

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 Приемы работы с библиотекой анализа временных рядов Darts	8
2 XGBoost	36
3 LightGBM	59
4 Приемы работы с библиотекой подбора гиперпараметров Optuna	78
5 Приемы работы с AutoML-библиотекой FLAML	91
6 Масштабирование признаков с выбросами	101
7 Дисбаланс классов	101
8 Метрики MAPE, SMAPE, WAPE, WMAPE etc.	104
9 Внедрение моделей машинного обучения в промышленную эксплуатацию	105
10 ML System Design Doc	110
11 Квантильная регрессия	110
12 MAE vs MSE. Устойчивость среднеабсолютной ошибки к выбросам	111
13 Оптимизационные задачи и теорема Куна-Таккера	111
14 Векторное дифференцирование	112
15 Классические алгоритмы машинного обучения	115
16 Глубокое обучение	128
17 Приемы работы с библиотекой plotly	132
18 Приемы работы с библиотекой анализа временных рядов ETNA	134
19 Генерация признаков и кодирование категориальных признаков	149

20 Перестановочная важность признаков и важность признаков по Шепли	152
21 Приемы работы с библиотекой Polars	155
22 Приемы работы с библиотекой Catboost	161
23 Приемы работы с решателем SCIP	167
24 Приемы работы с библиотеками Gym и Ecole	169
25 Нейронные сети	176
26 Приемы работы с библиотекой <code>marshmallow</code>	176
27 Борьба с переобучением в нейронных сетях	178
28 Графовые нейронные сети	179
29 Отбор признаков с библиотекой BoostARoota	183
30 Классический и байесовский бутстреп	183
31 HDI	183
32 Площадь по ROC-кривой	184
33 Приемы работы с Gurobi	186
34 Кластеризация. K-means, MeanShift, Affinity Prop, HDBSCAN и детектор выбросов GLOSH	187
Список иллюстраций	191
Список литературы	191

Содержание

1 Приемы работы с библиотекой анализа временных рядов Darts	8
1.1 Быстры старт	8
1.1.1 Чтение данных и построение <code>TimeSeries</code>	8
1.1.2 Некоторые операции над <code>TimeSeries</code>	9
1.1.3 Разбиение на обучающий и валидационный поднаборы	9
1.1.4 Обучение модели и построение прогноза	9
1.1.5 Сезонная компонента	10
1.1.6 Вычисление метрик	10
1.1.7 Быстрое сравнение моделей	10
1.1.8 Обратное тестирование (Backtesting)	10
1.1.9 Машинное обучение	12
1.1.10 Использование моделей глубокого обучения. N-BEATS	13
1.1.11 Ковариаты. Использование внешних данных	13

1.1.12	Кодирование для ковариат	15
1.1.13	Регрессионные модели прогнозирования	16
1.1.14	Вероятностное прогнозирование	17
1.1.15	Ансамбли моделей	18
1.1.16	Фильтрация. Фильтра Калмана	19
1.1.17	Гауссовские процессы	19
1.2	TimeSeries	20
1.3	Обзор моделей прогнозирования	21
1.4	Torch-ие модели	24
1.4.1	Как используются входные данные в TFM при обучении и построении прогноза	24
1.4.2	Повторное обучение или подбор гиперпараметров предобученной модели	27
1.4.3	Обратные вызовы	28
1.4.4	Рекомендации по производительности	29
1.5	Ковариаты	31
1.6	Подбор гиперпараметров в Darts	32
1.6.1	Подбор гиперпараметров с помощью Optuna	32
1.6.2	Подбор гиперпараметров с помощью Ray Tune	34
2	XGBoost	36
2.1	Установка	36
2.2	Вводные замечания	36
2.3	Оценка структуры	37
2.4	Введение в Model IO	38
2.5	DART booster	38
2.6	Ограничения на монотонность	38
2.7	Ограничение на взаимодействие признаков	38
2.8	Пользовательские целевые функции и метрики качества	39
2.9	Категориальные признаки	40
2.10	Замечания о подборе гиперпараметров в XGBoost	41
2.11	Параметры XGBoost	42
2.11.1	Общие параметры	42
2.11.2	Параметры бустера	42
2.11.3	Параметры задачи	44
2.12	Python Package	45
2.12.1	Интерфейс данных	45
2.12.2	Ранняя остановка	45
2.12.3	Прогноз	46
2.12.4	Scikit-learn интерфейс	46
2.12.5	Core Data Structure	46
2.12.6	Learning API	48
2.12.7	Scikit-Learn API	50
2.13	Подбор гиперпараметров	50
2.14	Подбор гиперпараметров XGBoost на отложенной выборке с подрезкой «слабых» запусков	52

2.15 Подбор гиперпараметров XGBoost с перекрестной проверкой и подрезкой «слабых» запусков	53
2.15.1 Конспект статьи Chen T. XGBoost: A Scalable Tree Boosting System, 2016	54
2.15.2 Целевая функция с регуляризацией	55
2.15.3 Градиентный бустинг	55
2.15.4 Сжатие (shrinkage) и подвыборка по признакам	56
2.15.5 Алгоритм поиска разбиения	57
2.15.6 Блоки столбцов для параллельного обучения	58
2.15.7 Блоки для внеядерных (out-of-core) вычислений	59
3 LightGBM	59
3.1 Установка	59
3.2 Быстрый старт	60
3.2.1 Работа с входными данными	60
3.2.2 Поддержка категориальных признаков	60
3.2.3 Эффективное использование памяти	61
3.2.4 Настройка параметров	61
3.2.5 Обучение	62
3.2.6 Ранняя остановка	63
3.2.7 Прогноз	63
3.3 Особенности реализации LightGBM	64
3.4 Стратегия выращивания деревьев в LightGBM	64
3.5 Оптимальное разбиение для категориальных признаков	65
3.6 Расспараллеливание	65
3.7 Параметры	66
3.8 Замечания о подборе гиперпараметров в LightGBM	68
3.8.1 <code>max_depth</code> и <code>num_leaves</code> для снижения времени обучения	68
3.8.2 <code>min_gain_to_split</code>	68
3.8.3 <code>min_data_in_leaf</code> и <code>min_sum_hessian_in_leaf</code>	69
3.8.4 <code>learning_rate</code> и <code>num_iterations</code>	69
3.8.5 <code>max_bin</code> или <code>max_bin_by_feature</code>	69
3.8.6 <code>feature_fraction</code>	69
3.8.7 <code>bagging_fraction</code> и <code>bagging_freq</code>	70
3.8.8 Для улучшения метрик	70
3.8.9 Для снижения эффекта переобучения	70
3.9 Подбор гиперпараметров LightGBM с подрезкой «слабых» запусков	70
3.10 Подбор гиперпараметров для LightGBM с помощью Optuna LightGBMTuner	71
3.11 Распределенное обучение с LightGBM	73
3.12 Конспект статьи G. Ke LightGBM. A Highly Efficient Gradient	73
3.12.1 GBDT и анализ сложности	74
3.12.2 Gradient-based One-Side Sampling, GOSS	75
3.12.3 Exclusive Feature Bundling	77
3.12.4 Обсуждение	78

4 Приемы работы с библиотекой подбора гиперпараметров Optuna	78
4.1 Эффективные алгоритмы оптимизации	79
4.1.1 Алгоритмы выборки	79
4.1.2 Алгоритмы подрезки	79
4.2 Сохранение и восстановление сессии исследований с помощью реляционной базы данных	80
4.3 Мульти-целевая оптимизация в Optuna	81
4.4 Пользовательские атрибуты	81
4.5 Пользовательские семплеры	82
4.6 Пользовательские пранеры	84
4.7 Пользовательские callbacks для метода <code>optimize</code>	85
4.8 Ask-and-Tell Interface	86
4.9 Define-and-Run	87
4.10 Пакетная оптимизация	87
4.11 Переиспользование лучшего запуска	88
4.12 Создание целевой функции с дополнительными аргументами	88
4.13 Optuna можно использовать и без RDB-серверов	90
4.14 Как сохранить модель машинного обучения, обученную в функции <code>objective</code>	90
4.15 Как можно тестировать целевые функции	91
5 Приемы работы с AutoML-библиотекой FLAML	91
5.1 Установка	92
5.2 Быстрый старт	92
5.3 Основные параметры	92
5.4 Кастомизация пространства поиска	94
5.5 Stacking	96
5.6 Стратегии разбиения на обучающий и валидационный поднаборы данных	97
5.7 Теплый старт	97
5.8 Получение лучшей модели	98
5.9 Как настраивать ограничение по времени	98
5.10 Алгоритмы подбора гиперпараметров	98
5.11 Zero Shot AutoML	99
5.12 Мультицель в задаче регрессии	100
5.13 FLAML + Sklearn pipeline	100
6 Масштабирование признаков с выбросами	101
7 Дисбаланс классов	101
8 Метрики MAPE, SMAPE, WAPE, WMAPE etc.	104
8.1 MAPE	104
8.2 SMAPE	104
8.3 WAPE / WMAPE	105

9 Внедрение моделей машинного обучения в промышленную эксплуатацию	105
9.1 Паттерны внедрения ML-моделей в промышленную эксплуатацию	106
9.1.1 Модель как услуга (Model-as-Service)	106
9.1.2 Модель как зависимость (Model-as-Dependency)	107
9.1.3 Предварительный расчет (Precompute)	107
9.1.4 Модель по запросу (Model-on-Demand) с Apache Kafka	108
9.1.5 Гибридная модель обслуживания (Hybrid Model Serving)	108
9.2 Стратегии внедрения ML-модели в прод	109
9.2.1 Разворачивание с помощью Docker-контейнеров	109
9.2.2 Бессерверные вычисления (serverless)	109
10 ML System Design Doc	110
11 Квантильная регрессия	110
12 MAE vs MSE. Устойчивость среднеабсолютной ошибки к выбросам	111
13 Оптимизационные задачи и теорема Куна-Таккера	111
14 Векторное дифференцирование	112
14.1 Решение задачи регрессии для многомерного случая	114
14.2 Градиентный спуск	115
15 Классические алгоритмы машинного обучения	115
15.1 Линейная регрессия	115
15.2 Логистическая регрессия	116
15.3 Деревья решений	117
15.4 Метод опорных векторов	118
15.4.1 Из документации scikit-learn	118
15.4.2 Из книги Жерона	119
15.4.3 Из курса лекций Соколова	122
15.4.4 Из книги Буркова	124
15.5 Метод k ближайших соседей	126
15.6 Многослойный персепtron	127
16 Глубокое обучение	128
16.1 Сврточная нейронная сеть	128
16.2 Рекуррентная нейронная сеть	131
17 Приемы работы с библиотекой plotly	132
18 Приемы работы с библиотекой анализа временных рядов ETNA	134
18.1 Перекрестная проверка на временных рядах	134
18.2 CatBoost. Базовая модель с конструированием признаков	137
18.3 Пользовательские классы для вычисления скользящих статистик	139
18.4 Работа с несколькими временными рядами	146

19 Генерация признаков и кодирование категориальных признаков	149
19.1 Кодирование одного категориального признака по другому категориальному признаку с помощью сингулярного разложения	149
20 Перестановочная важность признаков и важность признаков по Шепли	152
21 Приемы работы с библиотекой Polars	155
21.1 Установка	155
21.2 Вводные замечания	155
21.3 Polars-выражения	156
21.4 Оконные функции	159
21.5 Универсальные NumPy-функции	160
21.6 Примеры	160
21.7 Работа с временными рядами	161
22 Приемы работы с библиотекой Catboost	161
22.1 Установка CatBoost	161
22.2 Ключевые особенности пакета	162
22.3 Параметры	162
22.4 Классификатор CatBoostClassifier	162
22.5 Регрессор CatBoostRegressor	164
22.6 Функции потерь и метрики качества	165
22.6.1 Для классификации	165
22.6.2 Для регрессии	166
23 Приемы работы с решателем SCIP	167
23.1 Общие сведения	168
23.2 Emphasis Settings	168
23.3 Проблемы «using pseudo solution instead»	168
24 Приемы работы с библиотеками Gym и Ecole	169
24.1 Gym	169
24.2 Ecole	169
24.2.1 Общие сведения	170
24.2.2 Observations	170
24.2.3 Анализ примера работы связки Ecole+GNN	171
25 Нейронные сети	176
26 Приемы работы с библиотекой marshmallow	176
27 Борьба с переобучением в нейронных сетях	178
27.1 Нормировка	178
27.2 Инициализация весов	178
27.3 Продвинутая оптимизация	179
28 Графовые нейронные сети	179

29 Отбор признаков с библиотекой BoostARoota	183
30 Классический и байесовский бутстреп	183
31 HDI	183
32 Площадь по ROC-кривой	184
33 Приемы работы с Gurobi	186
34 Кластеризация. K-means, MeanShift, Affinity Prop, HDBSCAN и детектор выбросов GLOSH	187
34.1 Краткая сводка по основным кластеризаторам	187
34.1.1 KMeans	187
34.1.2 Affinity Propagation	188
34.1.3 MeanShift	188
34.1.4 Spectral Clustering	188
34.1.5 Agglomerative Clustering	189
34.1.6 DBSCAN	189
34.1.7 HDBSCAN	189
34.2 Одноклассовая классификация	189
34.3 Обнаружение выбросов с помощью GLOSH	190
Список иллюстраций	191
Список литературы	191

1. Приемы работы с библиотекой анализа временных рядов Darts

<https://unit8co.github.io/darts/>

1.1. Быстры старт

`TimeSeries` это самый главный класс в Darts. Он представляет и одномерные и многомерные временные ряды.

Временной индекс может быть:

- типа `pandas.DatetimeIndex` (содержит временные метки),
- типа `pandas.RangeIndex` (содержит целые числа; полезно для представления последовательности данных без временных меток).

В некоторых случаях `TimeSeries` может представлять даже вероятностные временные ряды для получения доверительных интервалов. Все модели Darts принимают `TimeSeries` и возвращают `TimeSeries`.

1.1.1. Чтение данных и построение `TimeSeries`

Экземпляр `TimeSeries` может быть легко построен с помощью следующих методов:

- на базе кадра данных `DataFrame`: `TimeSeries.from_dataframe()`,
- на базе временного индекса и массива: `TimeSeires.from_times_and_values()`,

- о на базе NumPy-массива: `TimeSeries.from_series()`,
- о на базе серии `Series`: `TimeSeries.from_series()`,
- о на базе `xarray.DataArray`: `TimeSeries.from_xarray()`,
- о на базе CSV-файлов: `TimeSeries.from_csv()`.

```
from darts import TimeSeries
from darts.datasets import AirPassengersDataset

series = AirPassengersDataset().load()
```

1.1.2. Некоторые операции над TimeSeries

```
series1, series2 = series.split_before(0.75)
# series1, series2 = series[:-36], series[-36:]
```

Арифметические операции

```
series_noise = TimeSeries.from_times_and_values(
    series.time_index,
    np.random.randn(len(series))
)
ser = series / 2 + 20 * series_noise - 10
```

```
series.map(np.log)
series.map(lambda ts, x: x / ts.days_in_month)
```

Можно из временной метки вытащить какой-нибудь атрибут и добавить его, например, в качестве второго измерения

```
series.add_datetime_attribute("month")
series.add_holiday("US")
```

Заполнение пропущенных значений `np.nan`

```
from darts.utils.missing_values import fill_missing_values

values = np.arange(50, step=0.5)
values[10:30] = np.nan
values[60:95] = np.nan

series_ = TimeSeries.from_values(values)
fill_missing_values(series_)
```

1.1.3. Разбиение на обучающий и валидационный поднаборы

```
train, val = series.split_before(pd.Timestamp("19580101"))
train.plot(label="training")
val.plot(label="validation")
```

1.1.4. Обучение модели и построение прогноза

```
from darts.models import NaiveSeasonal

naive_model = NaiveSeasonal(K=1)
naive_model.fit(train)
```

```
naive_forecast = naive_model.predict(36)

series.plot(label="actual")
naive_forecast.plot(label="naive forecast")
```

1.1.5. Сезонная компонента

Для того чтобы проверить наличие сезонной компоненты в наборе данных, можно построить автокорреляционную функцию

```
from darts.utils.statistics import plot_acf, check_seasonality

plot_acf(train, m=12, alpha=0.05)
```

Если в наборе есть сезонная компонента, ее лаг ($m=12$) будет посвечен красным. Еще можно проверить статистическую гипотезу по лагу-кандидату

```
for m in range(2, 25):
    is_seasonal, period = check_seasonality(train, m=m, alpha=0.05)
    if is_seasonal:
        print(f"There is seasonality of order {period}")
```

1.1.6. Вычисление метрик

ВАЖНО: на практике есть веские причины не использовать MAPE, здесь MAPE приводится просто в качестве примера.

```
from darts.metrics import mape

print(
    "Mean absolute percentage error for the combined naive drift + seasonal: {:.2f}%".format(
        mape(series, combined_forecast)
    )
```

1.1.7. Быстрое сравнение моделей

```
from darts.models import ExponentialSmoothing, TBATS, AutoARIMA, Theta

def eval_model(model):
    model.fit(train)
    forecast = model.predict(len(val))
    print(mape(val, forecast))

eval_model(ExponentialSmoothing())
eval_model(TBATS())
eval_model(AutoARIMA())
eval_model(Theta())
```

1.1.8. Обратное тестирование (Backtesting)

Обратное тестирование (Backtesting) строит прогнозы с помощью рассматриваемой модели на данных из «прошлого» таким образом, давая возможность изучить поведение модели на старых данных. То есть: «какая эффективность была бы у нашей модели, если бы она работала еще тогда, на данных из прошлого»

```

historical_fcast_theta = best_theta_model.historical_forecasts(
    series, start=0.6, forecast_horizon=3, verbose=True
)
series.plot(label="data")
historical_fcast_theta.plot(label="backtest 3-months ahead forecast (Theta)")
print("MAPE={:.2f}%".format(mape(historical_fcast_theta, series)))

```

Можно посмотреть «остатки» обученой модели (рис. 1)

```

from darts.utils.statistics import plot_residuals_analysis

plot_residuals_analysis(best_theta_model.residuals(series))

```

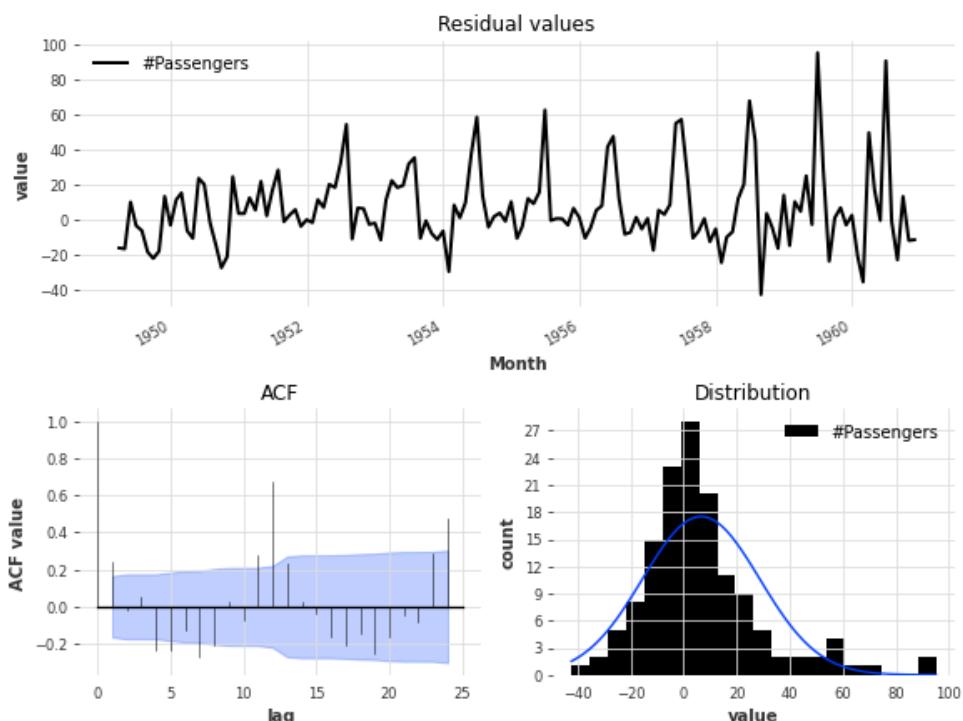


Рис. 1. Значения остатков простой модели Θ

Анализ остатков (рис. 1) показывает, что распределение не центрировано на 0, а значит наша модель *смещена*. Еще можно заметить большие значения автокорреляционной функции на 12-ом лаге. Это означает, что остатки содержат информацию, не учтенную моделью.

Может быть удастся исправить ситуацию с помощью модели ExponentialSmoothing

```

model_es = ExponentialSmoothing(seasonal_periods=12)
historical_fcast_es = model_es.historical_forecasts(
    series, start=0.6, forecast_horizon=3, verbose=True
)

series.plot(label="data")
historical_fcast_es.plot(label="backtest 3-months ahead forecast (Exp. Smoothing)")
print("MAPE = {:.2f}%".format(mape(historical_fcast_es, series)))

```

Смотрим остатки (рис. 2)

```
plot_residuals_analysis(model_es.residuals(series))
```

Как можно видеть на рис. 2 распределение остатков стало центрированным, а значения автокорреляционной функции меньше (хотя их нельзя считать незначимыми).

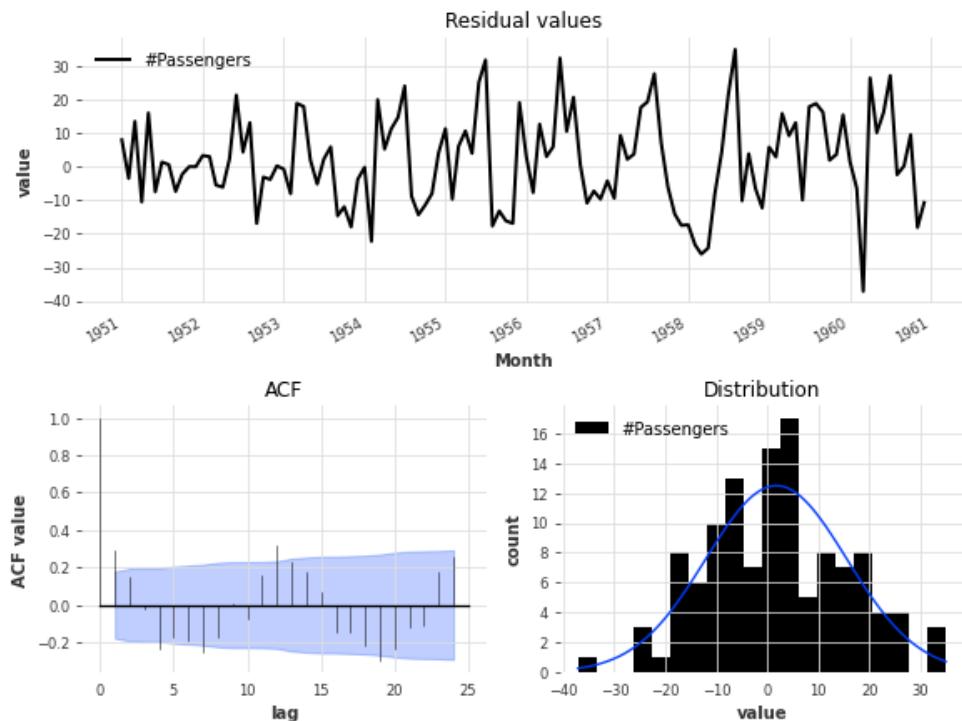


Рис. 2. Значения остатков модели ExponentialSmoothing

1.1.9. Машинное обучение

Обычно (это больше касается моделей глубокого обучения) обучение требует большого количества данных, что означает большое количество отдельных, но связанных временных рядов.

В Darts набор (multiple) временных рядов представляется последовательностью (**Sequence**) временных рядов (**TimeSeries**) (например, список временных **TimeSeries**).

Для примера рассмотрим два временных ряда, относящихся к различным времененным периодам

```
from darts.datasets import AirPassengersDataset, MonthlyDataset

series_air = AirPassengersDataset().load().astype(np.float32)
series_milk = MonthlyMilkDataset().load().astype(np.float32)

train_air, val_air = series_air[:-36], series_air[-36:]
train_milk, val_milk = series_milk[:-36], series_milk[-36:]

train_air.plot()
val_air.plot()
train_milk.plot()
val_milk.plot()
```

Приведем временные ряды к диапазону от 0 до 1

```
from darts.dataprocessing.transformers import Scaler

scaler = Scaler()
train_air_scaled, train_milk_scaled = scaler.fit_transform([train_air, train_milk])

train_air_scaled.plot()
train_milk_scaled.plot()
```

1.1.10. Использование моделей глубокого обучения. N-BEATS

```
from darts.models import NBEATSMModel

model = NBEATSMModel(
    input_chunk_length=24,
    output_chunk_length=12,
    random_seed=42
)

model.fit(
    [train_air_scaled, train_milk_scaled],
    epochs=50,
    verbose=True
)
```

Строим прогноз на 36 месяцев вперед

```
pred_air = model.predict(series=train_air_scaled, n=36)
pred_milk = model.predict(series=train_milk_scaled, n=36)

pred_air, pred_milk = scaler.inverse_transform([pred_air, pred_milk])
```

1.1.11. Ковариаты. Использование внешних данных

В дополнение к целевому временному ряду (target series) (то есть временному ряду, который мы пытаемся прогнозировать), многие модели Darts еще поддерживают *ковариаты* (covariates). Ковариаты – это временные ряды, которые нас не интересуют с точки зрения прогноза, но которые могут содержать полезную информацию для прогнозирования целевого временного ряда. И целевой временной ряд, и ковариаты могут быть как одномерными, так и многомерными.

Darts поддерживает два вида ковариатов (рис. 3):

- **past_covariates:** это временные ряды, которые не обязательно известны на горизонте прогноза; модели не используют значения этих ковариатов с горизонта прогнозирования,
- **future_covariates:** это временные ряды, известные вплоть до горизонта прогнозирования; модели учитывают значения этих ковариатов с горизонта прогнозирования.

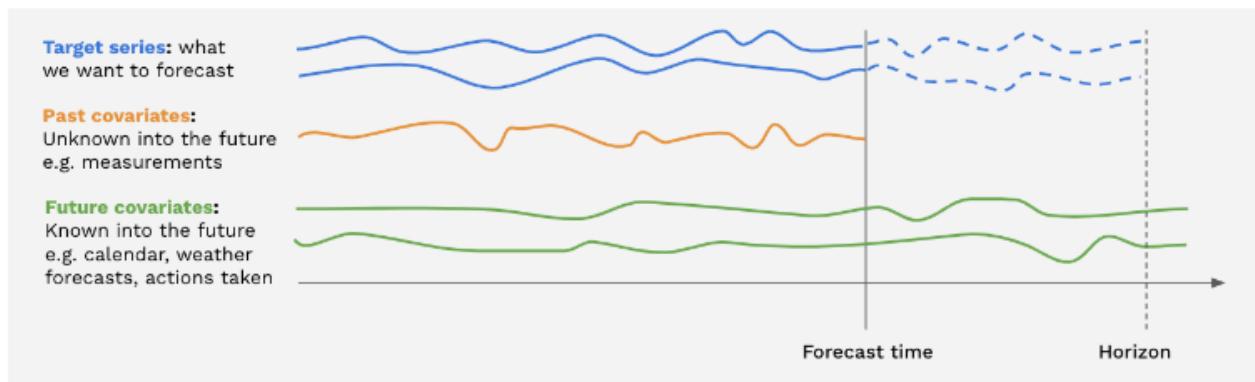


Рис. 3. Ковариаты

Каждый ковариат может быть многомерным. Если есть несколько ковариатов, то их можно объединить с помощью `stack()` или `concatenate()` в многомерный временной ряд

```

from darts import concatenate
from darts.utils.timeseries_generation import datetime_attribute_timeseries as dt_attr

air_covs = concatenate(
    [
        dt_attr(series_air.time_index, "month", dtype=np.float32) / 12,
        (dt_attr(series_air.time_index, "year", dtype=np.float32) - 1948) / 12,
    ],
    axis="component",
)

milk_covs = concatenate(
    [
        dt_attr(series_milk.time_index, "month", dtype=np.float32) / 12,
        (dt_attr(series_milk.time_index, "year", dtype=np.float32) - 1962) / 13,
    ],
    axis="component",
)

```

Не все модели поддерживают все типы ковариатов. Например, NBEATSModel поддерживает только past_covariates. Для обучения модели нужно просто передать ковариаты методу fit()

```

model = NBEATSModel(
    input_chunk_length=24,
    output_chunk_length=12,
    random_seed=42,
)

model.fit(
    [train_air_scaled, train_milk_scaled],
    past_covariates=[air_covs, milk_covs],
    epochs=50,
    verbose=True,
)

```

Строим прогноз

```

pred_air = model.predict(
    series=train_air_scaled,
    past_covariates=air_covs, n=36
)
pred_milk = model.predict(
    series=train_milk_scaled,
    past_covariates=milk_covs, n=36
)

# scale back:
pred_air, pred_milk = scaler.inverse_transform([pred_air, pred_milk])

plt.figure(figsize=(10, 6))
series_air.plot(label="actual (air)")
series_milk.plot(label="actual (milk)")
pred_air.plot(label="forecast (air)")
pred_milk.plot(label="forecast (milk)")

```

1.1.12. Кодирование для ковариат

Чтобы добавить ковариаты в модель достаточно просто указать `add_encoders`. Этот параметр принимает словарь, который описывает что должно быть закодировано в качестве внешних ковариат. Например (здесь поддерживаются и «прошлые» и «будущие» ковариаты)

```
encoders = {
    "cyclic": {"future": ["month"]},
    "datetime_attribute": {"future": ["hour", "dayofweek"]},
    "position": {"past": ["absolute"], "future": ["relative"]},
    "custom": {"past": [lambda idx: (idx.year - 1950) / 50]},
    "transformer": Scaler(),
}
```

В этом словаре:

- Месяцы будут использованы как будущие ковариаты (future covariates) с гармоническим кодированием (sin/cos),
- Часы и дни недели будут использовать как будущие ковариаты,
- Абсолютная позиция (временной шаг во временном ряду) будет использоваться как прошлый ковариат,
- Относительная позиция (относительно времени прогнозирования) будет использоваться как будущий ковариат,
- Дополнительная пользовательская функция будет использоваться как прошлый ковариат,
- Все ковариаты будут масштабироваться с помощью `Scaler`,

```
encoders = {
    "datetime_attribute": {
        "past": ["month", "year"]
    },
    "transformer": Scaler(),
}

model = NBEATSMModel(
    input_chunk_length=24,
    output_chunk_length=12,
    add_encoders=encoders,
    random_state=42,
)

model.fit(
    # при обучении передается последовательность временных рядов
    [train_air_scaled, train_milk_scaled],
    epochs=50,
    verbose=True,
)
```

Строим прогноз только для пассажиров

```
pred_air = model.predict(series=train_air_scaled, n=36)

# scale back:
pred_air = scaler.inverse_transform(pred_air)
```

1.1.13. Регрессионные модели прогнозирования

Регрессионные модели Darts это модели, совместимые с регрессионными моделями Sklearn. Внутренние регрессионные модели используются будущих значений целевого временного ряда, как функции от лагов целевого временного ряда, прошлых и будущих ковариатов. Под капотом временной ряд табулизируется, то есть превращается в матрицу признакового описания объекта.

По умолчанию `RegressionModel` это модели *линейной регрессии*. Кроме того Darts из коробки поддерживает:

- `sklearn.ensemble.RandomForestRegressor`,
- `LightGBMModel`,
- `sklearn.linear_model.LinearRegression`.

Для примера рассмотрим Байесовскую гребневую регрессию

```
from darts.models import RegressionModel
from sklearn.linear_model import BayesianRidge

model = RegressionModel(
    lags=72,
    lags_future_covariates=[-6, 0],
    model=BayesianRidge(),
)

model.fit(
    [train_air_scaled, train_milk_scaled],
    future_covariates=[air_covs, milk_covs]
)
```

Строим прогноз

```
pred_air, pred_milk = model.predict(
    series=[train_air_scaled, train_milk_scaled],
    future_covariates=[air_covs, milk_covs],
    n=36,
)

# scale back:
pred_air, pred_milk = scaler.inverse_transform([pred_air, pred_milk])
```

Обратное тестирование

```
bayes_ridge_model = RegressionModel(
    lags=72,
    lags_future_covariates=[0],
    model=BayesianRidge()
)

backtest = bayes_ridge_model.historical_forecasts(
    series_air,
    future_covariates=air_covs,
    start=0.6,
    forecast_horizon=3,
    verbose=True,
)

print(mape(backtest, series_air))
```

1.1.14. Вероятностное прогнозирование

Некоторые модели в Darts поддерживают вероятностное прогнозирование. Например, для ARIMA и ExponentialSmoothing достаточно просто указать `num_samples` в `predict()`. `num_samples` это число точек выборки по Монте-Карло, описывающих распределение (преимущество использования метода Монте-Карло в том, что таким образом можно описывать и параметрические и непараметрические распределения) и вычислять произвольные квантили

```
model_es = ExponentialSmoothing()
model_es.fit(train)
probabilistic_forecast = model_es.predict(len(val), num_samples=500)

series.plot(label="actual")
probabilistic_forecast.plot(label="probabilistic forecast")
```

При работе с нейросетевыми моделями можно указать правдоподобие

```
from darts.models import TCNModel
from darts.utils.likelihood_models import LaplaceLikelihood

model = TCNModel(
    input_chunk_length=24,
    output_chunk_length=12,
    random_state=42,
    likelihood=LaplaceLikelihood(),
)

model.fit(train_air_scaled, epochs=400, verbose=True);
```

При построении прогноза нужно указать `num_samples >> 1`

```
pred = model.predict(n=36, num_samples=500)

# scale back:
pred = scaler.inverse_transform(pred)
```

Кроме того, если есть априорная уверенность в том, что масштаб распределения составляет, скажем, 0.1, то

```
model = TCNModel(
    input_chunk_length=24,
    output_chunk_length=12,
    random_seed=42,
    likelihood=LaplaceLikelihood(prior_b=0.1),
)

model.fit(train_air_scaled, epochs=400, verbose=True)
pred = model.predict(n=36, num_samples=500)
pred = scaler.inverse_transform(pred)
```

По умолчанию `TimeSeries.plot()` показывает медиану, а также 5-ый и 95-ый процентили, но этим можно управлять

```
pred.plot(low_quantile=0.01, high_quantile=0.99, label="1-99th percentiles")
pred.plot(low_quantile=0.2, high_quantile=0.8, label="20-80th percentiles")
```

Правдоподобие должно быть согласовано с областью значений временного ряда. Например, `PoissonLikelihood` может использоваться для положительных дискретных значений, `ExponentialLikelihood`

– для положительных вещественных, а BetaLikelihood – для вещественных значений из диапазона (0, 1).

Качество вероятностного прогноза можно оценить с помощью ρ -риска (или квантильной потери), который вычисляет ошибку для каждого квантиля

```
from darts.metrics import rho_risk

print("MAPE of median forecast: %.2f" % mape(series_air, pred))
for rho in [0.05, 0.1, 0.5, 0.9, 0.95]:
    rr = rho_risk(series_air, pred, rho=rho)
    print("rho-risk at quantile %.2f: %.2f" % (rho, rr))
```

Можно работать с квантильной регрессией QuantileRegression напрямую

```
from darts.utils.likelihood_models import QuantileRegression

model = TCNModel(
    input_chunk_length=24,
    output_chunk_length=12,
    random_state=42,
    likelihood=QuantileRegression([0.05, 0.1, 0.5, 0.9, 0.95]),
)

model.fit(train_air_scaled, epochs=400, verbose=True);
```

Строим прогноз

```
pred = model.predict(n=36, num_samples=500)
pred = scaler.inverse_transform(pred)

series_air.plot()
pred.plot()

print("MAPE of median forecast: %.2f" % mape(series_air, pred))
for rho in [0.05, 0.1, 0.5, 0.9, 0.95]:
    rr = rho_risk(series_air, pred, rho=rho)
    print("rho-risk at quantile %.2f: %.2f" % (rho, rr))
```

1.1.15. Ансамбли моделей

Для примера мы объединим наивную модель с сезонной компонентой и модель с трендом. Наивное объединение просто берет среднее прогнозов нескольких моделей

```
from darts.models import NaiveEnsembleModel

models = [NaiveDrift(), NaiveSeasonal(12)]

ensemble_model = NaiveEnsembleModel(models=models)

backtest = ensemble_model.historical_forecasts(
    series_air, start=0.6, forecast_horizon=3, verbose=True
)

print("MAPE = %.2f" % (mape(backtest, series_air)))
series_air.plot()
backtest.plot()
```

Пример использования RegressionEnsembleModel

```

from darts.models import RegressionEnsembleModel

models = [NaiveDrift(), NaiveSeasonal(12)]

ensemble_model = RegressionEnsembleModel(
    forecasting_models=models, regression_train_n_points=12
)

backtest = ensemble_model.historical_forecasts(
    series_air, start=0.6, forecast_horizon=3, verbose=True
)

print("MAPE = %.2f" % (mape(backtest, series_air)))
series_air.plot()
backtest.plot()

```

Еще можно посмотреть веса моделей в ансамбле

```

ensemble_model.fit(series_air)
ensemble_model.regression_model.model.coef_

```

1.1.16. Фильтрация. Фильтра Калмана

```

from darts.models import KalmanFilter

kf = KalmanFilter(dim_x=3)
kf.fit(train_air_scaled)
filtered_series = kf.filter(train_air_scaled, num_samples=100)

train_air_scaled.plot()
filtered_series.plot()

```

1.1.17. Гауссовские процессы

Еще Darts содержит GaussianProcessFilter, который может использоваться для вероятностного моделирования рядов

```

from darts.models import GaussianProcessFilter
from sklearn.gaussian_process.kernels import RBF

# create a series with holes:
values = train_air_scaled.values()
values[20:22] = np.nan
values[28:32] = np.nan
values[55:59] = np.nan
values[72:80] = np.nan
series_holes = TimeSeries.from_times_and_values(train_air_scaled.time_index, values)
series_holes.plot()

kernel = RBF()

gpf = GaussianProcessFilter(kernel=kernel, alpha=0.1, normalize_y=True)
filtered_series = gpf.filter(series_holes, num_samples=100)

filtered_series.plot()

```

1.2. TimeSeries

`TimeSeries` это главный класс в Darts. Этот класс представляет и одномерные и многомерные временные ряды. Временной индекс может быть как типа `pandas.DatatimeIndex` (содержит метки времени) или типа `pandas.RangeIndex` (содержит целочисленные значения; полезен для представления последовательности данных без временных меток). Класс `TimeSeries` может даже представлять вероятностные временные ряды, для получения доверительных интервалов.

Следует различать одномерные (univariate) и многомерные (multivariate) временные ряды:

- *Многомерные* временные ряды содержат несколько измерений (то есть несколько значений на один временной шаг); еще можно сказать, что многомерный временной ряд это пакет временных рядов, то есть несколько временных рядов, наложенных один на другой (рис. 4).
- *Одномерные* временные ряды содержат только одно измерение (то есть только одно скалярное значение на один временной шаг).

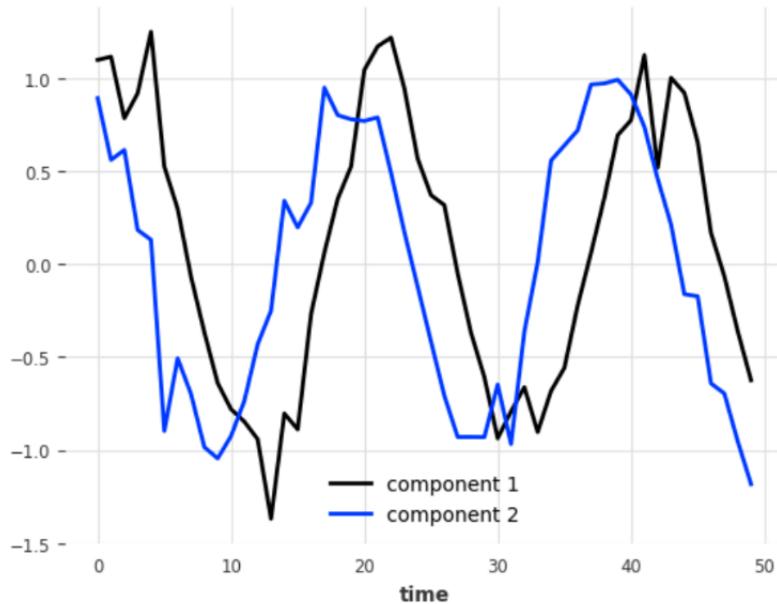


Рис. 4. Один экземпляр многомерного временного ряда. Один пакет временных рядов

Иногда измерения называют *компонентами*. Одиночный объект `TimeSeries` можно быть как одномерным (то есть содержать только одну компоненту) или многомерным (то есть содержать несколько компонент). В многомерном временном ряду, все компоненты разделяют одну и ту же временную ось, то есть они разделяют одни и те же временные шаги.

Некоторые модели в Darts (и все модели машинного обучения) поддерживают многомерные временные ряды.

Кроме того, некоторые модели могут работать с *множественными* (multiple) временными рядами, которые представляют собой последовательность нескольких объектов `TimeSeries` – например, простой список объектов `TimeSeries` (рис. 5). Множественные временные ряды еще называют *панельными данными*. Такие временные ряды не обязаны разделять один и тот же временной индекс. Например, один временной ряд может начинаться с 1990 года, а другой – с 2000. Они даже не обязаны иметь одну и ту же частоту.

Пример многомерного временного ряда: кровяное давление и ЧСС одного пациента (один многомерный ряд с 2 компонентами).

