*Дата последней сборки документа:*
*2 февраля 2024 г. в 13:04*

# Заметки по прогнозированию временных рядов

## Содержание

# 1. Вводные замечания

Возможно самой простой формой прогнозирования является *скользящее среднее*. Часто скользящее среднее используется в качестве *метода сглаживания*, чтобы найти более гладкую линию для данных с большим количеством вариаций [2]. Каждую точку данных можно описать средним значением $n$ окружающих точек данных, где $n$ – обозначает размер окна. При размере окна 10 мы опишем точку данных так, чтобы ее значения представляло собой среднее значение 5 значений, предшествующих точке, и 5 значений, следующих после точки. При прогнозировании будущие значения рассчитываются как среднее $n$ предыдущих значений, поэтому при размере окна 10 речь будет идти о среднем значении 10 предыдущих значений.

Суть сглаживания скользящим средним заключается в том, что вам нужен большой размер окна, чтобы сгладить шум и уловить фактический тренд, но при большом размере окна ваши прогнозы будут значительно отставать от тренда, поскольку вы будете использовать все более ранние наблюдения для вычисления среднего.

Идея экспоненциального сглаживания заключается в том, что мы применяем *экспоненциально уменьшающиеся веса* к усредняемым по времени значениям, придавая недавним значениям больший вес, а большее ранним значениям – меньший.

Хольт усовершенствовал технику экспоненциального сглаживания, чтобы она позволяла учитывать тренд, и назвал ее *двойным экспоненциальным сглаживанием* (double exponential smoothing). А в сотрудничестве с Винтерсом Хольт добавил поддержку сезонности в 1960 году, и техника получала название *тройного экспоненциального сглаживания* (экспоненциальное сглаживание Хольта-Винтерса).

Недостатком этих методов прогнозирования является то, что *они могут медленно приспосабливаться к новым тенденциям*, и поэтому прогнозируемые значения отстают от реальности – *они плохо работают, когда требуется более длительные горизонты прогнозирования*, и существует мнрожество гиперпараметров для настройки, что может быть трудным и очень времязатратным процессом [2, стр. 14].

ARIMA и модель Бокса-Дженкинса часто используется как вазимозаменяемые термины, хотя технически модель Бокса-Дженкинса относится к методу оптимизации параметров для ARIMA-модели.

ARIMA – это аббривиатура трех понятий: *автогрессия*, *интегрированное* и *скользящее среднее*. *Авторегрессия* означает, что модель использует зависимость между точкой данных и некоторым количеством запаздывающих точек данных (лагов). То есть модель делает прогнозы на основе предыдущих значений. Это похоже на предсказание того, что завтра будет тепло, потому что до сих пор было тепло всю неделю.

Интегрированное означает, что вместо применения любой исходной точки данных используется разница между этой точкой данных и некоторой предыдущей точкой данных. По сути, это означает, что мы преобразуем ряд значений в ряд изменений значений. Интуитивно это предпалагает, что завтра будет более или менее такая же температура, как сегодня, потому что температура всю неделю сильно не менялась.

Проблема с ARIMA-моделями заключается в том, что они не поддерживают сезонность или данные с повторяющимися циклами, такими как повышение температуры днем и падение ночью или повышение летом и падение зимой. SARIMA (Seasonal ARIMA) была разработана для преодоления данного недостатка. Подобно нотации ARIMA, нотация для модели SARIMA представляет собой $SARIMA(p,d,q)(P,D,Q)m$, где $P$ – порядок сезонной авторегрессии, $D$ – порядок сезонного дифференциирования, $Q$ – порядок сезонного скользящего среднего, а $m$ – периодичность (количество периодов в полном сезонном цикле).

VARIMA для случаев с несколькими временными рядами в качестве векторов. FARIMA – дробная ARIMA и ARFIMA – дробная проинтегрированная ARIMA. Последние две включают дробную степень дифференциирования, обеспечивающую длительную память в том смысле, что наблюдения, удаленные друг от друга с точки зрения времени, могут иметь несущественные зависимости.

SARIMAX – сезонная ARIMA, где X означает экзогенные или дополнительные переменные, добавленные в модель, например добавление прогнгоза осадков в модель прогнозирования температуры.

