\mathbf{Z})] = \frac{1}{B-1} \sum_{b=1}^B \left(S(\mathbf{Z}^{*b}) - \bar{S}^* \right)^2, \quad \bar{S}^* = \sum_b S(\mathbf{Z}^{*b})/B.$$

16. HDI

Highest Density Interval (HDI) – интервал высокой плотности – показывает какие точки распределения наиболее достоверны/правдоподобны и охватывают большую часть распределения. Каждая точка внутри интервала имеет более высокую *достоверность*, чем любая точка вне интервала.

17. Площадь по ROC-кривой

Построение ROC-кривой происходит следующим образом (рис. 15):

1. Сначала сортируем все наблюдения по убыванию спрогнозированной вероятности положительного класса,
2. Берем единичный квадрат на координатной плоскости. Значения оси абсцисс будут значениями 1 - специфичности (цена деления оси задается значением $1/\text{neg}$), а значения оси ординат будут значениями чувствительности (цена деления оси задается значением $1/\text{pos}$). При этом pos — это количество наблюдений положительного класса, а neg — количество наблюдений отрицательного класса,
3. Задаем точку с координатами $(0, 0)$ и для каждого отсортированного наблюдения x :
 - если x принадлежит положительному классу, двигаемся на $1/\text{pos}$ вверх,
 - если x принадлежит отрицательному классу, двигаемся на $1/\text{neg}$ вправо.

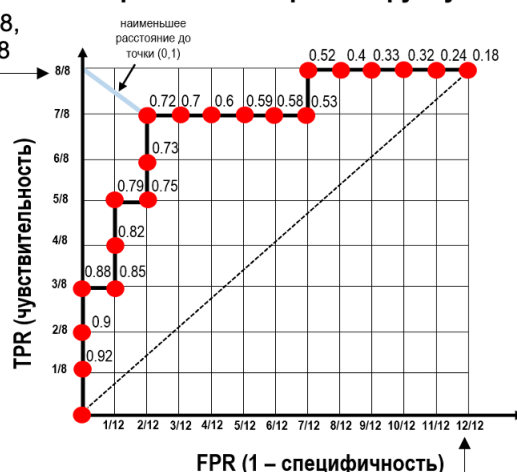
Значение вероятности положительного класса, при котором ROC-кривая находится на минимальном расстоянии от верхнего левого угла — точки с координатами $(0, 1)$, дает наибольшую правильность классификации. В данном случае (рис. 16) будет 0.72.

**Спрогнозированные вероятности
положительного класса,
отсортированные по убыванию**

№	фактический класс	спрогнозированная вероятность положительного класса
20	P	0.92
19	P	0.9
18	P	0.88
12	N	0.85
17	P	0.82
16	P	0.79
11	N	0.75
15	P	0.73
14	P	0.72
10	N	0.7
9	N	0.6
8	N	0.59
7	N	0.58
6	N	0.53
13	P	0.52
5	N	0.4
4	N	0.33
3	N	0.32
2	N	0.24
1	N	0.18

Цена деления $1/8$, поскольку у нас 8 наблюдений положительного класса

Построение ROC-кривой вручную



Вместо 1 – специфичности можно отложить специфичность, но тогда произойдет инверсия шкалы: $12/12$, $11/12$, ..., $1/12$, что не очень удобно для интерпретации

Цена деления $1/12$, поскольку у нас 12 наблюдений отрицательного класса

Рис. 15. Построение ROC-кривой

Площадь под ROC-кривой (ROC-AUC) можно интерпретировать как вероятность события, состоящего в том, что классификатор присвоит более высокий ранг (например, вероятность) случайно выбранному экземпляру положительного класса, чем случайно выбранному экземпляру отрицательного класса (если не рассматривать вариант равенства значений рангов).