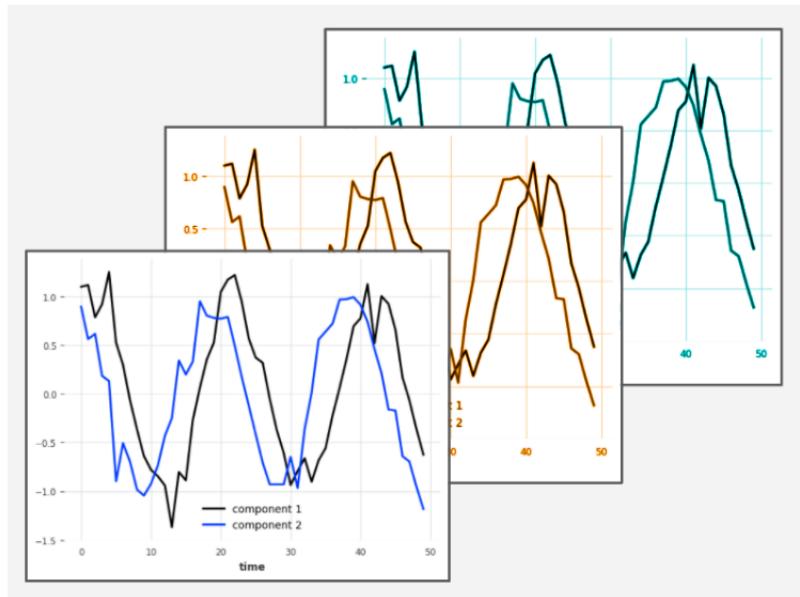


Рис. 5. Панельные данные. Множественные временные ряды – последовательность временных рядов. Они могут разделять одну и туже временную ось, а могут и не разделять. Могут быть многомерными, а могут и не быть

Пример панельного временного ряда: кровяное давление и ЧСС нескольких пациентов, измеренные в разное время (получается один многомерный ряд с 2 компонентами на одного пациента).

В Darts вероятностное прогнозирование осуществляется с помощью процедуры Монте-Карло. Этот подход позволяет представлять совместные распределения произвольной сложности (и по времени, и по компонентам) без ограничений параметрических моделей.

С помощью `concatenate()` можно объединять несколько временных рядов по различным осям. Ось `axis=0` объединяет по времени, ось `axis=1` – по компонентам, а ось `axis=2` – по стохастическим экземплярам.

Например

```
from darts import concatenate

my_multivariate_series = concatenate([series1, series2, ...], axis=1)
```

объединяет несколько временных рядов в один многомерный.

Опционально временные ряды могут содержать так называемые *статические ковариаты* (что-то вроде метаинформации). Например:

- Расположение хранилища,
- ID продукта,
- и пр.

1.3. Обзор моделей прогнозирования

Все модели прогнозирования устроены одинаково: создаем экземпляр класса модели, обучаем с помощью `fit()` и строим прогноз `predict()`

```
from darts.models import NaiveSeasonal

naive_model = NaiveSeasonal(K=1)          # init
naive_model.fit(train)                    # fit
naive_forecast = naive_model.predict(n=36) # predict
```

Аргумент `n` в `predict()` это горизонт прогноза (число точек, для которых нужно построить прогноз).

Сохранение и загрузка моделей

```
from darts.models import RegressionModel

model = RegressionModel(lags=4)

model.save("my_model.pkl")
model_loaded = RegressionModel.load("my_model.pkl")
```

С torch-моделями сохраняются и параметры модели и состояние обучения

```
from darts.models import NBEATSMModel

model = NBEATSMModel(input_chunk_length=24,
output_chunk_length=12)

model.save("my_model.pt")
model_loaded = NBEATSMModel.load("my_model.pt")
```

Пример работы с многомерными временными рядами (рис. 6)

```
import darts.utils.timeseries_generation as tg
from darts.models import KalmanForecaster
import matplotlib.pyplot as plt

series1 = tg.sine_timeseries(value_frequency=0.05, length=100) + 0.1 * tg.gaussian_timeseries(
    length=100)
series2 = tg.sine_timeseries(value_frequency=0.02, length=100) + 0.2 * tg.gaussian_timeseries(
    length=100)

multivariate_series = series1.stack(series2)

model = KalmanForecaster(dim_x=4)
model.fit(multivariate_series)
pred = model.predict(n=50, num_samples=100)

plt.figure(figsize=(8,6))
multivariate_series.plot(lw=3)
pred.plot(lw=3, label='forecast')
```

В нейросетевых моделях можно использовать *квантильную регрессию*. Это часто оказывается полезен на практике в случаях, когда нет уверенности в «истинном» распределении или когда параметрические модели дают слабые результаты

```
from darts.datasets import AirPassengersDataset
from darts import TimeSeries
from darts.models import TCNModel
from darts.dataprocessing.transformers import Scaler
from darts.utils.likelihood_models import QuantileRegression

series = AirPassengersDataset().load()
train, val = series[:-36], series[-36:]

scaler = Scaler()
train = scaler.fit_transform(train)
val = scaler.transform(val)
series = scaler.transform(series)
```

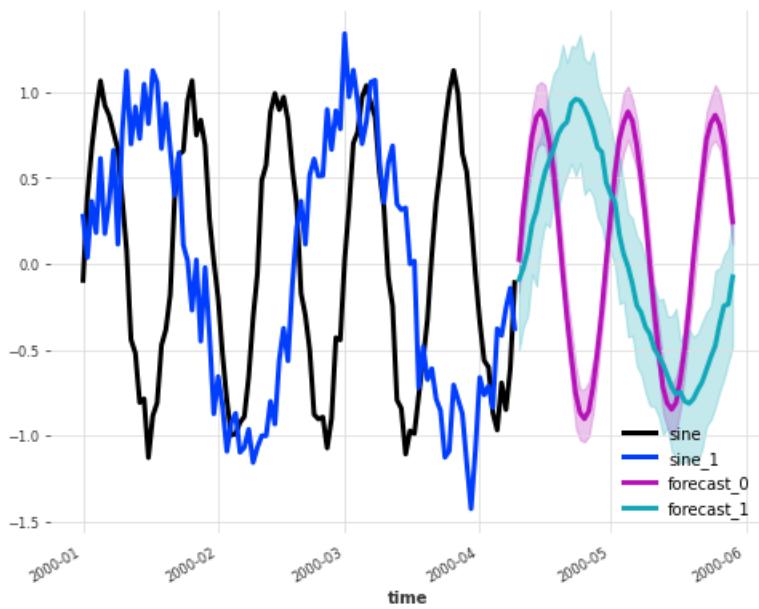


Рис. 6. Прогноз на многомерном временном ряду

```
model = TCNModel(
    input_chunk_length=30,
    output_chunk_length=12,
    likelihood=QuantileRegression(quantiles=[0.01, 0.05, 0.2, 0.5, 0.8, 0.95, 0.99])
)
model.fit(train, epochs=400)
pred = model.predict(n=36, num_samples=500)

series.plot()
pred.plot(label='forecast')
```

В Darts прореживание (dropout) это еще один способ учета неопределенности. Для примера обучим модель TCNModel (с квадратической функцией потерь) с 10%-ым прореживанием и построим прогноз с прореживанием по Монте-Карло

```
from darts.datasets import AirPassengersDataset
from darts import TimeSeries
from darts.models import TCNModel
from darts.dataprocessing.transformers import Scaler
from darts.utils.likelihood_models import QuantileRegression

series = AirPassengersDataset().load()
train, val = series[:-36], series[-36:]

scaler = Scaler()
train = scaler.fit_transform(train)
val = scaler.transform(val)
series = scaler.transform(series)

model = TCNModel(
    input_chunk_length=30,
    output_chunk_length=12,
    dropout=0.1
)
model.fit(train, epochs=400)
pred = model.predict(
    n=36,
```

```

        mc_dropout=True,
        num_samples=500
    )

series.plot()
pred.plot(label='forecast')

```

Некоторые регрессионные модели поддерживают вероятностное прогнозирование. У классов `LinearRegressionModel`, `LightGBMModel` и `XGBModel` есть аргумент `likelihood`. Если задать `"poisson"`, то модель будет обучаться распределению Пуассона, а если `"quantile"`, то модель будет использовать линейные потери (pinball loss) для квантильной регрессии

```

from darts.datasets import AirPassengersDataset
from darts import TimeSeries
from darts.models import LinearRegressionModel

series = AirPassengersDataset().load()
train, val = series[:-36], series[-36:]

model = LinearRegressionModel(
    lags=30,
    likelihood="quantile",
    quantiles=[0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95])
model.fit(train)
pred = model.predict(n=36, num_samples=500)

series.plot()
pred.plot(label='forecast')

```

1.4. Torch-ие модели

Каждый фрагмент (chunk) содержит входной фрагмент – представляет прошлое – и выходной фрагмент – представляет будущее

```

# model that looks 7 time steps back (past) and 1 time step ahead (future)
model = SomeTorchForecastingModel(
    input_chunk_length=7,
    output_chunk_length=1,
    **model_kwargs
)

```

Все Torch Forecast Models могут быть обучены на одиночном (`single`) или множественном (`multiple`) целевом временном ряде.

1.4.1. Как используются входные данные в TFM при обучении и построении прогноза

Пусть требуется построить модель прогнозирования уровня продаж мороженного. Тогда

- прошлые значения целевого временного ряда (`past target`): фактические продажи мороженого `ice_cream_sales`,
- будущие значения целевого временного ряда (`future target`): предсказанное значение продаж на следующий день,
- ковариаты «прошлого времени»: измеренная средняя дневная температура `temperature`,
- ковариаты «будущего времени»: день недели в прошлом и будущем `weekday`.

Допустим, что есть недельный паттерн в данных, поэтому `input_chunk_length=7`.

Тогда `output_chunk_length` можно задать равным 1, чтобы строить прогноз на следующий день

```
from darts.models import TFTModel

model = TFTModel(
    input_chunk_length=7,
    output_chunk_length=1,
)

model.fit(
    series=ice_cream_sales,
    past_covariates=temperature,
    future_covariates=weekday,
)
```

ВАЖНО! Длинный целевой ряд `target` может привести к очень большому числу обучающих последовательностей / экземпляров. Можно задать верхнюю границу для числа последовательностей / экземпляров с помощью `fit()`-параметра `max_samples_per_ts`

```
# fit only on the 10 "most recent" sequences
model.fit(target, max_samples_per_ts=10)
```

ВАЖНО! Точка начала каждой последовательности выбирается *случайно*.

After having completed computations on the first sequence, the model moves to the next one and performs the same training steps. The starting point of each sequence is selected randomly from the sequential dataset.

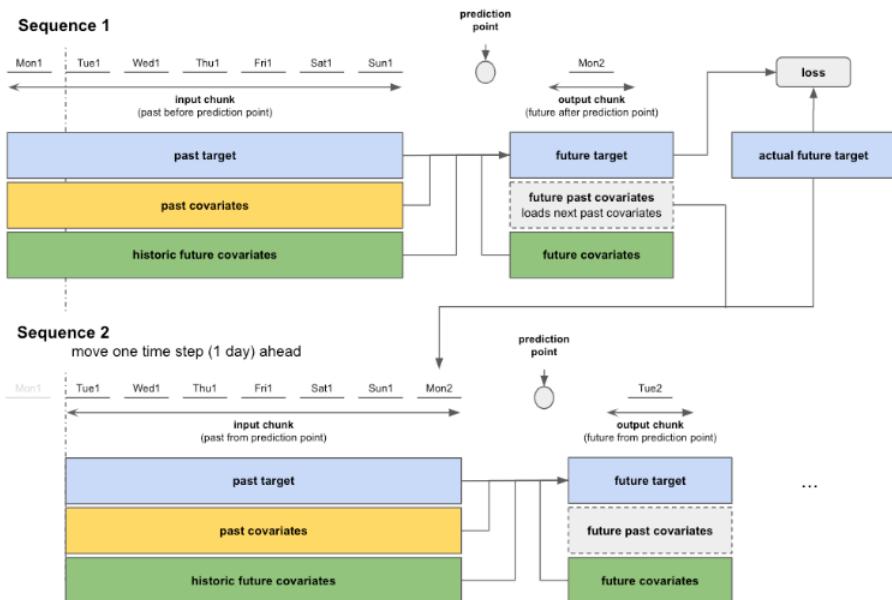


Figure 4: Sequence-to-sequence: Move to next sequence and repeat training steps

Рис. 7. Процедура обучения модели TFM

Еще можно обучать модель с валидационным поднабором

```
# create train and validation sets
ice_cream_sales_train, ice_cream_sales_val = ice_cream_sales.split_after(training_cutoff)

# train with validation set
model.fit(series=ice_cream_sales_train,
```

```

past_covariates=temperature,
future_covariates=weekday,
val_series=ice_cream_sales_val,
val_past_covariates=temperature,
val_future_covariates=weekday)

```

Модель будет обучаться как и раньше, но дополнительно еще будут вычисляться потери на валидационном поднаборе.

Следует различать два случая:

- Если $n \leq \text{output_chunk_length}$: мы можем предсказать n за один раз (используя один «внутренний вызов модели»); в данном примере $n = 1$ (рис. 8),
- Если $n > \text{output_chunk_length}$: мы можем предсказать n , вызвав внутреннюю модель несколько раз. Каждый вызов возвращает $\text{output_chunk_length}$ точек прогноза.

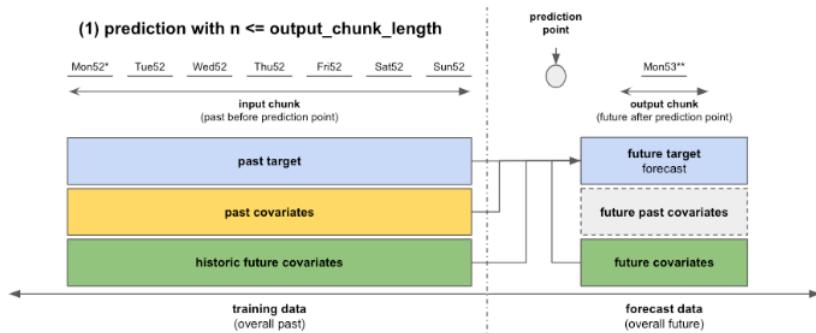
В данном примере $n = 3$. То есть чтобы построить прогноз, мы должны предоставить следующие $n - \text{output_chunk_length} = 2$ временные метки (дня). К сожалению, у нас нет измеренных значений температуры в будущем, но если у нас есть прогноз температуры на следующие 2 дня, то можно эти прогнозные точки учесть (рис. 9)

```

temperature = temperature.concatenate(
    temperature_forecast,
    axis=0
)

prediction = model.predict(
    n=n,
    past_covariates=temperature,
    future_covariates=weekday,
)

```



**Figure 5: Forecast with a single sequence for $n \leq \text{output_chunk_length}$ **

Рис. 8. Построение прогноза на одной последовательности для $n \leq \text{output_chunk_length}$

Для управления версиями модели можно использовать точки проверки (checkpoints)

```

model = SomeTorchForecastingModel(..., model_name='my_model', save_checkpoints=True)

# checkpoints are saved automatically
model.fit(...)

# load the model state that performed best on validation set
best_model = model.load_from_checkpoint(model_name='my_model', best=True)

```

Можно сохранить и загрузить модель вручную

```

model.save("/your/path/to/save/model.pt")
loaded_model = model.load("/your/path/to/save/model.pt")

```

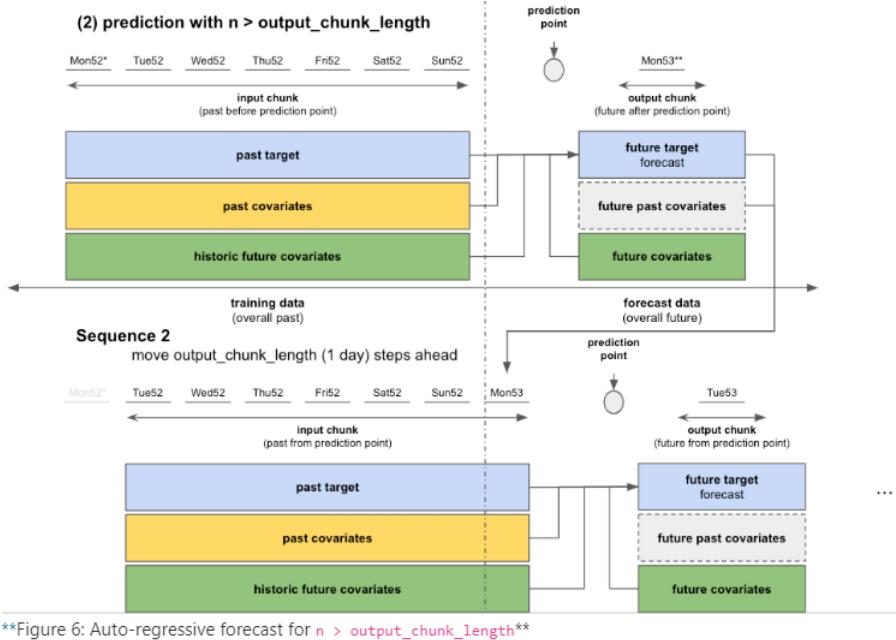


Рис. 9. Построение прогноза для $n > \text{output_chunk_length}$

Можно модель, обученную и сохраненную на GPU, загрузить на CPU

```
# define a model using gpu as accelerator
model = SomeTorchForecastingModel(...,
    model_name='my_model',
    save_checkpoints=True,
    pl_trainer_kwarg={

        "accelerator": "gpu",
        "devices": -1,
    })

# train the model, automatic checkpoints will be created
model.fit(...)

# specify the device to which the model should be loaded
loaded_model = SomeTorchForecastingModel.load_from_checkpoint(model_name='my_model',
    best=True,
    map_location="cpu")
loaded_model.to_cpu()

# run inference
loaded_model.predict(...)
```

Еще модель можно загрузить на CPU вручную

```
model.save("/your/path/to/save/model.pt")
loaded_model = model.load("/your/path/to/save/model.pt", map_location="cpu")
loaded_model.to_cpu()
```

1.4.2. Повторное обучение или подбор гиперпараметров предобученной модели

Для того чтобы повторно обучить или подобрать гиперпараметры модели с использованием различных оптимизаторов и / или планировщиков темпа обучения, можно загрузить веса из точек сохранения (checkpoints) в новую модель

```
# model with identical architecture but different optimizer (default: torch.optim.Adam)
```

```

model_finetune = SomeTorchForecastingModel(..., # use identical parameters & values as in
    original model
    optimizer_cls=torch.optim.SGD,
    optimizer_kwargs={"lr": 0.001})

# load the weights from a checkpoint
model_finetune.load_weights_from_checkpoint(model_name='my_model', best=True)

model_finetune.fit(...)

```

И аналогичным образом можно загрузить модель в ручную

```

# model with identical architecture but different lr scheduler (default: None)
model_finetune = SomeTorchForecastingModel(..., # use identical parameters & values as in
    original model
    lr_scheduler_cls=torch.optim.lr_scheduler.ExponentialLR,
    lr_scheduler_kwargs={"gamma": 0.09})

# load the weights from a manual save
model_finetune.load_weights("/your/path/to/save/model.pt")

```

1.4.3. Обратные вызовы

Обратные вызовы это очень мощный способ управления поведением модели во время обучения (мониторинг производительности модели, ранняя остановка и пр.).

Ранняя остановка это эффективный способ борьбы с переобучением (плюс снижается время обучения). Обучение прекращается, когда метрика качества на валидационном наборе перестает значимо улучшаться.

Раннюю остановку можно использовать с любой моделью TorchForecastingModel

```

import pandas as pd
from pytorch_lightning.callbacks import EarlyStopping
from torchmetrics import MeanAbsolutePercentageError

from darts.dataprocessing.transformers import Scaler
from darts.datasets import AirPassengersDataset
from darts.models import NBEATSModel

# read data
series = AirPassengersDataset().load()

# create training and validation sets:
train, val = series.split_after(pd.Timestamp(year=1957, month=12, day=1))

# normalize the time series
transformer = Scaler()
train = transformer.fit_transform(train)
val = transformer.transform(val)

# any TorchMetric or val_loss can be used as the monitor
torch_metrics = MeanAbsolutePercentageError()

# early stop callback
my_stopper = EarlyStopping(
    monitor="val_MeanAbsolutePercentageError", # "val_loss",
    patience=5,
    min_delta=0.05,
    mode='min',
)

```

```

)
pl_trainer_kwarg = { "callbacks": [my_stopper]}

# create the model
model = NBEATSMModel(
    input_chunk_length=24,
    output_chunk_length=12,
    n_epochs=500,
    torch_metrics=torch_metrics,
    pl_trainer_kwarg=pl_trainer_kwarg)

# use validation set for early stopping
model.fit(
    series=train,
    val_series=val,
)

```

Потери на обучающем и валидационном поднаборе могут автоматически логироваться с помощью tensorboard. После активации, Darts будет по умолчанию писать логи в директорию `darts_logs` в текущей директории проекта. После установки библиотеки `tensorboard` можно визуализировать логи из командной строки

```
$ tensorboard --log_dir darts_logs
```

Можно реализовать свой собственный обратный вызов

```

from pytorch_lightning.callbacks import Callback

class LossLogger(Callback):
    def __init__(self):
        self.train_loss = []
        self.val_loss = []

    # will automatically be called at the end of each epoch
    def on_train_epoch_end(self, trainer: "pl.Trainer", pl_module: "pl.LightningModule") -> None:
        self.train_loss.append(float(trainer.callback_metrics["train_loss"]))

    def on_validation_epoch_end(self, trainer: "pl.Trainer", pl_module: "pl.LightningModule") ->
        None:
        self.val_loss.append(float(trainer.callback_metrics["val_loss"]))

loss_logger = LossLogger()

model = SomeTorchForecastingModel(
    ...,
    nr_epochs_val_period=1, # perform validation after every epoch
    pl_trainer_kwarg={"callbacks": [loss_logger]}
)

# fit must include validation set for "val_loss"
model.fit(...)

```

1.4.4. Рекомендации по производительности

- о Экземпляр класса `TimeSeries` лучше строить в 32-битном представлении. Большой прирост производительности можно получить, когда и данные, и модель имеют 32-битное представ-

ление. Можно построить экземпляр `TimeSeries` на базе массива с типом `np.float32`. Или можно просто `my_series32 = my_series.astype(np.float32)`.

- Использование GPU часто повышает производительность по сравнению с CPU.
- Большой размер пакета ускоряет обучение поскольку сокращает количество обратных проходов за эпоху и позволяет лучше распараллелить вычисления. Однако, больший размер пакета отвечает и большему потреблению памяти. Нужно подбирать размер пакета.
- Имеет смысл подобрать значение параметра `num_loader_workers`. Все глубокие модели в Darts имеют параметр `num_loader_workers` в методах `fit()` и `predict()`, которые конфигурируют `num_workers` в PyTorch `DataLoaders`. По умолчанию этот параметр выставлен в 0, что означает, что главный процесс позаботится и о загрузке данных. Если выставить `num_workers > 0`, то будут использоваться дополнительные воркеры. Это повышает потребление памяти, но в некоторых случаях может повысить и производительность. Оптимальное значение зависит от многих факторов: размер пакета, используется ли GPU и пр.
- Разумеется на производительность модели сильное влияние оказывает ее размер (количество параметров) и количество операций требуемых для прямого и обратного проходов. Модели в Darts можно настраивать (число слоев, число «голов внимания» и пр.) и эти гиперпараметры оказывают сильное влияние на производительность. Рекомендуется начинать с компактных моделей.
- Полезно заранее загрузить все объекты `TimeSeries` в память, если это возможно. Darts позволяет обучать модель на любой `Sequence[TimeSeries]`, что означает, что на больших наборах можно написать свою собственную реализацию `Sequence` и читать временные ряды с диска *лениво*. Рекомендуется при обучении модели на панельных временных рядах (`multiple series`), сначала попробовать обучить модель на простом списке временных рядов `List[Sequence]`.
- Не используйте *все* доступные фрагменты (sub-series) временного ряда для обучения. По умолчанию, при вызове `fit()` модели в Darts строят экземпляр `TrainingDataset`, согласованный с используемой моделью (например, `PastCovariatesTorchModel`, `FutureCovariatesTorchModel` и т.д.). По умолчанию, эти наборы данных часто будут содержать *все* возможные фрагменты (и для входа, и для выхода) в каждом объекте `TimeSeires`. Если объект `TimeSeries` длинный, то в результате получиться очень большое число обучающих экземпляров, которое линейно влияет на время обучения модели (на одну эпоху). Чтобы ограничить этот эффект можно:
 - Задать `max_samples_per_ts` в `fit()`. Тогда для обучения модели будут использоваться только последние `max_samples_per_ts` экземпляров для каждого объекта `TimeSeries`.
 - Если этот параметр не помогает, то можно реализовать свой собственный `TrainingDataset` и определить как будет разбиваться объект `TimeSeries`.

Например в наборе данных `darts.datasets.EnergyDataset` 28 050 временных меток и 28 измерений. Если задать `input_chunk_length=48` и `output_chunk_length=12`, получится 27 991 обучающий экземпляр.

1.5. Ковариаты

В Darts *ковариаты* относятся к внешним данным, которые могут быть использованы моделью для улучшения прогноза. В контексте прогнозирования модели целевой временной ряд предсказывается, а ковариаты нет. Различают следующие типы ковариатов:

- ковариаты «прошлого времени» (past covariates): известны только в прошлом (например, измерения какой-то величины),
- ковариаты «будущего времени» (future covariates): известны в будущем (например, прогноз погоды),
- статические ковариаты: не изменяются во времени (содержат не изменяющуюся во времени информацию о целевом временном ряду).

```
# create one of Darts' forecasting models
model = SomeForecastingModel(...)

# fitting model with past and future covariates
model.fit(target=target,
           past_covariates=past_covariates_train,
           future_covariates=future_covariates_train)

# predict the next n=12 steps
model.predict(n=12,
              series=target, # only required for Global Forecasting Models
              past_covariates=past_covariates_pred,
              future_covariates=future_covariates_pred)
```

Если модель обучается с использованием `past_covariates`, то придется эти ковариаты `past_covariates` передать и на шаге построения прогноза. То же самое касается и `future_covariates` с тем отличием, что ковариаты прошлого времени должны иметь по крайней мере те же временные метки, что и у целевого временного ряда, а ковариаты будущего времени должны иметь те же метки, что и у целевого временного ряда + `n` шагов в будущее (рис. 10).

You can use the same ``*_covariates`` for both training and prediction, given that they contain the required time spans.

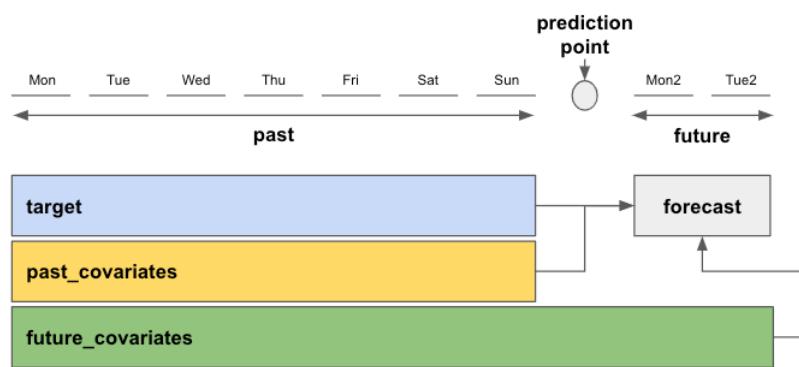


Рис. 10. Пояснения к построению прогноза на горизонт $n=2$

Примеры:

- `past_covariates`: обычно это какие-то измерения или временные атрибуты:
 - среднесуточная температура (измеренная и известная только в прошлом),
 - день недели, месяц, год.

- `future_covariates`: обычно это прогнозы (известные в будущем) или временные атрибуты:
 - среднесуточная температура (*спрогнозированная* и известная в будущем),
 - день недели, месяц, год.
- `static_covariates`: не зависимые от времени/постоянные/статичные характеристики целевого временного ряда:
 - категориальные: место размещения целевого ряда (страна, город и т.д.),
 - числовые: население страны, для которой записывается целевой временной ряд,
 - средняя температура региона, для которого записывается целевой временной ряд.

Временные атрибуты очень полезны, так как помогают модели уловить сезонные паттерны. Эмпирическое правило, позволяющее узнать какие ковариаты должны использоваться: если значения известны заранее, то они могут быть как ковариатами будущего времени, так и ковариатами прошлого времени; а если значения заранее не известны, то они могут быть только ковариатами прошлого времени.

Локальные модели (Local Forecast Models, LFMs) обычно обучаются на *полном* наборе данных `target` и `future_covariates` (если поддерживаются).

Глобальные модели (Global Forecast Models, GFMs) обучаются и предсказывают на *фрагментах фиксированной длины* (fixed-length chunks) целевого временного ряда `target` и `*_covariates` (если поддерживаются).

1.6. Подбор гиперпараметров в Darts

Когда дело касается подбора гиперпараметров, то в Darts нет ничего особенного.

1.6.1. Подбор гиперпараметров с помощью Optuna

```
import numpy as np
import optuna
import torch
from optuna.integration import PyTorchLightningPruningCallback
from pytorch_lightning.callbacks import EarlyStopping
from sklearn.preprocessing import MaxAbsScaler

from darts.dataprocessing.transformers import Scale
from darts.datasets import AirPassengersDataset
from darts.metrics import smape
from darts.models import TCNModel
from darts.utils.likelihood_models import GaussianLikelihood

# load data
series = AirPassengersDataset().load().astype(np.float32)

# split in train / validation (note: in practice we would also need a test set)
VAL_LEN = 36
train, val = series[:-VAL_LEN], series[-VAL_LEN:]

# scale
scaler = Scaler(MaxAbsScaler())
train = scaler.fit_transform(train)
val = scaler.transform(val)
```

```

# define objective function
def objective(trial):
    # select input and output chunk lengths
    in_len = trial.suggest_int("in_len", 12, 36)
    out_len = trial.suggest_int("out_len", 1, in_len-1)

    # Other hyperparameters
    kernel_size = trial.suggest_int("kernel_size", 2, 5)
    num_filters = trial.suggest_int("num_filters", 1, 5)
    weight_norm = trial.suggest_categorical("weight_norm", [False, True])
    dilation_base = trial.suggest_int("dilation_base", 2, 4)
    dropout = trial.suggest_float("dropout", 0.0, 0.4)
    lr = trial.suggest_float("lr", 5e-5, 1e-3, log=True)
    include_year = trial.suggest_categorical("year", [False, True])

    # throughout training we'll monitor the validation loss for both pruning and early stopping
    pruner = PyTorchLightningPruningCallback(trial, monitor="val_loss")
    early_stopper = EarlyStopping("val_loss", min_delta=0.001, patience=3, verbose=True)
    callbacks = [pruner, early_stopper]

    # detect if a GPU is available
    if torch.cuda.is_available():
        pl_trainer_kwargs = {
            "accelerator": "gpu",
            "gpus": -1,
            "auto_select_gpus": True,
            "callbacks": callbacks,
        }
        num_workers = 4
    else:
        pl_trainer_kwargs = {"callbacks": callbacks}
        num_workers = 0

    # optionally also add the (scaled) year value as a past covariate
    if include_year:
        encoders = {"datetime_attribute": {"past": ["year"]},
                    "transformer": Scaler()}
    else:
        encoders = None

    # reproducibility
    torch.manual_seed(42)

    # build the TCN model
    model = TCNModel(
        input_chunk_length=in_len,
        output_chunk_length=out_len,
        batch_size=32,
        n_epochs=100,
        nr_epochs_val_period=1,
        kernel_size=kernel_size,
        num_filters=num_filters,
        weight_norm=weight_norm,
        dilation_base=dilation_base,
        dropout=dropout,
        optimizer_kwargs={"lr": lr},
        add_encoders=encoders,
        likelihood=GaussianLikelihood(),
        pl_trainer_kwargs=pl_trainer_kwargs,
        model_name="tcn_model",
    )

```

```

        force_reset=True,
        save_checkpoints=True,
    )

# when validating during training, we can use a slightly longer validation
# set which also contains the first input_chunk_length time steps
model_val_set = scaler.transform(series[-(VAL_LEN + in_len) :])

# train the model
model.fit(
    series=train,
    val_series=model_val_set,
    num_loader_workers=num_workers,
)

# reload best model over course of training
model = TCNModel.load_from_checkpoint("tcn_model")

# Evaluate how good it is on the validation set, using sMAPE
preds = model.predict(series=train, n=val_len)
smapes = smape(val, preds, n_jobs=-1, verbose=True)
smape_val = np.mean(smaps)

return smape_val if smape_val != np.nan else float("inf")

# for convenience, print some optimization trials information
def print_callback(study, trial):
    print(f"Current value: {trial.value}, Current params: {trial.params}")
    print(f"Best value: {study.best_value}, Best params: {study.best_trial.params}")

# optimize hyperparameters by minimizing the sMAPE on the validation set
study = optuna.create_study(direction="minimize")
study.optimize(objective, n_trials=100, callbacks=[print_callback])

```

1.6.2. Подбор гиперпараметров с помощью Ray Tune

```

import pandas as pd
from pytorch_lightning.callbacks import EarlyStopping
from ray import tune
from ray.tune import CLIReporter
from ray.tune.integration.pytorch_lightning import TuneReportCallback
from ray.tune.schedulers import ASHAScheduler
from torchmetrics import MeanAbsoluteError, MeanAbsolutePercentageError, MetricCollection

from darts.dataprocessing.transformers import Scaler
from darts.datasets import AirPassengersDataset
from darts.models import NBEATSMModel

def train_model(model_args, callbacks, train, val):
    torch_metrics = MetricCollection([MeanAbsolutePercentageError(), MeanAbsoluteError()])
# Create the model using model_args from Ray Tune
model = NBEATSMModel(
    input_chunk_length=24,
    output_chunk_length=12,
    n_epochs=500,
    torch_metrics=torch_metrics,
)

```

```

pl_trainer_kwargs={"callbacks": callbacks, "enable_progress_bar": False},
**model_args)

model.fit(
    series=train,
    val_series=val,
)

# Read data:
series = AirPassengersDataset().load()

# Create training and validation sets:
train, val = series.split_after(pd.Timestamp(year=1957, month=12, day=1))

# Normalize the time series (note: we avoid fitting the transformer on the validation set)
transformer = Scaler()
transformer.fit(train)
train = transformer.transform(train)
val = transformer.transform(val)

# Early stop callback
my_stopper = EarlyStopping(
    monitor="val_MeanAbsolutePercentageError",
    patience=5,
    min_delta=0.05,
    mode='min',
)

# set up ray tune callback
tune_callback = TuneReportCallback(
{
    "loss": "val_Loss",
    "MAPE": "val_MeanAbsolutePercentageError",
},
on="validation_end",
)

# define the hyperparameter space
config = {
    "batch_size": tune.choice([16, 32, 64, 128]),
    "num_blocks": tune.choice([1, 2, 3, 4, 5]),
    "num_stacks": tune.choice([32, 64, 128]),
    "dropout": tune.uniform(0, 0.2),
}

reporter = CLIReporter(
    parameter_columns=list(config.keys()),
    metric_columns=["loss", "MAPE", "training_iteration"],
)

resources_per_trial = {"cpu": 8, "gpu": 1}

# the number of combinations to try
num_samples = 10

scheduler = ASHAScheduler(max_t=1000, grace_period=3, reduction_factor=2)

train_fn_with_parameters = tune.with_parameters(
    train_model, callbacks=[my_stopper, tune_callback], train=train, val=val,
)

```

```

# optimize hyperparameters by minimizing the MAPE on the validation set
analysis = tune.run(
    train_fn_with_parameters,
    resources_per_trial=resources_per_trial,
    # Using a metric instead of loss allows for
    # comparison between different likelihood or loss functions.
    metric="MAPE", # any value in TuneReportCallback.
    mode="min",
    config=config,
    num_samples=num_samples,
    scheduler=scheduler,
    progress_reporter=reporter,
    name="tune_darts",
)
print("Best hyperparameters found were: ", analysis.best_config)

```

2. XGBoost

Awesome XGBoost <https://github.com/dmlc/xgboost/tree/master/demo>

2.1. Установка

Установить библиотеку можно с помощью менеджера пакетов pip

```
pip install xgboost
```

2.2. Вводные замечания

XGBoost пытается строить *полные бинарные деревья в глубину* (*depth/level-wise*), а LightGBM строит деревья *по листьям* (*leaf-wise*) (*несимметричные деревья*) с ограничением числа терминальных узлов (*num_leaves*).

Целевая функция в градиентном бустинге – это *потери на обучающем наборе данных + регуляризация*

$$obj = L(\theta) + \Omega(\theta),$$

где L – функция потерь на обучающем наборе данных, Ω – член регуляризации.

Перепишем целевую функцию

$$obj = \sum_i^n l(y_i, \hat{y}_i) + \sum_{k=1}^K w(f_k),$$

где $w(f_k)$ – сложность дерева f_k , K – число деревьев.

В XGBoost *сложность дерева* $w(f)$ (complexity of the tree) определяется как

$$w(f) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2,$$

где w_j – вес листа в дереве f .

То есть здесь штрафуется сложность разбиения пространства (число листьев в дереве) и большие веса листьев.

Разумеется, существует много способов определить сложность дерева, но вариант, принятый в XGBoost работает на практике как правило хорошо.

2.3. Оценка структуры

Целевая функция для t -ой итерации градиентного бустинга (шаг добавления в ансамбль t -ого базового алгоритма) запишется в виде

$$\begin{aligned} obj^{(t)} &\approx \sum_{i=1}^n \left[g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2 \right] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \\ &= \sum_{j=1}^T \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T, \end{aligned}$$

где $I_j = \{i | q(x_i) = j\}$ – множество индексов экземпляров обучающего набора, попавших в j -ый лист, T – число листьев в дереве.

Во второй строке мы изменяем индекс суммирования, потому что все экземпляры обучающего набора данных, попавшие в один и тот же лист имеют одну и ту же оценку.

Это соотношение можно переписать в еще более компактной форме

$$obj^{(t)} = \sum_{j=1}^T \left[G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2 \right] + \gamma T,$$

где $G_j = \sum_{i \in I_j} g_i$ – сумма градиентов по экземплярам обучающего набора данных, попавших в один лист, $H_j = \sum_{i \in I_j} h_i$ – сумма гессианов по экземплярам обучающего набора данных, попавшим в один лист.

Можно переписать так

$$\begin{aligned} w_j^* &= -\frac{G_j}{H_j + \lambda}, \\ obj^* &= -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T \end{aligned}$$

Последнее соотношение оценивает *качество структуры дерева* (чем меньше оценка, тем лучше структура дерева). Теперь у нас есть оценка качества дерева и в идеале мы могли бы рассмотреть все возможные варианты деревьев и выбрать лучшее. Но на практике это невозможно, поэтому мы оптимизируем одно разбиение в дереве за раз.

То есть мы пытаемся разбить родительский узел на два дочерних и оценить информационный прирост (gain)

$$Gain = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma.$$

Эту формулу можно разбить на:

1. оценку нового левого листа,
2. оценку нового правого листа,

3. оценку родительского листа,
4. регуляризацию.

Выходит, что если выражение в квадратных скобках меньше γ , то лучше ветку не добавлять. Это в чистом виде *техника подрезки* (pruning).

Таким образом, дерево строится *жадно* с максимизацией информационного прироста *Gain* для каждого разбиения. В «обычном» градиентном бустинге дерево тоже строится жадно и с максимизацией информационного прироста относительно разброса вокруг среднего.

2.4. Введение в Model IO

В XGBoost 1.0.0 появилась поддержка JSON для сохранения / загрузки моделей XGBoost и связанных с ними параметров. Позже в XGBoost 1.6.0 появилась поддержка [Universal Binary JSON](#). Далее для сохранения / загрузки моделей будут использоваться два формата: `.json` и `.ubj` для бинарного JSON.

Для того чтобы сохранить модель (сохраняются только деревья и целевая функция), нужно в качестве расширения файла указать `.json` или `.ubj`

```
bst.save_model("model_file_name.ubj")
```

ВАЖНО: формат `pickle` не стабильный формат сериализации и не работает ни в разных версиях Python, ни в разных версиях XGBoost.

Для длительного хранения модели следует сохранять модель с помощью метода `save_model`. Метод `dump_model` предназначен исключительно для интерпретации и визуализации модели и не предполагает, что модель будет обратно загружена в XGBoost.

2.5. DART booster

<https://xgboost.readthedocs.io/en/latest/tutorials/dart.html>

XGBoost обычно состоит из огромного числа *регрессионных деревьев* с малым темпом обучения. В таком случае деревья добавленные на раннем этапе имеют большее значение, чем деревья добавленные позже.

Виньяк (Vinayak) и Гилад-Бачрач (Gilad-Bachrach) [7] предложили новый метод прореживания (dropout), который часто дает хорошие результаты – DART.

Однако, из-за внесения случайности в процедуру обучения надо иметь в виду, что:

- обучение DART-деревьев может быть медленнее GBTREE-деревьев,
- ранняя остановка может быть нестабильна.

2.6. Ограничения на монотонность

<https://xgboost.readthedocs.io/en/latest/tutorials/monotonic.html>

Если есть сильная априорная гипотеза о монотонной связи признака и целевой переменной, то эту информацию можно учесть с помощью ограничений на монотонность.

2.7. Ограничение на взаимодействие признаков

Деревья решений это мощный инструмент исследования взаимодействий между *независимыми признаками*. Признаки, которые встречаются на пути обхода дерева от корня к терминалльному листу, взаимодействуют друг с другом.

Когда глубина дерева больше 1, многие признаки взаимодействуют только для того, чтобы минимизировать ошибку на обучающем наборе данных и потому решающие деревья могут выявлять ложные зависимости (шум). Ограничение на взаимодействие признаков позволяет определить разрешенные взаимодействия признаков.