ARIMA обычно показывает очень хорошие результаты, но недостатками являются сложность подбора параметров и необходимость тщательного разведочного анализа. Настройка и оптимизация моделей ARIMA часто требуют значительных вычислительных ресурсов, а успешные результаты могут зависеть от навыков и опыта прогнозиста.

Когда дисперсия данных не является постоянной во времени, ARIMA-модели сталкиваются с проблемами [2, стр. 16]. В экономике и финансах непостоянство дисперсии может быть обычным явлением. Для решения этой проблемы были разработаны модели *авторегрессии условной гетероскедостичности* (Autoregressive Conditional Heteroscedasticity – ARCH). Гетероскедостичность – это способ сказать, что дисперсия или разброс данных не являются постоянными повсюду, а противоположенным термином является гомоскедостичность.

Тим Боллерслев и Стивен Тейлор в 1986 году дополнили модель ARCH компонентой *скользящего среднего*, предложив модель Generalized ARCH, или GARCH. Когда колебания случайны, может быть полезна GARCH.

Обе модели ARCH и GARCH не могут обрабатывать ни тренд, ни сезонность, хотя на практике часто для работы с *сезонными колебаниями* и *трендом* временного ряда применяется ARIMA-модель, а затем для моделирования *ожидаемой дисперсии* используется ARCH-модель.

Градиентный бустинг сейчас стал популярен для прогнозирования временных рядов. Следует помнить, что деревья не умеют экстраполировать! Если временной ряд содержит тренд, можно попробовать детрендинг. Мы удаляем из ряда тренд, предварительно спрогнозированный линейной моделью. На полученных остатках обучаем градиентный бустинг и получаем прогнозы, к ним добавляем тренд. Преимуществом градиентного бустинга является способность фиксировать нелинейные зависимости и взаимодействия высокого порядка.

# 2. Математический аппарат Facebook Prophet

## 2.1. Библиотека Facebook Prophet

Используется модель разложимых временных рядов (Harvey & Peters, 1990) с тремя основными компонентами:

- тренд,

- сезонность,
- праздники

$$y(t) = g(t) + s(t) + h(t) + \varepsilon_t,$$

где $g(t)$ – функция тренда, моделирующая непериодические изменения значений временного ряда, $s(t)$ представляет собой периодические изменения (например, еженедельную и ежегодную сезонность), а $h(t)$ – представляет собой эффекты праздников, которые возникают в течение одного или нескольких дней. Член ошибки $\varepsilon_t$ представляет собой любые случайные возмущения, которые не учитываются моделью.

Эта спецификация аналогична *обобщенной аддитивной модели* (GAM) (Hastie& Tibshirani, 1987), классу регрессионных моделей с потенциально нелинейными сглаживаниями, применяемыми к регрессорам.

Выбор в пользу GAM имеет то преимущество, что она легко декомпозируется и допускает включение новых компонент по мере необходимости, например при выявлении нового источника сезонности. GAM также обучается очень быстро, либо с помощью бэкфиттинга, либо с помощью L-BFGS (Byrd et al., 1995) (мы предпочитаем последнее), чтобы пользователь мог интерактивно изменять параметры модели.

По сути, мы формулируем проблему прогнозирования как задачу подгонки кривой, которая внутренне отличается от моделей временных рядов, поскольку те явно учитывают структуру временной зависимости в данных. Хотя мы отказываемся от некоторых важных преимуществ использования генеративной модели типа ARIMA, выбор в пользу *GAM* обеспечивает ряд практических преимуществ [2, стр. 24]:

- гибкость, заключающуюся в том, что мы можем *легко учесть сезонность с несколькими периодами* и позволить аналитикам выдвинуть разные предположения о трендах,
- в отличие от моделей ARIMA, *измерения не должны находиться на одинаковом расстоянии друг от друга* и нам не нужно интерполировать пропущенные значения, возникшие, например, по причине удаления выборосов,
- обучение выполняется очень быстро, что позволяет аналитику интерактивно исследовать большое количество спецификаций модели,
- прогнозная модель имеет легко интерпретируемые параметры, которые аналитик может изменить согласно выдвинутым предположениям относительно прогнозов.