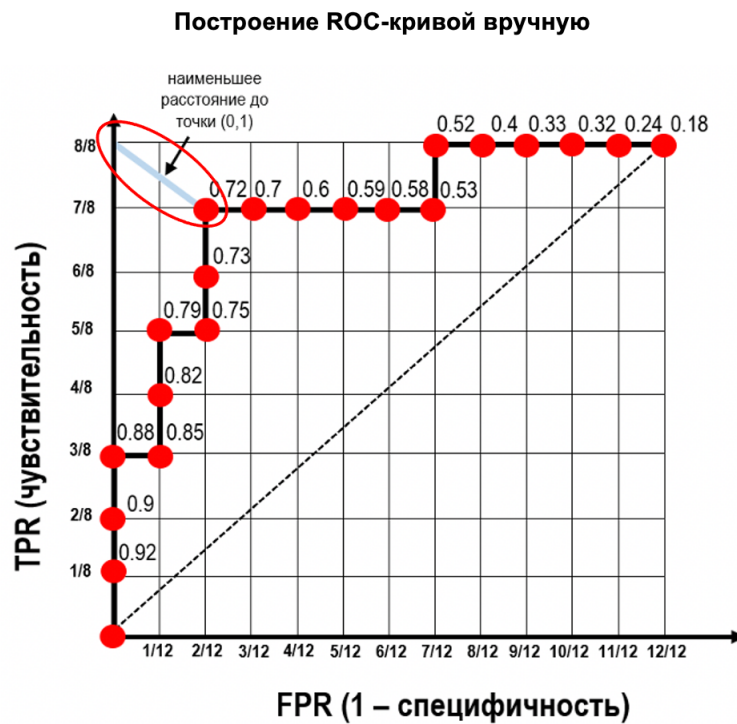


Рис. 16. ROC-кривая. Порог отсечения 0.72

Замечание

На ROC-кривые не влияет баланс классов (при достаточном объеме выборки) и они могут чрезмерно оптимистично оценивать качество работы алгоритма в случае дисбалансов. Лучше пользоваться гармоническим средним или PR-кривыми

Однако недостаток такой интерпретации заключается в том, что мы пренебрегаем часто встречающейся ситуацией равенства вероятностей. Поэтому правильнее будет сказать, что ROC-AUC равен доле пар вида (экземпляр положительного класса, экземпляр отрицательного класса), которые алгоритм верно упорядочил в соответствии с формулой

$$\frac{\sum_{i,j=1}^{n_i, n_j} s(x_i, x_j)}{n_i n_j}, \quad s(x_i, x_j) = \begin{cases} 1, & x_i > x_j, \\ 1/2, & x_i = x_j, \\ 0, & x_i < x_j, \end{cases} \quad (2)$$

где x_i – ответ алгоритма для положительного экземпляра, x_j – ответ алгоритма для отрицательного экземпляра.

По сути числитель дроби представляет собой сумму количеств j -ых наблюдений отрицательного класса, лежащих ниже каждого i -ого наблюдения положительного класса. Каждое такое количество мы берем по каждому i -ому наблюдению положительного класса в последовательности, отсортированной по мере убывания вероятности положительного класса. Знаменатель дроби – это произведение количества наблюдений положительного класса и наблюдений отрицательного класса.

Если говорить более точно, мы берем наблюдение положительного класса под номером 20 и каждый раз образуем пару с наблюдением отрицательного класса (рис. 17), у нас 12 пар, 12

раз наблюдение положительного класса под номером 20 было проранжировано выше наблюдений отрицательного класса 12, 11, 10 и т.д. Записываем число 12 напротив наблюдения 20.

Разные модели нельзя сравнивать только по ROC-AUC. ROC-AUC оценивает разные классификатор, используя метрику, которая сама зависит от классификатора. То есть ROC-AUC оценивает разные классификаторы, используя разные метрики.

Замечание

Если часть ROC-кривой лежит ниже диагональной линии, а часть – выше, то это означает, что классы не являются линейно-сепарабельными, а при этом используется линейная модель

При одинаковой ROC-AUC у разных моделей (соответственно с разными ROC-кривыми) будет разное распределение стоимостей ошибочной классификации. Проще говоря, мы можем вычислить ROC-AUC для классификатора A и получить 0.7, а затем вычислить ROC-AUC для второго классификатора и снова получить 0.7, но это не обязательно означает, что у них одна и та же эффективность.

18. Приемы работы с Gurobi

Полезный ресурс https://www.gams.com/latest/docs/S_GUROBI.html#GUROBI_GAMS_GUROBI_LOG_FILE

Чтобы запустить Gurobi в интерактивном режиме, следует в командной оболочке набрать gurobi

Сессия GUROBI

```
gurobi> m = read("./ikp_milp_problem.lp")
gurobi> m.optimize()
gurobi> vars = m.getVars()
gurobi> help(m)
# вывести 2-картежи целочисленных переменных с отличным от нуля значением
gurobi> [(var.varName, var.x) for var in vars if (var.x > 0) and (var.vType == "I")]
gurobi> m.write("res.sol") # записать решение
gurobi> help(GRM.param) # параметры GUROBI
gurobi> m.getParamInfo("TimeLimit") # ('TimeLimit', <class 'float'>, inf, 0.0, inf, inf)
gurobi> m.getParamInfo("MIPGap") # ('MIPGap', <class 'float'>, 0.0001, 0.0, inf, 0.0001)
gurobi> m.setParam("MIPGap", 65)
gurobi> m.setParam("TimeLimit", 100)
```