Преимущество использования:

- Более высокое качество модели за счет фокусирования внимания на взаимодействиях, которые работают – будь то с помощью знаний о предметной области или еще как-то,
- Меньше шума в прогнозах, более высокая обобщаемость,
- Больше контроля над моделью

Ограничения на взаимодействия признаков выражаются в терминах групп признаков, которым разрешено взаимодействовать. Например, $[[0, 1], [2, 3, 4]]$ признакам x_0, x_1 разрешено взаимодействовать друг с другом, а признакам x_2, x_3, x_4 соответственно между собой, но взаимодействие признаков из этих двух групп запрещено.

2.8. Пользовательские целевые функции и метрики качества

Корректная целевая функция должна принимать два входа: прогноз и метки. Например, для квадратической логарифмической ошибки (Squared Log Error, SLE) $1/2[\log(pred+1)-\log(label+1)]^2$

```
import numpy as np
import xgboost as xgb
import typing as t

def gradient(predt: np.ndarray, dtrain: xgb.DMatrix) -> np.ndarray:
    y = dtrain.get_label()

    return (np.log1p(predt) - np.log1p(y)) / (predt + 1)

def hessian(predt: np.ndarray, dtrain: xgb.DMatrix) -> np.ndarray:
    y = dtrain.get_label()

    return ((-np.log1p(predt) + np.log1p(y) + 1) / np.power(predt + 1, 2))

def squared_log(predt: np.ndarray, dtrain: xgb.DMatrix) -> t.Tuple[np.ndarray, np.ndarray]:
    predt[predt < 1] = -1 + 1e-6
    grad = gradient(predt, dtrain)
    hess = hessian(predt, dtrain)

    return grad, hess
```

Теперь эту целевую функцию можно передать в качестве обратного вызова `xgb.train`

```
xgb.train(
{
    "tree_method": "hist",
    "seed": 1994,
},
dtrain = dtrain,
num_boost_round = 10,
obj = squared_log # <- NB!
)
```

Метрикой по умолчанию для целевой функции SLE будет RMSLE. Таким образом мы снова определяем обратный вызов

```

def rmsle(predt: np.ndarray, dtrain: xgb.DMatrix) -> t.Tuple[str, float]:
    y = dtrain.get_label()
    predt[predt < -1] = -1 + 1e-6
    elements = np.power(np.log1p(y) - np.log1p(predt), 2)

    return "PyRMSLE", float(np.sqrt(np.sum(elements) / len(y)))

```

Теперь пользовательскую метрку качества можно передать аргументу `custom_metric` (аргумент `feval` отменен)

```

dtrain = xgb.DMatrix(X_train, y_train)
dtest = xgb.DMatrix(X_test)

xgb.train(
{
    "tree_method": "hist",
    "seed": 1994,
    "disable_default_eval_metric": 1,
},
dtrain = dtrain,
num_boost_round = 10,
obj = squared_log,
custom_metric = rmsle, # <- NB
evals = [(dtrain, "dtrain"), (dtest, "dtest")],
evals_result=results
)
[0]      dtrain-rmse:0.31750      dtrain-PyRMSLE:0.21845  dtest-rmse:0.33568  dtest-PyRMSLE
       :0.22994
[1]      dtrain-rmse:0.24315      dtrain-PyRMSLE:0.15925  dtest-rmse:0.26395  dtest-PyRMSLE
       :0.17356
[2]      dtrain-rmse:0.18940      dtrain-PyRMSLE:0.11989  dtest-rmse:0.21486  dtest-PyRMSLE
       :0.13937
[3]      dtrain-rmse:0.15031      dtrain-PyRMSLE:0.09275  dtest-rmse:0.18768  dtest-PyRMSLE
       :0.12227
[4]      dtrain-rmse:0.11876      dtrain-PyRMSLE:0.07169  dtest-rmse:0.16087  dtest-PyRMSLE
       :0.10648
[5]      dtrain-rmse:0.09478      dtrain-PyRMSLE:0.05636  dtest-rmse:0.14231  dtest-PyRMSLE
       :0.09637
[6]      dtrain-rmse:0.07597      dtrain-PyRMSLE:0.04466  dtest-rmse:0.13128  dtest-PyRMSLE
       :0.09147
[7]      dtrain-rmse:0.06306      dtrain-PyRMSLE:0.03678  dtest-rmse:0.12255  dtest-PyRMSLE
       :0.08764
[8]      dtrain-rmse:0.05368      dtrain-PyRMSLE:0.03121  dtest-rmse:0.11661  dtest-PyRMSLE
       :0.08521
[9]      dtrain-rmse:0.04541      dtrain-PyRMSLE:0.02627  dtest-rmse:0.11207  dtest-PyRMSLE
       :0.08318

```

2.9. Категориальные признаки

Разбиение по числовым признакам строится как $value < threshold$. Для категориальных признаков разбиение зависит от того как представляется категориальный признак: в виде секций или в виде результата кодирования с одним активным состоянием. В случае секционирования разбиение строится как $value \in categories$, где `categories` это множество уникальных значений признака. Если используется техника кодирования одного активного состояния, то разбиение строится как $value == category$.

Простейший способ сообщить XGBoost о категориальных признаках – это использовать pandas-типа category и scikit-learn интерфейс

```
# Все категориальные признаки привести к типу category
X[ "cat_feature" ] = X[ "cat_feature" ].astype( "category" )

# И включить флаг enable_categorical
clf = xgb.XGBClassifier(
    tree_method="hist",
    enable_categorical=True # <- NB
)
clf.fit(X, y)
# Must use JSON/UBJSON for serialization, otherwise the information is lost
clf.save_model("categorical-model.ubj")
```

В XGBoost есть параметр max_cat_to_onehot, который управляет стратегией кодирования. Либо используется техника кодирования с одним активным состоянием, либо секционирование.

Sklearn-интерфейс удобный, приятный, но реализует не все возможности нативного интерфейса

```
Xy = xgb.DMatrix(X, y, enable_categorical=True)
booster = xgb.train({ "tree_method": "hist", "max_cat_to_onehot": 5}, Xy)
# categorical features are listed as "c"
booster.feature_types # ["float", ..., "c"]
booster.save_model("categorical-model.json")
```

Чтобы посчитать значения Шепли нужно

```
SHAP = booster.predict(Xy, pred_interactions=True)
```

2.10. Замечания о подборе гиперпараметров в XGBoost

Если качество на обучающем наборе значительно выше чем тестовом, то очень вероятно, что модель переобучилась.

Есть два способа управлять переобучением:

1. Напрямую управлять сложностью модели:
 - max_depth, min_child_weight и gamma,
2. Добавить случайность, чтобы сделать обучение устойчивым к шуму (а более устойчивая к шуму модель, это модель с меньшей дисперсией):
 - subsample, colsample_bytree,
 - еще можно уменьшить темп обучения eta (тогда нужно будет увеличить число итераций градиентного бустинга num_round, так как эти параметры сильно связаны и подбираются в паре).

Рекомендации по значениям гиперпараметров XGBoost:

- gamma [0, +∞): от 0 до ≈ 1000 в логарифмическом масштабе,
- alpha: параметр L_2 -регуляризации, от 0 до ≈ 1000 в логарифмическом масштабе,
- lambda: параметр L_1 -регуляризации, от 0 до ≈ 1000 в логарифмическом масштабе,
- max_depth [0, +∞): от 1 до 15-20 в линейном масштабе,
- min_child_weight [0, +∞): от 0 до 15-20 линейном масштабе,

- `learning_rate` [0, 1]: 0.0001, 0.001, 0.01, 0.1 и 1.0 (температура обучения ограничивает вклад каждого базового алгоритма в ансамбль, поэтому разумно ограничиться единичкой справа),
- `subsample` (0, 1]: от 0.5 до 1.0 в линейном масштабе,
- `colsample_bytree` (0, 1]: от 0.5 до 1.0 в линейной масштабе; полезен для больших наборов данных или когда есть проблема *мультиколлинеарности*,
- `colsample_bylevel` & `colsample_bynode` (0, 1]: от 0.5 до 1.0 в линейной масштабе; полезен для наборов данных с *большим числом сильно коррелированных признаков*.

Чтобы увеличить скорость обучения модели, можно параметру `tree_method` передать значение `hist` или `gpu_hist`.

Что касается дисбаланса, то:

- Если интересует только значение метрики качества самое по себе, то можно положительные и отрицательные экземпляры можно сбалансировать с помощью параметра `scale_pos_weight`,
 - Если же интересуют правильные оценки вероятности, то перебалансировку делать нельзя!!!
- Можно задать параметр `max_delta_step` для того, чтобы помочь решению сойтись.

2.11. Параметры XGBoost

<https://xgboost.readthedocs.io/en/latest/parameter.html#cat-param>

Перед запуском XGBoost мы должны задать параметры трех типов:

- общие параметры,
- параметры бустеров,
- параметры задачи.

2.11.1. Общие параметры

- `booster`: определяет какой бустер будет использоваться. Допустимые значения: `gbtree`, `gblinear` или `dart`.

2.11.2. Параметры бустера

- `eta` (он же `learning_rate`): темп обучения. Уменьшает вклад базовых алгоритмов в ансамбль и тем самым делает модель менее склонной к переобучению.
- `gamma` (он же `min_split_loss`): управляет сложностью пространства. Чем больше `gamma`, тем более консервативным (то есть более простым) будет алгоритм.
- `max_depth`: максимальная глубина дерева. Большие значения этого параметра делают модель более сложной и повышают вероятность переобучения.
- `min_child_weight`: минимальный суммарный вес экземпляров (гессианов). Если сумма весов экземпляров не превышает `min_child_weight`, то узел не будет разбиваться, а значит не будет новых листьев. Чем больше `min_child_weight`, тем более консервативным (более простым) будет алгоритм. Другими словами, чем больше `min_child_weight`, тем меньше листьев у дерева, то есть тем проще пространство.
- `max_delta_step`: максимальный шаг, который мы допускаем для выхода листа. Если задать положительное значение, это поможет сделать шаг более консервативным. Обычно этот параметр не нужен, но может помочь в логистической регрессии, когда классы несбалансированы.

- **subsample**: доля экземпляров обучающего набора данных, которые принимают участие в процедуре обучения. Если взять, например, долю 0.5, то это означит, что XGBoost будет выращивать деревья на половине случайно выбранных экземпляров обучающего набора. Это позволит сделать модель менее склонной к переобучению. Случайная выборка будет выполняться на каждой итерации градиентного бустинга. Обычно **subsample** ≥ 0.5 .
- **sampling_method**: метод выбора экземпляров обучающего набора данных. Возможны два значения – **uniform** и **gradient_based** (поддерживается только если **tree_method=gpu_hist**)
- **colsample_bytree**, **colsample_bylevel**, **colsample_bynode**: это семейство параметров подвыборки по признакам.
- **lambda** (он же **reg_lambda**): L_2 -штраф. Большие значения этого параметра отвечают более сильной регуляризации, то есть модель будет более консервативной (более простой).
- **alpha** (он же **reg_alpha**): L_1 -штраф. Большие значения этого параметра отвечают более сильной регуляризации, то есть модель будет более консервативной (более простой).
- **tree_method**: метод построения деревьев в XGBoost. Поддерживаются следующие значения: **auto**, **exact**, **approx**, **hist** и **gpu_hist**. Для больших наборов данных рекомендуется использовать **hist**.
 - **exact**: точный жадный алгоритм; нумерует всех кандидатов для разбиения,
 - **approx**: приближенный жадный алгоритм; использует квантильный «набросок» и градиентную гистограмму.
 - **hist**: быстрый гистограммный приближенный жадный алгоритм.
 - **gpu_hist**: то же самое что **hist**, но на GPU.
- **scale_pos_weight**: управляет балансом положительных и отрицательных весов (задает отношение положительных и отрицательных экземпляров). Обычно вычисляется как число отрицательных экземпляров к числу положительных экземпляров $\frac{\#neg_instances}{\#pos_instances}$. Полезен для несбалансированных наборов. Например <https://github.com/dmlc/xgboost/blob/master/demo/kaggle-higgs/higgs-cv.py>

```

param = {"max_depth": 6, "eta": 0.1, "objective": "binary:logitraw"}

# define the preprocessing function
# used to return the preprocessed training, test data, and parameter
# we can use this to do weight rescale, etc.
# as a example, we try to set scale_pos_weight
def fprepoc(dtrain, dtest, param):
    label = dtrain.get_label()
    # Вычисляем соотношение положительных и отрицательных экземпляров.
    # Предполагается, что набор данных был разбит стратифицировано
    # и потому нет никакой разницы по какому набору считать долю
    ratio = float(np.sum(label == 0)) / np.sum(label == 1)
    param["scale_pos_weight"] = ratio

    wtrain = dtrain.get_weight()
    wtest = dtest.get_weight()

    # Нужно еще пересчитать веса экземпляров для обучающего и тестового набора!!!
    sum_weight = sum(wtrain) + sum(wtest)
    wtrain *= sum_weight / sum(wtrain)
    wtest *= sum_weight / sum(wtest)

    dtrain.set_weight(wtrain)
  
```

```

dtest.set_weight(wtest)

return (dtrain, dtest, param)

# do cross validation, for each fold
# the (dtrain, dtest, param) will be passed into fpreproc
# then the return value of fpreproc will be used to generate
# results of that fold
xgb.cv(
    param,
    dtrain,
    num_round,
    nfold=5,
    metric={"auc"},
    seed=0,
    fpreproc=fpreproc
)

```

- `max_cat_to_onehot`: если число уникальных значений категориального признака меньше значения этого параметра, то используется кодирование одним активным состоянием.

2.11.3. Параметры задачи

- `objective`: целевая функция:
 - `reg:squarederror`: квадратическая функция потерь для задач регрессии,
 - `reg:squaredlogerror`: квадратическая log loss для задач регрессии $1/2[\log(pred+1)-\log(label+1)]^2$,
 - `reg:logistic`: логистическая регрессия,
 - `reg:pseudohubererror`: псевдогуберовская функция потерь для задач регрессии,
 - `reg:absoluteerror`: регрессия с L_1 -штрафом.
 - `reg:quantileerror`: квантильная функция потерь (она же pinball loss).
 - `binary:logistic`: логистическая регрессия для задач бинарной классификации (вероятности).
 - `binary:logitraw`: логистическая регрессия для задач бинарной классификации (до логистического преобразования).
 - `binary:hinge`: кусочно-линейная функция потерь (hinge loss) для задач бинарной классификации (возвращает 0 или 1, вместо вероятностей).
 - `count:poisson`: пуассоновская регрессия (возвращает среднее пуассоновского распределения).
 - `survaval:cox`: регрессия Кокса для правых цензурированных временных данных.
 - `multi:softmax`: целевая softmax для задач мультиклассовой классификации (нужно еще задать число классов).
 - `multi:softprob`: то же, что и softmax, но выход это вектор `ndata * nclass`. Результат содержит предсказанную вероятность для каждой точки каждого класса.
 - `rank:map`: использует LambdaMART.
 - `reg:gamma`: гамма-регрессия. Выход это среднее гамма-распределения. Может быть полезна для моделирования страховых случаев.

2.12. Python Package

2.12.1. Интерфейс данных

XGBoost поддерживает numpy-массивы, pandas-кадры, XGBoost binary buffer file, LIBSVM etc.

Создать матрицу данных на базе numpy-матрицы

```
data = np.random.rand(5, 10)
label = np.random.randint(2, size=5)
dtrain = xgb.DMatrix(data, label=label)
```

Создать матрицу данных на базе pandas-кадра

```
data = pd.DataFrame(np.arange(12).reshape((4, 3)), columns=list("abc"))
label = pd.Series(np.random.randint(2, size=4))
dtrain = xgb.DMatrix(data, label=label)
```

Сохранение матрицы данных в бинарный файл для ускорения загрузки

```
dtrain = xgb.DMatrix("train.sum.txt")
dtrain.save_binary("train.buffer")
```

Можно передать веса

```
w = np.random.rand(5, 1)
dtrain = xgb.DMatrix(data, label=label, missing=np.nan, weight=w)
```

При построении матрицы данных на базе csv-файла с помощью `label_column` можно указать какой столбец считать целевой переменной

```
# 5-ый столбец будет целевой переменной
dtrain = xgb.DMatrix("train.csv?format=csv&label_column=5")
```

ВАЖНО: парсер XGBoost имеет ограниченную функциональность, поэтому рекомендуется использовать pandas-интерфейс – `read_csv`.

Обучение модели

```
bst = xgb.train(
    params={"max_depth": 2, "eta": 1, "objective": "binary:logistic"},
    dtrain=dtrain,
    num_boost_round=10,
    evals=[(dtrain, "train"), (dtest, "test")]
)
```

После обучения модель можно сохранить

```
bst.save_model("0001.model")

bst_restore = xgb.Booster()
bst_restore.load_model("0001.model")
```

2.12.2. Ранняя остановка

Если есть валидационный набор данных, то оптимальное число итераций градиентного бустинга можно подборать с помощью ранней остановки. Ранняя остановка требует хотя бы одного набора данных в `eval`

```
bst = xgb.train(..., evals=evals, early_stopping_rounds=10)
```

Модель будет обучаться до тех пор пока метрика качества на валидационном наборе данных не перестанет улучшаться. Ошибка на валидационном наборе должна уменьшаться как минимум каждые `early_stopping_rounds` для того чтобы процедура обучения продолжалась.

Когда ранняя остановка завершиться, у модели появится два дополнительных поля: `bst.best_score` и `bst.best_iteration`.

ВАЖНО: `xgb.train()` возвращает модель последней итерации, а не лучшей!

2.12.3. Прогноз

Прогноз можно построить так

```
ypred = bst.predict(dtest)
```

Если число итераций градиентного бустинга подбиралось с помощью ранней остановки, то прогноз для лучшей итерации можно получить так

```
ypred = bst.predict(dtest, iteration_range=(0, bst.best_iteration + 1))
```

Как отмечается в документации XGBoost 2.0.0-dev, если используется ранняя остановка, то функции построения прогноза, включая методы `xgboost.XGBModel.predict()`, `xgboost.XGBModel.score()` и `xgboost.XGBModel.apply()` будут автоматически использовать лучшую модель.

When early stopping is enabled, prediction functions including the `xgboost.XGBModel.predict()`, `xgboost.XGBModel.score()`, and `xgboost.XGBModel.apply()` methods will use the best model automatically.

Meaning the `xgboost.XGBModel.best_iteration` is used to specify the range of trees used in prediction

Чтобы закешировать результаты инкрементного прогноза, следует воспользоваться методом `xgboost.Booster.predict()`.

2.12.4. Scikit-learn интерфейс

Можно использовать scikit-learn интерфейс

```
reg = xgb.XGBRegressor(tree_method="gpu_hist")
reg.fit(X, y)
reg.save_model("regressor.json")
```

При необходимости можно преобразовать модель в бустер (booster). Бустер это модель XGBoost, которая содержит низкоуровневые процедуры для обучения, предсказания и вычисления

```
booster: xgb.Booster = reg.get_booster()
```

2.12.5. Core Data Structure

`xgboost.DMatrix`: это внутренняя структура данных XGBoost, оптимизированная и по памяти, и по скорости обучения.

```
dtrain = xgb.DMatrix(X_train, y_train)
dtest = xgb.DMatrix(X_test)

bst = xgb.train(
    params,
    dtrain,
    evals=[(dtrain, "train"), (xgb.DMatrix(X_test, y_test), "test")],
    num_boost_round=350,
```

```
    early_stopping_rounds=10,  
)  
  
bst.predict(dtest)
```

`xgboost.QuantileDMatrix`: это вариант `DMatrix`, который используется гистограммными методами для сохранения памяти (`hist` и `gpu_hist`). Этот класс был разработан специально для снижения потребления памяти в ходе обучения. Нельзя использовать квантильную матрицу данных `QuantileDMatrix` в качестве валидационного / тестового набора данных *без* привязки (параметр `ref`) к обучающему набору данных `QuantileDMatrix`, так как в ходе квантенизации может быть потеряна часть информации.

ВАЖНО: если на большом наборе данных возникают ошибки нехватки памяти, то имеет смысл попробовать `xgboost.QuantileDMatrix`. Если данные уже находятся на графическом процессоре, то `inplace_predict` может быть более предпочтительным вариантом, чем `predict`. `QuantileDMatrix` и `inplace_predict` используются по умолчанию, если работа с моделью ведется через scikit-learn интерфейс.

If you are getting out-of-memory errors on a big dataset, try the or `xgboost.QuantileDMatrix` or external memory version. Note that when external memory is used for GPU hist, it's best to employ gradient based sampling as well. Last but not least, `inplace_predict` can be preferred over `predict` when data is already on GPU. Both `QuantileDMatrix` and `inplace_predict` are automatically enabled if you are using the scikit-learn interface.

Как отмечается в документации XGBoost 2.0.0-dev, когда параметр `tree_method` получает значение `hist` или `gpu_hist` для экономии памяти вместо матрицы `DMatrix` будет использоваться матрица `QuantileDMatrix`. Однако, когда устройство с входным данными (device of input data) несогласовано с алгоритмом, то производительность может снизиться. Например, если вход это numpy-массив на CPU, но параметр `tree_method` получил значение `gpu_hist`, то данные сначала будут обрабатываться на CPU, а только потом на GPU.

Метод бустера `get_score(fmap="", importance_type="weight")` можно вычислить важность признаков. Для деревянных моделей важность определяется как

- "weight": количество раз, когда признак использовался в разбиениях по всем деревьям,
- "gain": средний информационный прирост по всем разбиениям, в которых принимал участие рассматриваемый признак,
- "cover": среднее покрытие по всем разбиениям, в которых признак принимал участие,
- "total_gain": общий информационный прирост по всем разбиениям, в которых признак принимал участие,
- "total_cover": общее покрытие по всем разбиениям, в которых признак принимал участие.

Метод бустера `inplace_predict` строит прогноз «на месте». В отличие от метода `predict`, предсказание на месте не кеширует результат прогноза

```
# Прогноз строится на X_test, а не на dtest!  
np.rint(clf.get_booster().inplace_predict(X_test)).astype(np.int_)
```



```
# Для сравнения  
clf.predict(X_test)  
np.rint(clf.get_booster().predict(dtest)).astype(np.int_) # Здесь нужно передавать dtest
```

В методе `predict` можно указать флаг `pred_contribs=True`, и тогда вернется матрица размера $n_{samples} \times (n_{feats} + 1)$, в которой каждая строка означает *вклад признаков* (значения SHAP) в соответствующий прогноз. Последний столбец это смещение.

Флаг `iteration_range` метода `predict` определяет какой уровень деревьев используется для прогноза.

Метод бустера `trees_to_dataframe()` парсит модель и возвращает в виде pandas-структуры

Tree	Node	ID	Feature	Split	Yes	No	Missing	Gain	Cover	Category
0	0	0-0	f1	2.95	0-1	0-2	0-1	6.663281	112.0	NaN
1	0	1	0-1	f3	1.65	0-3	0-4	0-3	7.100669	44.0
2	0	2	0-2	f2	3.05	0-5	0-6	0-5	1.103385	68.0
3	0	3	0-3	f0	4.70	0-7	0-8	0-7	1.525521	31.0
4	0	4	0-4	Leaf	NaN	NaN	NaN	-0.139286	13.0	NaN
5	0	5	0-5	Leaf	NaN	NaN	NaN	-0.145833	35.0	NaN
6	0	6	0-6	f2	5.00	0-9	0-10	0-9	4.627291	33.0
7	0	7	0-7	Leaf	NaN	NaN	NaN	-0.100000	2.0	NaN
8	0	8	0-8	f2	5.25	0-11	0-12	0-11	0.808621	29.0
9	0	9	0-9	f3	1.70	0-13	0-14	0-13	0.819444	11.0
...										

2.12.6. Learning API

`xgboost.train()`: обучает бустер с заданными параметрами:

- `params`: параметры бустера.
- `dtrain`: обучающий набор данных.
- `num_boost_round`: число итераций градиентного бустинга.
- `evals`: коллекция валидационных наборов данных, на которых будут вычисляться указанные метрики в ходе обучения. Валидационные метрики помогают отслеживать производительность модели.
- `obj`: пользовательская целевая функция.
- `early_stopping_rounds`: активирует раннюю остановку. Валидационные метрики должны улучшаться по крайней мере каждые `early_stopping_rounds` раундов, чтобы обучение продолжалось.

`xgboost.cv()`: перекрестная проверка с заданными параметрами:

- `params`: параметры бустера.
- `dtrain`: обучающий набор данных.
- `num_boost_round`: число итераций градиентного бустинга.
- `nfold`: число фолдов перекрестной проверки.
- `stratified`: стратифицированное разбиение.
- `metrics`: метрика, которая будет вычисляться на перекрестной проверке.
- `obj`: пользовательская целевая функция.
- `early_stopping_rounds`: активирует раннюю остановку. Валидационные метрики должны улучшаться по крайней мере каждые `early_stopping_rounds` раундов, чтобы обучение продолжалось.
- `fpreproc`: функция предобработки, которая принимает (`dtrain`, `dtest`, `param`) и возвращает их новое представление.
- `shuffle`: перетасовывает данные перед разбиением на фолды.

```
import json

bst = xgb.train(
    {   # params
        "objective": "binary:logistic",
```

```

    "eta": 0.05,
    "max_depth": 3,
    "subsample": 0.85,
    "colsample_bytree": 0.75,
    "tree_method": "hist"
},
dtrain,
num_boost_round=1_000,
evals=[(dtrain, "train"), (dtest, "test")],
callbacks=[xgb.callback.EarlyStopping(5)],
)

print(json.loads(bst.save_config()))
"""
{'learner': {'generic_param': {'fail_on_invalid_gpu_id': '0',
  'gpu_id': '-1',
  'n_jobs': '0',
  'nthread': '0',
  'random_state': '0',
  'seed': '0',
  'seed_per_iteration': '0',
  'validate_parameters': '1'},
  'gradient_booster': {'gbtree_model_param': {'num_parallel_tree': '1',
    'num_trees': '145',
    'size_leaf_vector': '0'},
    'gbtree_train_param': {'predictor': 'auto',
      'process_type': 'default',
      'tree_method': 'hist',
      'updater': 'grow_quantile_histmaker',
      'updater_seq': 'grow_quantile_histmaker'},
    'name': 'gbtree',
    'specified_updater': False,
    'updater': {'grow_quantile_histmaker': {'train_param': {'alpha': '0',
      'cache_opt': '1',
      'colsample_bylevel': '1',
      'colsample_bynode': '1',
      'colsample_bytree': '0.75',
      'eta': '0.0500000007',
      'gamma': '0',
      'grow_policy': 'depthwise',
      'interaction_constraints': '',
      'lambda': '1',
      'learning_rate': '0.0500000007',
      'max_bin': '256',
      'max_cat_threshold': '64',
      'max_cat_to_onehot': '4',
      'max_delta_step': '0',
      'max_depth': '3',
      'max_leaves': '0',
      'min_child_weight': '1',
      'min_split_loss': '0',
      'monotone_constraints': '()',
      'refresh_leaf': '1',
      'reg_alpha': '0',
      'reg_lambda': '1',
      'sampling_method': 'uniform',
      'sketch_ratio': '2',
      'sparse_threshold': '0.20000000000000001',
      'subsample': '0.850000024'}}}}},
  'learner_model_param': {'base_score': '5E-1',

```

```

'boost_from_average': '1',
'num_class': '0',
'num_feature': '30',
'num_target': '1'},
'learner_train_param': {'booster': 'gbtree',
'disable_default_eval_metric': '0',
'dsplit': 'auto',
'objective': 'binary:logistic'},
'metrics': [{name': 'logloss'}],
'objective': {'name': 'binary:logistic',
'reg_loss_param': {'scale_pos_weight': '1'}}},
'version': [1, 7, 5]}
"""

```

2.12.7. Scikit-Learn API

См. https://xgboost.readthedocs.io/en/stable/python/python_api.html

```

from sklearn.metrics import mean_absolute_error

reg = xgb.XGBRegressor(
    tree_method="hist",
    eval_metric=mean_absolute_error,
)
reg.fit(X, y, eval_set=[(X, y)])

```

2.13. Подбор гиперпараметров

Можно использовать Scikit-learn интерфейс

```

xgb_model = xgb.XGBRegressor(n_jobs=1)

clf = GridSearchCV(
    # <- NB: используется модель градиентного бустинга XGBoostr (но интерфейс scikit-learn)
    xgb_model,
    {
        "max_depth": [2, 4],
        "n_estimators": [50, 100],
    },
    verbose=1,
    n_jobs=1,
    cv=3,
)

clf.fit(X, y)
clf.best_score_
clf.best_params_

# Можно сохранить обученную модель
pickle.dump(clf, open("best_calif.pkl", "wb"))
clf2 = pickle.load(open("best_calif.pkl", "rb"))

clf2.predict(X)

```

Можно организовать раннюю остановку

```

X = digits["data"]
y = digits["target"]

```

```

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
clf = xgb.XGBClassifier(n_jobs=1)
clf.fit(
    X_train, y_train,
    early_stopping_rounds=10,
    eval_metric="auc",
    eval_set=[(X_test, y_test)])
)

```

Для более сложных случаев подбора гиперпараметров следует воспользоваться библиотекой Optuna <https://optuna.readthedocs.io/en/stable/index.html>.

Пример использования связки XGBoost + Optuna

```

from sklearn.model_selection import ShuffleSplit
import pandas as pd
import numpy as np
import xgboost as xgb
import optuna

# The Veterans' Administration Lung Cancer Trial
# The Statistical Analysis of Failure Time Data by Kalbfleisch J. and Prentice R (1980)
df = pd.read_csv('../data/veterans_lung_cancer.csv')
print('Training data:')
print(df)

# Split features and labels
y_lower_bound = df['Survival_label_lower_bound']
y_upper_bound = df['Survival_label_upper_bound']
X = df.drop(['Survival_label_lower_bound', 'Survival_label_upper_bound'], axis=1)

# Split data into training and validation sets
rs = ShuffleSplit(n_splits=2, test_size=.7, random_state=0)
train_index, valid_index = next(rs.split(X))
dtrain = xgb.DMatrix(X.values[train_index, :])
dtrain.set_float_info('label_lower_bound', y_lower_bound[train_index])
dtrain.set_float_info('label_upper_bound', y_upper_bound[train_index])
dvalid = xgb.DMatrix(X.values[valid_index, :])
dvalid.set_float_info('label_lower_bound', y_lower_bound[valid_index])
dvalid.set_float_info('label_upper_bound', y_upper_bound[valid_index])

# Define hyperparameter search space
base_params = {'verbosity': 0,
               'objective': 'survival:aft',
               'eval_metric': 'aft-nloglik',
               'tree_method': 'hist'} # Hyperparameters common to all trials
def objective(trial):
    params = {'learning_rate': trial.suggest_loguniform('learning_rate', 0.01, 1.0),
              'aft_loss_distribution': trial.suggest_categorical('aft_loss_distribution',
                                                               ['normal', 'logistic', 'extreme']),
              'aft_loss_distribution_scale': trial.suggest_loguniform('aft_loss_distribution_scale', 0.1,
                                                                     10.0),
              'max_depth': trial.suggest_int('max_depth', 3, 8),
              'lambda': trial.suggest_loguniform('lambda', 1e-8, 1.0),
              'alpha': trial.suggest_loguniform('alpha', 1e-8, 1.0)} # Search space
    params.update(base_params)
    pruning_callback = optuna.integration.XGBoostPruningCallback(trial, 'valid-aft-nloglik')
    bst = xgb.train(params, dtrain, num_boost_round=10000,
                    evals=[(dtrain, 'train'), (dvalid, 'valid')],
                    early_stopping_rounds=50, verbose_eval=False, callbacks=[pruning_callback])
    if bst.best_iteration >= 25:

```

```

        return bst.best_score
    else:
        return np.inf # Reject models with < 25 trees

# Run hyperparameter search
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=200)
print('Completed hyperparameter tuning with best aft-nloglik = {}.'.format(study.best_trial.
    value))
params = {}
params.update(base_params)
params.update(study.best_trial.params)

# Re-run training with the best hyperparameter combination
print('Re-running the best trial... params = {}'.format(params))
bst = xgb.train(params, dtrain, num_boost_round=10000,
    evals=[(dtrain, 'train'), (dvalid, 'valid')],
    early_stopping_rounds=50)

# Run prediction on the validation set
df = pd.DataFrame({'Label (lower bound)': y_lower_bound[valid_index],
    'Label (upper bound)': y_upper_bound[valid_index],
    'Predicted label': bst.predict(dvalid)})
print(df)
# Show only data points with right-censored labels
print(df[np.isinf(df['Label (upper bound)'])])

# Save trained model
bst.save_model('aft_best_model.json')

```

2.14. Подбор гиперпараметров XGBoost на отложенной выборке с подрезкой «слабых» запусков

```

import numpy as np
import optuna

import sklearn.datasets
import sklearn.metrics
from sklearn.model_selection import train_test_split
import xgboost as xgb

# FYI: Objective functions can take additional arguments
# (https://optuna.readthedocs.io/en/stable/faq.html#objective-func-additional-args).
def objective(trial):
    data, target = sklearn.datasets.load_breast_cancer(return_X_y=True)
    train_x, valid_x, train_y, valid_y = train_test_split(data, target, test_size=0.25)
    dtrain = xgb.DMatrix(train_x, label=train_y)
    dvalid = xgb.DMatrix(valid_x, label=valid_y)

    param = {
        "verbosity": 0,
        "objective": "binary:logistic",
        "eval_metric": "auc",
        "booster": trial.suggest_categorical("booster", ["gbtree", "gblinear", "dart"]),
        "lambda": trial.suggest_float("lambda", 1e-8, 1.0, log=True),
        "alpha": trial.suggest_float("alpha", 1e-8, 1.0, log=True),
    }

```

```

if param["booster"] == "gbtree" or param["booster"] == "dart":
    param["max_depth"] = trial.suggest_int("max_depth", 1, 9)
    param["eta"] = trial.suggest_float("eta", 1e-8, 1.0, log=True)
    param["gamma"] = trial.suggest_float("gamma", 1e-8, 1.0, log=True)
    param["grow_policy"] = trial.suggest_categorical("grow_policy", ["depthwise", "lossguide"])
if param["booster"] == "dart":
    param["sample_type"] = trial.suggest_categorical("sample_type", ["uniform", "weighted"])
    param["normalize_type"] = trial.suggest_categorical("normalize_type", ["tree", "forest"])
    param["rate_drop"] = trial.suggest_float("rate_drop", 1e-8, 1.0, log=True)
    param["skip_drop"] = trial.suggest_float("skip_drop", 1e-8, 1.0, log=True)

# Add a callback for pruning.
pruning_callback = optuna.integration.XGBoostPruningCallback(trial, "validation-auc")
bst = xgb.train(param, dtrain, evals=[(dvalid, "validation")], callbacks=[pruning_callback])
preds = bst.predict(dvalid)
pred_labels = np.rint(preds)
accuracy = sklearn.metrics.accuracy_score(valid_y, pred_labels)
return accuracy

if __name__ == "__main__":
    study = optuna.create_study(
        pruner=optuna.pruners.MedianPruner(n_warmup_steps=5), direction="maximize" # <- NB
    )
    study.optimize(objective, n_trials=100)
    print(study.best_trial)

```

2.15. Подбор гиперпараметров XGBoost с перекрестной проверкой и подрезкой «слабых» запусков

ВАЖНО: в Optuna диапазоны изменения значений гиперпараметров задаются явно; если требуется искать значения в логарифмическом масштабе, то просто добавляется флаг `log=True`. В hyperopt в качестве верхних и нижних границ указывается результат преобразования $\exp(lower)$ или $\exp(upper)$; то есть если к примеру темп обучения изменяется в диапазоне от значения близкого к нулю до единицы, то в hyperopt придется писать так

`"learning_rate": hp.loguniform("learning_rate", -7, 0)` так как $e^{-7} \approx 0$, а $e^0 = 1$.

Или пусть параметр регуляризации λ изменяется в диапазоне от значения близкого к нулю до нескольких тысяч, тогда `"lambda": hp.loguniform("lambda", -10, 10)`, так как $e^{-10} \approx 0$, а $e^{10} > 10\,000$.

```

import optuna

import sklearn.datasets
import xgboost as xgb

def objective(trial):
    train_x, train_y = sklearn.datasets.load_breast_cancer(return_X_y=True)
    dtrain = xgb.DMatrix(train_x, label=train_y)

    param = {
        "verbosity": 0,
        "objective": "binary:logistic",
        "eval_metric": "auc", # <- NB
        "booster": trial.suggest_categorical("booster", ["gbtree", "gblinear", "dart"]),
    }

```

```

"lambda": trial.suggest_float("lambda", 1e-8, 1.0, log=True),
"alpha": trial.suggest_float("alpha", 1e-8, 1.0, log=True),
}

if param["booster"] == "gbtree" or param["booster"] == "dart":
    param["max_depth"] = trial.suggest_int("max_depth", 1, 9)
    param["eta"] = trial.suggest_float("eta", 1e-8, 1.0, log=True)
    param["gamma"] = trial.suggest_float("gamma", 1e-8, 1.0, log=True)
    param["grow_policy"] = trial.suggest_categorical("grow_policy", ["depthwise", "lossguide"])
if param["booster"] == "dart":
    param["sample_type"] = trial.suggest_categorical("sample_type", ["uniform", "weighted"])
    param["normalize_type"] = trial.suggest_categorical("normalize_type", ["tree", "forest"])
    param["rate_drop"] = trial.suggest_float("rate_drop", 1e-8, 1.0, log=True)
    param["skip_drop"] = trial.suggest_float("skip_drop", 1e-8, 1.0, log=True)

pruning_callback = optuna.integration.XGBoostPruningCallback(trial, "test-auc")
history = xgb.cv(param, dtrain, num_boost_round=100, callbacks=[pruning_callback])

mean_auc = history["test-auc-mean"].values[-1] # <- NB
return mean_auc

if __name__ == "__main__":
    pruner = optuna.pruners.MedianPruner(n_warmup_steps=5)
    study = optuna.create_study(pruner=pruner, direction="maximize")
    study.optimize(objective, n_trials=100)

    print("Number of finished trials: {}".format(len(study.trials)))

    print("Best trial:")
    trial = study.best_trial

    print("Value: {}".format(trial.value))

    print("Params: ")
    for key, value in trial.params.items():
        print("{}: {}".format(key, value))

```

2.15.1. Конспект статьи Chen T. XGBoost: A Scalable Tree Boosting System, 2016

В статье отмечается, что самый главный фактор успешности библиотека – это ее масштабируемость во всех сценариях. Масштабируемость XGBoost обусловлена несколькими важными алгоритмическими оптимизациями, включающими в себя новый алгоритм обучения моделей деревьев принятия решений на *разреженных данных*. Теоретически обоснованная *процедура создания эскиза взвешенного квантиля* (weighted quantile sketch procedure) позволяет обрабатывать веса экземпляров при приближенном обучении дерева.

Что еще более важно, XGBoost использует out-of-core вычисления (то есть вычисления на таких больших наборах данных, работа с которыми не возможна в оперативной памяти), что позволяет работать с миллионами экземпляров на локальной машине.

2.15.2. Целевая функция с регуляризацией

Пусть дан обучающий набора данных с n экземплярами и m признаками $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}$ ($|\mathcal{D}| = n, \mathbf{x}_i \in \mathbb{R}^m, y_i \in \mathbb{R}$), а ансамбль содержит K функций

$$\hat{y}_i = \varphi(\mathbf{x}_i) = \sum_{k=1}^K f_k(\mathbf{x}_i), \quad f_k \in \mathcal{F},$$

где $\mathcal{F} = \{f(\mathbf{x}) = w_{q(\mathbf{x})}\}(q : R^m \rightarrow \{1, 2, \dots, T, w \in \mathbb{R}^T\})$ – пространство *регрессионных деревьев* (еще их называют CART – Classification and Regression Trees); q – структура каждого дерева, которое отображает экземпляр (строку матрицы признакового описания объекта) на *индекс листа*; T – число листьев в дереве; w_i – оценка (вес) i -ого листа.

Целевая функция – *потери на обучающем наборе данных* (train loss) и *штраф* (regularization term)

$$\mathcal{L}(\varphi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k), \quad (1)$$

где $\Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2 = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$; l – дифференцируемая выпуклая функция потерь, которая оценивает как сильно прогноз \hat{y}_i отличается от цели y_i .

Если штраф опустить, то получится целевая функция обычного градиентного бустинга.

2.15.3. Градиентный бустинг

Уравнение (1) включает функции как параметры и не может быть оптимизировано с использованием традиционных методов оптимизации в Евклидовом пространстве. Поэтому модель обучается в *аддитивной* манере.

Пусть \hat{y}_i это прогноз на i -ом экземпляре на t -ой итерации и нам требуется минимизировать следующую целевую функцию

$$\mathcal{L}^{(t)} = \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)) + \Omega(f_t)$$

Это означает, что мы *жадно* добавляем f_t так, чтобы получить наибольшее улучшение модели. Удобно переписать целевую с помощью аппроксимации второго порядка

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^n \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i) \right] + \Omega(f_t)$$

Целевую можно переписать в более простой форме

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^n \left[g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i) \right] + \Omega(f_t)$$

Если определить $I_j = \{i | q(\mathbf{x}_i) = j\}$ – множество экземпляров, попавших в j -ый лист, то последнее соотношение можно переписать так

$$\tilde{\mathcal{L}}^{(t)} = \sum_{j=1}^T \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T$$

Для фиксированной структуры $q(\mathbf{x})$, мы можем вычислить *оптимальный вес* w_j^* листа j как

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda},$$

и вычислить соответствующее *оптимальное значение целевой*

$$\tilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^T \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T. \quad (2)$$

Уравнение (2) (**теоретически!**) может использоваться для оценки качества структуры дерева. Эта оценка похожа на оценку загрязненности (impurity score) в решающих деревьях, за исключением того, что оценка $\tilde{\mathcal{L}}^{(t)}(q)$ распространяется на более широкий класс целевых функций.