### 2.1.1. Модель тренда

В Facebook реализованы две модели тренда:

- кусочно-логистическая модель роста с насыщением,
- кусочно-линейную модель.

**Нелинейные рост с насыщением** Для прогнозирования роста основной компонентой процесса генерации данных является модель, предсказывающая, как выросла численность население и как она будет расти дальше. Моделирование роста в Facebook обычно похоже на рост населения в естественных экосистемах, где наблюдается нелинейный рост, который насыщается при предельной пропускной способности. Предельной пропускной способностью для количества пользователей Facebook в определенном регионе может быть количество людей, имеющих доступ к

сети Интернет. Такой рост обычно моделируется с помощью логистической модели роста, которая имеет вид

$$g(t) = \frac{C}{1 + \exp\left(-k(t-m)\right)},$$

где $C$ – верхний порог (пропускная способность), $k$ – скорость роста, $m$ – параметр смещения, позволяющий сдвигать функцию вдоль оси времени.

Предельная пропускная способность непостоянна, поскольку количество людей в мире, которые имеют доступ к Интернету, увеличивается. Таким образом, мы заменяем фиксированную пропускную способность $C$ на изменяющуюся во времени пропускную способность $C(t)$. Во-вторых, скорость роста непостоянна.

Мы включаем изменения тренда в модель роста, явно определяя точки изменения (change points), в которых скорость роста может измениться.

Тогда кусочно-логистическая модель роста принимает вид [2, стр. 26]

$$g(t) = \frac{C(t)}{1 + \exp\left(-(k + a(t)^T)\delta\right)(t - (m + a(t)^T)\gamma)}$$

По сути $C(t)$ – это функция верхней границы тренда.

**Линейный тренд с точками изменения** При прогнозировании задач, в которых нет роста с насыщением, кусочно-постоянная скорость роста дает экономную и часто полезную модель. В таком случае модель тренда выглядит следующим образом

$$g(t) = (i + a(t)^T\delta)t + (m + a(t)^T\gamma),$$

где $k$ – скорость роста, $\delta$ содержит корректировки скорости, $m$ – параметр смещения, а $\gamma_i$ устанавливается равной $-s_j\delta_j$, чтобы функция была непрерывной.

Автоматический отбор можно выполнить вполне естественным путем, выбрав априорное распределение Лапласа для $\delta$. Что немаловажно, применение распределения Лапласа для корректировок $\delta$ не влияет на первоначальную скорость роста $\tau$, поэтому когда $\tau$ стремится к 0, обучение сводится к стандартному (а не кусочному) логистическому или линейному росту.

Когда происходит экстраполяция модели за пределы исторических данных для получения прогноза, тренд получает постоянную скорость. Каждая из этих точек имеет изменение скорости $\delta_j \approx Lapalce(0, \tau)$. Параметр $\tau$ – коэффициент масштаба для автоматического выбора точек смены тренда. Мы симулируем значения будущих изменений скорости, которые подражают прошлым значениям путем замены $\tau$ на дисперсию, оцениваемую по имеющимся данным.

Предположение, что тренд продолжит меняться с той же частотой и скоростью изменений, что и в исторических данных, является довольно сильным, поэтому мы не ожидаем высокой точности от доваерительных интервалов. Однако они являются полезным показателем уровня неопределенности и в особенности *показателем переобучения.*

Временные ряды в бизнес-задачах часто имеют *многопериодную сезонность* как результат человеческого поведения, которые они отражают. Например, 5-дневная рабочая неделя может оказывать влияние на временной ряд, повторяющееся каждую неделю, в то время как графики отпусков и школьных каникул могут вызывать эффекты, повторяющиеся каждый год.