Список иллюстраций

1	Многослойный персептрон с двумерным входом, двумя слоями по четыре узла в каждом и выходным слоем с единственным узлом	10
2	Свертка двух матриц	11
3	Фильтр, свертывающий изображение	12
4	Первые два слоя в рекуррентной нейронной сети. На вход подается двумерный вектор признаков. Каждый слой имеет два узла	14
5	Перекрестная проверка на временном ряду <i>расширяющимся</i> окном	15
6	Перекрестная проверка <i>на скользящем</i> окне	15
7	Модифицированная перекрестная проверка <i>расширяющимся</i> окном	16

Отсортированные спрогнозированные вероятности положительного класса

№	фактический класс	спрогно- зированная вероятность положитель- ного класса	скоринговое правило $S(x_i, x_j)$ $= \begin{cases} 1, x_i > x_j, \\ \frac{1}{2}, x_i = x_j, \\ 0, x_i < x_j \end{cases}$	количество наблюдений отрицательного класса, лежащих ниже соответствующего наблюдения положительного класса
20	P	0,92	0	12
19	P	0,9	0	12
18	P	0,88	0	12
12	N	0,85	1	
17	P	0,82	0	11
16	P	0,79	0	11
11	N	0,75	1	
15	P	0,73	0	10
14	P	0,72	0	10
10	N	0,7	1	
9	N	0,6	1	
8	N	0,59	1	
7	N	0,58	1	
6	N	0,53	1	
13	P	0,52	0	5
5	N	0,4	1	
4	N	0,33	1	
3	N	0,32	1	
2	N	0,24	1	
1	N	0,18	1	

Рис. 17. Расчет ROC-AUC по формуле (2)

8	Модифицированная перекрестная проверка скользящим окном	17
9	Лаги, у которых порядок равен горизонту прогнозирования или превышает его, не используют тестовую выборку	26
10	К вопросу о важности признака по частоте его выбора. Признак по оси x выбирается только один раз, а признак по оси y выбирается три раза, но очевидно, что первый признак лучше справляется с задачей	33
11	Перестановочная важность признаков, вычисленная на отложенной выборке	34
12	Прямое распространение сигнала и обратное распространение градиента	56
13	Настройка темпа обучения	59
14	Пример вычисления обновленного представления узла A на шаге 1 в графовой сверточной нейронной сети	61
15	Построение ROC-кривой	64
16	ROC-кривая. Порог отсечения 0.72	65
17	Расчет ROC-AUC по формуле (2)	67
18	Расчет ROC-AUC по формуле (2) для случая равных вероятностей принадлежности экземпляра положительному классу	68

Список литературы

1. Лутц М. Изучаем Python, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 1280 с.

**Отсортированные спрогнозированные вероятности
положительного класса
случай равенства вероятностей**


№	фактический класс	спрогно- зированная вероятность положитель- ного класса	скоринговое правило $S(x_i, x_j)$ $= \begin{cases} 1, x_i > x_j, \\ \frac{1}{2}, x_i = x_j, \\ 0, x_i < x_j \end{cases}$	количество наблюдений отрицательного класса, лежащих ниже соответствующего наблюдения положительного класса	
20	P	0,92	0	12	<div>Считаем количество отрицательных ниже каждого наблюдения положи- тельного класса</div> 
19	P	0,9	0	12	
18	P	0,88	0,5	11,5	
12	N	0,88			
17	P	0,82	0	11	
16	P	0,79	0	11	
11	N	0,75	1		
15	P	0,73	0	10	
14	P	0,72	0	10	
10	N	0,7	1		
9	N	0,6	1		
8	N	0,59	1		
7	N	0,58	1		
6	N	0,53	1		
13	P	0,52	0	5	
5	N	0,4	1		
4	N	0,33	1		
3	N	0,32	1		
2	N	0,24	1		
1	N	0,18	1		

Рис. 18. Расчет ROC-AUC по формуле (2) для случая равных вероятностей принадлежности экземпляра положительному классу

2. Бурков А. Машинное обучение без лишних слов. – СПб.: Питер, 2020. – 192 с.
3. Бизли Д. Python. Подробный справочник. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 864 с.