Однако, на практике обычно крайне затруднительно рассмотреть все возможные варианты структуры q . Поэтому используется *энсейджный* алгоритм построения дерева. А снижение потерь после разбиения (информационный прирост) вычисляется как

$$\mathcal{L}_{split} = \frac{1}{2} \left[\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma,$$

где $I = I_L \cup I_R$; I_L, I_R – подмножество экземпляров соответственно левого и правого дочернего узла.

Эта формула обычно применяется на практике при оценке разбиений в дереве. Побеждает разбиение с наибольшим значением информационного прироста (gain) \mathcal{L}_{split} .

2.15.4. Сжатие (shrinkage) и подвыборка по признакам

Помимо целевой функции с регуляризацией, упомянутой в предыдущем разделе, используются еще две техники для снижения эффекта переобучения:

- *сжатие* (shrinkage, learning rate) $\eta \in (0, 1]$, введенное Фридманом: также как и темп обучения в стохастической оптимизации, сжатие *снижает влияние* каждого отдельного дерева на ансамбль¹ (ограничивает вклад каждого базового алгоритма в ансамбль),
- подвыборка по признакам (column subsampling).

В работе отмечается, что согласно отзывам пользователей библиотеки *подвыборка по признакам* (column subsampling) помогает бороться с переобучением даже эффективнее, чем традиционная *подвыборка по экземплярам* (row subsampling).

¹Другими словами [курс ШАД по машинному обучению], присутствие темпа обучения означает, что каждый *базовый алгоритм* вносит относительно *небольшой вклад* во всю композицию: $a_{k+1}(x) = b_1(x) + \eta b_2(x) + \dots + \eta b_{k+1}(x)$

2.15.5. Алгоритм поиска разбиения

Базовый точный жадный алгоритм (Basic Exact Greedy Algorithm) Одна из ключевых проблем в процедуре обучения дерева – это *поиск наилучшего разбиения*. Для поиска лучшего разбиения алгоритм перебирает все возможные варианты разбиений для каждого признака. Авторы XGBoost называют этот алгоритм *точным жадным алгоритмом* (exact greedy algorithm). Для того чтобы эффективно перебрать все возможные разбиения в непрерывном признаке алгоритм должен сначала отсортировать экземпляры по значениям признака (для каждого признака). В своих работах авторы LightGBM называют точный жадный алгоритм *алгоритмом предварительной сортировки* (pre-sorted algorithm). Многие существующие реализации градиентного бустинга для локальной машины – то есть без поддержки масштабирования – (например, scikit-learn и gbm), также как и «локальная» реализация XGBoost поддерживают точный жадный алгоритм.

ЗАМЕЧАНИЕ: точный жадный алгоритм поиска разбиения в узле (exact greedy algorithm) в терминологии XGBoost это тоже самое, что и алгоритм предварительной сортировки (pre-sorted algorithm) в терминологии LightGBM.

Приближенный алгоритм (Approximate Algorithm) Точный жадный алгоритм очень мощный, так как перебирает все возможные варианты разбиения жадно. Однако это невозможно сделать эффективно, если данные не помещаются в память.

Приближенный алгоритм предлагает точки разбиения на основе процентиелей распределения признаков. Затем алгоритм отображает непрерывные признаки на бакеты, разделенные этими точками (candidate split points), агрегирует статистики и ищет лучшее решение.

Существует два варианта алгоритма в зависимости от того, когда «подается предложение». *Глобальный* вариант предлагает все возможные варианты разбиения на начальном этапе построения дерева и использует одни и те же предложения для поиска разбиений на всех уровнях. А *локальный* вариант пересматривает предложения после каждого разбиения. Глобальный метод требует меньше шагов-предложений, чем локальный. Однако, обычно большее количество точек разбиения требуется глобальному методу, потому как точки-кандидаты не уточняются после каждого разбиения. Локальный вариант уточняет точки-кандидаты после каждого разбиения и может быть более подходящим для глубоких деревьев.

Эскиз взвешенного квантиля Важный шаг в приближенном алгоритме – это предложение точек разбиения (candidate split points). Обычно процентили признака используются для построения равномерно распределенных точек-кандидатов.

Формально, пусть есть набор $\mathcal{D}_k = \{(x_{1k}, h_1), (x_{2k}, h_2), \dots, (x_{nk}, h_n)\}$, представляющий k -ый признак и значение гессиана на каждом экземпляре обучающего поднабора. Можно определить функции ранжирования $r_k : \mathbb{R} \rightarrow [0, +\infty)$

$$r_k(z) = \frac{1}{\sum_{(x,h) \in \mathcal{D}_k} h} \sum_{(x,h) \in \mathcal{D}_k, x < z} h,$$

которые представляют долю экземпляров, значение признака которых меньше z .

Цель найти такую точку разбиения $\{s_{k1}, s_{k2}, \dots, s_{kl}\}$, чтобы

$$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \varepsilon, \quad s_{k1} = \min_i \mathbf{x}_{ik}, \quad s_{kl} = \max_i \mathbf{x}_{ik},$$

где ε – фактор аппроксимации. Интуитивно это означает, что существует примерно $1/\varepsilon$ точек-кандидатов. Здесь каждая экземпляр набора данных имеет вес h_i .

Перепишем целевую функцию

$$\sum_{i=1}^n \frac{1}{2} h_i (f_t(\mathbf{x}_i) - g_i/h_i)^2 + \Omega(f_t) + \text{constant},$$

что по сути взвешенная квадратическая функция потерь с метками g_i/h_i и весами h_i .

Для больших наборов данных очень не просто найти точки разбиения, удовлетворяющие этому критерию. Когда веса у всех экземпляров одинаковые, то алгоритм называется *квантильный наборосок* (quantile sketch). Однако, не существует квантильного набороска для взвешенного набора данных. Поэтому большинство существующих приближенных алгоритмов либо прибегают к сортировке на случайном подмножестве набора данных (есть вероятность сбоя), либо к эвристикам, которые не имеют теоретического обоснования.

Чтобы обойти эту проблему авторы XGBoost предлагают новый distributed weighted quantile sketch algorithm, который может взвешивать экземпляры набора данных с теоретической гарантией. Общая идея состоит в том, чтобы предложить структуру данных, которая поддерживает операции *слияния и подрезки*.

Поиск разбиения с учетом разреженности Во многих реальных приложениях часто оказывается, что вход X разреженный. Существует множество причин разреженности:

- наличие пропущенных значений в данных,
- частые нулевые значения,
- артифакты подготовки признаков (например, кодирование с помощью техники одного активного состояния).

Важно, чтобы алгоритм знал о разреженности в данных. Поэтому авторы XGBoost предлагают добавить *направление по умолчанию* в каждый узел дерева. Когда значение пропущено в разреженной матрице признакового описания объекта X , экземпляр классифицируется как принадлежащий этому направлению по умолчанию. Однако, направление по умолчанию может быть левым (left-child) или правым (right-child) и в процессе построения дерева определяется наилучшее направление, которое может снизить потери на обучающем наборе данных.

По умолчанию все экземпляры с пропущенными значениями попадают в левый дочерний узел.

XGBoost обрабатывает все случаи разреженности в одной и той же манере. Что гораздо важнее, XGBoost алгоритм поиска разбиений на разреженных данных использует разреженность, чтобы сделать сложность вычислений *линейной* по числу *непропущенных* значений в наборе данных.

2.15.6. Блоки столбцов для параллельного обучения

Большую часть времени в процедуре обучения дерева занимает сортировка экземпляров набора данных. Чтобы снизить затраты на сортировку авторы XGBoost предлагают хранить данные в блоках памяти (in-memory units) или просто *блоках*. Данные в каждом блоке хранятся в формате сжатого разреженного столбца (compressed column, CSC). Причем каждый столбец отсортирован по значению признака. Этот подход требует построения блоков только один раз перед обучением.

В точном жадном алгоритме мы храним весь набор данных в одном блоке и запускаем алгоритм поиска разбиения путем линейного сканирования предварительно отсортированного входа. Мы выполняем поиск разделения по всем листьям, поэтому при одном проходе по блоку будет собрана вся статистика. Блочная структура также помогает при использовании приближенного алгоритма. Различные блоки могут быть распределены по различным машинам или сохранены на диске в режиме «вне ядра».

С использованием отсортированной структуры, этап поиска квантиля превращается в линейное сканирование отсортированных столбцов.

Сбор статистик может быть распараллелен по каждому столбцу, что позволяет распараллелить алгоритм поиска разбиений. Важно отметить, что блочная структура столбцов также поддерживает подвыборку по признакам (column subsampling), так как легко выбрать подмножество столбцов в блоке.

2.15.7. Блоки для внеядерных (out-of-core) вычислений

Помимо процессоров и памяти важно еще использовать и дисковое пространство, когда данные не помещаются в основную память.

Чтобы обеспечить возможность выполнения *внеядерных вычислений*, XGBoost разбивает данные на *блоки* и хранит их *на диске*. Во время вычислений важно использовать независимые потоки для предварительной выборки блоков в буфер основной памяти, чтобы вычисления могли выполняться одновременно с чтением с диска. Однако, это не решает проблему полностью, так как чтение с диска составляет большую часть вычислительного времени. Важно увеличить пропускную способность дискового ввода-вывода.

XGBoost использует две техники повышения эффективности *внеядерных вычислений*:

- Block Compression. Блоки сжимаются по столбцам и распаковываются *на лету* независимыми потоками при загрузке в основную память. XGBoost использует алгоритм сжатия общего назначения для сжатия значений признаков.
- Block Sharding. Поток предварительной выборки (pre-fetcher thread) назначается каждому диску и извлекает данные в буфер памяти. Затем обучающий поток поочередно считывает данные из каждого буфера. Это помогает *увеличить пропускную способность чтения с диска* при наличии нескольких дисков.

Spark MLLib и H2O это системы аналитики данных в памяти (in-memory analytics frameworks), которым требуется, чтобы данные хранились в оперативной памяти, в то время как XGBoost может переключаться на внеядерный режим, когда память заканчивается.

3. LightGBM

3.1. Установка

Установить библиотеку можно с помощью менеджера пакетов `pip`

```
pip install lightgbm
```

3.2. Быстрый старт

3.2.1. Работа с входными данными

LightGBM поддерживает входные данные в форматах CSV, TSV, LibSVM, NumPy 2D массивы, pandas DataFrame, H2O DataTable's Frame, SciPy sparse matrix, LightGBM binary file, LightGBM Sequence objects.

Данные хранятся в Dataset объектах

```
dtrain = lgb.Dataset(X_train, y_train)
dtrain.data # вернет X_train
dtrain.label # или dtrain.get_label() вернет y_train
```

Сохранить набор данных можно так

```
dtrain.save_binary("regression.train.from_hdf.bin")
```

Сохранение набора данных в бинарном формате LightGBM делает загрузку более быстрой

```
train_data = lgb.Dataset("train.sum.txt")
train_data.save_binary("train.bin")
```

Поддерживаются веса экземпляров (но нужно назначить эти веса).

Для чтения бинарных файлов можно реализовать Sequence-интерфейс

```
import h5py

class HDFSequence(lgb.Sequence):
    def __init__(self, hdf_dataset, batch_size):
        self.data = hdf_dataset
        self.batch_size = batch_size

    def __getitem__(self, idx):
        return self.data[idx]

    def __len__(self):
        return len(self.data)

f = h5py.File("train.hdf5", "r")
train_data = lgb.Dataset(HDFSequence(f["X"], 8192), label=f["Y"][:])
```

Особенности использования Sequence-интерфейса:

- Выборка данных выполняется случайно и потому не требуется полный проход через весь набор,
- Данные читаются пакетами, что экономит память при конструировании Dataset,
- Поддерживает создание Dataset из нескольких файлов.

3.2.2. Поддержка категориальных признаков

В документации LightGBM утверждается, что библиотека может работать с категориальными признаками напрямую (без ОНЕ – кодирования одним активным состоянием). Если я правильно понял, то они всегда (не зависимо от кардинальности категориального признака) используют технику партиционирования. То есть категории признака они собирают в группы и для каждой группы вычисляют $\frac{\sum \text{gradient}}{\sum \text{hessian}}$. Затем сортируют по этой оценке и оценивают $K - 1$ разбиений. А в XGBoost, если я правильно понял, используется либо техника кодирования одним

активным состоянием, либо техники партиционирования в зависимости от значения параметра `max_cat_to_hot`. К слову, в гистограммной реализации градиентного бустинга в Scikit-learn `HistGradientBoosting`* категории тоже собираются в группы, затем для каждой группы вычисляется дисперсия целевой переменной, по которой потом эти группы и сортируются. В итоге для K категорий категориального признака получается K групп и требуется оценить лишь $K - 1$ вариантов разбиения, вместо $2^{K-1} - 1$ в обычном случае. Группы категорий признака нужно сначала отсортировать, а затем оценить $K - 1$ вариантов разбиений и потому итоговая времененная сложность $O(K \log K + K)$, вместо $O(2^K)$. Если я правильно понял, то на шаге построения прогноза для тестовой точки мы просто смотрим значение признака тестовой точки (например, `green`), идем в соответствующий дочерний узел и если это лист, то просто забираем значение этого листа.

Перед построением `Dataset` требуется категориальные фичи привести к целочисленному типу (как и в XGBoost)

```
X_train["color"] = X_train["color"].astype("category")
dtrain = lgb.Dataset(X_train, y_train)
```

ВАЖНО: при создании `Dataset` LightGBM не проверяет корректность данных, а XGBoost проверяет, то есть XGBoost просто не позволит создать экземпляр матрицы данных, в которой есть нарушение типа категориального признака, используются не существующие имена признаков и пр., а LightGBM заговорит о нарушениях только на этапе обучения модели

Если при создании экземпляра `Dataset` не передавать параметру `feature_name` список значений, то есть если `feature_name="auto"` и матрица признакового описания это pandas-DataFrame, то имена столбцов будут извлекаться из атрибута `columns`.

Если параметру `categorical_features` передать список целых чисел, то он будет интерпретироваться как список индексов категориальных признаков А если как список строк (имен столбцов), то нужно будет еще параметру `feature_name` передать список имен столбцов

```
lgb.Dataset(
    X_train,
    y_train,
    # перечислить нужно имена всех признаков
    feature_name=[ "col1", "col2", "col3", "color" ],
    categorical_feature=[ "color" ]
)
```

3.2.3. Эффективное использование памяти

Объект `Dataset` в LightGBM имеет очень эффективную по памяти реализацию, ему нужно только сохранить бины. Однако, numpy/array/pandas объекты требуют больших затрат памяти. Снизить потребление памяти можно так

- При создании `Dataset` выставить `free_raw_data=True`,
- После создания `Dataset` явно указать `raw_data=None`,
- Вызвать `gc`.

3.2.4. Настройка параметров

LightGBM для настройки параметров может использовать словарь

```
params = {"num_leaves": 31, "objective": "binary"}  
params["metirc"] = "auc" # ["auc", "binary_logloss"]
```

3.2.5. Обучение

Обучение модели

```
num_round = 100  
  
bst = lgb.train(  
    params,  
    dtrain,  
    num_round,  
    valid_sets=[dval]  
)
```

Сохранить обученную модель можно так

```
bst.save_model("model.txt")
```

Еще можно обученную модель представить в виде словаря

```
print(bst.dump_load())  
"""  
{'name': 'tree',  
 'version': 'v3',  
 'num_class': 1,  
 'num_tree_per_iteration': 1,  
 'label_index': 0,  
 'max_feature_idx': 3,  
 'objective': 'regression',  
 'average_output': False,  
 'feature_names': ['col1', 'col2', 'col3', 'color'],  
 'monotone_constraints': [],  
 'feature_infos': {'color': {'min_value': -1,  
 'max_value': 2,  
 'values': [-1, 2, 0, 1]}},  
 'tree_info': [{tree_index': 0,  
 'num_leaves': 1,  
 'num_cat': 0,  
 'shrinkage': 1,  
 'tree_structure': {'leaf_value': 0.6}},  
 'feature_importances': {},  
 'pandas_categorical': [['blue', 'green', 'red']]}  
"""
```

В XGBoost метод `dump_model()` рекомендуется использовать только для визуализации и местного изучения структуры модели. Для длительного хранения модель нужно сохранять с помощью метода `save_model` в форматах json или ubj. И в XGBoost метод `dump_model` сохраняет объект обученной модели в файл, а в LightGBM этот метод просто возвращает модель в виде словаря.

Сохраненную модель в LightGBM можно загрузить в бустер-пустышку

```
# Нельзя создать экземпляр бустера без указания пути до модели  
bst = bst.Booster(model_file="model.txt")
```

А в XGBoost можно создать пустой бустер, а затем загрузить в него модель

```
bst = xgb.Booster()
```

```
bst.load_model("./simple_xgb.ubj")
# а можно сразу указать путь до модели
bst = xgb.Booster(model_file="./simple_xgb.ubj")
```

Можно обучаться с помощью перекрестной проверки

```
lgb.cv(params, dtrain, num_round, nfold=5)
```

3.2.6. Ранняя остановка

Если есть валидационный набор данных, то подобрать оптимальное число итераций градиентного бустинга с помощью ранней остановки. Ранняя остановка требует хотя бы одного набора данных в `valid_sets`. Если их больше одного, то будут использоваться все кроме обучающего поднабора.

```
bst = lgb.train(
    params,
    dtrain,
    valid_sets=[dval],
    callbacks=[lgb.early_stopping(stopping_rounds=5)])
bst.save_model("model.txt", num_iteration=bst.best_iteration)
```

Модель будет обучаться до тех пор, пока оценка на валидационном поднаборе не перестанет улучшаться. Оценка на валидационном поднаборе должна улучшаться по крайней мере каждые `stopping_rounds` для продолжения обучения.

В LightGBM `train()` возвращает модель лучшей итерации, а в XGBoost – последней (не лучшей!).

Это работает и с ошибками (L2, log loss etc.), и с метриками (NDCG, AUC etc.). Если указать более одной оценки (метрики или ошибки), то все эти оценки будут использоваться для ранней остановки. Если нужно, чтобы использовалась только первая оценка, то можно выставить `first_metric_only=True` в callback `early_stopping`.

3.2.7. Прогноз

Обученная или загруженная модель может использоваться для построения прогноза

```
data = np.random.rand(7, 10)
ypred = bst.predict(data)
```

В «быстром страте» говорится, что если использовалась ранняя остановка, то прогноз для лучшей итерации можно получить так

```
ypred = bst.predict(data, num_iteration=bst.best_iteration)
```

Но в документации LightGBM говорится, что если `num_iteration=None` (по умолчанию), лучшая итерация существует и `start_iteration <= 0`, то будет использоваться *лучшая итерация*. То есть можно просто использовать `bst.predict(X_test)`.

В XGBoost вот так

```
ypred = bst.predict(dtest, iteration_range=(0, bst.best_iteration + 1))
```

В XGBoost действительно всегда нужно указывать диапазон итераций, если на фазе обучения число итераций градиентного бустинга определяется с помощью ранней остановки.

3.3. Особенности реализации LightGBM

LightGBM использует алгоритмы, основанные на гистограммах (histogram-based algorithms), которые распределяют вещественные признаки (атрибуты) по бинам. Это повышает скорость обучения и снижает потребление памяти.

Преимущества гистограммных алгоритмов в следующем:

- Снижает издержки на вычисление информационного прироста для каждого разбиения:
 - Pre-sort-based algorithms имеют временную сложность $O(\#data)$,
 - У операции построения гистограммы такая же временная сложность – $O(\#data)$, но когда гистограмма будет построена, временная сложность гистограммного алгоритма будет $O(\#bins)$, а $\#bins$ значительно меньше $\#data$.
- Использует разность гистограмм для ускорения:
 - Для получения гистограмм одного листа бинарного дерева, использует разность родительской и соседней гистограмм,
- Снижает потребление памяти:
 - Заменяет непрерывные значения дискретными бинами. Если число $\#bins$ маленькое, то можно использовать соответствующие типы данных (например, `uint8`) для сохранения обучающего набора,
 - не требуется сохранять дополнительную информацию о предварительно отсортированных признаках.
- Снижает коммуникационные издержки для распределенного обучения.

3.4. Стратегия выращивания деревьев в LightGBM

Как отмечается в документации LightGBM, большинство алгоритмов построения деревьев строят деревья в глубину, по уровням (level-wise или depth-wise) рис. 11. То есть другими словами дерево строится так, чтобы был заполнен каждый уровень.

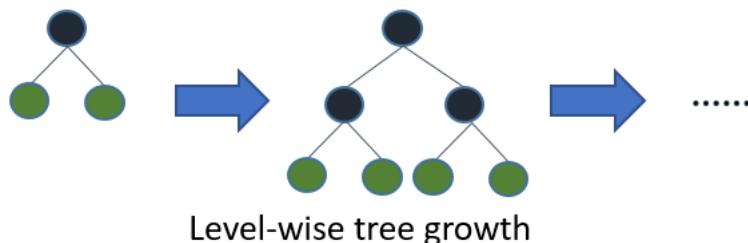


Рис. 11. Стратегия выращивания деревьев в глубину (level-wise)

LightGBM строит деревья по листьям (leaf-wise или best-first) рис. 12 с ограничением на число терминальных узлов. Это приводит к тому, что деревья получаются несимметричными: одно поддерево может иметь глубину 2, а другое – глубину 15. Выращивание деревьев по листьям может стать причиной переобучения, когда обучающих экземпляров $\#data$ мало, поэтому LightGBM ограничивает еще и глубину дерева (`max_depth`).

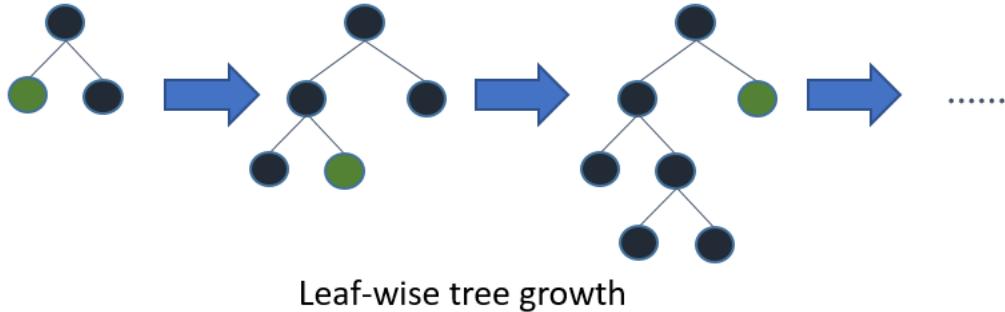


Рис. 12. Стратегия LightGBM выращивания деревьев в глубину (level-wise). Получаются несимметричные деревья

3.5. Оптимальное разбиение для категориальных признаков

Обычно категориальные признаки представляют с помощью техники одного активного состояния (ОНЕ), но этот подход неоптимален для «деревянных» моделей. Особенно для высоко кардинальных категориальных признаков дерево, построенное на признаках, закодированных с помощью ОНЕ, должно быть *очень глубоким*.

LightGBM сортирует гистограмму (для категориального признака) согласно накопленному значению `sum_gradient / sum_hessian` и затем ищет лучшее разбиение по этой отсортированной гистограмме. Если я правильно понял, то они просто собирают категории в k групп и для каждой группы вычисляют отношение суммы градиентов к сумме гессианов, а затем сортируют группы по этой оценке и рассматривают $k - 1$ вариантов разбиения.

3.6. Распараллеливание

Традиционный алгоритм поиска лучшего разбиения в дереве:

- Матрица признакового описания объекта разбивается вертикально, то есть каждая машина получает свой поднабор признаков,
- Воркер ищет лучшую локальную точку разбиения (то есть пару «признак–порог») на своем локальном подмножестве признаков,
- Из этих локальных лучших пар «признак–порог» выбирается лучшая пара,
- Воркер с лучшим разбиением делает разбиение, а затем отправляет другим воркерам результат разбиения данных,
- Другие воркеры разбивают данные в соответствие с полученными данными.

Распараллеливание по признакам не может значительно повысить производительность, когда данных много, поэтому LightGBM предлагает вместо разбиения набора данных вертикально передавать *полный набор* данных каждому воркеру. В этом случае снимаются накладные расходы на коммуникацию, так как каждый воркер знает как разбить данные.

Однако, остаются вычислительные издержки, когда данных много. Предлагается распараллеливать по данным.

Традиционный алгоритм распараллеливания по данным:

- Данные разбиваются горизонтально,
- Воркеры используют локальные поднаборы для построения локальных гистограмм,
- Локальные гистограммы сливаются в глобальную,
- Ищется лучшее разбиение на «склееной» глобальной гистограмме.

Недостаток: высокие издержки на коммуникацию.

LightGBM распараллеливает по данным так. Вместо объединения локальных гистограмм в одну глобальную гистограмму, LightGBM использует «Reduce Scatter» для объединения гистограмм различных неперекрывающихся признаков различных воркеров. Временная сложность $O(0.5 \cdot \#features \cdot \#bins)$.

3.7. Параметры

`force_col_wise`: используется только на CPU; `force_col_wise=True` для принудительного поколоночного построения гистограммы; рекомендуется использовать, когда

- много признаков или бинов,
- используется много потоков `num_threads > 20`,
- требуется снизить потребление памяти.

ВАЖНО: когда и `force_col_wise=False` и `force_row_wise=False`, LightGBM пробует оба эти параметра, а затем выбирает более быстрый. Чтобы не тратить время на тестирование вариантов, можно вручную выбрать более быстры и его выставить в `True`. Использовать можно только один из этих параметров.

`force_row_wise`: используется только на CPU, `force_row_wise=True` для принудительного построкового построения гистограммы; рекомендуется использовать, когда

- много точек данных и общее количество бинов невелико,
- `num_threads` относительно невелико (≤ 16),
- требуется использовать низкие значения `bagging_fraction` или `goss` стратегию выборки для ускорения.

ВАЖНО: если выставить `force_row_wise` в `True`, потребление памяти Dataset удвоится. Если памяти недостаточно, то можно попробовать выставить `force_col_wise` в `True`.

`min_data_in_leaf` (default = 20): минимальное число экземпляров в листе; может использоваться для борьбы с переобучением.

`min_sum_hessian_in_leaf`: минимальная сумма гессианов в листе; так же как и `min_data_in_leaf` может применяться для борьбы с переобучением.

`bagging_fraction`: доля случайно выбранных экземпляров обучающего набора; может применяться для сокращения временных издержек на фазе обучения модели, а также для борьбы с переобучением; требуется установить ненулевое значение для параметра `bagging_freq`.

`pos_bagging_fraction`: используется только в задачах бинарной классификации; может быть полезен для несбалансированных задач; выбирает `#pos_samples * pos_bagging_fraction` положительных экземпляров; должен использоваться вместе с `neg_bagging_fraction`; чтобы использовать `pos_bagging_fraction` нужно выставить еще `bagging_freq` и `neg_bagging_freq`.

`bagging_freq`: частота беггинга; k означает, что беггинг выполняется на каждой k -ой итерации. То есть LightGBM на каждой k -ой итерации случайно выбирает `bagging_fraction * 100 %` от обучающего набора данных для следующих k итераций.

`feature_fraction`: LightGBM на каждой итерации (то есть каждого дерева) будет случайно отбирать указанную долю признаков, если `feature_fraction < 1.0`; может использоваться для снижения временных издержек на фазе обучения модели, а также для борьбы с переобучением.

`feature_fraction_bynode`: LightGBM будет случайно выбирать указанную долю признаков в каждом узле дерева, если `feature_fraction_bynode < 1.0`.

`lambda_11`: L_1 -регуляризация.

`lambda_12`: L_2 -регуляризация.

`two_round`: рекомендуется выставить этот параметр в `True`, если набор данных слишком большой для обучения в памяти; по умолчанию LightGBM отображает данные на память и загружает признаки из памяти; это значительно ускоряет загрузку, но может привести ошибкам памяти, если набор слишком большой; работает только, если данные загружаются напрямую из текстового файла.

`predict_contrib`: если выставить в `True`, то вернутся значения по Шепли, представляющие вклад каждого признака в прогноз.

`is_unbalanced`: используется только в задачах *бинарной* и *многоклассовой классификации по схеме One-vs-All* (или что то же самое One-vs-Rest); если обучающий набор данных *несбалансирован*, то нужно выставить в `True`; этот параметр нельзя использовать одновременно с параметром `scale_pos_weight`; нужно выбрать что-то одно.

https://en.wikipedia.org/wiki/Multiclass_classification#One-vs.-rest One-vs-Rest (или One-vs-All) – это стратегия, идея которой состоит в том, чтобы один класс (положительный) был противопоставлен остальным классам, которые «сливаются» в единый отрицательный класс. Каждый класс из нескольких должен побывать на месте положительного класса, противопоставленного отрицательному классу, состоящему из остальных классов. Эта стратегия требует, чтобы базовые классификаторы выдавали реальный показатель достоверности своего решения, а не просто метку класса. При построении прогноза к новому тестовому экземпляру x применяются все обученные классификаторы и прогнозируется метка класса с наивысшей оценкой (confidence score)

$$\hat{y} = \arg \max_{k \in \{1, \dots, K\}} f_k(x)$$

ВАЖНО! Не смотря на то, что эта стратегия очень популярна, она обладает серьезными недостатками:

- Масштаб оценок принятия решений (confidence values) может быть различным для бинарных классификаторов,
- Даже если исходный набор данных сбалансирован, то схема One-vs-All (One-vs-Rest) делает набор несбалансированным, потому как экземпляров положительного класса обычно значительно меньше экземпляров отрицательного класса.

One-vs-one обучает $(K-1)K/2$ классификаторов для задачи мультиклассовой классификации на K -классов. Например, если решается задача на 3 класса, то нужно построить 3 классификатора: «1–2», «1–3», «2–3». На шаге построения прогноза применяется схема голосования: все $(K-1)K/2$ классификаторы применяются к тестовой точке x и прогнозируется класс, получивший наибольшее число голосов.

ВАЖНО! Так же как и One-vs-Rest, One-vs-One страдает от неопределенности, так как в некоторых случаях прогнозы могут быть несогласованы. Например, «1–2» \rightarrow 2, «1–3» \rightarrow 1 и «2–3» \rightarrow 3.

`scale_pos_weight`: взвешивает метки экземпляров положительного класса; используется только в задачах *бинарной binary* и *многоклассовой классификации по схеме One-vs-All* (One-vs-Rest) *multiclassova*; этот параметр нельзя использовать одновременно с параметром `is_unbalanced`; нужно выбрать что-то одно.

`reg_sqrt`: используется только в задачах регрессии; на шаге обучения модели вместо исходных меток использует \sqrt{label} , а на шаге построения прогноза – $prediction^2$; может быть полезен в случаях, когда значения целевой переменной изменяются в широком диапазоне.

3.8. Замечания о подборе гиперпараметров в LightGBM

LightGBM строит деревья по листьям (leaf-wise), однако, для того чтобы модель не переобучалась нужно настраивать следующие параметры:

- `num_leaves`: это главный параметр в LightGBM, контролирующий сложность деревьев. Теоретически, мы могли бы задать `num_leaves = 2^(max_depth)`, чтобы получить то же дерево, что и в XGBoost (строит полные бинарные деревья в глубину, по уровням, и потому число уровней в таком дереве будет логарифмически зависеть от числа листьев дерева, $max_depth = \log_2(num_leaves) \rightarrow num_leaves = 2^{max_depth}$).

Однако, это правило для LightGBM на практике работает плохо, так как leaf-wise деревья (то есть деревья, которые строятся по листьям) гораздо глубже depth-wise деревьев (деревья, которые строятся по уровням) при фиксированном числе листьев. Такими деревьями легко переобучиться. Таким образом, при подборе значения параметра `num_leaves` нужно задавать значения < 2^{max_depth} . Например, если `max_depth=7` в depth-wise дереве (XGBoost), то в leaf-wise дереве (LightGBM) число листьев должно быть меньше 127, скажем 70-80.

- `min_data_in_leaf`: это очень важный параметр для предотвращения переобучения leaf-wise деревьев. Оптимальное значение зависит от числа экземпляров обучающего набора данных и числа листьев num_leaves. Большие значения делают деревья менее глубокими, что может привести к недообучению. На практике этот параметр имеет порядок нескольких сотен (≈ 100).
- `max_depth`: можно использовать `max_depth`, чтобы явно ограничить глубину дерева.

Для того чтобы поднять скорость вычислений можно выставить параметр `num_threads` в число доступных ядер процессора. То есть обучение можно ускорить, перейдя на машину с большим количеством доступных ядер процессора. Еще можно воспользоваться моделью распределенного обучения. Или можно попробовать LightGBM с поддержкой GPU.

3.8.1. `max_depth` и `num_leaves` для снижения времени обучения

Чем меньше `max_depth`, тем меньше время обучения модели. Тоже самое касается и параметра `num_leaves`: чем меньше листьев разрешается в дереве, тем меньше время обучения модели.

3.8.2. `min_gain_to_split`

Когда добавляется новый узел в дерево, LightGBM выбирает точку разбиения с наибольшим информационным приростом. Информационный прирост по сути показывает снижение потерь на обучающем наборе данных при добавлении рассматриваемой точки разбиения. По умолчанию LightGBM выставляет параметр `min_gain_to_split` в 0, что означает «мы рады любому, даже самому маленькому информационному приросту». Однако, на практике можно заметить, что небольшое снижение потерь на обучающем наборе данных не оказывает значимого влияния на обобщающую способность модели. Большие значения `min_gain_to_split` снижают время обучения модели (потому как в модели будет меньше листьев).

3.8.3. `min_data_in_leaf` и `min_sum_hessian_in_leaf`

- `min_data_in_leaf`: минимальное число наблюдений (экземпляров), которое должно попасть в узел, чтобы он был добавлен в дерево,
- `min_sum_hessian_in_leaf` (то же что и `min_child_weight` XGBoost'a): минимальная сумма гессианов (вторая производная от функции потерь, которая вычисляется для каждого наблюдения) наблюдений в листе.

Пояснение

- Для регрессии потеря на i -ом экземпляре в узле будет $1/2(y_i - \hat{y}_i)^2$. Вторая производная от этого выражения (гессиан) по \hat{y}_i будет 1. Таким образом, сумма вторых производных (гессианов) по всем экземплярам узла это просто *число экземпляров в этом узле*. То есть в данном случае `min_child_weight` (это XGBoost'ый параметр; получается, что это тоже самое что и `min_sum_hessian_in_leaf` в LightGBM) означает «прекратите разбивать узел, как только число экземпляров в узле окажется меньше заданного порога». Потому что нам не нужны листья с малым числом экземпляров (вырожденный случай лист с одним экземпляром).
- Для бинарной классификации (модель логистической регрессии) гессиан для каждой точки в узле будет $\sigma(\hat{y}_i)(1 - \sigma(\hat{y}_i))$, где σ – это логистический сигмоид. Пусть узел будет чистым положительным (то есть в узле лежат экземпляры только одного положительного класса). Тогда все \hat{y}_i будут большими положительными числами, и потому $\sigma(\hat{y}_i)$ будут числами близкими к 1. Таким образом, все гессианы будут близки к 0. Рассуждения справедливы и для случая, когда все экземпляры в узле отрицательные. В этом случае `min_child_weight` можно интерпретировать как: «прекратите разбивать узел, как только узел достигнет определенной степени чистоты». Или лучше сказать, что не нужно разбивать узел, если его чистота выше некоторого порога, то есть не нужно разбивать узел, если он «слишком чистый». Нам не нужны чистые листья! Можем переобучиться.

3.8.4. `learning_rate` и `num_iterations`

Выбор правильного значения для пары `learning_rate` и `num_iterations` сильно зависит от данных и целевой функции, поэтому эти гиперпараметры обычно подбирают. Общее правило: меньшие значения `learning_rate` отвечают большему числу итераций градиентного бустинга. Меньшие значения числа итераций градиентного бустинга отвечают меньшему времени обучения.

3.8.5. `max_bin` или `max_bin_by_feature`

LightGBM чтобы ускорить обучение модели и снизить потребление памяти непрерывные признаки разбивает по бинам. Разбиение по бинам выполняется на этапе конструирования `Dataset`. Число разбиений, учитываемых при разбиении узла равно $O(\#feature \cdot \#bin)$. Таким образом, уменьшение числа бинов для каждого признака может снизить число разбиений, которые нужно рассмотреть.

Меньшие значения `max_bin` или `max_bin_by_feature` снижают время обучения модели.

3.8.6. `feature_fraction`

Меньшие значения `feature_fraction` снижают время обучения модели, так как уменьшается общее число разбиений, которое нужно оценить для того чтобы добавить узел в дерево.

3.8.7. bagging_fraction и bagging_freq

По умолчанию LightGBM на каждой итерации градиентного бустинга использует все экземпляры обучающего набора данных, однако можно попросить использовать не все экземпляры, а какую-то долю случайно выбранных (эта процедура называется баггингом).

Например, если сказать: `bagging_freq=5`, а `bagging_fraction=0.75`, то это будет значить «делай выборку без возвращения каждые 5 итераций и выбирай 75% экземпляров обучающего набора».

Меньшие значения `bagging_fraction` снижают время обучения модели.

3.8.8. Для улучшения метрик

Для получения более высоких значений метрик качества:

- используйте большие значения `max_bin` (может быть медленным),
- используйте меньшие значения `learning_rate` с большими значениями `num_iterations`,
- используйте большие значения `num_leaves` (опасно! можно переобучиться),
- используйте больше обучающих экземпляров,
- попробуйте `dart`.

3.8.9. Для снижения эффекта переобучения

Чтобы снизить переобучение:

- используйте низкие значения `max_bin`,
- используйте низкие значения `num_leaves`,
- используйте `min_data_in_leaf` и `min_sum_hessian_in_leaf`,
- используйте подвыборку по экземплярам: `bagging_fraction` и `bagging_freq`,
- используйте подвыборку по признакам: `feature_fraction`,
- используйте больший набор данных; при фиксированной емкости модели чем больше набор данных, тем сложнее его описывать (обогащение набора новыми данными – регуляризация на уровне данных),
- используйте `lambda_l1`, `lambda_l2` и `min_gain_to_split` (если я правильно понял, то это тоже, что и параметр `gamma` в XGBoost) для регуляризации,
- ограничьте `max_depth`, чтобы не строить глубоки деревья.

В документации LightGBM ничего нет о регуляризующем эффекте низких `learning_rate`, но в документации XGBoost низкие значения темпа обучения встречаются в контексте борьбы с переобучением.

3.9. Подбор гиперпараметров LightGBM с подрезкой «слабых» запусков

```
import numpy as np
import optuna

import lightgbm as lgb
import sklearn.datasets
import sklearn.metrics
from sklearn.model_selection import train_test_split

# FYI: Objective functions can take additional arguments
```

```

# (https://optuna.readthedocs.io/en/stable/faq.html#objective-func-additional-args).
def objective(trial):
    data, target = sklearn.datasets.load_breast_cancer(return_X_y=True)
    train_x, valid_x, train_y, valid_y = train_test_split(data, target, test_size=0.25)
    dtrain = lgb.Dataset(train_x, label=train_y)
    dvalid = lgb.Dataset(valid_x, label=valid_y)

    param = {
        "objective": "binary",
        "metric": "auc",
        "verbosity": -1,
        "boosting_type": "gbdt",
        "lambda_l1": trial.suggest_float("lambda_l1", 1e-8, 10.0, log=True),
        "lambda_l2": trial.suggest_float("lambda_l2", 1e-8, 10.0, log=True),
        "num_leaves": trial.suggest_int("num_leaves", 2, 256),
        "feature_fraction": trial.suggest_float("feature_fraction", 0.4, 1.0),
        "bagging_fraction": trial.suggest_float("bagging_fraction", 0.4, 1.0),
        "bagging_freq": trial.suggest_int("bagging_freq", 1, 7),
        "min_child_samples": trial.suggest_int("min_child_samples", 5, 100),
    }

    # Add a callback for pruning.
    pruning_callback = optuna.integration.LightGBMPruningCallback(trial, "auc")
    gbm = lgb.train(param, dtrain, valid_sets=[dvalid], callbacks=[pruning_callback])

    preds = gbm.predict(valid_x)
    pred_labels = np.rint(preds)
    accuracy = sklearn.metrics.accuracy_score(valid_y, pred_labels)
    return accuracy

if __name__ == "__main__":
    study = optuna.create_study()
    pruner=optuna.pruners.MedianPruner(n_warmup_steps=10), direction="maximize" # <- NB
    )
    study.optimize(objective, n_trials=100)

    print("Number of finished trials: {}".format(len(study.trials)))

    print("Best trial:")
    trial = study.best_trial

    print("  Value: {}".format(trial.value))

    print("  Params: ")
    for key, value in trial.params.items():
        print("    {}: {}".format(key, value))

```

3.10. Подбор гиперпараметров для LightGBM с помощью Optuna LightGBT Tuner

<https://medium.com/optuna/lightgbm-tuner-new-optuna-integration-for-hyperparameter-optimization-43a2f3a2a2>

Наивный способ подбора гиперпараметров с помощью Optuna мог бы выглядеть так

```

def objective(trial):
    data, target = sklearn.datasets.load_breast_cancer(return_X_y=True)
    train_x, test_x, train_y, test_y = train_test_split(data, target, test_size=0.25)
    dtrain = lgb.Dataset(train_x, label=train_y)

    param = {

```

```

'objective': 'binary',
'metric': 'binary_logloss',
'lambda_l1': trial.suggest_loguniform('lambda_l1', 1e-8, 10.0),
'lambda_l2': trial.suggest_loguniform('lambda_l2', 1e-8, 10.0),
'num_leaves': trial.suggest_int('num_leaves', 2, 256),
'feature_fraction': trial.suggest_uniform('feature_fraction', 0.4, 1.0),
'bagging_fraction': trial.suggest_uniform('bagging_fraction', 0.4, 1.0),
'bagging_freq': trial.suggest_int('bagging_freq', 1, 7),
'min_child_samples': trial.suggest_int('min_child_samples', 5, 100),
}

gbm = lgb.train(param, dtrain)
preds = gbm.predict(test_x)
pred_labels = np.rint(preds)
accuracy = sklearn.metrics.accuracy_score(test_y, pred_labels)
return accuracy

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)

print('Number of finished trials:', len(study.trials))
print('Best trial:', study.best_trial.params)

```

В этом примере Optuna пытается найти наилучшую комбинацию 7 различных гиперпараметров (параметр регуляризации L_1 , число терминальных узлов `num_leaves` и пр.). Суммарное количество комбинаций это произведение всех пространств поиска каждого гиперпараметра.