Мы предложили использовать ряды Фурье, чтобы получить гибкую модель периодических изменений (Harvey & Shephard, 1993). Пусть $P$ – постоянное значение периода для рассматриваемого временного ряда (например, $P = 365.2$ для годовых данных или $P = 7$ для недельных данных). Мы можем аппроксимировать произвольные сезонные эффекты с помощью стандартного ряда Фурье

$$s(t) = \sum_{n}^{N} \left( a_n \cos \frac{2\pi nt}{P} + b_n \sin \frac{2\pi nt}{P} \right)$$

Подгонка сезонности требует оценки $2N$ параметров $[a_1, b_1, \ldots, a_N, b_N]^T$. Это делается путем построения матрицы векторов сезонности для каждого значения $t$ в наших исторических и прогнозных данных, например ниже приведен пример для годовой сезонности и $N = 10$

$$X(t) = \left[ \cos\left(\frac{2\pi \cdot 1 \cdot t}{365.25}\right), \ldots, \sin\left(\frac{2\pi \cdot 10 \cdot t}{365.25}\right) \right]$$

Тогда сезонная компонента будет иметь вид

$$s(t) = X(t)\beta.$$

В нашей генеративной модели мы берем $\beta \sim Normal(0, \sigma^2)$, чтобы сгладить сезонность. Для годовой и недельной сезонности были найдены значения $N = 10$ и $N = 3$ соответственно, которые работают достаточно хорошо для большинства задач. Выбори этих параметров можно автоматизировать с помощью критерия отбора модели типа AIC [2, стр. 28].

Для оценивания параметров обучаемой модели используются принципы байесовской статистики: либо поиск апостериорного максимума (MAP) с помощью L-BFGS, либо полный байесовский вывод.

Выбрать наблюдения из данного диапазона

```
df.set_index("date").between_time("8:00", "18:00")  # работает только временными индексами
# или так
df[(df["date"].dt.hour >= 8) & (df["date"].dt.hour < 18)].set_index("date")
```

По умолчанию Prophet для годовой сезонности использует ряд Фурье порядка 10, для недельной сезонности – ряд Фурье порядка 3 и для дневной сезонности – ряд Фурье порядка 4. Обычно эти значения по умолчанию работают очень хорошо, и настройка не нужна.

## 2.2. Добавление условных сезонностей

Условная сезонность должна иметь цикл, который короче периода, в рамках которого она действует. Так, например, не имеет смысла задавть годовую сезонность, которая будет действовать всего несколько месяцев.

Для прогнозирования использования электроэнергии в студенческом городе вам потребуется задать либо дневную, либо недельную сезонность, а возможно, даже и то, и другое, в зависимости от паттерна поведения студентов, одну дневную/недельную сезонность для летних месяцев, когда студенты вернулись в свои родные города, и другую – дневную/недельную сезонность для остального времени года. В идеале *условная сезонность* должна иметь *как минимум два полных цикла* всякий раз, когда она действует [2, стр. 77].

Процедура добавления этой условной сезонности заключается в добавлении новых булевых столбцов в обучающий датафрейм (а затем соответствующих столбцов в датафрейм, удлиненный на горизонт прогнозирования), указывающих, является ли данное наблюдение выходным или будним днем.

Например можно создать два булевых столбца в наборе данных, которые будут маркировать будни и выходные. То есть у модели получается как бы дополнительный категориальный признак с двумя уникальными значениями для описания дневной сезонности.

```
def is_weekend(ds):
    date = pd.to_datetime(ds)
    return (date.dayofweek == 5 or date.dayofweek == 6)

df["weekend"] = df["ds"].apply(is_weekend)
df["weekday"] = ~df["weekend"]

model = Prophet(
    seaonlity_mode="multiplicative",  # мультипликативная сезонность
    yearly_seasonality=6,  # порядок Фурье для годовой сезонности
    weekly_seasonality=6,  # порядок Фурье для недельной сезонности
    daily_seasonality=False  # отключаем дневную сезонность по умолчанию
)
```

Для создания условных сезонностей мы используем метод `.add_seasonality()` с параметром `condition_name`

```
model.add_seasonality(
    name="daily_weekend",
    period=1,
    fourier_order=3,
    condition_name="weekend",
)

model.add_seasonality(
    name="daily_weekday",
    period=1,
    fourier_order=3,
    condition_name="weekday",
)

model.fit(df)
future = model.make_future_dataframe(periods=365 * 24)
# индикаторные переменные нужно задать и для прогнгоза
future["weekend"] = future["ds"].apply(is_weekend)
future["weekday"] = future["weekend"]

forecast = model.predict(future)
```

Первый способ применить регуляризацию сезонности – применить регуляризацию глобально, одинаково влияя на все сезонности в модели. Параметр `seasonality_prior_scale` по умолчанию принимает значение 10. Уменьшение этого числа соответствует большей регуляризации, т.е. меньшей гибкости кривой сезонности модели.

```
model = Prophet(
    seasonality_mode="multiplicative",
    yearly_seasonality=4,
    seasonality_prior_scale=10,  # глобальная регуляризация
)
model.fit(df)
```

```
future = model.make_future_dataframe(periods=365)
forecast = model.predict(future)
```

Допустим, вас устраивает кривая годовой сезонности с настройками регуляризации по умолчанию, но кривая недельной сезонности имеет много экстремальных значений и характеризуется переобучением. В этом случае можно использовать метод `.add_seasonality()` для создания новой недельной сезонности с другими значениями параметра `seasonality_prior_scale`

```
model = Prophet(
    seasonality_mode="multiplicative",
    yearly_seasonality=4,
    weekly_seasonality=False,  # отключаем недельную сезонность по умолчанию
)

model.add_seasonality(
    name="weekly",
    period=7,
    fourier_order=4,
    prior_scale=0.01,
)

model.fit(df)
future = model.make_future_dataframe(periods=365)
forecast = model.predict(future)
```

Все сезонности могут иметь разную силу регуляризации, для этого используйте вызовы метода `.add_seasonality()`. Разумные значения для параметра `prior_scale` варьируют от 10 до 0.01.

Регуляризация просто уменьшает масштаб влияния праздника. В Prophet есть параметр `holiday_prior_s`
Праздники, как сезонность, можно регуляризовать глобально и локально.

Глобальная регуляризация влияния праздников настраивается с помощью парааметра `holiday_prior_sca`
Разумные значения `holiday_prior_scale` варьируют от 10 до 0.01. Как и в случае с параметром `seasonality_prior_scale` в большинстве случаев значени параметров `holiday_prior_scale` от 10 до 0.01 будут работать хорошо.

Использование параметра `holiday_prior_scale` позволяет скорректировать влияние всех праздников глобально: каждый праздник регуляризуетсяя одинаково. Однако Prophet позволяет поразному регуляризовать отдельно взятые праздники

```
holidays = make_holidays_df(year_list=year_list, country="US")
black_friday = pd.DataFrame({
    "holiday": "Black Friday",
    "ds": pd.to_datetime([
        "2014-11-28",
        "2015-11-27",
        ...
    ]),
    "prior_scale": 1,
})

taste_of_chicago = pd.DataFrame({
    "holiday": "Taste of Chicago",
    "ds": pd.to_datetime([
        "2014-07-09",
        ...
    ]),
    ""lower_window": 0,
```

```
    "upper_window": 4,
    "prior_scale" 0.1,
})

holidays = pd.concat([
    holidays,
    taste_of_chicago
]).sort_values("ds").reset_index(drop=True)
```

Модель Prophet – это *обобщенная аддитивная модель*. Тренд – это самый фундаментальный строительный блок нашего прогноза. Мы добавляем к нему сезонности, праздники и дополнительные регрессоры.

При логистическом росте Prophet всегда требует, чтобы был установлен «потолок» – значение, которое ваш прогноз никогда не превзойдет. В Prophet этот параметр называется `cap`. Чтобы добавить его в Prophet, нам нужно создать новый столбец под названием cap в датафрейме исторических данных, а также продублировать его в удлиненном датафрейме.

По умолчанию Prophet разместит 25 потенциальных точек изменения в первых 80% временного ряда. Чтобы управлять процедурой автоматического обнаружения точек изменения в Prophet, вы можете изменить оба этих параметра с помощью параметров `n_changepoints` (количество точек изменения) и `changepoint_range` (диапазон охвата данных точками изменения, то есть первые `changepoint_range` процентов данных, в которых необходимо разместить точки изменения) при создании экземпляра модели.

```
model = Prophet(
    seasonality_mode="multiplicative",
    yearly_seasonality=4,
    n_changepoints=5,
)
```

Это приводит к выводу пяти равномерно распределенных потенциальных точек изменения в первых 80% данных.