Не смотря на то, что существует много методов поиска наилучшей комбинации гиперпараметров за наименьшее количество запусков, все же имеет смысл рассмотреть альтернативный *метод последовательного подбора* (stepwise algorithm) – он, как понятно из названия, подбирает параметры последовательно и итоговое пространство поиска будет представлять собой *сумму* (а не произведение!!!) пространств поиска.

Класс `optuna.integration.lightgbm.LightGBMTuner` реализует именно *последовательный алгоритм подбора гиперпараметров*.

`LightGBMTuner` *последовательно* (stepwise manner) подбирает следующие гиперпараметры :

- `lambda_l1`,
- `lambda_l2`,
- `num_leaves`,
- `featuer_fraction`,
- `bagging_fraction`,
- `bagging_freq`,
- `min_child_samples` (то же что и `min_data_in_leaf`).

Для того чтобы использовать `LightGBMTuner` нужно изменить лишь строку импорта библиотеки `ligthgbm`

```

import lightgbm as lgb # было
import optuna.integration.lightgm as lgb # стало!

from lightgbm import early_stopping
from lightgbm import log_evaluation
import sklearn.datasets
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

```

```

if __name__ == "__main__":
    data, target = sklearn.datasets.load_breast_cancer(return_X_y=True)
    train_x, val_x, train_y, val_y = train_test_split(data, target, test_size=0.25)
    dtrain = lgb.Dataset(train_x, label=train_y)
    dval = lgb.Dataset(val_x, label=val_y)

    params = {
        "objective": "binary",
        "metric": "binary_logloss",
        "verbosity": -1,
        "boosting_type": "gbdt",
    }

    model = lgb.train(  # здесь начинается ПОСЛЕДОВАТЕЛЬНЫЙ подбор гиперпараметров
        params,
        dtrain,
        valid_sets=[dtrain, dval],
        callbacks=[early_stopping(100), log_evaluation(100)],
        time_budget=600,
        model_dir="../boosters",  # сюда будут записываться модели бустера, например, 14.pkl
    )

    prediction = np.rint(model.predict(val_x, num_iteration=model.best_iteration))
    accuracy = accuracy_score(val_y, prediction)

    best_params = model.params
    print("Best params:", best_params)
    print(" Accuracy = {}".format(accuracy))
    print(" Params: ")
    for key, value in best_params.items():
        print(" {}: {}".format(key, value))

```

Подбор гиперпараметров начинается, когда вызывается функция `lgb.train()`, а построить прогноз можно так

```
cohen_kappa_score(y_test, np.rint(model.predict(X_test))) # 0.9750292056074766
```

3.11. Распределенное обучение с LightGBM

Получить доступ к LightGBM для работы в распределенном режиме на кластере Spark можно с помощью библиотеки SynapseML <https://microsoft.github.io/SynapseML/>.

3.12. Конспект статьи G. Ke LightGBM. A Highly Efficient Gradient ...

В статье предлагается две техники:

- Gradient-based One-Side Sampling (GOSS): авторы считают, что экземпляры данных с большими градиентами играют более важную роль в вычислении информационного прироста и GOSS может получить более аккуратные оценки информационного прироста на значительно меньшем наборе данных,
- Exclusive Feature Bundling (EFB): с помощью этой техники авторы предлагают объединять взаимоисключающие признаки (то есть признаки, которые редко принимают ненулевые значения одновременно) для снижения размерности задачи по признакам. Поиск оптимального

объединения взаимоисключающих признаков это NP-трудная задача, но авторы показывают, что жадный алгоритм может найти приемлемую аппроксимацию решения (и таким образом эффективно снизить размерность задачи по признакам без значительного снижения точности определения точки разбиения в узле).

Эту реализацию деревьев решения градиентного бустинга (Gradient Boosting Decision Tree, GBDT) с GOSS и EFB авторы называют LightGBM.

Gradient-based One-Side Sampling (GOSS). Экземпляры данных с различными градиентами дают различный вклад при вычислении информационного прироста. В частности, согласно определению информационного прироста, экземпляры с *большим градиентом*² (то есть недообученные экземпляры) будут давать *больший вклад* в информационный прирост (information gain). Поэтому для того чтобы сохранить точность информационного прироста выгоднее сохранять экземпляры с большим градиентом (то есть с градиентом, превышающим некоторый порог, или с градиентом верхних персентиляй), а случайно отбрасывать экземпляры с малым градиентом. Авторы показывают, что такая подвыборка экземпляров может давать более аккуратные оценки информационного прироста, чем простая равномерная выборка.

Exclusive Feature Bundling (EFB). Обычно, несмотря на большое количество признаков в матрице признакового описания объекта, пространство признаков *довольно разреженное*. В разреженном признаковом пространстве многие признаки (почти) уникальные, то есть они редко принимают ненулевые значения одновременно. Авторы сводят задачу *оптимального объединения взаимоисключающих признаков* к задаче о *раскраске графа* (используя в качестве вершин графа признаки и соединяя каждые два признака ребром, если они *не взаимоисключающие*) и решают ее с помощью жадного алгоритма.

3.12.1. GBDT и анализ сложности

GBDT – это ансамбль моделей решающих деревьев, которые обучаются *последовательно*. На каждой итерации градиентного бустинга решающие деревья (то есть базовые алгоритмы) подгоняются под вектор *антиградиента* (еще его называют псевдо-остатками, residual errors).

Основные затраты в градиентном бустинге связаны с обучением (лучше сказать с *постстроением*) решающих деревьев, а самая затратная по времени часть это *поиск наилучших точек разбиения* (the best split points).

Один из самых популярных алгоритмов поиска точек разбиения – это *алгоритм предварительной сортировки* (pre-sorted algorithm), который перебирает все возможные точки разбиения на предварительно отсортированных значениях признака. Этот алгоритм простой и может найти оптимальные точки разбиения, но он неэффективен ни с точки зрения скорости обучения, ни с точки зрения потребления памяти.

Другой популярный алгоритм поиска точек разбиения – это *гистограммный алгоритм* (histogram-based algorithm). Вместо того чтобы искать точки разбиения на отсортированном признаке, гистограммный алгоритм разбивает непрерывные вещественные признаки на бины и использует эти бины для построения гистограмм признаков во время обучения. Этот алгоритм эффективен и по скорости обучения, и по потреблению памяти (и потому авторы LightGBM решили свою работу развивать именно на нем).

У операции *построения гистограммы* временная сложность $O(\#data \times \#feature)$, а у операции *поиска разбиения* – $O(\#bin \times \#feature)$. Поскольку бинов $\#bin$ обычно значительно меньше,

²В этой статье под большими или малыми градиентами авторы понимают абсолютные значения градиентов

чем экземпляров $\#data$, временная сложность операции построения гистограммы значительно больше временной сложности операции поиска разбиения. Если мы сможем снизить $\#data$ или $\#feature$, то получится значительно ускорить обучение градиентного бустинга.

В Python-библиотеке Scikit-learn и R-библиотеке gbm реализован алгоритм *предварительной сортировки* (плохо, неэффективный алгоритм), а XGBoost поддерживает и алгоритм предварительной сортировки (`tree_method=exact`), и гистограммный алгоритм (`tree_method=hist`).

Для снижения размера обучающего набора данных обычно делают подвыборку. Например, экземпляры могут быть отфильтрованы по их весу, который не превышает некоторый заданный порог. Стохастический градиентный бустинг Фридмана (Stochastic Gradient Boosting) на каждой итерации для обучения слабых учеников (weak learners) использует случайную подвыборку.

Для того чтобы сократить число признаков, кажется естественным отфильтровать слабые признаки. Обычно эта процедура выполняется, например, с помощью метода главных компонент (PCA). Однако, PCA предполагает, что признаки содержать избыточную информацию, что не всегда так (признаки обычно проектируются с учетом их индивидуального вклада и удаление любого из них может повлиять на точность обучения).

Обычно крупномасштабные наборе данных, которые используются в реальных приложениях довольно *разрежены*. GBDT с алгоритмом предварительной сортировки может снизить временные издержки на обучение, игнорируя признаки с нулевыми значениями. Однако, GBDT с гистограммным алгоритмом не имеет эффективного решения для разреженной оптимизации. Причина в том, что гистограммный алгоритм должен извлекать значения признаков из бинов.

3.12.2. Gradient-based One-Side Sampling, GOSS

Описание алгоритма В адаптивном бустинге вес экземпляра служит хорошим показателем важности этого экземпляра. Однако в градиентном бустинге у экземпляров нет своих весов и потому метод отбора, предлагаемый AdaBoost, не может быть применен на прямую. К счастью, мы замечаем, что градиент каждого экземпляра в градиентном бустинге сообщает полезную информацию по выборке. Если у экземпляра *небольшой градиент*, то значит и *ошибка* на этом экземпляре тоже *небольшая*. Идея состоит в том, чтобы отбросить экземпляры с малым градиентом. Однако, это приведет к тому, что распределение данных изменится и снизит точность обучающей модели. Чтобы обойти эту проблему авторы предлагают метод GOSS.

GOSS учитывает *все* экземпляры с *большим градиентом* и *случайно* отбирает экземпляры с *малым градиентом*. Для того чтобы компенсировать искажение распределения данных, при вычислении информационного прироста GOSS вводит постоянный множитель для экземпляров с малым градиентом. Сначала GOSS сортирует экземпляры обучающего набора по абсолютной величине градиента и выбирает верхние $a \times 100\%$ экземпляров. Затем он случайно отбирает $b \times 100\%$ экземпляров из оставшегося набора данных. После чего при вычислении информационного прироста GOSS умножает экземпляры с малым градиентом на $\frac{1-a}{b}$. Поступая таким образом, мы делаем акцент на недообученных экземплярах, не изменяя существенно исходное распределение данных.

Теоретический анализ GBDT использует решающие деревья для обучения функции, отображающей входное пространство \mathcal{X}^s в пространство градиентов \mathcal{G} . Предположим, что у нас есть обучающее множество n независимых одинаково распределенных (i.i.d) экземпляров $\{x_i, \dots, x_n\}$, в котором каждый x_i это вектор размерности s в пространстве \mathcal{X}^s . На каждой итерации гради-

ентного бустинга отрицательный градиент целевой функции по отношению к выходу модели обозначается как $\{g_1, \dots, g_n\}$. Модель решающего дерева каждый узел разбивает *по самому информативному признаку* (то есть *по признаку с наибольшим информационным приростом*).

Для GBDT информационный прирост обычно оценивается по дисперсии после разбиения

$$V_{j|O}(d) = \frac{1}{n_O} \left(\frac{(\sum_{\{x_i \in O : x_{ij} \leq d\}} g_i)^2}{n_{l|O}^j(d)} + \frac{(\sum_{\{x_i \in O : x_{ij} > d\}} g_i)^2}{n_{r|O}^j(d)} \right),$$

где O – обучающий набор данных, попавших в рассматриваемый узел дерева; j – признак, по которому строится разбиение; d – пороговое значение; $n_O = \sum I[x_i \in O]$ – число экземпляров, попавших в рассматриваемый узел дерева; $n_{l|O}^j(d) = \sum I[x_i \in O : x_{ij} \leq d]$ – число экземпляров, попавших в левый дочерний узел; $n_{r|O}^j(d) = \sum I[x_i \in O : x_{ij} > d]$ – число экземпляров, попавших в правый дочерний узел.

Для признака j алгоритм решающего дерева выбирает такое пороговое значение d , которое максимизирует дисперсию – то есть $d_j^* = \arg \max_d V_j(d)$ – и вычисляет наибольший информационный прирост $V_j(d_j^*)$. Затем данные разбиваются по признаку j^* и значению d_{j^*} на левый и правый дочерние узлы.

В предлагаемом методе GOSS:

- обучающие экземпляры сначала сортируются по их абсолютному значению градиента в порядке убывания,
- затем сохраняется заданная доля верхних $a \times 100\%$ экземпляров с большим градиентом и получается множество экземпляров A ,
- потом из оставшегося множества A^c , состоящего из $(1 - a) \times 100\%$ экземпляров с малым градиентом, случайно отбираются экземпляры множества B размера $b \times |A^c|$,
- и наконец, экземпляры разбиваются по оценке дисперсии прироста (variance gain) $\tilde{V}_j(d)$

$$\tilde{V}_j(d) = \frac{1}{n} \left(\frac{(\sum_{x_i \in A_l} g_i + \frac{1-a}{b} \sum_{x_i \in B_l} g_i)^2}{n_l^j(d)} + \frac{(\sum_{x_i \in A_r} g_i + \frac{1-a}{b} \sum_{x_i \in B_r} g_i)^2}{n_r^j(d)} \right),$$

где $A_l = \{x_i \in A : x_{ij} \leq d\}$ – множество экземпляров с большим градиентом (попавших в левый дочерний узел) со значением j -ого признака непревышающим значение порога d ; $A_r = \{x_i \in A : x_{ij} > d\}$ – множество экземпляров с большим градиентом (попавших в правый дочерний узел) со значением j -ого признака меньшим значения порога d ; $B_l = \{x_i \in B : x_{ij} \leq d\}$ – множество экземпляров с малым значением градиента (попавших в левый дочерний узел) со значением j -ого признака непревышающего значения порога d ; $B_r = \{x_i \in B : x_{ij} > d\}$ – множество экземпляров с малым градиентом (попавших в правый дочерний узел) со значением j -ого признака меньшим значения порога d и коэффициент $\frac{1-a}{b}$ используемый для нормализации суммы градиентов по B .

Таким образом, в GOSS для вычисления точки разбиения используется оценка дисперсии прироста по малому подмножеству экземпляров $\tilde{V}_j(d)$ вместо точной оценки $V_j(d)$ по всем экземплярам и потому вычислительные затраты могут быть значительно снижены. В статье приводится теорема, которая показывает, что GOSS «не просадит» точность модели на обучающем поднаборе данных и превзойдет случайную выборку.

Из теоремы следует, что:

- на больших наборах данных предлагаемая аппроксимация обладает достаточной точностью,
- случайная подвыборка это специальный случай GOSS при $a = 0$.

3.12.3. Exclusive Feature Bundling

Высокоразмерные данные обычно *сильно разрежены*. В разреженном пространстве признаков многие признаки являются взаимоисключающими, то есть они никогда не принимают ненулевые значения одновременно. Мы можем безопасно объединить взаимоисключающие признаки в один. Тогда временная сложность построения гистограммы изменится с $O(\#data \times \#features)$ на $O(\#data \times \#bundle)$, при том, что $\#bundle \ll \#feature$.

Однако сначала нужно ответить на следующие вопросы: какие признаки должны быть объединены? и как построить объединение (bundle)?

Задача объединения признаков в меньшее количество групп (bundles) это NP-трудная задача. Доказательство: описанная задача сводится к задаче о раскраске графа, а она NP-трудная.

Признаки это вершины графа. Легко видеть, что набор вершин графа, окрашенных в один и тот же цвет, представляют собой связку взаимоисключающих признаков (exclusive bundle of features). Задача поиска оптимального объединения признаков относится к классу NP-трудных, а значит невозможно получить точное решение за полиномиальное время.

Для того чтобы найти хорошее приближение, авторы сводят задачу оптимального объединения к задаче о раскраске графа, принимая признаки в качестве вершин графа и соединяя ребрами каждые два признака, если они *не являются взаимоисключающими*. В итоге удается получить достаточно эффективный жадный алгоритм (с постоянным коэффициентом приближения). Кроме того, обычно многие признаки (хотя и не являются на 100% взаимоисключающими) также редко принимают ненулевые значения одновременно. Если алгоритм сможет разрешить небольшую долю конфликтов, то мы сможем еще сократить количество объединений и как следствие снизить вычислительную нагрузку.

Итак, сначала мы строим граф с взвешенными ребрами, веса которого указывают на суммарное число конфликтов между признаками. Затем мы сортируем признаки по их степеням в графе (в порядке убывания). И наконец, мы проверяем каждый признак в упорядоченном списке и либо назначаем его существующему объединению с малым конфликтом (контролируется параметром γ), либо создаем новое объединение. Временная сложность жадного объединения будет $O(\#feature^2)$ и эта процедура выполняется только один раз, перед обучением. Такая временная сложность приемлема, если признаков немного.

Авторы предлагают более эффективную стратегию объединения без построения графа (Merge Exclusive Features): упорядочивание по числу ненулевых значений, что эквивалентно упорядочиванию по степеням, поскольку большее число ненулевых значений обычно приводит к более высокой вероятности конфликтов. Поскольку гистограммный алгоритм хранит дискретные бины вместо непрерывных значений признаков, мы можем сконструировать объединение признаков, разрешив эксклюзивным признакам находиться в разных бинах. Этого можно добиться добавив смещение ко всем значениям исходных признаков. Например, пусть у нас есть объединение из двух признаков. Изначально, признак A принимает значения из $[0, 10)$, а признак B – из диапазона $[0, 20)$. Тогда ко всем значениям признака B можно добавить смещение 10 и тогда признак B будет принимать значение из $[10, 30)$. После этого можно *слиять* (merge) признаки A и B та-

ким образом, чтобы получился *новый признак* (замещающий два старых A и B), принимающий значения из диапазона $[0, 30]$.

EFB алгоритм объединяет множество экзклузивных признаков в гораздо меньшее число плотных признаков, что позволяет избежать ненужных вычислений для нулевых признаков. Еще можно оптимизировать гистограммный алгоритм таким образом, чтобы он игнорировал нулевые значения признаков, используя таблицу для каждого признака для записи ненулевых значений. Тогда временная сложность построения гистограммы для одного признака изменится с $O(\#data)$ на $O(\#non_zero_data)$. Однако этот метод требует дополнительной памяти и вычислительных затрат для поддержания этих таблиц (одна таблица для каждого признака) на протяжении всего процесса роста дерева.

3.12.4. Обсуждение

Как показывают вычислительные эксперименты алгоритм выборки GOSS значительно более эффективен, чем случайная выборка. EFB очень эффективен на разреженных признаковых пространствах (потому как сливают большое число разреженных признаков в небольшое число подгрупп плотных признаков).

LightGBM значительно эффективнее XGBoost и алгоритма стохастического градиентного бустинга как с точки зрения потребления памяти, так и с точки зрения временных издержек.

4. Приемы работы с библиотекой подбора гиперпараметров Optuna

<https://optuna.org/>

Базовые понятия:

- Study – оптимизация базируется на целевой функции,
- Trial – одиночный запуск целевой функции.

Optuna прекрасно работает и с Scikit-learn, и с XGBoost, и с LightGBM, и с прочими популярными библиотеками.

Обычно Optuna используется для подбора гиперпараметров, но может быть использована и для оптимизации пользовательских функций. Для примера рассмотрим простую квадратическую функцию $(x - 2)^2$

```
import optune

def objective(trail):
    x = trail.suggest_float("x", -10, 10)
    return (x - 2) ** 2

study = optuna.create_study()
study.optimize(objective, n_trials=100)
```

Лучшие найденные параметры можно получить так

```
study.best_params # {'x': 1.9914500091892555}
```

Можно запустить оптимизацию еще на 100 запусков

```
study.optimize(objective, n_trials=100)
len(study.trials) # 200
```

Для выбора значений гиперпараметров Optuna предлагает следующие функции:

- o `optuna.trial.Trial.suggest_categorical()` для категориальных параметров,
- o `optuna.trial.Trial.suggest_int()` для целочисленных параметров,
- o `optuna.trial.Trial.suggest_float()` для вещественных параметров.

Можно указывать шаг (`step`) и можно строить логарифмическую шкалу (`log`)

```
import optuna

def objective(trial):
    optimizer = trial.suggest_categorical("optimizer", ["MomentumSGD", "Adam"])
    num_layers = trial.suggest_int("num_layers", 1, 3)
    num_channels = trial.suggest_int("num_channels", 32, 512, log=True)
    num_units = trial.suggest_int("num_units", 10, 100, step=5)
    dropout_rate = trial.suggest_float("dropout_rate", 1e-5, 1e-2, log=True)
    drop_path_rate = trial.suggest_float("drop_path_rate", 0.0, 1.0, step=0.1)
```

4.1. Эффективные алгоритмы оптимизации

4.1.1. Алгоритмы выборки

Optuna предлагает следующие алгоритмы выборки:

- o Grid Search реализован в `GridSampler`,
- o Random Search реализован в `RandomSampler`,
- o Парзеновский алгоритм реализован в `TPESampler` (default),
- o CMA-ES реализован в `CmaEsSampler`,
- o `PartialFixedSampler`,
- o Генетический алгоритм недоминирующей сортировки `NSGAIISampler`,
- o Алгоритм квази Монте-Карло реализован в `QMCSampler`.

```
study = optuna.create_study()
print(study.sampler) # <optuna.samplers._tpe.sampler.TPESampler at 0x7fd7587ded60>
```

По умолчанию используются Парзеновские деревья, но если требуется использовать какую-то другую стратегию выборки, то можно сделать так

```
study = optuna.create_study(sampler=optuna.samplers.RandomSampler())
print(study.sampler) # <optuna.samplers._random.RandomSampler at 0x7fd758737bb0>

study = optuna.create_study(sampler=optuna.samplers.CmaEsSampler())
print(study.sampler) # <optuna.samplers._cmaes.CmaEsSampler at 0x7fd7587de850>
```

4.1.2. Алгоритмы подрезки

Алгоритмы подрезки (pruners) автоматически останавливают запуски с низким потенциалом (automated early-stopping).

Optuna предлагает следующие алгоритмы подрезки:

- o Алгоритм медианной подрезки `MedianPruner` (чаще используется этот алгоритм),
- o Алгоритм без подрезки `NopPruner`,
- o Алгоритм, управляющий подрезкой с помощью допуска `PatientPruner`,
- o Алгоритм подрезки заданного процентиля `PercentilePruner`,
- o Асинхронный алгоритм последовательного деления пополам `SuccessiveHalvingPruner`,
- o `HyperbandPruner`,

- Алгоритм подрезки по порогу `ThresholdPruner`.

Для задач, не связанных с глубоким обучением, эффективнее всего следующие пары «алгоритм выборки – алгоритм подрезки»:

- Для `RandomSampler`, лучше `MedianPruner`,
- Для `TPESampler`, лучше `HyperbandPruner`.

Для XGBoost использовать подрезку проще всего с помощью специального класса `XGBoostPruningCallback`

```
def objective(trial):
    ...
    pruning_callback = optuna.integration.XGBoostPruningCallback(trial, "validation-error")

    bst = xgb.train(
        param,
        dtrain,
        evals=[(dvalid, "validation")],
        callbacks=[pruning_callback] # <- NB
    )

study = optuna.create_study()
study.optimize(objective, n_trials=200)
study.best_trial.params
```

4.2. Сохранение и восстановление сессии исследований с помощью реляционной базы данных

Можно создать устойчивую сессию исследований следующим образом

```
import logging
import sys

import optuna

# Add stream handler of stdout to show the messages
optuna.logging.get_logger("optuna").addHandler(logging.StreamHandler(sys.stdout))
study_name = "example-study" # Unique identifier of the study.
storage_name = "sqlite:///{}.db".format(study_name)
study = optuna.create_study(study_name=study_name, storage=storage_name)
# A new study created in RDB with name: example-study
```

Пусть целевая выглядит так

```
def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    return (x - 2) ** 2

study.optimize(objective, n_trials=3)
"""

Trial 0 finished with value: 88.4056436341533 and parameters: {'x': -7.402427539425831}. Best is
trial 0 with value: 88.4056436341533.
Trial 1 finished with value: 2.2903082473653392e-05 and parameters: {'x': 1.9952142834942244}.
Best is trial 1 with value: 2.2903082473653392e-05.
Trial 2 finished with value: 0.8965355981899926 and parameters: {'x': 1.0531443625398893}. Best
is trial 1 with value: 2.2903082473653392e-05.
"""


```

Восстановить сессию можно так

```

study = optuna.create_study(study_name=study_name, storage=storage_name, load_if_exists=True)
study.optimize(objective, n_trials=3)
"""
Using an existing study with name 'example-study' instead of creating a new one.
Trial 3 finished with value: 55.80972760725184 and parameters: {'x': -5.47059084726582}. Best is
    trial 1 with value: 2.2903082473653392e-05.
Trial 4 finished with value: 8.03129355496931 and parameters: {'x': -0.8339247265050869}. Best
    is trial 1 with value: 2.2903082473653392e-05.
Trial 5 finished with value: 49.229175309978025 and parameters: {'x': 9.016350569204622}. Best
    is trial 1 with value: 2.2903082473653392e-05.
"""

```

Изучить наполнение базы данных можно так

```

import sqlite3

conn = sqlite3.connect("./example-study.db")
cur = conn.cursor()

cur.execute("SELECT name FROM sqlite_master WHERE type='table';") # <sqlite3.Cursor at 0
x7f289382be30>
print(cur.fetchall())
"""
[('studies',),
 ('version_info',),
 ('study_directions',),
 ('study_user_attributes',),
 ('study_system_attributes',),
 ('trials',),
 ('trial_user_attributes',),
 ('trial_system_attributes',),
 ('trial_params',),
 ('trial_values',),
 ('trial_intermediate_values',),
 ('trial_heartbeats',),
 ('alembic_version',)]
"""

cur.execute("SELECT * FROM study_user_attributes;").fetchall()
# [(1, 1, 'contributors', ['Leor Finkelberg'])]

```

4.3. Мульти-целевая оптимизация в Optuna

Если требуется минимизировать одну целевую функцию и максимизировать другую, то обе функции можно описать в функции `objective(trial)` и возвращать кортеж значений, ассоциированных с этими функциями

```

def objective(trial):
    ...
    return flops, accuracy

study = optuna.create_study(direction=["minimize", "maximize"])
study.optimize(objective, n_trials_30, timeout=300)

```

4.4. Пользовательские атрибуты

Пользовательские атрибуты можно добавить и в сессию исследования (`study`), и в запуск (`trial`)

```

import sklearn.datasets
import sklearn.model_selection
import sklearn.svm

import optuna

study = optuna.create_study(storage="sqlite:///example.db")
study.set_user_attr("contributors", ["Akiba", "Sano"])
study.set_user_attr("dataset", "MNIST")

```

Вывести все пользовательские атрибуты можно так

```

study.user_attrs
study_summaries = optuna.get_all_study_summaries("sqlite:///example-study.db")
study_summaries[0].user_attrs # {"contributors": ["Akiba", "Sano"], "dataset": "MNIST"}

```

А так можно добавить пользовательские атрибуты в запуск

```

def objective(trial):
    iris = sklearn.datasets.load_iris()
    x, y = iris.data, iris.target

    svc_c = trial.suggest_float("svc_c", 1e-10, 1e10, log=True)
    clf = sklearn.svm.SVC(C=svc_c)
    accuracy = sklearn.model_selection.cross_val_score(clf, x, y).mean()

    trial.set_user_attr("accuracy", accuracy)

    return 1.0 - accuracy # return error for minimization

study.optimize(objective, n_trials=1)

study.trials[0].user_attrs

```

4.5. Пользовательские семплеры

Благодаря пользовательским семплерам (алгоритмам выборки) можно:

- экспериментировать со своими алгоритмами выборки,
- реализовывать алгоритмы под задачу для повышения оптимизации,
- оборачивать другие библиотеки оптимизации, чтобы потом их внедрить в конвейеры Optuna.

Для того чтобы создать новый семплер, нужно определить класс, наследующий от `BaseSampler`.

У этого базового класса три абстрактных метода: `infer_relative_search_space()`, `sample_relative()` и `sample_independent()`.

Optuna поддерживает два типа семплирования:

- относительное семплирование (relative sampling), которое может учитывать корреляцию параметров в запуске,
- независимое семплирование (independent sampling), которое семплирует параметры независимо.

Пример реализации семплера на базе метода иметации отжига

```

import numpy as np
import optuna

```

```

class SimulatedAnnealingSampler(optuna.samplers.BaseSampler):
    def __init__(self, temperature=100):
        self._rng = np.random.RandomState()
        self._temperature = temperature # Current temperature.
        self._current_trial = None # Current state.

    def sample_relative(self, study, trial, search_space):
        if search_space == {}:
            return {}

        # Simulated Annealing algorithm.
        # 1. Calculate transition probability.
        prev_trial = study.trials[-2]
        if self._current_trial is None or prev_trial.value <= self._current_trial.value:
            probability = 1.0
        else:
            probability = np.exp(
                (self._current_trial.value - prev_trial.value) / self._temperature
            )
        self._temperature *= 0.9 # Decrease temperature.

        # 2. Transit the current state if the previous result is accepted.
        if self._rng.uniform(0, 1) < probability:
            self._current_trial = prev_trial

        # 3. Sample parameters from the neighborhood of the current point.
        # The sampled parameters will be used during the next execution of
        # the objective function passed to the study.
        params = {}
        for param_name, param_distribution in search_space.items():
            if (
                not isinstance(param_distribution, optuna.distributions.FloatDistribution)
                or (param_distribution.step is not None and param_distribution.step != 1)
                or param_distribution.log
            ):
                msg = (
                    "Only suggest_float() with 'step' 'None' or 1.0 and"
                    " 'log' 'False' is supported"
                )
                raise NotImplementedError(msg)

            current_value = self._current_trial.params[param_name]
            width = (param_distribution.high - param_distribution.low) * 0.1
            neighbor_low = max(current_value - width, param_distribution.low)
            neighbor_high = min(current_value + width, param_distribution.high)
            params[param_name] = self._rng.uniform(neighbor_low, neighbor_high)

        return params

    # The rest are unrelated to SA algorithm: boilerplate
    def infer_relative_search_space(self, study, trial):
        return optuna.samplers.intersection_search_space(study)

    def sample_independent(self, study, trial, param_name, param_distribution):
        independent_sampler = optuna.samplers.RandomSampler()
        return independent_sampler.sample_independent(study, trial, param_name, param_distribution)

```

Использовать пользовательский сэмплер можно также как и встроенные сэмплеры

```

def objective(trial):
    x = trial.suggest_float("x", -10, 10)
    y = trial.suggest_float("y", -5, 5)
    return x**2 + y

sampler = SimulatedAnnealingSampler()
study = optuna.create_study(sampler=sampler)
study.optimize(objective, n_trials=100)

best_trial = study.best_trial
print("Best value: ", best_trial.value)
print("Parameters that achieve the best value: ", best_trial.params)

```

4.6. Пользовательские пранеры

Функция `create_study()` принимает необязательный аргумент, наследующий от класса `BasePruner`. Пранер должен переопределить абстрактный метод `prune()`, который принимает аргументы, связанные с `Study` и `Trial` и возвращает булево значение: `True`, если запуск был остановлен и `False` в противном случае.

Пример

```

import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import SGDClassifier

import optuna
from optuna.pruners import BasePruner
from optuna.trial._state import TrialState

class LastPlacePruner(BasePruner):
    def __init__(self, warmup_steps, warmup_trials):
        self._warmup_steps = warmup_steps
        self._warmup_trials = warmup_trials

    def prune(self, study: "optuna.study.Study", trial: "optuna.trial.FrozenTrial") -> bool:
        # Get the latest score reported from this trial
        step = trial.last_step

        if step: # trial.last_step == None when no scores have been reported yet
            this_score = trial.intermediate_values[step]

            # Get scores from other trials in the study reported at the same step
            completed_trials = study.get_trials(deepcopy=False, states=(TrialState.COMPLETE,))
            other_scores = [
                t.intermediate_values[step]
                for t in completed_trials
                if step in t.intermediate_values
            ]
            other_scores = sorted(other_scores)

            # Prune if this trial at this step has a lower value than all completed trials
            # at the same step. Note that steps will begin numbering at 0 in the objective
            # function definition below.
            if step >= self._warmup_steps and len(other_scores) > self._warmup_trials:
                if this_score < other_scores[0]:

```

```

print(f "prune() True: Trial {trial.number}, Step {step}, Score {this_score}")
return True

return False

def objective(trial):
    iris = load_iris()
    classes = np.unique(iris.target)
    X_train, X_valid, y_train, y_valid = train_test_split(
        iris.data, iris.target, train_size=100, test_size=50, random_state=0
    )

    loss = trial.suggest_categorical("loss", ["hinge", "log_loss", "perceptron"])
    alpha = trial.suggest_float("alpha", 0.00001, 0.001, log=True)
    clf = SGDClassifier(loss=loss, alpha=alpha, random_state=0)
    score = 0

    for step in range(0, 5):
        clf.partial_fit(X_train, y_train, classes=classes)
        score = clf.score(X_valid, y_valid)

        trial.report(score, step)

        if trial.should_prune():
            raise optuna.TrialPruned()

    return score

pruner = LastPlacePruner(warmup_steps=1, warmup_trials=5)
study = optuna.create_study(direction="maximize", pruner=pruner)
study.optimize(objective, n_trials=50)

```

4.7. Пользовательские callbacks для метода optimize

Следующий класс останавливает оптимизацию, когда количество остановок запуска порядд превышает некоторый порог

```

import optuna

class StopWhenTrialKeepBeingPrunedCallback:
    def __init__(self, threshold: int):
        self.threshold = threshold
        self._consecutive_pruned_count = 0

    def __call__(self, study: optuna.study.Study, trial: optuna.trial.FrozenTrial) -> None:
        if trial.state == optuna.trial.TrialState.PRUNED:
            self._consecutive_pruned_count += 1
        else:
            self._consecutive_pruned_count = 0

        if self._consecutive_pruned_count >= self.threshold:
            study.stop()

```

Целевая устроена так, чтобы запуски останавливались подряд после 5 итераций

```

def objective(trial):
    if trial.number > 4:

```

```

    raise optuna.TrialPruned

    return trial.suggest_float("x", 0, 1)

```

И запуск

```

import logging
import sys

# Add stream handler of stdout to show the messages
optuna.logging.get_logger("optuna").addHandler(logging.StreamHandler(sys.stdout))

study_stop_cb = StopWhenTrialKeepBeingPrunedCallback(2)
study = optuna.create_study()
study.optimize(objective, n_trials=10, callbacks=[study_stop_cb])

```

4.8. Ask-and-Tell Interface

Варианта с функцией `objective` часто оказывается достаточно, но все-таки у него есть ограничения. Более гибкий вариант можно построить так

```

study = optuna.create_study(direction="maximize")

n_trials = 10
for _ in range(n_trials):
    trial = study.ask() # 'trial' is a 'Trial' and not a 'FrozenTrial'.

    C = trial.suggest_float("C", 1e-7, 10.0, log=True)
    solver = trial.suggest_categorical("solver", ("lbfgs", "saga"))

    clf = LogisticRegression(C=C, solver=solver)
    clf.fit(X_train, y_train)
    val_accuracy = clf.score(X_test, y_test)

study.tell(trial, val_accuracy) # tell the pair of trial and objective value

```

`optuna.study.Study.ask()` создает запуск (триал), чтобы можно было семплировать гиперпараметры, а `optuna.study.Study.tell()` завершает триал. Таким образом можно подбирать гиперпараметры без функции `objective`.

Если требуется сделать оптимизацию более быстрой, то нужно явно передать экземпляр пранера

```

import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split

import optuna

X, y = load_iris(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
classes = np.unique(y)
n_train_iter = 100

# define study with hyperband pruner.
study = optuna.create_study(
    direction="maximize",

```

```

pruner=optuna.pruners.HyperbandPruner(
    min_resource=1, max_resource=n_train_iter, reduction_factor=3
),
)

for _ in range(20):
    trial = study.ask()

    alpha = trial.suggest_float("alpha", 0.0, 1.0)

    clf = SGDClassifier(alpha=alpha)
    pruned_trial = False

    for step in range(n_train_iter):
        clf.partial_fit(X_train, y_train, classes=classes)

        intermediate_value = clf.score(X_valid, y_valid)
        trial.report(intermediate_value, step)

        if trial.should_prune():
            pruned_trial = True
            break

    if pruned_trial:
        study.tell(trial, state=optuna.trial.TrialState.PRUNED) # tell the pruned state
    else:
        score = clf.score(X_valid, y_valid)
        study.tell(trial, score) # tell objective value

```

4.9. Define-and-Run

Еще можно сначала описать распределения гиперпараметров, а потом вызвать `optuna.study.Study.ask()`

```

distributions = {
    "C": optuna.distributions.FloatDistribution(1e-7, 10.0, log=True)
    "solver": optuna.distributions.CategoricalDistribution(("lbfgs", "saga")),
}

study = optuna.create_study(direction="maximize")
n_trials = 10

for _ in range(n_trials):
    trial = study.ask(distributions)

    C = trial.params["C"]
    solver = trial.params["solver"]

    clf = LogisticRegression(C=C, solver=solver)
    clf.fit(X_train, y_train)
    val_accuracy = clf.score(X_test, y_test)

    study.tell(trial, val_accuracy)

```

4.10. Пакетная оптимизация

Для ускорения оптимизации можно передавать массивы

```

def batched_objective(xs: np.ndarray, ys: np.ndarray):
    return xs**2 + ys

batch_size = 10
study = optuna.create_study(sampler=optuna.samplers.CmaEsSampler())

for _ in range(3):

    # create batch
    trial_numbers = []
    x_batch = []
    y_batch = []
    for _ in range(batch_size):
        trial = study.ask()
        trial_numbers.append(trial.number)
        x_batch.append(trial.suggest_float("x", -10, 10))
        y_batch.append(trial.suggest_float("y", -10, 10))

    # evaluate batched objective
    x_batch = np.array(x_batch)
    y_batch = np.array(y_batch)
    objectives = batched_objective(x_batch, y_batch)

    # finish all trials in the batch
    for trial_number, objective in zip(trial_numbers, objectives):
        study.tell(trial_number, objective)

```

4.11. Переиспользование лучшего запуска

Пусть есть гиперпараметры, подобранные на некотором фрагменте обучающего набора (для снижения вычислительной нагрузки), и теперь требуется обучить модель на полном обучающем наборе данных с использованием лучшей комбинации гиперпараметров.

`best_trial` предлагает интерфейс для повторного вычисления целевой функции с текущим лучшим набором гиперпараметров и возвращает объект `FrozenTrial`.

4.12. Создание целевой функции с дополнительными аргументами

Если нужно передать в целевую функцию что-то еще кроме `trial`, то можно:

1. Написать свой класс

```

import optuna

class Objective:
    def __init__(self, min_x, max_x):
        # Hold this implementation specific arguments as the fields of the class.
        self.min_x = min_x
        self.max_x = max_x

    def __call__(self, trial):
        # Calculate an objective value by using the extra arguments.
        x = trial.suggest_float("x", self.min_x, self.max_x)
        return (x - 2) ** 2

    # Execute an optimization by using an 'Objective' instance.

```

```
study = optuna.create_study()
study.optimize(Objective(-100, 100), n_trials=100)
```

2. Использовать замыкания

```
import optuna

# Objective function that takes three arguments.
def objective(trial, min_x, max_x):
    x = trial.suggest_float("x", min_x, max_x)
    return (x - 2) ** 2

# Extra arguments.
min_x = -100
max_x = 100

# Execute an optimization by using the above objective function wrapped by 'lambda'.
study = optuna.create_study()
study.optimize(lambda trial: objective(trial, min_x, max_x), n_trials=100)
```

Чтобы *переиспользовать* набор данных, а не загружать его для каждого триала можно передать ссылку на набор данных при создании экземпляра класса целевой функции https://github.com/optuna/optuna-examples/blob/main/sklearn/sklearn_additional_args.py

```
"""
It takes the Iris dataset by a constructor's argument
instead of loading it in each trial execution. This will speed up the execution of each trial
compared to 'sklearn_simple.py'.
"""

import optuna

import sklearn.datasets
import sklearn.ensemble
import sklearn.model_selection
import sklearn.svm


class Objective(object):
    def __init__(self, iris):
        self.iris = iris

    def __call__(self, trial):
        x, y = self.iris.data, self.iris.target

        classifier_name = trial.suggest_categorical("classifier", ["SVC", "RandomForest"])
        if classifier_name == "SVC":
            svc_c = trial.suggest_float("svc_c", 1e-10, 1e10, log=True)
            classifier_obj = sklearn.svm.SVC(C=svc_c, gamma="auto")
        else:
            rf_max_depth = trial.suggest_int("rf_max_depth", 2, 32, log=True)
            classifier_obj = sklearn.ensemble.RandomForestClassifier(
                max_depth=rf_max_depth, n_estimators=10
            )

        score = sklearn.model_selection.cross_val_score(classifier_obj, x, y, n_jobs=-1, cv=3)
        accuracy = score.mean()
        return accuracy
```

```

if __name__ == "__main__":
    # Load the dataset in advance for reusing it each trial execution.
    iris = sklearn.datasets.load_iris()
    objective = Objective(iris)

    study = optuna.create_study(direction="maximize")
    study.optimize(objective, n_trials=100)
    print(study.best_trial)

```

В этом примере роль функции `objective` играет метод `__call__()`. Обязательно у метода должен быть параметр `trial`.