А можно расположить все 25 потенциальных точек не в первых 80% обучающих данных, а в первых 50%

```
model = Prophet(
    seasonality_mode="multiplicative",
    yearly_seasonality=4,
    changepoint_range=0.5,  # в первых 50% данных
)
```

Важно помнить, что вы размещаете лишь *потенциальные точки изменения*. Prophet по-прежнему будет пытаться занулить как можно больше потенциальных точек изменений. Prophet никогда не разместить точку изменения в будущих данных. Вот почему по умолчанию Prophet будет использовать только первые 80% данных, чтобы предотвратить выбор плохой точки изменения с помощью нескольких последующих точек.

Модель, у которой будет много точек изменения с большими значениями, будет иметь более широкий доверительный интервал.

Как правило, если вы задаете точки изменения в очень поздние моменты временного ряда, ваша модель будет иметь более высокую вероятность переобучения

```
model = Prophet(
    seasonality_mode="multiplicative",
    yearly_seasonality=4,
```

```
    changepoints=["2017-11-01"],
    changepoint_prior_scale=50,  # регуляризация для точек изменения тренда
)
```

Не нужно достаточно часто настраивать количество точек изменения или диапазон охвата данных точками изменения. Значения по умолчанию почти всегда работают очень хорошо. Если вы обнаружите, что Prophet увеличивает или уменьшает количество точек изменения, лучше контролировать это с помощью регуляриации. Для регуляризации мы используем параметр `changepoint_prior_scale`. Слишком гибкая (сложная) модель имеет высокую вероятность переобучения данных, то есть моделирует шум помимо истинного сигнала. Недостаточно гибкая модель плохо соответсвует данным или не улавливает весь доступный сигнал.

Разумные значения для `changepoint_prior_scale` обычно находятся в диапозоне от 0.001 до 0.5.

Собственные точки изменения

```
wc_2024 = pd.DataFrame({
    "holiday": "World Cup 2014",
    "ds": pd.to_datetime(["2014-06-12"]),
    "lower_window": 0,
    "upper_window": 31,
})

wc_2018 = pd.DataFrame({
    "holiday": "World Cup 2018",
    "ds": pd.to_datetime(["2018-06-14"]),
    "lower_window": 0,
    "upper_window": 31,
})

signing = pd.DataFrame({
    "holiday": "Bayern Munich",
    "ds": pd.to_datetime(["2017-07-11"]),
    "lower_window": 0,
    "upper_window": 14,
})

special_events = pd.concat([wc_2014, wc_2018, signing])

changepoints = [
    "2014-06-12",
    "2014-07-13",
    "2017-07-11",
    "2017-07-31",
    "2018-06-14",
    "2018-07-15",
]
```

Для каждого особого события мы добавляем одну потенциальную точку изменения в начале события и одну – в конце.

Иногда даже в тех случаях, когда в данных нет никакой сезонности, можно использовать ту или иную модель сезонности. Потому как она может влиять, к примеру, на праздники. Еще важно уделять внимание регуляризации точек изменения тренда, так как значение по умолчанию часто излишне регуляризует данные [2, стр. 138]

```
model = Prophet(
    seasonality_mode="multiplicative",
```

```
    holidays=special_events,
    yearly_seasonality=False,
    weekly_seasonality=False,
    changepoint_prior_scale=1,
    changepoints=changepoints,
)
```

### 2.3. Добавление регрессоров

Prophet предлагает обобщенный метод добавления любого дополнительного *регрессора*, как бинарного, так и непрерывного.

Следует учитывать, что при использовании дополнительных регрессоров, бинарных и непрерывных, нужно знать *будущие* значения *для всего периода прогноза*. Это не является проблемой для праздников, потому что мы точно знаем, когда наступит каждый праздник в будущем. Все будущие значения должны быть либо известны, как в случае с праздниками, либо спрогнозированы отдельно. Однако вы должны быть осторожно при построении прогноза с использованием данных, которые сами были предсказаны: ошибка в первом прогнозе увеличит ошибку во втором прогнозе, и ошибки будут постоянно накапливаться.

Чтобы добавить в модель новый регрессор, следует воспользоваться методом `.add_regressor()`. В метод `.add_regressor()` мы передаем имя регрессора, которое является именем соответсвующего столбца в кадре данных. Кроме того, можно воспользоваться параметром `prior_scale`, чтобы применить регуляризацию к регрессору, как мы это делали с праздниками, сезонностью и точками изменения тренда.