4.13. Optuna можно использовать и без RDB-серверов

В простейшем случае Optuna работает в оперативной памяти (in-memory storage)

```

study = optuna.create_study()
study.optimize(objective)

```

Если нужно сохранять информацию по сессиям исследований, то можно воспользоваться SQLite в качестве локального хранилища

```

study = optuna.create_study(study_name="foo_study", storage="sqlite:///example.db")
study.optimize(objective) # # The state of 'study' will be persisted to the local SQLite file.

```

В простейшем случае, когда используется хранилище в памяти, можно просто сохранить объект сессии исследований (`study`) с помощью `pickle` или `joblib`

```

study = optuna.create_study()
joblib.dump(study, "study.pkl")

```

А затем восстановить

```

study = joblib.load("study.pkl")
study.best_trial.value
study.best_trial.params

```

ВАЖНО: Optuna не поддерживает сохранение/загрузку в разных версиях Optuna с `pickle`. Если требуется сохранять/восстанавливать сессии исследований в различных версиях Optuna, то следует использовать СУБД и обновить схему хранилища, если требуется.

4.14. Как сохранить модель машинного обучения, обученную в функции `objective`

Optuna сохраняет значения гиперпараметров, но не сохраняет модели машинного обучения или веса нейронных сетей. Чтобы сохранить модель, следует воспользоваться средствами используемой библиотеки машинного обучения. Рекомендуется сохранять номер запуска

`optuna.trial.Trial.number`

```

def objective(trial):
    svc_c = trial.suggest_float("svc_c", 1e-10, 1e10, log=True)
    clf = sklearn.svm.SVC(C=svc_c)
    clf.fit(X_train, y_train)

    # Save a trained model to a file.
    with open("{} .pickle".format(trial.number), "wb") as fout:
        pickle.dump(clf, fout)
    return 1.0 - accuracy_score(y_valid, clf.predict(X_valid))

```

```

study = optuna.create_study()
study.optimize(objective, n_trials=100)

# Load the best model.
with open("{}.pickle".format(study.best_trial.number), "rb") as fin:
    best_clf = pickle.load(fin)
print(accuracy_score(y_valid, best_clf.predict(X_valid)))

```

Чтобы получить воспроизводимые результаты следует зафиксировать ключ генератора псевдо-случайных чисел `seed` семплера

```

sampler = TPESampler(seed=10) # Make the sampler behave in a deterministic way.
study = optuna.create_study(sampler=sampler)
study.optimize(objective)

```

Однако, следует иметь ввиду, что в распределенном или параллельном режиме очень сложно получить воспроизводимые результаты, поэтому если нужно получить строго воспроизводимые результаты, то следует запускать исследования последовательно.

4.15. Как можно тестировать целевые функции

Для того чтобы целевая функция возвращала одни и те же значения, можно вызывать целевую функцию `objective` со словарем параметров, обернутым классом `FixedTrial`

```

# A test function of pytest
def test_objective():
    assert 1.0 == objective(FixedTrial({"x": 1.0, "y": 0}))
    assert -1.0 == objective(FixedTrial({"x": 0.0, "y": -1}))
    assert 0.0 == objective(FixedTrial({"x": -1.0, "y": 1}))

```

Если во время выполнения большого числа запусков ощущается нехватка памяти, то можно попробовать периодически запускать сборщик мусора `gc_after_trial`

```

def objective(trial):
    x = trial.suggest_float("x", -1.0, 1.0)
    y = trial.suggest_int("y", -5, 5)
    return x + y

study = optuna.create_study()
study.optimize(objective, n_trials=10, gc_after_trial=True)

# 'gc_after_trial=True' is more or less identical to the following.
study.optimize(objective, n_trials=10, callbacks=[lambda study, trial: gc.collect()])

```

ВАЖНО: Optuna поддерживает только «мягкие» ограничения (то есть ограничение можно нарушить, но целевая будет штрафовать за такое нарушение).

5. Приемы работы с AutoML-библиотекой FLAML

FLAML <https://github.com/microsoft/FLAML> – это легковесная библиотека эффективной автоматизации процессов машинного обучения, включая выбор модели, подбор гиперпараметров и пр.

5.1. Установка

Установить можно с помощью менеджера пакетов pip как

```
pip install flaml
```

5.2. Быстрый старт

Простейший пример использования

```
from flaml import AutoML

automl = AutoML()
automl.fit(X_train, y_train, task="regression")
automl.best_config_per_estimator
automl.best_estimator # rf
reg = automl.model # <flaml.automl.model.RandomForestEstimator at 0x7f8f5b056cd0>
reg.get_params()
reg.predict(X_test)
```

Можно сохранить лучшую конфигурацию

```
import json

automl.save_best_config("rf_best_config.json")

with open("./rf_best_config.json") as f:
    config: dict = json.load(f)

# В scikit-learn параметр max_leaves называется max_leaf_nodes
# и у параметра criterion нет значения 'entropy'
config["hyperparameters"]["max_leaf_nodes"] = config["hyperparameters"].pop("max_leaves")
config["hyperparameters"].update({"criterion": "friedman_mse"})

reg = RandomForestRegressor(**config)
reg.predict(X_test)
```

Можно сохранить обученную модель в формате pickle

```
import joblib
...
with open("./rf_automl.pkl", "wb") as f:
    joblib.dump(reg, f)

with open("./rf_automl.pkl", "rb") as f:
    reg = joblib.load(f)

reg.predict(X_test)
```

5.3. Основные параметры

Поддерживаются следующие типы задач:

- "classification": классификация на табличных данных,
- "regression": регрессия на табличных данных,
- "ts_forecast": прогнозирование на горизонт временного ряда,
- "ts_forecast_classification": классификация временных рядов,
- "ts_forecast_panel": прогнозирование на пакете временных рядов (multiple time series),

- и пр.

Поддерживаются следующие ошибки (или метрики качества, приведенные к ошибке)

- "log_loss": метрика по умолчанию для мультиклассовой классификации,
- "r2": $1 - R^2$ – определяется как метрика для минимизации; по умолчанию используется в задачах регрессии,
- rmse: RMSE,
- mse: MSE,
- mae: MAE,
- mape: MAPE,
- roc_auc: $1 - ROC_AUC$; по умолчанию используется в задачах бинарной классификации,
- roc_auc_ovr: $1 - ROC_AUC$ с multi_class="ovr",
- roc_auc_ovo: $1 - ROC_AUC$ с multi_class="ovo",
- roc_auc_weighted: $1 - ROC_AUC$ с average="weighted",
- roc_auc_ovr_weighted: $1 - ROC_AUC$ с multi_class="ovr" и average="weighted",
- roc_auc_ovo_weighted: $1 - ROC_AUC$ с multi_class="ovo" и average="weighted",
- f1: $1 - F_1$,
- micro_f1: $1 - F_1$ с average="micro",
- macro_f1: $1 - F_1$ с average="macro",
- и пр.

Поддерживаются следующие модели:

- lgbm: LGBEstimator для задач классификации, регрессии, ранжирования, прогнозирования на горизонт, классификации временных рядов. Гиперпараметры:
 - n_estimators,
 - num_leaves,
 - min_child_samples,
 - learning_rate,
 - log_max_bin,
 - colsample_bytree,
 - reg_alpha,
 - reg_lambda.
- xgboost: XGBoostSklearnEstimator для задач классификации, регрессии, ранжирования, прогнозирования на горизонт, классификации временных рядов. Гиперпараметры:
 - n_estimators,
 - num_leaves,
 - min_child_weight,
 - learning_rate,
 - log_max_bin,
 - colsample_bytree,
 - reg_alpha,
 - reg_lambda.

- **rf**: RandomForestEstimator для задач классификации, регрессии, прогнозирования на горизонт, классификации временных рядов. Гиперпараметры:
 - `n_estimators`,
 - `max_features`,
 - `max_leaves`,
 - `criterion` (только для классификации)
- **arima**: ARIMA для задачи "ts_forecast". Гиперпараметры: `p`, `d`, `q`,
- **sarimax**: SARIMAX для задачи "ts_forecast". Гиперпараметры: `p`, `d`, `q`, `P`, `D`, `Q`, `s`.
- **holt-winters**: модель Хольта-Винтерса (модель тройного экспоненциального сглаживания) для задачи "ts_forecast", Гиперпараметры:
 - `seasonal_periods`,
 - `seasonal`,
 - `use_boxcox`,
 - `trend`,
 - `damped_trend`.
- **temporal_fusion_transformer**: TemporalFusionTransformerEstimators для задачи "ts_forecast". Гиперпараметры:
 - `gradient_clip_val`,
 - `hidden_size`,
 - `hidden_continuous_size`,
 - `attention_head_size`,
 - `dropout`,
 - `learning_rate`.

5.4. Кастомизация пространства поиска

Чтобы кастомизировать пространство поиска для встроенных моделей, нужно переопределить метод `search_space`. Причем ключи-гиперпараметры должны включать: `domain` для описания области поиска и дополнительно `init_value` (просто значение по умолчанию гиперпараметра) и/или `low_cost_init_value`, которое задает самое дешевое в вычислительном плане значение гиперпараметра

ВАЖНО: если диапазон изменения параметра большой, то рекомендуется выполнять выборку в логарифмической шкале

```
{
  # широкий диапазон -> логарифмическая шкала
  "learning_rate": tune.loguniform(lower=1 / 1024, upper=1.0)
}
```

Если же диапазон невелик, то выборка выполняется в линейном масштабе

```
{
  # узкий диапазон -> линейная шкала
  "learning_rate": tune.uniform(lower=0.1, upper=0.2)
}
```

Если требуется квантованлизовать пространство поиска, то можно использовать `tune.qlograndint`, `tune.qloguniform` и пр.

```
{ # 0.1, 0.12, 0.14, ...
  "learning_rate": tune.quniform(lower=0.1, upper=0.2, q=0.02)
}
```

Пространство поиска можно скоректировать с помощью пользовательского класса

```
from flaml import tune
from flaml.automl.model import XGBoostSklearnEstimator

# или class XGBoost2D(XGBoostEstimator):
class XGBoost2D(XGBoostSklearnEstimator):
    @classmethod
    def search_space(cls, data_size, task):
        upper = min(32_768, int(data_size))

        # подбираем только 2 гиперпараметра
        return {
            "n_estimators": {
                "domain": tune.lograndint(lower=4, upper=upper),
                "low_cost_init_value": 4,
            },
            "max_leaves": {
                "domain": tune.lograndint(lower=4, upper=upper),
                "low_cost_init_value": 4,
            }
        }
```

После нужно обязательно зарегистрировать пользовательский класс в экземпляре AutoML и указать либо максимальное число итераций, либо бюджет времени

```
automl = AutoML()
automl.add_learner(learner_name="my_xgb", learner_class=XGBoost2D)
automl.fit(X_train, y_train, task="regression", estimator_list=["my_xgb"], max_iter=100)
```

Или можно передать методу `fit` экземпляра класса модели AutoML (обязательно нужно задать `max_iter` или `time_budget` (задается в секундах), в противном случае изменения не будут учитываться)

```
custom_hp = {
    "xgboost": {
        "n_estimators": {
            "domain": tune.lograndint(lower=new_lower, upper=new_upper),
            "low_cost_init_value": new_lower,
        },
    },
    "rf": {
        "max_leaves": {
            "domain": None # disable search
        },
    },
    "lgbm": {
        "subsample": {
            "domain": tune.uniform(lower=0.1, upper=0.9),
            "init_value": 1.0,
        },
        "subsample_freq": {
            "domain": 1 # subsample_freq must > 0 to enable subsample
        }
    }
}
```

```
        },
    },
}
```

То есть

```
from flaml import tune

automl = AutoML()

custom_hp = {
    "xgboost": {
        "gamma": {
            "domain": tune.uniform(lower=1, upper=10),
            "init_value": 1,
            "low_cost_init_value": 10,
        },
        "reg_alpha": {
            "domain": None, # disable search
        },
        "n_estimators": {
            "domain": tune.lograndint(lower=100, upper=350),
            "low_cost_init_value": 100,
        },
    }
}

automl.fit(
    X_train, y_train,
    task="regression",
    estimators=[ "xgboost" ],
    custom_hp=custom_hp,
    time_budget=120 # или max_iter=100
)
```

Можно задать ограничение на время (в секундах) обучения модели и на время построения прогноза с помощью параметров `train_time_limit` и `pred_time_limit`.

Еще можно ограничить ошибки на обучающем и/или валидационном поднаборах

```
metric_constraints = [( "train_loss", "<=", 0.1), ( "val_loss", "<=", 0.1)]
automl.fit(X_train, y_train, max_iter=100, train_time_limit=1, metric_constraints=
    metric_constraints)
```

5.5. Stacking

Модели с подобранными гиперпараметрами можно собрать в ансамбль с помощью процедуры стекинга: либо выставить `ensemble=True`, либо передать словарь. Когда `ensemble=True`, автоматически строится стекинговая модель

```
automl.fit(
    X_train, y_train, task="regression",
    ensemble={
        "final_estimator": RidgeCV(), # мета-модель
        "passthrough": False, # не передавать исходные признаки
    },
    max_iter=100,
)
```

```

automl.model
"""
StackingRegressor(estimators=[('rf',
    <flaml.automl.model.RandomForestEstimator object at 0x7f3425faddee0>),
('extra_tree',
    <flaml.automl.model.ExtraTreesEstimator object at 0x7f3425fb2a30>),
('lgbm',
    <flaml.automl.model.LGBMEstimator object at 0x7f3425fb2550>),
('xgboost',
    <flaml.automl.model.XGBoostSklearnEstimator object at 0x7f3425fb2a60>),
('xgb_limitdepth',
    <flaml.automl.model.XGBoostLimitDepthEstimator object at 0x7f3425fb4580>)],
final_estimator=RidgeCV(), n_jobs=1)
"""
automl.model.predict(X_test)

```

5.6. Стратегии разбиения на обучающий и валидационный поднаборы данных

По умолчанию FLAML разбивает набор данных автоматически в зависимости от размера набора данных и ограничении на время поиска конфигурации. Но можно управлять разбиением и вручную. Можно либо использовать отложенную выборку (`eval_method="holdout"`), либо перекрестную проверку `eval_method="cv"`.

Для отложенной выборки:

- `split_ratio`: доля валидационного набора (по умолчанию 0.1),
- `X_val, y_val`: отдельный валидационный набор. Если передан, то валидационные метрики вычисляются на этом валидационном наборе. Если не передан, то валидационный набор выделяется из обучающего набора. После подбора гиперпараметров, модель заново обучается на лучшей комбинации на полном обучающем наборе.

Для перекрестной проверки можно задать число фолдов `n_splits` (по умолчанию 5).

Поддерживаются следующие стратегии `split_type`:

- для задач классификации: `auto (stratified)`, `stratified`, `uniform`, `time`, `group`,
- для задач регрессии: `auto (uniform)`, `uniform`, `time`,
- для временных рядов: `auto` или `time`
- и пр.

ВАЖНО: когда последовательный и параллельный подбор гиперпараметров имеют схожие временные издержки, то предпочтение следует отдавать *последовательной* стратегии подбора.

5.7. Теплый старт

Можно использовать «теплый старт» на базе полученных конфигураций гиперпараметров для каждой модели.

```

automl1 = AutoML()
# обучаемся в течение 1 часа
automl1.fit(X_train, y_train, time_budget=3600)

automl2 = AutoML()
automl2.fit(
    X_train, y_train,
    # продолжаем обучение в течение еще 2 часов
    time_budget=7200,
    starting_points=automl1.best_config_per_estimator

```

5.8. Получение лучшей модели

Чтобы получить лучшую модель, нужно запросить значение атрибута `model`

```
automl.fit(X_train, y_train, task="regression")
automl.best_estimator # xgboost
automl.model # <flaml.automl.model.XGBoostSklearnEstimator at 0x7f3425adea00>
```

Получить саму обученную модель можно так

```
automl.model.estimator
"""
XGBRegressor(base_score=None, booster=None, callbacks=[],
    colsample_bylevel=0.9573549467024939, colsample_bynode=None,
    colsample_bytree=0.7496135390795017, early_stopping_rounds=None,
    enable_categorical=False, eval_metric=None, feature_types=None,
    gamma=None, gpu_id=None, grow_policy='lossguide',
    importance_type=None, interaction_constraints=None,
    learning_rate=0.07299317690396159, max_bin=None,
    max_cat_threshold=None, max_cat_to_onehot=None,
    max_delta_step=None, max_depth=0, max_leaves=22,
    min_child_weight=18.541277231179432, missing=nan,
    monotone_constraints=None, n_estimators=57, n_jobs=-1,
    num_parallel_tree=None, predictor=None, random_state=None, ...)
"""


```

5.9. Как настраивать ограничение по времени

Если, например, нижняя граница на поиска (скажем 60 сек) и верхняя граница (пусть 2400 сек), то можно поробовать задать `time_budget=60` и никаких сообщений и рекомендаций по увеличению временного бюджета не будет, то на этом можно остановиться. Если рекомендуется увеличить бюджет по времени, то пробуем верхнюю границу с `early_stop=True`.

Для того чтобы прикинуть сколько времени потребуется на поиск наилучшей конфигурации, можно запустить обучение с двумя итерациями, а затем изучить логи

```
automl.fit(
    X_train, y_train, task="regression",
    custom_hp={"xgboost": { "n_estimators": { "domain": tune.lograndint(10, 25)} } },
    max_iter=2
)
# INFO - iteration 0, current learner lgbm
# INFO - Estimated sufficient time budget=10000s. Estimated necessary time budget=10s.
```

То есть предполагаемый необходимый временной бюджет 10 секунд (estimated sufficient time budget), а предполагаемый достаточный – 10 000 секунд (estimator necessary time budget). Это очень грубая оценка, просто ориентир.

Если временной бюджет слишком низкий, то может получиться, что ни одна модель за отведенное время не обучилась. В этом случае лучше использовать `max_iter`, а не `time_budget`.

5.10. Алгоритмы подбора гиперпараметров

FLAML использует два метода:

- CFO (Frugal Optimizatoin for Cost-related Hyperparameters),
- BlendSearch.

CFO использует метод случайного прямого поиска *FLOW*² с адаптивным размером шага и случайнм рестартом. Он требует начальной точки низкой стоимости (low-cost initial point).

*FLOW*² это простой эффективный метод стохастического прямого поиска. Однако, если пространство поиска сложное, а алгоритм «залипает» на локальных минимумах, то имеет смысл рассмотреть алгоритм BlendSearch.

BlendSearch сочетает в себе локальный и глобальный поиск. Как и CFO BlendSearch требует начальной точки низкой стоимости.

5.11. Zero Shot AutoML

`flaml.default` – пакет zero-shot AutoML или «no-tuning» AutoML. Он использует `flaml.AutoML` и `flaml.default.portfolio` для поиска хороших конфигураций и предлагает ориентированные на данные конфигурации во время выполнения без дорого подбора гиперпараметров.

Достоинства Zero-shot AutoML:

- Вычислительные затраты только на обучение одной модели. Подбор гиперпараметров не требуется.
- Конфигурация предлагается мгновенно. Нет никаких накладных расходов.

Например, если сейчас решение выглядит так

```
from lightgbm import LGBRegressor

estimator = LGBMRegressor()
estimator.fit(X_train, y_train)
estimator.predict(X_test)
```

то нужно только заменить строку импорта

```
from flaml.default import LGBMRegressor
```

Ожидается, что в большинстве случаев модель будет работать также или лучше.

На текущий момент поддерживаются следующие модели:

- LGBMClassifier, LGBMRegressor,
- XGBClassifier, XGBRegressor,
- RandomForestClassifier, RandomForestRegressor,
- ExtraTreesClassifier, ExtraTreesRegressor.

`flaml.default.LGBMRegressor` принимает решение о конфигурации гиперпараметров на базе обучающего набора данных. Если прогнозируется прирост эффективности по сравнению с конфигурацией по умолчанию, то будет использована зависящая от данных конфигурация.

Перед запуском можно посмотреть предлагаемую конфигурацию

```
from flaml.default import LGBMRegressor

reg = LGBMRegressor()
hyperparams, estimator_name, X_transformed, y_transformed = reg.suggest_hyperparams(X_train,
    y_train)
print(hyperparams)
```

5.12. Мультицель в задаче регрессии

```
from flaml import AutoML
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.multioutput import MultiOutputRegressor

# целевая переменная будет содержать 3 подцели
X, y = make_regressor(n_targets=3)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=42)
model = MultiOutputRegressor(AutoML(task="regression", time_budget=60))
model.fit(X_train, y_train)

print(model.predict(X_test))
```

5.13. FLAML + Sklearn pipeline

```
from sklearn import set_config
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from flaml import AutoML

set_config(display='diagram')

imputer = SimpleImputer()
standardizer = StandardScaler()
automl = AutoML()

automl_pipeline = Pipeline([
    ("imputuer", imputer),
    ("standardizer", standardizer),
    ("automl", automl)
])
automl_pipeline
```

Запустить конвейер можно так

```
automl_settings = {
    "time_budget": 60, # total running time in seconds
    "metric": "accuracy", # primary metrics can be chosen from: ['accuracy', 'roc_auc', 'roc_auc_weighted', 'roc_auc_ovr', 'roc_auc_ovo', 'f1', 'log_loss', 'mae', 'mse', 'r2'] Check
    # the documentation for more details (https://microsoft.github.io/FLAML/docs/Use-Cases/Task-Oriented-AutoML#optimization-metric)
    "task": "classification", # task type
    "estimator_list": ["xgboost", "catboost", "lgbm"],
    "log_file_name": "airlines_experiment.log", # flaml log file
}
pipeline_settings = {
    f"automl_{key}": value for key, value in automl_settings.items()
}
automl_pipeline.fit(X_train, y_train, **pipeline_settings)
```

Забрать лучшую модель можно так

```
automl = automl_pipeline.steps[2][1]
# Get the best config and best learner
print('Best ML leaner:', automl.best_estimator)
```

```
print('Best hyperparameter config:', automl.best_config)
print('Best accuracy on validation data: {:.4g}'.format(1 - automl.best_loss))
print('Training duration of best run: {:.4g} s'.format(automl.best_config_train_time))
```

6. Масштабирование признаков с выбросами

https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html#original-data

Для того чтобы сделать признаки «более гауссовскими» можно попробовать степенное преобразование по модели Бокса-Кокса или Яо-Джонсона. Еще можно использовать квантильное преобразование с гауссовским выходом `QuantileTransformer(output_distribution="normal")`.

7. Дисбаланс классов

Несбалансированность классов может оказать существенное влияние на качество модели независимо от выбранного алгоритма обучения. Суть проблемы – крайне неравномерное распределение меток в обучающих данных [4, стр. 85].

Так бывает, например, когда классификатор должен отличать настоящие электронные транзакции от мошеннических: примеров настоящих транзакций гораздо больше. Типичный алгоритм машинного обучения старается классифицировать большинство обучающих примеров правильно. Он вынужден так делать, потому что предназначен для минимизации функции стоимости, которая обычно назначает положительную потерю каждому неправильно классифицированному примеру. Если потеря при неправильной классификации примеров редко встречающихся классов такая же, как для часто встречающихся, то может случиться, что алгоритм обучения сочтет выгодным «сдаваться» на многих примерах редко встречающихся классов, чтобы делать меньше ошибок на часто встречающихся.

Можно создавать синтетические примеры – для этого выберем значения признаков из нескольких примеров редко встречающегося класса и, комбинируя их, построим новый пример этого класса. Есть два популярных алгоритма выборки с избыtkом (*oversampling*) путем синтезирования примеров редко встречающегося класса: Synthetic Minority Oversampling Technique (SMOTE) и Adaptive Synthetic Sampling Method (ADASYN).

Методы SMOTE и ADASYN во многих отношениях похожи. Для заданного примера \mathbf{x}_i редко встречающегося класса оба метода выбирают k ближайших соседей. Обозначим этот набор k примеров S_k . Синтетический пример \mathbf{x}_{new} определяется как $\mathbf{x}_i + \lambda(\mathbf{x}_{zi} - \mathbf{x}_i)$, где \mathbf{x}_{zi} – пример редко встречающегося класса, случайно выбранный из S_k . Гиперпараметр интерполяции λ может быть произвольным числом из отрезка $[0, 1]$.

В ADASYN количество синтетических примеров, генерируемых для каждого \mathbf{x}_i , пропорционально числу примеров в S_k , не принадлежащих редко встречающемуся классу. Поэтому больше синтетических примеров генерируется в той области, где примеры редко встречающегося класса действительно редки.

Противоположный подход – выборка с недостатком (*undersampling*) – заключается в том, чтобы исключить из обучающего набора некоторые примеры часто встречающегося класса.

Связь Томека между примерами \mathbf{x}_i и \mathbf{x}_j , принадлежащими двум разным классам, существует, если в наборе данных нет примера \mathbf{x}_k , расстояние от которого до \mathbf{x}_i или \mathbf{x}_j меньше, чем расстояние

между ними самими. Расстояние можно определить, например, как *коэффициент Отцаи* или *евклидово расстояние*.

Кластерная выборка с недостатком. Решите, сколько примеров должно быть в часто встречающемся классе в результате выборки с недостатком. Обозначим это число через k . Примените алгоритм кластеризации на основе цетроидов к примерам часто встречающегося класса, задав в качестве количества кластеров k . Затем замените все примеры мажоритарных классов k центроидами.

Ансамблевое обучение – еще один способ сгладить остроту проблемы несбалансированных классов. Нужно случайным образом разбить мажоритарный примеры на H подмножеств, а затем создать H обучающих наборов. После обучения H моделей предсказание делается путем усреднения (или голосования) их выводов [4, стр. 204]. Для $H = 4$ этот процесс показан на рис. 13. Здесь мы преобразовали несбалансированную задачу бинарного обучения в четыре сбалансированные, разбив примеры мажоритарного класса на четыре подмножества. Примеры миноритарного класса полностью скопированы в каждое подмножество.

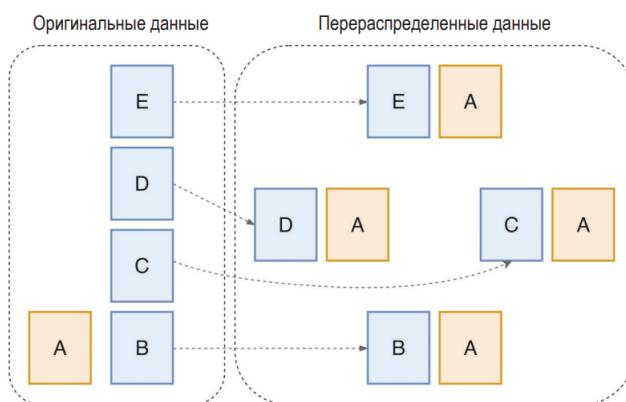


Рис. 13. Ансамбль перераспределенных наборов данных

Этот подход прост и хорошо масштабируется: мы можем обучить и выполнить модели на разных процессорных ядрах или узлах кластера. Кроме того, ансамбль моделей дает лучшие предсказания, чем каждая отдельная модель.

Тот же прием из книги Лакшманана [5, стр. 167]. Поникающий отбор также часто сочетается с паттерном «Ансамбли». Используя этот подход, вместо полного удаления случайной выборки из мажоритарного класса мы используем разные его подмножества для тренировки нескольких моделей, а затем совмещаем эти модели. В целях иллюстрации предположим, что у нас есть набор данных со 100 примерами миноритарного класса и 1000 примерами мажоритарного класса. Вместо удаления 900 примеров из мажоритарного класса мы бы разбили мажоритарные примеры случайно на 10 групп по 100 примеров в каждой, чтобы идеально сбалансировать набор данных. Затем мы бы натренировали 10 классификаторов, каждый с теми же 100 примерами из миноритарного класса и 100 разными, случайно отобранными значениями из мажоритарного класса.

Про подбор порога бинаризации для классификатора. В дополнение к сочетанию подходов, которые делают акцент на данных, мы также можем корректировать порог классификатора, чтобы выполнять оптимизацию под метрики точности или полноты в зависимости от варианта использования. Если бы мы были больше заинтересованы в том, чтобы модель была правильной всякий раз, когда она делает предсказание положительного класса (другими словами, если бы

мы хотели снизить число *ошибок первого рода*, т.е. FP), мы бы оптимизировали предсказательный порог под метрику точности, так как точность – это истинно-положительных экземпляров в группе экземпляров, которые классификатор посчитал положительными, и следовательно зависит от FP. Если же дороже обходится упущение потенциальной положительной классификации (ошибка второго рода, FN), даже если мы можем ошибаться, то мы оптимизируем нашу модель под метрику полноты.

Важное замечание про тестовый поднабор данных [5, стр. 156]. Независимо от того, как мы модифицируем набор данных для тренировки, мы должны оставлять тестовый набор как есть, чтобы он обеспечивал точное представление изначального набора данных. Другими словами, тестовый набор должен иметь примерно такой же баланс классов, как и изначальный набор данных. В приведенном примере это 5% мошенничества и 95% не мошенничества.

Поникающий отбор [5, стр. 157]. Пусть набор данных содержит 6,3 млн. примеров, из которых только 8 тыс. являются мошенническими транзакциями. Это число составляет всего лишь 0,1% всего набора данных. Хотя крупный набор данных нередко улучшает способность модели выявлять закономерности, он менее полезен, когда данные значительно несбалансированы. Мы можем решить эту проблему, удалив из набора данных большой кусок мажоритарного класса. Мы возьмем все 8 тыс. мошеннических примеров и отложим их в сторону, чтобы использовать во время тренировки модели. Затем возьмем небольшую случайную выборку не мошеннических транзакций. Далее мы объединим ее с 8 тыс. мошенническими примерами, перетасуем данные и применим этот новый, меньший набор данных для тренировки модели. После этого наш набор данных будет содержать мошеннических транзакций, являясь гораздо более сбалансированным, чем изначальный набор данных, в котором только 0.1% приходится на мажоритарный класс. При поникающем отборе неплохо поэкспериментировать с точным используемым балансом. Здесь мы использовали разбивку 25/75, но для достижения точности может потребоваться разбивка, более близкая к 50/50.

Поникающий отбор обычно сочетают с паттерном «Ансамбли», соблюдая следующие шаги:

1. Подвергнуть мажоритарный класс поникающему отбору и использовать все экземпляры мажоритарного класса,
2. Натренировать модель и добавить ее в ансамбль,
3. Повторить.

Во время предсказательного вывода следует брать *медианное* значение на выходе из *ансамблевых моделей*.

В статье Дьяконова прием подбора порога бинаризации классификатора описывается в следующем виде. Обычно модель получает некоторые оценки принадлежности к классам, а сама классификация – это результат бинаризации (по умолчанию порог 0.5). Но порог можно подбирать. Простая стратегия: при скользящем контроле по 10 фолдам получим оценки принадлежности классу 1 *на обучении* (функция `cross_val_predict(method="predict_proba")`), потом для заданного функционала качества подберем оптимальный порог бинаризации (при котором значение функционала максимально). *Этот же порог* будем потом использовать *на teste*.

Выводы по работе с несбалансированными наборами данных:

- Если дисбаланс в наборе небольшой (грубо «40 на 60») и признаки вещественные, то можно воспользоваться либо приемом обогащения мажоритарного класса синтетическими экземплярами, сгенерированными с помощью техник SMOTE или ADASYN (!), либо можно ничего не делать с набором данных, а использовать ансамблевые методы на деревьях принятия решений (в этом случае неважно какие типы признаков участвуют в описании объекта

- вещественные, категориальные или еще какие-либо). «Деревянные» модели часто и без устранения дисбаланса неплохо работают. Для оценки производительности можно использовать гармоническое среднее, каппу Коэна, коэффициент Метьюса. Как правило, дисбаланс набора данных для «деревянных» моделей практически не влияет на значение этих метрик, а сами метрики слабо чувствительны как к взешиванию классов, так и к подбору порога бинаризации классификатора,
- Если дисбаланс значительный (скажем, «20 на 80»), то можно один *сильно разбалансированный набор данных* представить в виде нескольких *сбалансированных наборов данных*. Из мажоритарного класса случайным образом выбираются экземпляры без возвращения в количестве равном мощности миноритарного класса, а экземпляры миноритарного класса просто копируются для каждого такого поднабора мажоритарного класса. В итоге получается N идеально сбалансированных наборов, на каждом из которых можно обучить модель, а потом просто выдать агрегат прогнозов (мажоритарным голосованием, мягким голосованием, усреднением и т.д.); прогноз на тестовом поднаборе данных будет генерироваться как и в случае баггинга: каждое дерево ансамбля, выращенное на своем идеально сбалансированном наборе данных дает прогноз, затем эти прогнозы агрегируются,
- В случае очень сильного дисбаланса (грубо «1 на 99») и если размер миноритарного класса позволяет (т.е. миноритарный класс имеет достаточное количество экземпляров для обучения и тестирования) можно просто удалить большую группу экземпляров в мажоритарном классе, а из случайных подвыборок мажоритарного класса сформировать наборы с менее серьезным дисбалансом. На каждом таком меньшем наборе данных обучаем модели, собираем их в ансамбль и агрегируем прогнозы.

8. Метрики MAPE, SMAPE, WAPE, WMAPE etc.

8.1. MAPE

Метрика MAPE (Mean Absolute Percentage Error) вычисляется как

$$MAPE = 100\% \frac{1}{n} \sum_{t=1}^n |1 - \frac{F_t}{A_t}|,$$

где A_t – истинное значение, F_t – прогноз.

Очевидно, что если $A_t = 0, F_t \neq 0$, значение в t -ой точке будет бесконечным. При низких значениях A_t, F_t ошибка будет завышенной. Поэтому если A_t, F_t принимают нулевые или близкие к нулю значения, рекомендуется использовать WAPE / WMAPE.

8.2. SMAPE

Метрику SMAPE рекомендуется вычислять по формуле

$$SMAPE = 100\% \frac{\sum_{t=1}^n |A_t - F_t|}{\sum_{t=1}^n A_t + F_t}$$

Выходит, что симметричный вариант MAPE не симметричный, а вот обычная MAPE симметричная.

8.3. WAPE / WMAPE

Метрика WAPE (Weighted Average Percentage Error) вычисляется по формуле

$$WAPE = 100\% \frac{\sum_{t=1}^n |A_t - F_t|}{\sum_{t=1}^n |A_t|}$$

а метрика WMAPE (Weighted Mean Absolute Percentage Error) – как

$$WMAPE = 100\% \frac{\sum_{t=1}^n w_t |A_t - F_t|}{\sum_{t=1}^n w_t |A_t|}$$

```
def weighted_mean_average_percentage_error(
    y_true: t.Iterable,
    y_pred: t.Iterable,
    *,
    weights: t.Optional[t.Iterable] = None
) -> float:
    """Calculates WAPE(weights is None) or WMAPE(weights is not None)"""
    if len(y_true) != len(y_pred):
        raise ValueError("Error! y_true.len must be equal to y_pred.len")
    else:
        if weights is not None and len(y_true) != len(weights):
            raise ValueError("Error! y_true.len or y_pred.len must be equal to weights.len")

    if weights is None:
        weights = np.ones_like(y_true)

    return 100 * (weights * np.abs(y_true - y_pred)).sum() / (weights * np.abs(y_true)).sum()
```

9. Внедрение моделей машинного обучения в промышленную эксплуатацию

<https://www.bigdataschool.ru/blog/mlops-deployment-patterns-and-strategies.html>

MLOps-энтузиасты выделяют следующие паттерны внедрения моделей машинного обучения в прод

- Модель как услуга или сервис (Model-as-Service),
- Модель как зависимость (Model-as-Dependency),
- Предварительный расчет (Precompute),
- Модель по запросу (Model-on-Demand),
- Гибридная модель обслуживания (Hybrid Model Serving) или Федеративное обучение (Federated Learning)

9.1. Паттерны внедрения ML-моделей в промышленную эксплуатацию

<https://ml-ops.org/content/three-levels-of-ml-software>

9.1.1. Модель как услуга (Model-as-Service)

Это весьма распространенный шаблон для предоставления модели машинного обучения в качестве независимой услуги. Обычно это реализуется через заключение ML-модели и интерпретатора в выделенную веб-службу, к которой приложения-потребители данных запрашивают через REST API или удаленный вызов процедур (RPC, Remote Procedure Call). Такой паттерн пригоден для различных рабочих процессов машинного обучения, от пакетного прогнозирования до онлайн-обучения модели в потоковом режиме (рис. 14).

REST API <https://skillbox.ru/media/code/rest-api-chto-eto-takoe-i-kak-rabotaet/> – архитектурный подход, который устанавливает ограничения для API: как они должны быть устроены и какие функции поддерживать. Это позволяет стандартизировать работу программных интерфейсов, сделать их более удобными и производительными.

В отличие от SOAP API, REST API – не протокол, а простой список рекомендаций, которым можно следовать или не следовать. Поэтому у него нет собственных методов. С другой стороны, его автор Рой Финлдинг создал еще и протокол HTTP, так что они очень хорошо сочетаются, и REST обычно используют в связке с HTTP. Хотя REST – это не только HTTP, а HTTP – не только REST.

Всего в REST есть шесть требований к проектированию API. Пять из них обязательные, одно – optionalное:

- Клиент-серверная модель (client-server model),
- Отсутствие состояния (statelessness),
- Кэширование (cacheability),
- Удинообразие интерфейса (uniform interface),
- Многоуровневая система (layered system),
- Код по требованию (code on demand) – необязательно.

В вебе ресурсами называют любые данные: текст, изображение, видео, аудио, программу и пр.

Так как REST – архитектурный подход, а не протокол, в нем не заложено никаких конкретных методов. На чаще всего его применяют со стандартом HTTP, в котором заложены собственные методы.

Если кратко, то в HTTP прописан набор действий, который можно описать аббревиатурой CRUD: create, read, update, delete. Для каждого такого действия существует один или несколько методов. Например, GET для чтения, а PUT или PATCH – для разных видов обновления.

Итак:

- REST – это *архитектурный стиль API*. Он не ограничивается никакими протоколами и не имеет собственных методов. Но обычно в RESTful-сервисах используют стандарт HTTP, а файлы передают в формате JSON или XML,
- Есть шесть принципов, на которых строится REST: клиент-серверная модель, отсутствие состояния, кэширование, удинообразие интерфейса, многоуровневая система, код по требованию. Последний из них необязателен,

- REST-подход к архитектуре позволяет сделать сервисы отказоустойчивыми, гибкими и производительными, а при их масштабировании и внесении изменений не возникает больших сложностей.



Рис. 14. Машинное обучение как услуга – самый простой шаблон MLPOps

9.1.2. Модель как зависимость (Model-as-Dependency)

Это наиболее простой способ упаковать модель машинного обучения, которая рассматривается как зависимость внутри программного приложения. Например, приложение использует ML-модель как обычную зависимость от jar-файла, вызывая метод прогнозирования и передавая значения. Возвращаемые значения такого -метода – некоторый прогноз, который выполняется предварительно обученной ML-моделью. Как правило, этот подход используется в задачах простого *пакетного прогнозирования* (рис. 15).

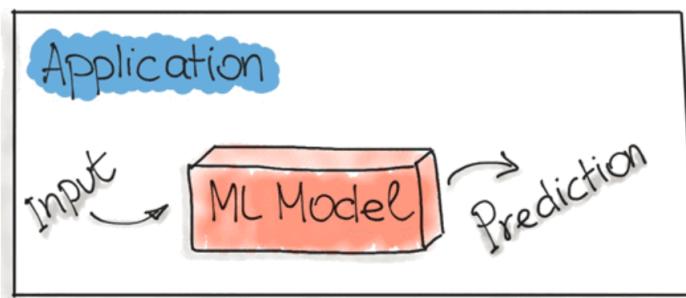


Рис. 15. Модель как зависимость (Model-as-Dependency)

9.1.3. Предварительный расчет (Precompute)

В этом случае прогнозы предварительно вычисляются с использованием уже обученной ML-модели для входящего пакета данных и сохраняются в базе. Далее к этой базе идет обращение при любом входном запросе, чтобы получить результат прогнозирования. С архитектурной точки зрения это похоже на Лямбда-шаблон, когда «горячая» обработка данных в потоковом режиме совмещается с «холодной», где пакеты исторических данных подгружаются из хранилища, в качестве которого обычно выступает Data Lake на Apache Hadoop рис. 16.

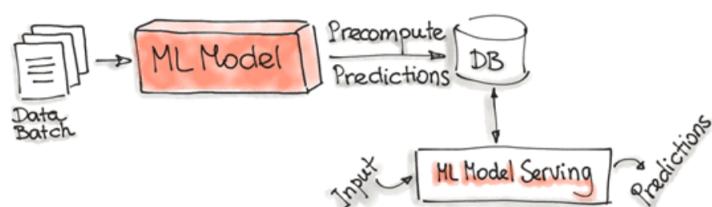


Рис. 16. Лямбда-архитектура для ML-систем в MLPOps (Precompute)

9.1.4. Модель по запросу (Model-on-Demand) с Apache Kafka

Этот вариант рассматривает модель машинного обучения как зависимость, доступную во время выполнения. Однако, в отличие от паттерна «Модель как зависимость», Model-on-Demand имеет собственный цикл выпуска и публикуется независимо. Для этой реализации обычно используется брокер сообщений, например, Apache Kafka или RabbitMQ. В этом случае применяется популярный в BigData подход потоковой обработки событий (event-stream processing), когда данные представляются в виде потока событий, объединенные в канал.

Каналы событий представляют собой *очереди сообщений*:

- входную,
- выходную

Брокер сообщений позволяет одному процессу записывать запросы на прогнозирование во входную очередь. Обработчик событий (event processor) содержит модель, обслуживающую среду выполнения, и ML-модель. Этот процесс подключается к брокеру, считывает из очереди запросы в пакетном режиме и отправляет их ML-модели для прогнозирования. Процесс обслуживания модели запускает генерацию прогнозов для входных данных и записывает полученные прогнозы в выходную очередь. Оттуда результаты прогнозирования отправляются в сервис, который инициировал запрос рис. 17.

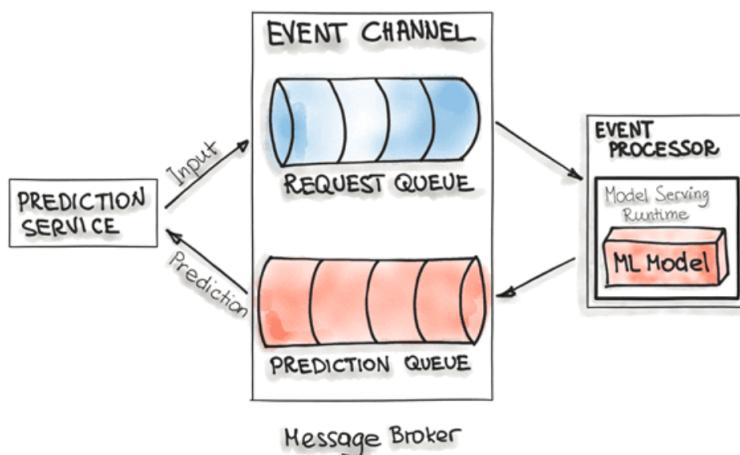


Рис. 17. Модель по запросу (Model-on-Demand)

9.1.5. Гибридная модель обслуживания (Hybrid Model Serving)

Уникальность федеративного обучения или гибридного обслуживания в том, что оно работает со множеством ML-моделей, индивидуальных для каждого пользователя в дополнение к той, которая хранится на сервере. Серверная модель обучается только один раз с реальными данными и выступает в качестве начального образца для каждого пользователя. Далее она может видоизменяться в пользовательских вариациях, даже на мобильных устройствах, параметры которых сегодня позволяют обучать собственные ML-алгоритмы. Периодически пользовательские устройства отправляют на сервер уже обученные данные своей ML-модели, корректируя серверный вариант. Оттуда эти изменения могут быть распространены на остальных пользователей, с учетом предварительной проверки и тестирования на функциональность.

Главным плюсом такого гибридного подхода является то, что обучающие и тестовые данные, которые носят исключительно личный характер, никогда не покидают пользовательские устрой-

ства, сохраняя при этом все доступные данные. Это позволяет обучать высокоточные ML-модели без необходимости хранить гигабайты данных в облаке. Однако, стоит помнить, что обычные алгоритмы машинного обучения ориентированы на однородные большие данные, которые обрабатываются на мощном оборудовании и всегда доступны для обучения. Современные мобильные устройства пока еще менее мощные, чем специализированные Big Data кластера, а также обучающие данные распределены по миллионам устройств, которые не всегда доступны. С учетом этого был создан отдельный фреймворк – TensorFlow Federated, облегченная форма TensorFlow для федеративного обучения (рис. 18).

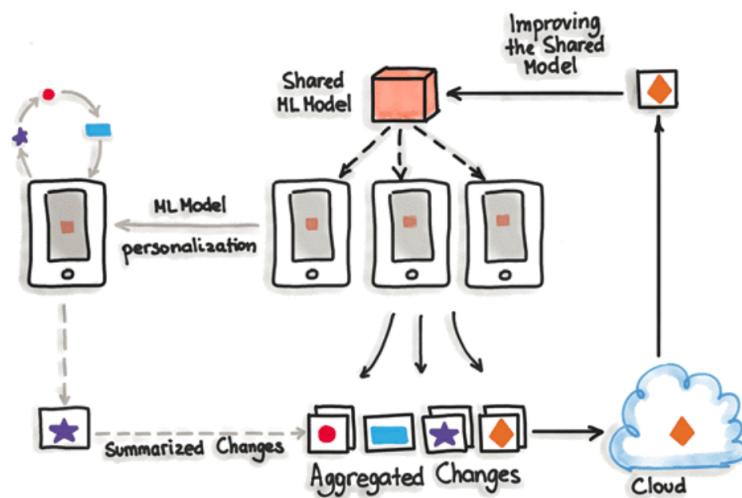


Рис. 18. Гибридная модель обслуживания или Федеративное обучение

9.2. Стратегии внедрения ML-модели в прод

9.2.1. Развёртывание с помощью Docker-контейнеров

Этот вариант подходит для *легковесных* ML-моделей *без сохранения состояния*. В этом случае код модели машинного обучения заключен в Docker-контейнер, который считается наиболее распространенной технологией контейнеризации для локального, облачного или гибридного развертывания. Оркестрация таких контейнеров обычно выполняется с помощью Kubernetes или альтернатив, таких как AWS Fargate. Функциональные возможности модели Machine Learning доступны через REST API, например, в виде приложения на Flask (рис. 19).

9.2.2. Бессерверные вычисления (serverless)

Код приложения и зависимости упаковываются в файлы .zip с одной функцией точки входа. Затем этой функцией могут управлять основные облачные провайдеры, такие как Azure Functions, AWS Lambda или Google Cloud Functions. Однако следует обратить внимание на возможные ограничения развертываемых артефактов, в частности, их размеры. Напомним, serverless-подход реализует PasS- или FaaS- (функция как услуга, Function-as-Service) стратегию, когда облако автоматически и динамически управляет выделением вычислительных ресурсов в зависимости от пользовательской нагрузки.

При этом для выполнения *каждого запроса* (вызыва функции) *создается отдельный контейнер* или виртуальная машина, уничтожающиеся после выполнения. Преимуществом этого является избавление пользователей от работы по выделению и настройки серверов, в т.ч. виртуальных

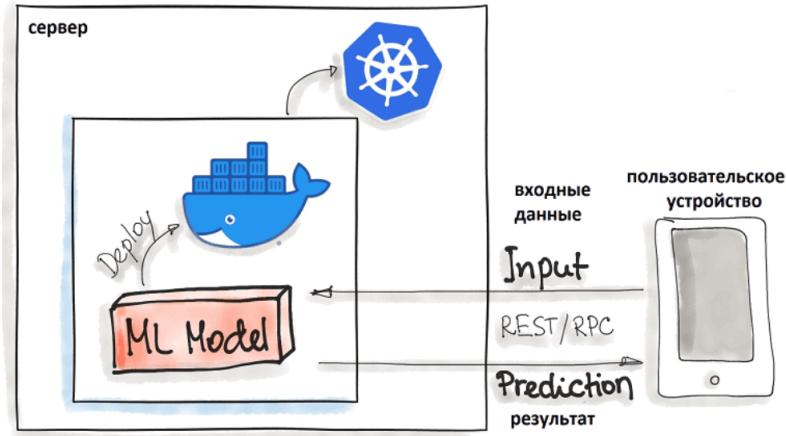


Рис. 19. Развёртывание ML-модели с помощью Docker и Kubernetes

машин, контейнеров, баз данных, приложений, экземпляров сред выполнения. Все конфигурации и планирование вычислительных ресурсов для запуска кода по требованию или по событию скрыты от пользователей и управляются облаком. Бессерверный код может быть частью приложений, построенных на традиционной архитектуре, например, на микросервисах. Обратной стороной этих достоинств являются зависимость от облачного провайдера, сложность в поиске причин случившихся ошибок из-за многослойной инкапсуляции внутреннего устройства всей системы, а также время на запуск облачной функции, которое может быть критично для бизнеса рис. 20



Рис. 20. Бессерверная стратегия внедрения ML-модели в прод

10. ML System Design Doc

Очень полезный шаблон для дорожной карты ML-продукта https://github.com/IrinaGoloschapova/ml_system_design_doc_ru

11. Квантильная регрессия

https://scikit-learn.org/stable/modules/linear_model.html#quantile-regression

Квантильная регрессия (модель линейной регрессии, которая предсказывает условные квантили) может быть полезна при построении интервальных, а не точечных оценок. Иногда интервальные прогнозы строятся в предположении о нормальном распределении ошибки с нулевым средним и постоянной дисперсией. Квантильная регрессия дает адекватные интервалы даже тогда, когда у ошибки непостоянная дисперсия или когда ошибка негауссовская.

Как и линейная модель, квантильная регрессия дает линейные прогнозы $\hat{x}(w, X) = X w$ для q -ого квантиля, $q \in (0, 1)$. Веса модели w находятся для следующей оптимизационной проблемы

$$\min_w \frac{1}{n_{samples}} \sum_i PB_q(y_i - X_i w) + \alpha \|w\|_1.$$

То есть состоит из линейной потери (pinball loss https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_pinball_loss.html#sklearn.metrics.mean_pinball_loss)

$$PB_q(t) = q \max(t, 0) + (1 - q) \max(-t, 0) = \begin{cases} qt, & t > 0, \\ 0, & t = 0, \\ (q - 1)t, & t < 0 \end{cases}$$

и L_1 -штрафа, как в модели Lasso.

Так как pinball loss линейна по остаткам, квантильная регрессия более устойчива к выбросам, чем среднее на базе квадратической ошибки.

12. MAE vs MSE. Устойчивость среднеабсолютной ошибки к выбросам

Средняя абсолютная ошибка более устойчива к выбросам по сравнению с квадратической ошибкой. Потому что, оптимальным решением на классе константных алгоритмов для квадратической ошибки MSE является *среднее арифметическое*, а для среднеабсолютной MAE – *медиана*.

Медиана, как известно, является более устойчивой к выбросам по сравнению со средним арифметическим. Несмотря на это, MSE обладает своими достоинствами – в частности, для нее можно выписать аналитическое решение в случае линейной регрессии, а также ее можно оптимизировать напрямую при помощи градиентного спуска, в отличие от MAE, которая не является дифференцируемой по w .

13. Оптимизационные задачи и теорема Куна-Таккера

Рассмотрим задачу минимизации

$$\begin{cases} f_0(x) \rightarrow \min_{x \in \mathbb{R}^d} \\ f_i(x) \leq 0, \quad i = 1, \dots, m, \\ h_i(x) = 0, \quad i = 1, \dots, p. \end{cases} \quad (3)$$

Если ограничения в этой задаче отсутствуют, то имеет место *необходимое условие экстремума*: если в точке x функция f_0 достигает своего минимума, то ее градиент в этой точке равен нулю. Значит, для решения задачи *безусловной оптимизации* (нет ограничений)

$$f_0(x) \rightarrow \min$$

достаточно найти все решения уравнения

$$\nabla f_0(x) = 0,$$

и выбрать то, в котором достигается наименьшее значение. Для решения *условных* задач оптимизации (в них есть ограничения) требуется более сложный подход.

Для задачи (3) можно записать *лагранжиан* (функцию Лагранжа)

$$L(x, \lambda, \nu) = f_0(x) + \sum_{i=1}^m \lambda_i f_i(x) + \sum_{i=1}^p \nu_i h_i(x), \quad \lambda_i \geq 0,$$

где λ_i, ν_i – множители Лагранжа (двойственные переменные).

Двойственной функцией для задачи (3) называется функция, получающаяся при взятии минимума лагранжиана по x

$$g(\lambda, \nu) = \inf_x L(x, \lambda, \nu).$$

Двойственная функция дает *нижнюю оценку* на минимум в исходной оптимационной задаче.

Условия Каруша-Куна-Таккера (необходимые условия экстремума)

$$\begin{aligned} \nabla f_0(x_*) + \sum_{i=1}^m \lambda_i^* \nabla f_i(x_*) + \sum_{i=1}^p \nu_i^* \nabla h_i(x_*) &= 0 \\ f_i(x_*) &\leq 0, \quad i = 1, \dots, m \\ h_i(x_*) &= 0, \quad i = 1, \dots, p \\ \lambda_i^* &\geq 0, \quad i = 1, \dots, m \\ \lambda_i^* f_i(x_*) &= 0, \quad i = 1, \dots, m \end{aligned}$$

Если задача (3) является выпуклой и удовлетворяет условию Слейтера, то условия Куна-Таккера становятся *необходимыми и достаточными*.

14. Векторное дифференцирование

Иногда при взятии производных по вектору или от вектор-функции удобно оперировать матричными операциями. Это упрощает запись и упрощает вывод формул.

Введем следующие определения:

- При отображении вектора в число $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$ (то есть когда нужно взять производную от скалярной функции векторного аргумента $f(x_1, \dots, x_n)$ по каждому аргументу x_i в отдельности)

$$\nabla_x f(x) = \left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right]^T,$$

- При отображении матрицы в число $f(A) : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}$ (когда нужно взять производную функции от матрицы A по всем элементам матрицы A_{ij})

$$\nabla_A f(A) = \left(\frac{\partial f}{\partial A_{ij}} \right)_{i,j=1}^{n,m}.$$

Мы хотим оценить, как функция изменяется по каждому из аргументов по отдельности. Поэтому производной от скалярной функции векторного аргумента по вектору будет *вектор*, по матрице – *матрица*.

Замечание

Когда говорят, что нужно взять производную от скалярной функции векторного аргумента $f(\underbrace{x_1, \dots, x_n}_x)$ по вектору x , это означает, что нужно взять производную от скалярной функции векторного аргумента $f(x)$ по каждому элементу вектора x_i . То есть запись $\nabla_x f(x_1, \dots, x_n)$ означает, частные производные $\frac{\partial}{\partial x_i}$ от $f(x)$. И аналогично, когда говорят, что нужно взять производную от функции матричного аргумента по матрице, это означает, что нужно взять производную функции матричного аргумента по каждому элементу матрицы. То есть запись $\nabla_A f(A)$ означает частные производные по каждому элементу матрицы $\frac{\partial}{\partial A_{ij}}$ от $f(A)$

Задача 1. Пусть $a \in \mathbb{R}^n$ – вектор параметров, а $x \in \mathbb{R}^n$ – вектор переменных. Необходимо найти производную их скалярного произведения по *вектору переменных* $\nabla_x a^T x$.

Так как $a^T x$ это просто линейная комбинация переменных

$$\frac{\partial}{\partial x_i} a^T x = \frac{\partial}{\partial x_i} \sum_j a_j x_j = a_i = \frac{\partial f}{\partial x_i},$$

то, собрав все n компонент $\frac{\partial f}{\partial x_i}$ вместе, получим $\nabla_x \underbrace{a^T x}_{f(x_1, \dots, x_n)} = a$. То есть результатом будет вектор параметров a .

Заметим, что $a^T x$ – это число, поэтому $a^T x = (a^T x)^T = x^T a$, следовательно $\boxed{\nabla_x x^T a = \nabla_x a^T x = a}$.

Задача 2. Пусть теперь $A \in \mathbb{R}^{n \times n}$. Необходимо найти $\nabla_x x^T A x$.

$$\begin{aligned} \frac{\partial}{\partial x_i} x^T A x &= \frac{\partial}{\partial x_i} \sum_j x_j (Ax)_j = \frac{\partial}{\partial x_i} \sum_j x_j \left(\sum_k a_{jk} x_k \right) = \frac{\partial}{\partial x_i} \sum_{j,k} a_{jk} x_j x_k = \\ &= \sum_{j \neq i} a_{ji} x_j + \sum_{k \neq i} a_{ik} x_k + 2a_{ii} x_i = \sum_j a_{ji} x_j + \sum_k a_{ik} x_k = \sum_j (a_{ji} + a_{ij}) x_j. \end{aligned}$$

Поэтому $\boxed{\nabla_x x^T A x = (A + A^T)x}$.

Задача 3. Пусть $A \in \mathbb{R}^{n \times n}$. Необходимо найти $\nabla_A \det A$. Воспользуемся теоремой Лапласа о разложении определителя по строке

$$\frac{\partial}{\partial A_{ij}} \det A = \frac{\partial}{\partial A_{ij}} \left[\sum_k (-1)^{i+k} A_{ik} M_{ik} \right] = (-1)^{i+j} M_{ij},$$

где M_{ik} – дополнительный минор³ матрицы A .

Элементы обратной матрицы

$$(A^{-1})_{ij} = \frac{1}{\det A} (-1)^{i+j} M_{ji}.$$

³Дополнительным минором M_{ij} элемента a_{ij} называется определитель порядка $n - 1$, полученный из матрицы A порядка n вычеркиванием i -ой строки и j -ого столбца

Подставляя выражение для дополнительного минора, получаем ответ

$$\boxed{\nabla_A \det A = (\det A) A^{-T}}$$

Действительно, так как $\det A(A^{-T})_{ij} = (-1)^{i+j} M_{ij}$, то $\frac{\partial}{\partial A_{ij}} \det A = \det A(A^{-T})_{ij}$.

Задача 4. Пусть $A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times n}$. Необходимо найти $\nabla_A \text{tr}(AB)$.

$$\frac{\partial}{\partial A_{ij}} \text{tr}(AB) = \frac{\partial}{\partial A_{ij}} \sum_k (AB)_{kk} = \frac{\partial}{\partial A_{ij}} = \sum_{k,l} A_{kl} B_{lk} = B_{ji}.$$

То есть $\boxed{\nabla_A \text{tr}(AB) = B^T}$.

Задача 5. Пусть $x \in \mathbb{R}^n, A \in \mathbb{R}^{n \times m}, y \in \mathbb{R}^m$. Необходимо найти $\nabla_A x^T A y$. Воспользоваться циклическим свойством следа матрицы (для матриц подходящего размера)

$$\text{tr}(ABC) = \text{tr}(BCA) = \text{tr}(CAB)$$

и результатом предыдущей задачи, получаем

$$\nabla_A \underbrace{x^T A y}_{(1 \times 1)} = \nabla_A \text{tr}(x^T A y) = \nabla_A \text{tr}\left(\underbrace{A(yx^T)}_B\right) = B^T = xy^T.$$

14.1. Решение задачи регрессии для многомерного случая

В общем случае мы имеем выборку $\{(x_i, y_i)_{i=1}^l\}$, $x_i \in \mathbb{R}^d$, $y \in \mathbb{R}$, $i = (1, \dots, l)$ и мы хотим найти наилучшие параметры модели $a(x) = \langle w, x \rangle$ с точки зрения минимизации функции ошибки (то есть с точки зрения квадратичной функции потерь)

$$Q(w) = (y - Xw)^T (y - Xw).$$

Здесь $X \in \mathbb{R}^{l \times d}$ – матрица «объекты-признаки» для обучающей выборки, $y \in \mathbb{R}^l$ – вектор значений целевой переменной на обучающей выборке, $w \in \mathbb{R}^d$ – вектор параметров.

Выпишем градиент функции ошибки по w (это просто скалярная функция векторного аргумента)

$$\nabla_w Q(w) = \nabla_w [y^T y - y^T X w - w^T X^T y + w^T X^T X w] = 0 - X^T y - X^T y + (X^T X + X^T X)w = 0.$$

Рассмотрим подробнее второй элемент в квадратных скобках $\nabla_w \left(\underbrace{y^T}_{(1 \times l)} \underbrace{X}_{(l \times d)} \underbrace{w}_{(d \times 1)} \right) = \nabla_w \left(\underbrace{y^T X}_{(1 \times d)} \underbrace{w}_{(d \times 1)} \right)$.

То есть в обозначениях $\nabla_x a^T x = a$, $y^T X = a^T$, а $w = x$. Следовательно, здесь $a = (y^T X)^T = X^T y$.

Аналогично для третьего элемента $\nabla_w (w^T X^T y)$ в терминах $\nabla_x x^T a = a$, элемент $w^T = x^T$, а элемент $X^T y = a$. То есть решением будет просто $a = X^T y$.

Для четвертого элемента следует использовать формулу $\nabla_x x^T A x = (A + A^T)x$.

Таким образом, искомый вектор параметров выражается так

$$w = (X^T X)^{-1} X^T y.$$

Покажем, что найденная точка – точка минимума, если матрица $X^T X$ обратима. Из курса матана мы знаем, что если матрица Гессе функции положительно определена в точке, градиент которой равен нулю, то эта точка является локальным минимумом (достаточное условие существования экстремума)

$$\nabla_w^2 Q(w) = \nabla_w (2X^T X w) = 2X^T X \nabla_w w = \underbrace{2X^T X}_{(d \times d)} \underbrace{E}_{(d \times d)} = 2X^T X.$$

Так как числовую матрицу соответствующих размеров можно выносить за знак производной.

Необходимо понять является ли матрица $X^T X$ положительно определенной. Запишем определение положительной определенности матрицы $X^T X$

$$z^T X^T X z > 0, \forall z \in \mathbb{R}^d, z \neq 0.$$

Видим, что тут записан квадрат нормы вектора Xz , то есть $\|Xz\|^2 \geq 0$. В случае, если матрица X имеет книжную ориентацию (строк не меньше, чем столбцов) и имеет *полный ранг*⁴ (нет линейно зависимых столбцов), то вектор Xz не может быть нулевым, а значит выполняется

$$z^T X^T X z = \|Xz\|^2 > 0, \forall z \in \mathbb{R}^d, z \neq 0.$$

Ранг матрицы равен наибольшему порядку отличного от нуля минора этой матрицы.

То есть $X^T X$ является положительно определенной матрицей. Если же строк оказывается меньше, чем столбцов, или X не является полноранговой (есть линейно зависимые столбцы), то $X^T X$ *необратима* ($\det X^T X = 0$) и решение w определено *неоднозначно*.

14.2. Градиентный спуск

Ситуация, когда нам удается найти решение оптимизационной задачи в явном виде, – большая удача. В общем случае оптимизационные задачи можно решать *итерационно* с помощью *градиентных методов* (или же методов, использующих как градиент, так и информацию о производных более высокого порядка).

Антаградиент $(-\nabla f(x_1, \dots, x_n))$ является направлением наискорейшего убывания функции в заданной точке.

15. Классические алгоритмы машинного обучения

15.1. 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}

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

⁴Говорят, что у матрицы $A \in \mathbb{R}^{n \times m}$ полный ранг, если $\text{rg}A = \min(n, m)$

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 [3, стр. 44]

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

Замечание

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

15.2. 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 [3, стр. 48].

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

ции, 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 навсегда 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

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

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

15.3. Деревья решений

Дерево решений – это ациклический граф, который можно использовать для принятия решений. В каждом ветвящемся узле графа исследуется j -ый признак из вектора 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 расщепления 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 расщепления,
- Расщепление уменьшает энтропию ниже некоторого значения ϵ ,
- Дерево достигает некоторой максимальной глубины d .

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

Наиболее широкомасштабная версия алгоритма обучения дерева решений называется *C4.5*. Версия алгоритма C4.5 имеет несколько дополнительных особенностей по сравнению с ID3 [3, стр. 54]:

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

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.

15.4. Метод опорных векторов

15.4.1. Из документации scikit-learn

<https://scikit-learn.org/stable/modules/svm.html#svm-classification>

Классификация с SVC. Общий случай Даны векторы $x_i \in \mathbb{R}^p, i = 1, \dots, n$ и целевой вектор $y \in \{+1, -1\}^n$.

Наша цель заключается в том, чтобы найти такой вектор $w \in \mathbb{R}^p$ и скаляр $b \in \mathbb{R}$, что прогноз, вычисленный по формуле $\text{sign}(w^T \varphi(x) + b)$, будет корректным для большинства экземпляров.

Прямая задача (primal problem)

$$\begin{aligned} \min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^n \zeta_i \\ y_i (w^T \varphi(x_i) + b) \geq 1 - \zeta_i \\ \zeta_i \geq 0, i = 1, \dots, n \end{aligned}$$

Интуитивно, мы пытаемся *максимизировать зазор* (минимизируя квадрат евклидовой нормы $\|w\|^2 = w^T w$). Параметр C управляет силой штрафа и действует как обратный параметр регуляризации.

Замечание

Гиперпараметр C связан с *силой регуляризации* обратно пропорциональной зависимостью. То есть низким значениям параметра C отвечает сильная регуляризация (модель упрощается), а высоким значениям – слабая регуляризация (модель усложняется)

Меньшее значение параметра C (что отвечает более сильной регуляризации \rightarrow модель упрощается) приводит к более широкой полосе, но большему числу нарушений зазора [2, стр. 201]

Классификация с LinearSVC. Случай линейного ядра Прямая задача

$$\min_{w,b} \frac{1}{2} w^T w + C \sum_{i=1}^n \max[0, 1 - y_i (w^T \varphi(x_i) + b)]$$

Здесь используется *кусочно-линейная функция потерь* (hing loss).

Регрессия с SVR. Общий случай Даны векторы $x_i \in \mathbb{R}^p, i = 1, \dots, n$ и целевой вектор $y \in \mathbb{R}^n$. Прямая задача

$$\begin{aligned} \min_{w,b,\zeta,\zeta^*} \frac{1}{2} w^T w + C \sum_{i=1}^n (\zeta_i + \zeta_i^*) \\ y_i - (w^T \varphi(x_i) + b) \leq \varepsilon + \zeta_i, \\ (w^T \varphi(x_i) + b) - y_i \leq \varepsilon + \zeta_i^*, \\ \zeta_i, \zeta_i^* \geq 0, i = 1, \dots, n \end{aligned}$$

Регрессия с LinearSVR. Случай линейного ядра Прямая задача

$$\min_{w,b} \frac{1}{2} w^T w + C \sum_{i=1} \max[0, |y_i - (w^T \varphi(x_i) + b)| - \varepsilon].$$

Здесь используется функция потерь, не чувствительная к ошибкам в ε -окрестности (epsilon-insensitive loss).

15.4.2. Из книги Жерона

Методы SVM чувствительны к масштабам признаков: если масштаб, скажем по оси y , будет значительно больше масштаба по оси x – например, $y = 0 \dots 1000$ против $x = 1 \dots 5$, то больший зазор получиться по оси y . Дело в том, что методы SVM пытаются обеспечить самую широкую, какую только возможно, полосу между классами, так что если обучающий набор не масштабирован, то методы SVM, будут иметь тенденцию игнорировать небольшие признаки [2, стр. 602] (рис. 21).

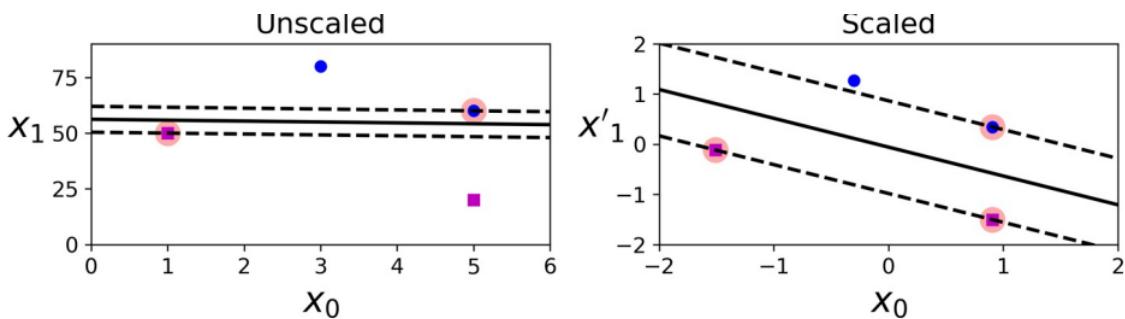


Рис. 21. Чувствительность к масштабам признаков. SVM посчитал горизонтальное положение полосы наиболее удачным, так как по оси x_1 масштаб больше и соответственно получается большее значение ширины зазора

Классификация с жестким зазором (hard margin classification) присущи две главные проблемы. Во-первых, она работает, только если данные линейно-сепарабельны. Во-вторых, она довольно чувствительна к выбросам.

Чтобы избежать этих проблем, предпочтительнее применять более гибкую модель. Цель заключается в том, чтобы отыскать хороший баланс между удержанием полосы как можно более широкой и ограничением количества нарушений зазора. Это называется *классификацией с мягким зазором* (soft margin classification)

В классах SVM библиотеки Scikit-Learn управлять упомянутым балансом можно с помощью параметра C : меньшее значение C ведет к более широкой полосе, но большему числу нарушений зазора.

`LinearSVC(C=1, loss="hinge")` похож на `SVC(kernel="linear", C=1)` с точки зрения функциональных возможностей, но основан на библиотеке liblinear⁵, а не libsvm, работает гораздо быстрее и лучше масштабируется на большие выборки [2, стр. 202]. Для улучшения производительности следует установить гиперпараметр `dual` в `False`. Флаг `dual=True` означает, что будет решаться *двойственная задача* (dual problem), а не *прямая задача* (primal problem).

⁵Библиотека liblinear реализует специальный алгоритм для линейных методов SVM – метод двойного покоординатного спуска (dual coordinate descent) <https://www.csie.ntu.edu.tw/~cjlin/papers/cddual.pdf>

Рекомендуется (см. [LinearSVC](#)) оптимизационную задачу решать в *прямой* постановке (т.е. `dual=False`), когда в матрице признакового описания объекта экземпляров больше, чем признаков – `n_samples > n_features` (книжная ориентация матрицы).

То есть оптимизационная задача для классификатора `LinearSVC` может быть сформулирована как в *прямой*, так и в *двойственной* постановке.

Замечание

Ядерный трюк предотвращает комбинаторно бурный рост количества признаков, поскольку в действительности мы не добавляем никаких признаков

Класс `LinearSVC` основан на библиотеке `liblinear`, не поддерживает ядерный трюк, но масштабируется почти линейно с ростом числа экземпляров m и числа признаков n , а его временная сложность составляет $\approx O(m \times n)$.

Класс `SVC` основан на библиотеке `libsvm` и поддерживает ядерный трюк. Временная сложность обычно находится между $O(m^2 \times n)$ и $O(m^3 \times n)$. Это означает, что он становится невероятно медленным при большом количестве обучающих экземпляров (порядка нескольких сотен тысяч). Такой алгоритм идеален для сложных, но небольших или средних обучающих наборов. Тем не менее, он хорошо масштабируется с ростом числа признаков, особенно разреженных.

Метод опорных векторов поддерживает не только линейную и нелинейную классификацию, но и линейную и нелинейную регрессию.

Регрессия SVM пытается уместить *на полосе* как можно больше экземпляров наряду с ограничением нарушений зазора (т.е. экземпляров вне полосы) [2, стр. 210]. Ширина полосы управляется гиперпараметром ε (рис. 22).

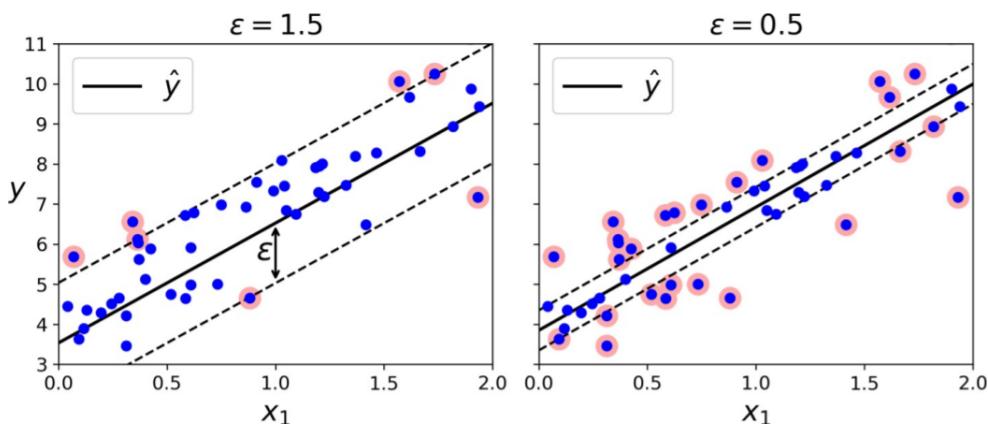


Рис. 22. Линейная регрессия с помощью `LinearSVR`

Добавление дополнительных обучающих экземпляров внутри зазора не влияет на прогнозы модели; потому говорят, что модель *нечувствительна* к ε .

Для решения задач *линейной* регрессии можно использовать класс `LinearSVR`, а для задач *нелинейной* регрессии – `SVR`.

Замечание

Класс `SVR`, как и `SVC` поддерживает ядерный трюк

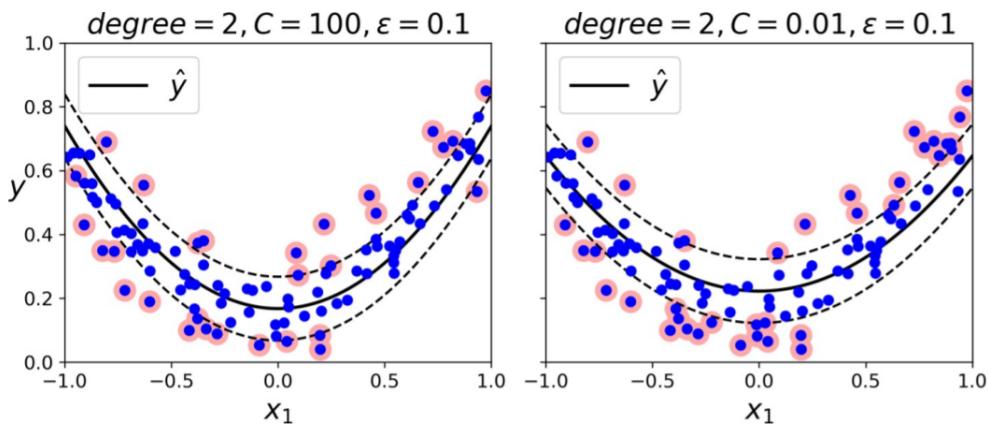


Рис. 23. Нелинейная регрессия с помощью SVR с полиномиальным ядром 2-ого порядка

Если в задаче n признаков, то *функция решения*⁶ – это n -мерная гиперплоскость (рис. 24), а *граница решения*⁷ (то есть множество точек, где функция решения равна нулю) – это $(n - 1)$ -мерная гиперплоскость [2, стр. 212].

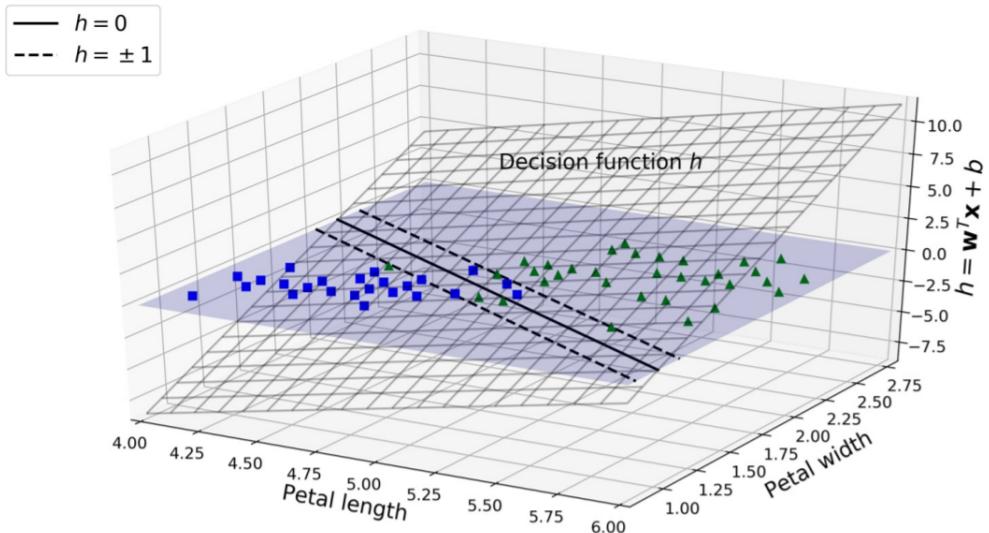


Рис. 24. Функция решения для набора данных об ирисах

Обучение линейного классификатора SVM означает нахождение таких значений $\{w_1, \dots, w_n\}, b$, которые делают *зазор* как можно более широким, одновременно избегая нарушений зазора (жесткий зазор) или ограничиваая их (мягкий зазор) [2, стр. 213].

Задачи *жесткого* и *мягкого* зазора являются задачами выпуклой квадратичной оптимизации с линейными ограничениями [2, стр. 215]. Такие задачи известны как задачи квадратичного программирования (Quadratic Programming – QP).

То есть перейдя от оригинальной задачи (*прямая задача*) с помощью метода множителей Лагранжа и записав условия Каруша-Куна-Таккера, приходим к *двойственной задаче* и решаем ее (находим множители Лагранжа) методами квадратичного программирования.

⁶decision function

⁷decision boundary

Двойственная задача решается быстрее прямой, когда количество обучающих экземпляров меньше количества признаков. Но что более важно, в случае двойственной задачи становится возможен ядерный трюк, в то время как при решении прямой задачи он невозможен.

Ядро – это функция, которая способна вычислять скалярное произведение $\varphi(\mathbf{a})^T \cdot \varphi(\mathbf{b})$, базируясь только на исходных векторах \mathbf{a} и \mathbf{b} , без необходимости вычислять трансформацию φ (или даже знать о ней) [2, стр. 218].

15.4.3. Из курса лекций Соколова

Вспомним, что метод опорных векторов сводится к решению задачи оптимизации (для общего случая классификации с мягким зазором с помощью SVC)

$$\begin{cases} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^l \xi_i \rightarrow \min_{w, b, \xi} \\ y_i(\langle w, x_i \rangle + b) \geq 1 - \xi_i, \quad i = 1, \dots, l, \\ \xi_i \geq 0, \quad i = 1, \dots, l. \end{cases}$$

Построим двойственную к ней. Запишем лагранжиан (ограничения⁸ просто суммируются с учетом множителей Лагранжа λ_i, μ_i и вычитаются из целевой функции)

$$L(w, b, \xi, \lambda, \mu) = \frac{1}{2} \|w\|^2 + C \sum_{i=1}^l \xi_i - \sum_{i=1}^l \lambda_i [y_i(\langle w, x_i \rangle + b) - 1 + \xi_i] - \sum_{i=1}^l \mu_i \xi_i.$$

Выпишем условия Каруша-Куна-Таккера

$$\nabla_w L = w - \sum_{i=1}^l \lambda_i y_i x_i = 0 \Rightarrow w = \sum_{i=1}^l \lambda_i y_i x_i \quad (4)$$

$$\nabla_b L = - \sum_{i=1}^l \lambda_i y_i = 0 \Rightarrow \sum_{i=1}^l \lambda_i y_i = 0 \quad (5)$$

$$\nabla_{\xi_i} L = C - \lambda_i - \mu_i \Rightarrow \lambda_i + \mu_i = C \quad (6)$$

$$\lambda_i [y_i(\langle w, x_i \rangle + b) - 1 + \xi_i] = 0 \Rightarrow (\lambda_i = 0) \text{ или } (y_i(\langle w, x_i \rangle + b) = 1 - \xi_i) \quad (7)$$

$$\mu_i \xi_i = 0 \Rightarrow (\mu_i = 0) \text{ или } (\xi_i = 0) \quad (8)$$

$$\xi_i \geq 0, \lambda_i \geq 0, \mu_i \geq 0. \quad (9)$$

При вычислении градиентов $\nabla_w, \nabla_b, \nabla_{\xi_i}$ просто берем частные производные от лагранжиана L по каждому элементу w_1, w_2, \dots вектора w , по каждому элементу b_1, b_2, \dots вектора b и т.д.

Проанализируем полученные условия. Из (4) следует, что вектор весов w , полученный в результате настройки SVM, можно записать как линейную комбинацию объектов x_i из объектов обучающей выборки, причем веса в этой линейной комбинации можно найти как решение двойственной задачи.

В зависимости от значений ξ_i и λ_i объекты x_i разбиваются на три категории:

1. $\xi_i = 0, \lambda_i = 0$: такие объекты не влияют на решение w (входят в него с нулевым весом λ_i), правильно классифицируются ($\xi_i = 0$) и лежат вне разделяющей полосы. Объекты этой категории называются *периферийными*.

⁸Ограничения должны быть сведены к виду $left_part \geq 0$

2. $\xi_i = 0, 0 < \lambda_i < C$: из условия (7) следует, что $y_i(\langle w, x_i \rangle + b) = 1$, то есть объект лежит *строго на границе разделяющей полосы*. Поскольку $\lambda_i > 0$, объект влияет на решение w . Объекты этой категории называются *опорными граничными*.
3. $\xi_i > 0, \lambda_i = C$: такие объекты могут лежать внутри разделяющей полосы ($0 < \xi_i < 2$) или выходить за ее пределы ($\xi_i \geq 2$). При этом если $0 < \xi_i < 1$, то объект классифицируется правильно, в противном случае неправильно. Объекты этой категории называются *опорными нарушителями*.

Итак, итоговый классификатор зависит от объектов, лежащих на границе разделяющей полосы, и от объектов-нарушителей (с $\xi_i > 0$).

Приходим к следующей *двойственной задаче*

$$\begin{cases} \sum_{i=1}^l \lambda_i - \frac{1}{2} \sum_{i,j=1}^l \lambda_i \lambda_j y_i y_j \langle x_i, x_j \rangle \rightarrow \max_{\lambda} \\ 0 \leq \lambda_i \leq C, i = 1, \dots, l, \\ \sum_{i=1}^l \lambda_i y_i = 0. \end{cases}$$

Она также является вогнутой, квадратичной и имеет единственный максимум.

Двойственная задача SVM зависит только от скалярных произведений объектов – отдельные признаковые описания никак в нее не входят. Значит, можно легко сделать ядерный переход

$$\begin{cases} \sum_{i=1}^l \lambda_i - \frac{1}{2} \sum_{i,j=1}^l \lambda_i \lambda_j y_i y_j K(x_i, x_j) \rightarrow \max_{\lambda} \\ 0 \leq \lambda_i \leq C, i = 1, \dots, l, \\ \sum_{i=1}^l \lambda_i y_i = 0. \end{cases}$$

Подставляя представление (4) в классификатор, получаем

$$a(\mathbf{x}) = \text{sign} \left(\sum_{i=1}^l \lambda_i y_i \langle x_i, \mathbf{x} \rangle + b \right).$$

Таким образом, классификатор измеряет *сходство нового объекта с объектами из обучающей выборки, вычисляя скалярное произведение между ними*. Это выражение также зависит только от скалярных произведений, поэтому в нем тоже можно сделать переход к ядру.