Еще с помощью параметра `mode` можно указать, должен ли регрессор быть аддитивным или мультипликативным. Чтобы избежать мультиколлинеарности в категориальном регрессоре, закодированном с помощью техники одного активного состояния, лучше удалить один из столбцов. Мультиколлинеарность может затруднить интерпретацию отдельного эффекта. Впрочем Prophet довольно устойчив к мультиколлинеарности, поэтому она не должна существенно повлиять на результаты.

```
model = Prophet(
    seasonality_mode="multiplicative",
    yearly_seasonality=4,
)
# добавляем регрессор clear
model.add_regressor(
    name="clear",
    prior_scale=10,
    standardize="auto",
    mode="multiplicative",
)
# добавляем регрессор not clear
model.add_regressor("not clear")

# добавляем регрессор rain or snow
model.add_regressor("rain or snow")
```

Помним, что нам нужны будущие данные для наших дополнительных регрессоров и мы собираемся прогнозировать только две недели.

```
from datetime import timedelta
# удаляем последние две недели обучающих данных
```

```
train = df[df["ds"] < df["ds"].max() - timedelta(weeks=2)]

mdoel.fit(train)
future = model.make_future_dataframe(periods=14)
future["clear"] = df["clear"]
future["not clear"] = df["not clear"]
future[rain or snow"] = df["rain or snow"]

forecast = model.predict(future)
```

С помощью фукнции `regressor_coefficients()` можно вывести регрессионные коэффициенты

```
from prophet.utilities import regressor_coefficients

regressor_coefficients(model)
```

В столбце coef сохраняются ожидаемые значения коэффициента. То есть ожидаемое влияние регрессора на y при увеличении регрессора на единицу. Например, коэффициент для регрессора temp равен 0.012282. Этот коэффициент говорит нам о том, что с каждым градусом выше среднего происходит увеличение количества поездок на 0.012282, то есть на 1,2% [2, стр. 149].

Для бинарного регрессора rain or snow коэффициент показывает, что в дожливые или снежные дни пассажиропоток снизится на 20,7% по сравнению с пасмурными днями, поскольку категория `claudy`, став исключенным столбцом, является у нас опорной категорией, с которой мы сравниваем остальные категории, ставшие теперь столбцами.

Если бы мы включили все 4 состояния погоды, мы бы сказали, что пассажиропоток снизился на 20,7% по сравнению со значением, прогнозируемым на тот же день модель без включения этих 4 состояний погоды.

Дополнительные регрессоры в Prophet всегда моделируются как *линейные зависимости*. Это означает, что, например, наш дополнительный регрессор temp, который увеличивает количество поездок на 1,2% с каждым градусом, моделирует тренд, который будет продолжаться до бесконечности. Если бы температура поднялась до 120 градусов по Фаренгейту, у нас не было бы возможности изменить линейную зависимость и сообщить Prophet, что количество поездок, вероятно, теперь уменьшится, поскольку на улице стало очень жарко. Хотя это является ограничением Prophet, на практике это не всегда является проблемой. Линейная зависимость очень часто предстает хорошим показателем реальной зависимости, особенно для небольшого диапазона данных [2, стр. 151].

Если во временном ряду есть групповая аномалия (несколько точек, идущих подряд), которая сильно искажает прогнозы, то ее можно просто вырезать [2, стр. 156]. Prophet очень хорошо обрабатывает пропущенные данные, поэтому небольшой гэп не вызовет никаких проблем.

```
# удаляем данные с выбросами
df2 = df[(df["ds"] < "2016-07-29") | (df["ds"] > "2016-09-01")]

model = Prophet(
    seasonality_mode="multiplicative",
    yearly_seasonality=6,
)
model.fit(df2)
future = model.make_future_dataframe(periods=365 * 2)
forecast = model.predict(future)
```

Аномалии можно не удалять из временного ряда, а передать им `None`. Тогда Prophet будет делать прогнозы для этих дат [2, стр. 158].

```
df3.loc[df3["ds"].dt.year == 2016, "y"] = None
```

## 2.4. Автоматическое обнаружение выбросов

Винзоризация – грубый инструмент, который, как правило, не работает с неплоскими трендами. Винзоризация требует, чтобы аналитик указал конкретный нижний и/или верхний процентиль, всем значениям выше или ниже этого процентиля принудительно присваиваеются значения, соответствующие нижнему и/или верхнему процентилю.

```
stats.mstats.winsorize(df["y"], limits=(0, 0.05), axis=0)
```

Тримминг – похожая техника, однако здесь экстремальные значения удаляются.