Если использовать гауссовское ядро (радиальную базисную функцию) в *методе опорных векторов*, то получится следующее *решающее правило*

$$a(\mathbf{x}) = \text{sign} \underbrace{\sum_{i=1}^l y_i \lambda_i \exp \left(- \frac{\|\mathbf{x} - x_i\|^2}{2\sigma^2} \right)}_{K(x, x_i)}$$

То есть мы просто в решающем правиле заменили *скалярное произведение* $\langle x_i, x \rangle$ на *ядро* $K(x, x_i)$.

15.4.4. Из книги Буркова

После обучения алгоритм метода опорных векторов будет определяться так

$$f(\mathbf{x}) = \text{sign}(\mathbf{w}^* \mathbf{x} - b^*)$$

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{w}^* 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 [3]

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

Желательно также, чтобы гиперплоскость отделяла положительные данные от отрицательных с максимальным зазором. Зазор – это расстояние между ближайшими образцами двух классов, отделляемых 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 $\|\mathbf{w}\| = \sqrt{\sum_{j=1}^D (w^{(j)})^2}$ [3, 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-приближенном обучении* (Probably Approximately Correct, PAC). Теория вероятностно-приближенного 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 зазором⁹)

$$\min \frac{1}{2}\|w\|^2, \text{ такое, что } y_i(\mathbf{x}_i \mathbf{w} - b) - 1 \geq 0, i = 1, \dots, N. \quad (10)$$

Чтобы распространить SVM на случаи, когда данные невозможны 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)

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

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

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 .

Для решения задачи оптимизации (10) 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.

⁹hard-margin SVM

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

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

15.5. 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.

15.6. 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 матрешки). Например, для трехслойной нейронной сети, возвращающей скаляр, f_{NN} выглядит так [3, стр. 91]

$$y = f_{NN}(\mathbf{x}) = f_3(f_2(f_1(\mathbf{x}))),$$

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.

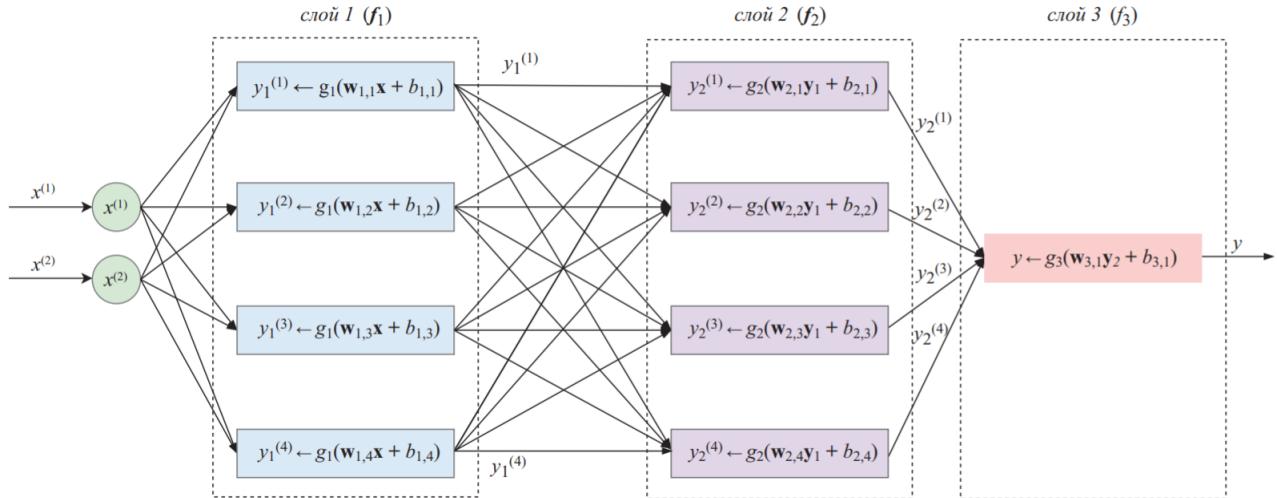


Рис. 25. Многослойный персептрон с двумерным входом, двумя слоями по четыре узла в каждом и выходным слоем с единственным узлом

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 $\mathbf{W}_l z + \mathbf{b}$ является линейной функцией, а линейная функция от линейной функции также является линейной.

16. 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.

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 *исчезающим* 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 [3, 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.

Замечание

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

16.1. 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 скользящ-

щего 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 персептрана, но имеют *только по одному слою* [3, 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. 26).

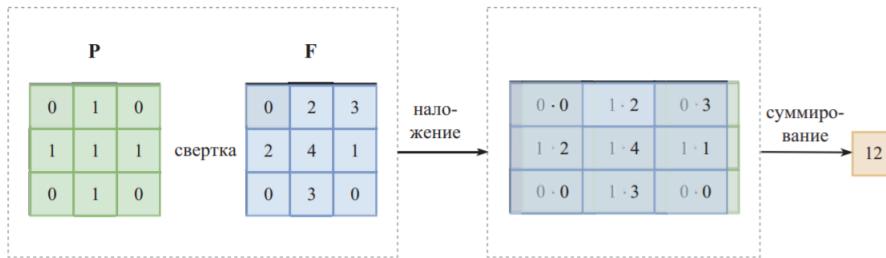


Рис. 26. Свертка двух матриц

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 вычисляется 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.

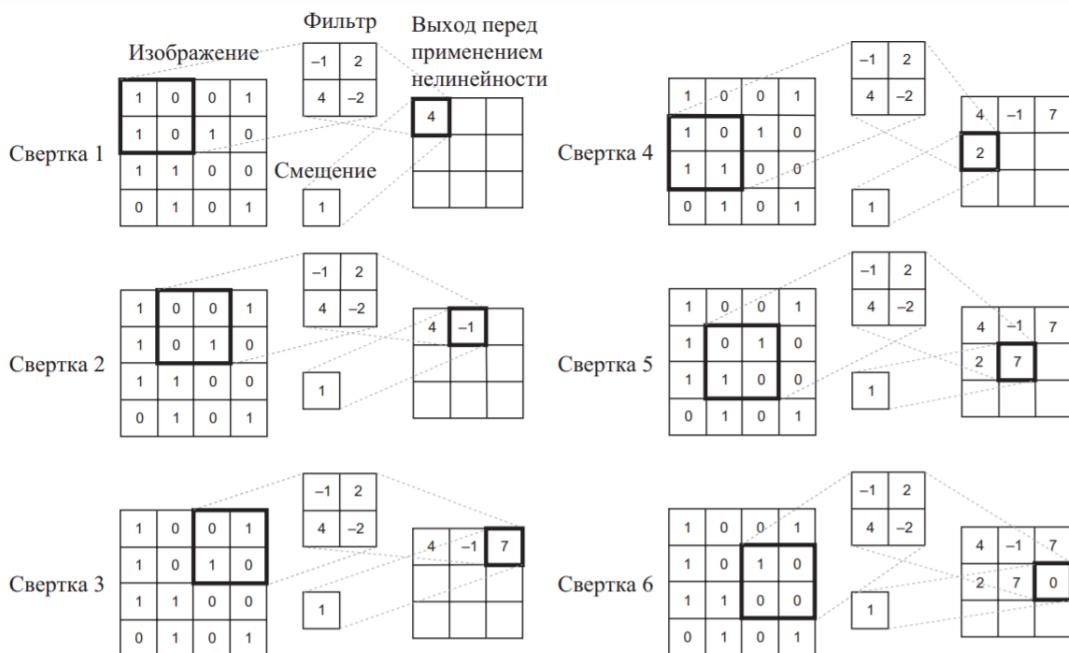


Рис. 27. Фильтр, свертывающий изображение

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.

16.2. Рекуррентная нейронная сеть

Рекуррентные нейронные сети (RNN) используется 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 i рекуррентного слоя l имеет *вещественное состояние* $h_{l,i}$. Состояние можно рассматривать как *память узла*. В RNN каждый узел i 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. 28).

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

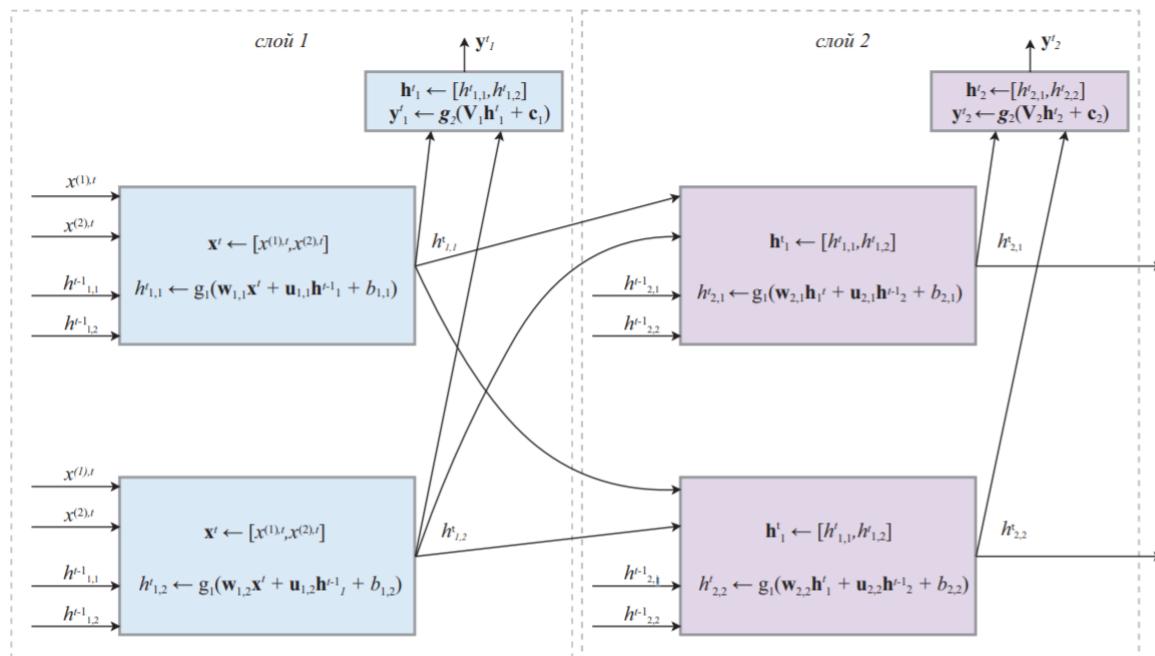


Рис. 28. Первые два слоя в рекуррентной нейронной сети. На вход подается двумерный вектор признаков. Каждый слой имеет два узла

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.

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 вентиляй. Есть несколько архитектур управляемых узлов. Простая, но эффективная называется *минимальным вентильным узлом* и состоит из *ячейки памяти* и *вентиля забывания*

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

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 $\tilde{h}_{l,u}^t$.

17. Приемы работы с библиотекой `plotly`

Шаблонный код для JupyterLab

```
# ячейка
import numpy as np
import plotly.graph_objs as go
from plotly.offline import (
    download_plotlyjs,
    init_notebook_mode,
```

```

plot,
iplot,
)
# ячейка
init_notebook_mode(connected=True)
# ячейка
fig = go.Figure()
# ячейка
fig.add_traces([
    go.Scatter(y=np.random.randn(100).cumsum(), name="curve-1"),
    go.Scatter(y=np.random.randn(100).cumsum(), name="curve-2"),
]);
# ячейка
fig.update_layout(
    title=dict(
        text=(
            "<i>Реализации значений переменных</i>"),
        font=dict(
            family="Arial",
            size=18,
            color="#07689F",
        ),
    ),
    xaxis_title="Имена переменных",
    yaxis_title="Значение переменной",
    xaxis=dict(
        showline=True,
        showgrid=False,
        showticklabels=True,
        linecolor="rgb(204, 204, 204)",
        linewidth=1.5,
        ticks="outside",
        tickfont=dict(
            family="Arial",
            size=15,
            color="rgb(82, 82, 82)",
        ),
    ),
    yaxis=dict(
        showgrid=False,
        zeroline=False,
        showline=True,
        showticklabels=True,
        linecolor="rgb(204, 204, 204)",
        linewidth=1.5,
        ticks="outside",
        tickfont=dict(
            family="Arial",
            size=15,
            color="rgb(82, 82, 82)",
        ),
    ),
    autosize=False,
    margin=dict(
        autoexpand=False,
        l=70,
        r=10,
        t=50,
    ),
    showlegend=True,
)

```

```

plot_bgcolor="white",
legend_title_text="<i>Имя блока легенды</i>",
legend=dict(
    orientation="v",
    yanchor="bottom",
    y=0.01,
    xanchor="right",
    x=0.99,
    font=dict(family="Arial", size=12, color="black"),
),
font=dict(
    family="Arial",
    size=13,
),
)
)

```

Если теперь воспользоваться функцией `plot`, то в текущей директории проекта будет создан `html`-файл интерактивного графика, который автоматически откроется в браузере

```
plot(fig, filename="test-sample.html")
```

18. Приемы работы с библиотекой анализа временных рядов ETNA

18.1. 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 расширяющиеся окном количество наблюдений, использованных для обучения в каждой итерации, растет с числом итераций, предоставляет все больший объем данных для обучения.



Рис. 29. 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. 30. 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**.

НВ 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-требованиями. 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.

Перекрестную 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.

Перекрестную 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.



Рис. 31. Модифицированная перекрестная проверка расширяющимся окном



Рис. 32. Модифицированная перекрестная проверка скользящим окном

При использовании перекрестной проверки расширяющимся окном модель в большей степени нацелена на обнаружение глобальных паттернов и менее склонна к изменениям, т.е. более консервативна. При использовании перекрестной 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 перекрестную проверку 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.

18.2. CatBoost. Базовая модель с конструированием признаков

В ETNA есть два класса-обертки над классом `CatboostRegressor`: `CatBoostModelPerSegment` 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 параметров `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 выборках 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
```

18.3. 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:
    """Вычисляем 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 понадобиться класс LGBRegressor и базовые классы библиотеки ETNA Model и PerSegmentModel.

Сначала надо написать ядро – внутренний класс _LGBMModel, в котором используется LGBMRegressor. 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 LGBMModelPerSegment, который строит
# отдельную модель LGBM для каждого сегмента
class LGBMModelPerSegment(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 `LGBModelMultiSegment`. 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):
    # преевращаем 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):
    # преевращаем 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. 33). 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:

- значения зависимой переменной в наблюдениях исходного набора, которые будут соответствовать будущей тестовой выборке (набору новых данных), заменяют значениями NaN,
- 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.

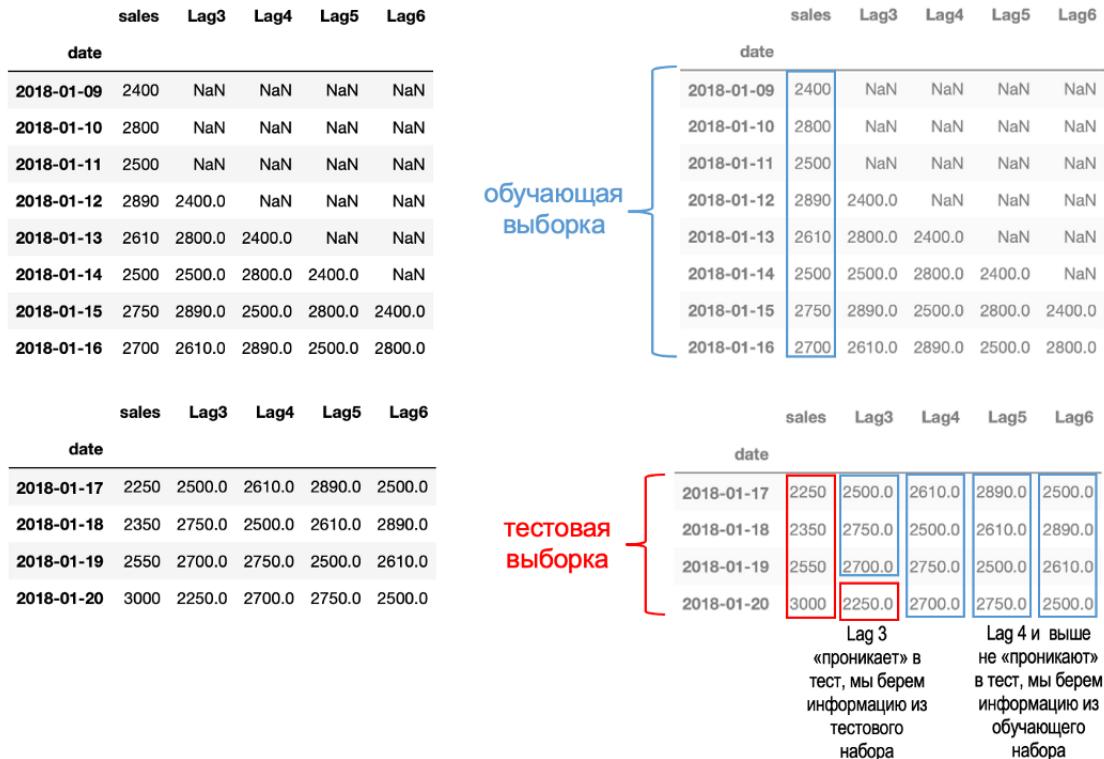


Рис. 33. Лаги, у которых порядок равен горизонту прогнозирования или превышает его, не используют тестовую выборку

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
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)
```

18.4. 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 экспоненцирования переменной (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 умещающих 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 получат 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.

19. Генерация признаков и кодирование категориальных признаков

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

- ОНЕ-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, категории, 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. $count + \epsilon$. Мелкие и новые категории объединяют в одну.

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

19.1. Кодирование одного категориального признака по другому категорциальному 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 псевдообращения 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 A 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

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

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. Так же столбцы матрицы V являются собственными векторами матрицы $A^T A$, 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 Encoding) 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!

20. Перестановочная важность признаков и важность признаков по Шепли

Полезный ресурс https://scikit-learn.org/stable/modules/permuation_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. 34). Как правило, все наоборот. Если признак выбирается часто, значит модель не может по каким-то причинам сразу получить от него нужную информацию.

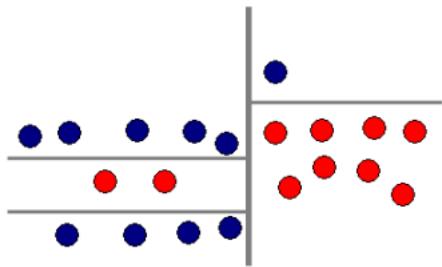


Рис. 34. К вопросу о важности признака по частоте его выбора. Признак по оси x выбирается только один раз, а признак по оси y выбирается три раза, но очевидно, что первый признак лучше справляется с задачей

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; то есть исходный набор данных разбивается на обучение и тест, модель обучается на обучающем поднаборе данных, на тестовом поднаборе мы вычисляем качество модели без модификации признаков, затем перетасовываем какой-то признак тестового поднабора и вычисляем качество на тестовом поднаборе с испорченным признаком, а затем делаем вывод о том на сколько качество изменилось (на сколько признак важен); Модель для вычисления важности признаков обучается на обучающем поднаборе данных, а *важность признаков* вычисляется на *отложенной выборке*, включая различные схемы валидации; то есть можно было бы разбить исходный набор данных на k фолдов: обучаемся на $k - 1$ фолдах, вычисляем качество на *валидационном* поднаборе данных без модификации признаков, затем применяем ту же самую модель (обученную на обучающем поднаборе данных первого разбиения) к валидационному поднабору, но с уже перетасованным i -ым признаком; и так поступаем для всех признаков f каждого фолда; теперь можно усреднить результаты по фолдам; в итоге мы получим f усредненных по фолдам важностей признаков,
- (+) можно применять на любых алгоритмах,
- (+) самый надежный метод,
- (+) в бутстрепе можно использовать ОOB-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. 35) и на любой схеме *валидации* (надежнее использовать валидацию).

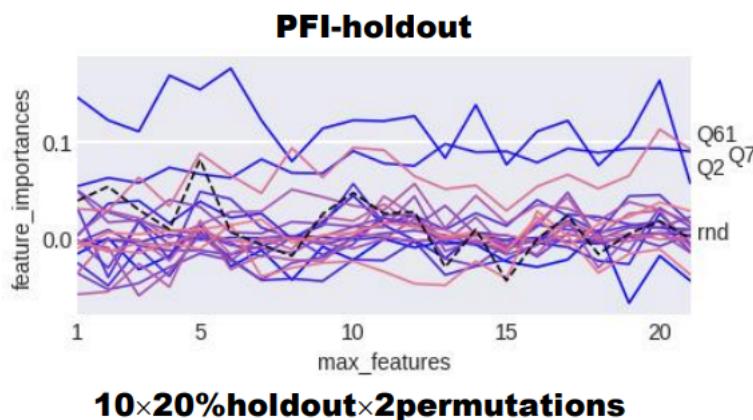


Рис. 35. Перестановочная важность признаков, вычисленная на отложенной выборке

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 признак 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).

21. Приемы работы с библиотекой Polars

21.1. 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.

21.2. Вводные замечания

Polars поддерживает режимы *жадных* (eager)¹⁰ и *отложенных* (lazy) вычислений.

Пример

¹⁰В этом режиме работа с данными выглядит как в pandas

```

import polars as pl

df = pl.read_csv("https://j.mp/iriscsv")
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/iriscsv")
    .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/iriscsv")
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.

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

21.3. 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 склейть 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 времененная 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()

```

21.4. Оконные функции

Окноную 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 `"WinCumsuм"` 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"),
])
]

```

21.5. 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
])
]

```

21.6. 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.

При克莱ить столбец справа можно с помощью метода `hstack`

```
df: pl.DataFrame
df.hstack([
    pl.Series("new_col", np.random.randint(10, 100, size=df.height))
])
```

Вставить столбец в произвольную позицию можно так

```
# insert_at_idx изменяет объект на месте
df.insert_at_idx(0, pl.Series("new_col", np.random.randn(df.height)))
```

21.7. Работа с временными рядами

Pandas-конструкцию `ts.resample(...).agg(...)` силами Polars можно реализовать так

```
ts.groupby_dynamic(
    "time_stamp",
    every="4h",
    closed="left",
).agg([
    pl.col("value").count().alias("value_cnt"),
    pl.col("value").mean().suffix("_mean"),
])
```

22. Приемы работы с библиотекой Catboost

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

22.1. 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
```

22.2. Ключевые особенности пакета

22.3. 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 бутстрата

22.4. 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`)

22.5. Регрессор CatBoostRegressor

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

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

22.6. Функции потерь и метрики качества

22.6.1. Для классификации

Для мультиклассификации <https://catboost.ai/en/docs/concepts/loss-functions-multiclassification.html>

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

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

- Kappa

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

- LogLikelihoodOfPrediction.

22.6.2. 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

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

23. 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
```

23.1. 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.

23.2. Emphasis Settings

SCIP поддерживает следующие настройки выразительности (emphasis):

- `set emphasis easycip`: 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`: для 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.

23.3. Проблемы «using pseudo solution instead»

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

Обратный вызов `CONSENFOPS` аналогичен обратному вызову `CONSENFOLP`, но имеет дело с псевдорешениями 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 псевдо-решение. В этом решении переменные устанавливаются на локальную границу, которая является наилучшей по отношению к целевой функции. О псевдо-решении можно думать как о релаксированном решении со всеми ограничениями за исключением удаляемых границ.

24. Приемы работы с библиотеками Gym и Ecole

24.1. 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([-4.8e-05 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8e-05 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.

24.2. Ecole

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

24.2.1. 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.

Секущие 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.

24.2.2. 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.

24.2.3. 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)

```

25. 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:

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

Замечание

Обратное распространение = SGD + дифференцирование сложных функций (рис. 36)

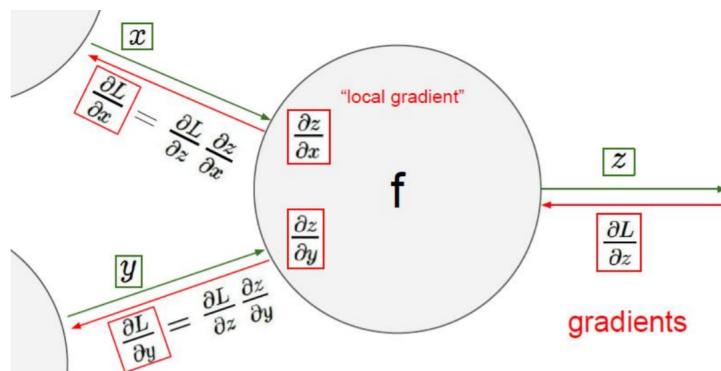


Рис. 36. Прямое распространение сигнала и обратное распространение градиента

26. 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("^\w+.*$"))
    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 на 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 оберачивать 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)

```

27. Борьба с переобучением в нейронных сетях

27.1. 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

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

27.2. 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. 37).

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

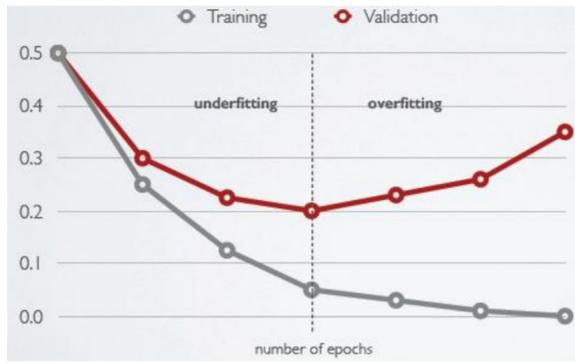


Рис. 37. Настройка темпа обучения

27.3. 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)

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

Метод Нестерова

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

28. 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

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

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]

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

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. 38). 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.

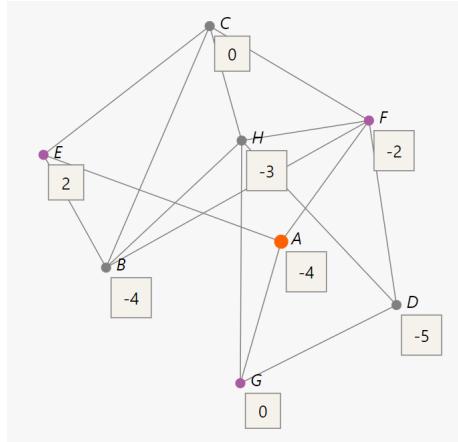


Рис. 38. Пример вычисления обновленного представления узла A на шаге 1 в графовой сверточной нейронной сети

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 потерю 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 номировками,
- 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.

29. 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 даммикодирование, 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 понадобиться еще *проверочная* выборка.

30. Классический и байесовский бутстреп

Бутстреп является универсальным инструментом для оценки статистической точности.

Байесовский бутстреп это байесовский аналог классического бутстрата. 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

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

31. 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.

32. Площадь по ROC-кривой

Построение ROC-кривой происходит следующим образом (рис. 39):

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. 40) 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

Построение ROC-кривой вручную

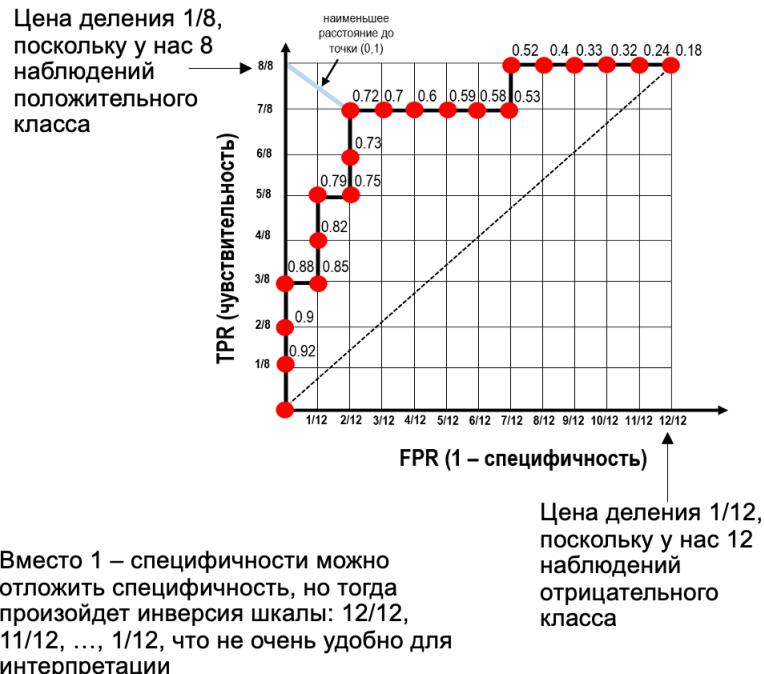


Рис. 39. Построение ROC-кривой

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).

Построение ROC-кривой вручную

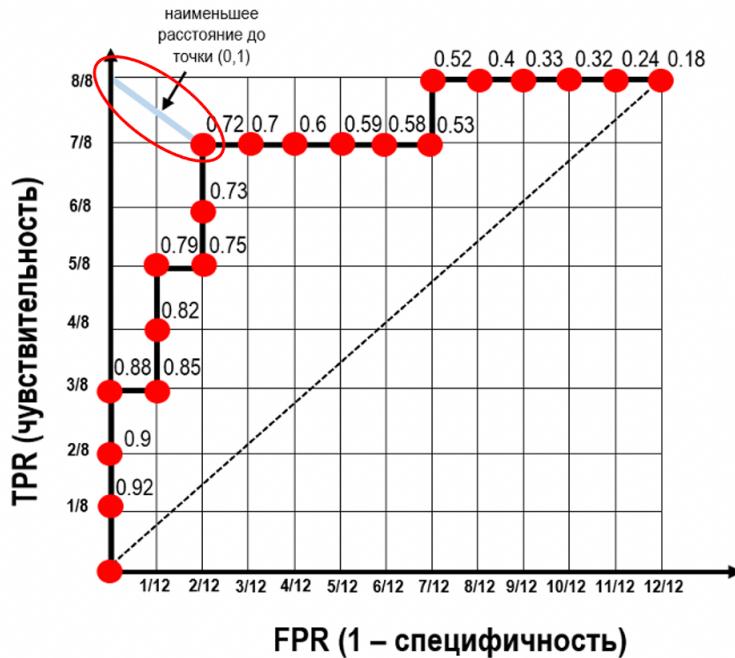


Рис. 40. ROC-кривая. Порог отсечения 0.72

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

$$\frac{\sum_{i,j=1}^{n_i, n_j} s(x_i, x_j)}{n_i n_j}, \quad s(x_i, x_j) = \begin{cases} 1, & x_i > x_j, \\ 1/2, & x_i = x_j, \\ 0, & x_i < x_j, \end{cases} \quad (11)$$

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 образовываем пару с наблюдением отрицательного класса (рис. 41), 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 классификаторы, 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 А 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.

Задача Чему равно значение метрики ROC AUC у классификатора, который для любого объекта возвращает значение 0.97, если доля положительного класса в выборке составляет 4%?

Первый способ. Вероятностная интерпретация. Метрика ROC AUC показывает долю верно упорядоченных пар. Константный классификатор не задает никакого порядка объектов. Это значит, что они упорядочиваются *случайным образом*. А ROC AUC случайного классификатора равен 0.5.

Второй способ. При отрисовке ROC-кривой, в случае одинаковых ответов классификатора, необходимо двигаться и вверх, и вправо одновременно. Значение всего одно, значит прямая тоже одна – это просто диагональная линия. Площадь получившегося треугольника 0.5.

33. 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-картычики 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)
```

Отсортированные спрогнозированные вероятности положительного класса

№	фактический класс	спрогнозированная вероятность положительного 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. 41. Расчет ROC-AUC по формуле (11)

34. Кластеризация. K-means, MeanShift, Affinity Prop, HDBSCAN и детектор выбросов GLOSH

34.1. Краткая сводка по основным кластеризаторам

https://hdbSCAN.readthedocs.io/en/latest/comparing_clustering_algorithms.html#k-means

34.1.1. KMeans

K-means быстрый, простой, легко интерпретируется. Однако, у него есть несколько проблем. Во-первых, строго говоря, это не алгоритм кластеризации, а алгоритм партицирования. То есть K-Means «не ищет» кластеры, а просто разбивает набор данных на заданное число «сферических» партиций (фрагментов). Отсюда вторая проблема. Нужно указать ожидаемое число кластеров.

K-Means зависит от инициализации. Различные случайные запуски будут приводить к различным результатам кластеризации. K-Means предполагает, что кластеры гиперсферические, и не умеет детектировать шумовые точки.

K-Means вычислительно эффективен и на по-настоящему больших данных может оставаться единственным вариантом.

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

№	фактический класс	спрогнозированная вероятность положительного класса	скоринговое правило $S(x_i, x_j)$	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	0,5	11,5
12	N	0,88	0	11
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. 42. Расчет ROC-AUC по формуле (11) для случая равных вероятностей принадлежности экземпляра положительному классу

34.1.2. Affinity Propagation

Как и K-Means предполагает, что кластеры гиперсферические и не умеет детектировать шумовые точки. Не нужно задавать число кластеров, но «правильные» значения его параметров *preference* и *damping* на практике найти не просто.

Очень медленный алгоритм.

34.1.3. MeanShift

Умеет выявлять шумовые точки, но так же как и K-Means предполагает кластеры гиперсферическими. Имеет параметры с более прозрачным смыслом. Зависит от инициализации (производительность может гулять).

Все-таки медленный алгоритм.

34.1.4. Spectral Clustering

Не предполагает, что кластеры гиперсферические, но не умеет выявлять шумовые точки (то есть кластеры загрязняются шумом). Нужно задавать число кластеров.

Медленный алгоритм.

34.1.5. Agglomerative Clustering

Не выдвигает гипотезы о гиперсферичности кластеров, но не умеет выявлять шумовые точки.
Нужно задавать число кластеров.
sklearn-реализация очень медленная.

34.1.6. DBSCAN

Кластеры не обязательно должны быть сферическими. Умеет детектировать шумовые точки. Скопления переменной плотности могут стать проблемой. Параметр ε на практике настроить нелегко. Обычно у алгоритма высокая производительность, но на наборах порядка 1'000'000 экземпляров все-таки работает не очень быстро.

Классический DBSCAN в отличие от DBSCAN* кроме *ядровых* (core objects) и *шумовых* (noise objects) экземпляров используется еще и *границевые* экземпляры (border objects).

34.1.7. HDBSCAN

Может эффективно работать на класетрах переменной плотности. Не делает предположения о гиперсферичности кластеров. Умеет выявлять шумовые точки. Не требует задавать параметр ε . Вместо него используется параметр `min_cluster_size`. Очень эффективная реализация.

34.2. Одноклассовая классификация

В отличие от классической задачи классификация, в одно-классовой классификации нам предоставляются только наблюдения одного класса, а модель должна сообщить принадлежат ли новые наблюдения этому классу или нет.

Если в задаче бинарной классификации требуется, чтобы модель разбила пространство признаков на две области, представляющие свой класс, то в одно-классовой классификации модель должна определить подобласть в пространстве признаков, точки которой будут считаться «нормальными», а точке вне этой подобласти – выбросами (рис. 43).

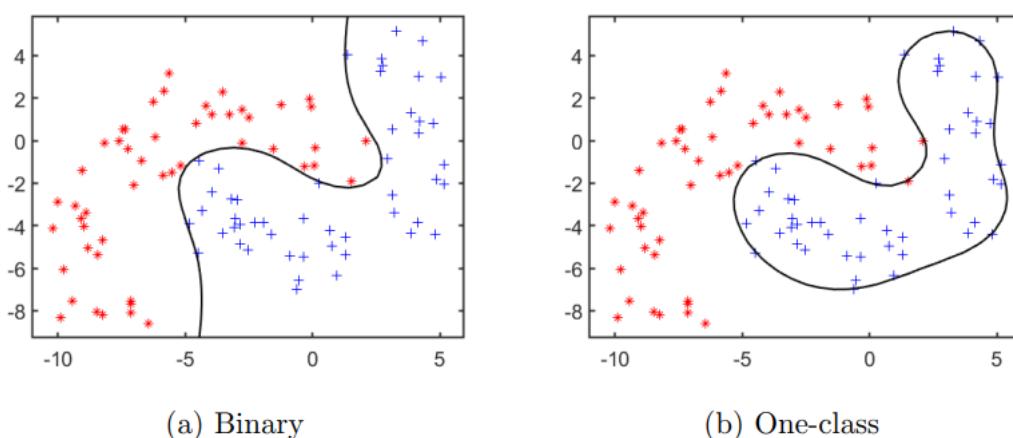


Рис. 43. Бинарная и одно-классовая классификация

34.3. Обнаружение выбросов с помощью GLOSH

GLOSH (Global-Local Outlier Score from Hierarchies) – это алгоритм обнаружения выбросов без использования информации о разметке, основанный на иерархический оценках плотности, предоставляемых HDBSCAN*.

Оценка по GLOSH (грубо можно интерпретировать как уверенность в том, что рассматриваемый экземпляр является выбросом) определяется следующим образом

$$\text{GLOSH}(x_i) = \frac{\lambda_{\max}(C_{x_i}) - \lambda(x_i)}{\lambda_{\max}(C_{x_i})},$$

где $\lambda(x_i)$ – плотность экземпляра x_i ; $\lambda_{\max}(C_{x_i})$ – плотность экземпляра $x_l \in C_{x_i}$ с наибольшей плотностью, где плотности определяются по порогу плотности ε – уровню плотности, отвечающему экземпляру помеченному как «шумовой» в иерархии HDBSCAN*.

Оценки по GLOSH принимают значения из диапазона $[0, 1]$, где оценка 0 означает, что экземпляр считается «штатным», а оценки близкие к 1 указывают на то, что есть основания считать экземпляр выбросом¹¹.

Список иллюстраций

1	Значения остатков простой модели Θ	11
2	Значения остатков модели <code>ExponentialSmoothing</code>	12
3	Ковариаты	13
4	Один экземпляр многомерного временного ряда. Один пакет временных рядов	20
5	Панельные данные. Множественные временные ряды – последовательность временных рядов. Они могут разделять одну и туже временную ось, а могут и не разделять. Могут быть многомерными, а могут и не быть	21
6	Прогноз на многомерном временном ряду	23
7	Процедура обучения модели TFM	25
8	Построение прогноза на одной последовательности для <code>n <= output_chunk_length</code>	26
9	Построение прогноза для <code>n > output_chunk_length</code>	27
10	Пояснения к построению прогноза на горизонт <code>n=2</code>	31
11	Стратегия выращивания деревьев в глубину (level-wise)	64
12	Стратегия LightGBM выращивания деревьев в глубину (level-wise). Получаются несимметричные деревья	65
13	Ансамбль перераспределенных наборов данных	102
14	Машинное обучение как услуга – самый простой шаблон MLPOps	107
15	Модель как зависимость (Model-as-Dependency)	107
16	Лямбда-архитектура для ML-систем в MLOps (Precompute)	107
17	Модель по запросу (Model-on-Demand)	108
18	Гибридная модель обслуживания или Федеративное обучение	109
19	Развертывание ML-модели с помощью Docker и Kubernetes	110
20	Бессерверная стратегия внедрения ML-модели в прод	110

¹¹The higher the score, the more likely the point is to be an outlier https://hdbSCAN.readthedocs.io/en/latest/outlier_detection.html

21	Чувствительность к масштабам признаков. SVM посчитал горизонтальное положение полосы наиболее удачным, так как по оси x_1 масштаб больше и соответственно получается большее значение ширины зазора	119
22	Линейная регрессия с помощью LinearSVR	120
23	Нелинейная регрессия с помощью SVR с полиномиальным ядром 2-ого порядка	121
24	Функция решения для набора данных об ирисах	121
25	Многослойный персептрон с двумерным входом, двумя слоями по четыре узла в каждом и выходным слоем с единственным узлом	127
26	Свертка двух матриц	129
27	Фильтр, свертывающий изображение	130
28	Первые два слоя в рекуррентной нейронной сети. На вход подается двумерный вектор признаков. Каждый слой имеет два узла	131
29	Перекрестная проверка на временном ряду <i>расширяющимся окном</i>	134
30	Перекрестная проверка <i>на скользящем окне</i>	135
31	Модифицированная перекрестная проверка расширяющимся окном	136
32	Модифицированная перекрестная проверка скользящим окном	136
33	Лаги, у которых порядок равен горизонту прогнозирования или превышает его, не используют тестовую выборку	145
34	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	153
35	Перестановочная важность признаков, вычисленная на отложенной выборке	154
36	Прямое распространение сигнала и обратное распространение градиента	176
37	Настройка темпа обучения	179
38	Пример вычисления обновленного представления узла A на шаге 1 в графовой сверточной нейронной сети	181
39	Построение ROC-кривой	184
40	ROC-кривая. Порог отсечения 0.72	185
41	Расчет ROC-AUC по формуле (11)	187
42	Расчет ROC-AUC по формуле (11) для случая равных вероятностей принадлежности экземпляра положительному классу	188
43	Бинарная и одно-классовая классификация	189

Список литературы

1. Лутц М. Изучаем Python, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 1280 с.
2. Жерон О. Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow: концепции, инструменты и техники для создания интеллектуальных систем. – СПб.: ООО «Альфа-книга», 2018. – 688 с.
3. Бурков А. Машинное обучение без лишних слов. – СПб.: Питер, 2020. – 192 с.
4. Бурков А. Инженерия машинного обучения. – М.:ДМК Пресс, 2022. – 306 с.
5. Лакшманан В. Машинное обучение. Паттерны проектирования. – СПб.: БХВ-Перетбург, 2022. – 448 с.
6. Бизли Д. Python. Подробный справочник. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 864 с.

7. Rashmi K.V., Gilad-Bachrach R. DART: Dropouts meet Multiple Additive Regression Trees, 2015
8. Ke G. etc. LightGBM: A Highly Efficient Gradient Boosting Decision Tree, 2017