Вместо использования процентилей иногда имеет смысл использовать стандартное отклонение. Выбросы просто удаляются. Точки, которые лежат на 1,65 стандартного отклонения выше среднего исключаются.

```
df[stats.zscore(df["y"]) < 1.65]
```

Этот метод тоже не подходит, когда данные имеют тренд. Очевидно, что точки, расположенные позже во временном ряду с восходящим трендом, будут обрезаны с большей вероятностью, чем те, которые лежат ранее [2, стр. 163].

Для вычисления среднего и стандартного отклонения можно воспользоваться скользящем средним.

```
df["moving_average"] = df.rolling(
    window=300,
    min_periods=1,
    center=True,
    on="ds"
)["y"].mean()

df["std_dev"] = df.rolling(
    window=300,
    min_periods=1,
    center=True,
    on="ds"
)["y"].std()

df["lower"] = df["moving_average"] - 1.65 * df["std_dev"]
df["upper"] = df["moving_average"] + 1.65 * df["std_dev"]

df = df[(df["y"] < df["upper"]) & (df["y"] > df["lower"])]
```

Важное преммущество этого метода состоит в том, что он учитывает тренд.

Еще можно объявить выбросом все, что выходит за границы доверительных интервалов. Однако использование этого метода основано на неявном предположении, что данные являются стационарными и имеют постоянную дисперсию [2, стр. 165].

## 2.5. Доверительные интервалы

Есть три источника неопределенности, которые составляют общую неопределенность модели Prophet:

- неопределенность в тренде,
- неопределенность в сезонности, праздниках и дополнительных регрессорах,
- неопределенность из-за шума в данных.

### 2.5.1. Моделирование неопределенности тренда

По умолчанию Prophet оценивает только неопределенность тренда плюс неопределенность, возникающую из-за случайного шума в данных. Шум моделируется как нормальное распределение вокруг тренда, а неопределенность тренда моделируется с помощью апостериорного максимума (MAP).

Чтобы получить оценки неопределенности для сезонности используется семплирование на основе MCMC. Самый большой источник неопределенности в прогнозах Prophet – это возможность изменений тренда в будущем.

```
model = Prophet(
    uncerainty_samples=1000
)
```

Разработчики Prophet отмечают, что их предположение о том, что изменения тренда в будущем никогда не будут более значительными, чем изменения тренда в будущем никогда не будут более значительными, чем изменения тренда в исторических данных, является очень ограничительным.

### 2.5.2. Моделирование неопределенности сезонности

MAP-оценка выполняется очень быстро, поскольку это режим Prophet по умолчанию, но он не работает с сезонными колебаниями, поэтому необходим другой метод. Чтобы смоделировать неопределенность сезонности, Prophet использует метод MCMC.

Семплирование на основе MCMC выполняется очень медленно. К сожалению, на компьютере с Windows API PyStan, который взаимодействует с моделью Prophet на языке Stan, есть проблемы, что означает, что семплирование на основе MCMC выполняется очень медленно. В зависимости от количества точек данных обучение модели на компьютере с Windows иногда может занять несколько часов.

Команда Prophet рекомендует пользователям с Windows-компьюетарами работать с Prophet в R или использовать Python на виртуальной машине Linux.

Моделируем сезонную неопределенность

```
# При mcmc_samples=0 Prophet вернется к MAP-оценке и вычислит неопределенность только для компон
    енты тренда
model = Prophet(mcmc_samples=300)
```

Если вы используете семплинг на основе MCMC, обязательно обратите внимание на увеличивашееся количество точек изменения. Если ваша линия тренда кажется слишком извилистой, вы можете просто уменьшить `changepoint_prior_scale`, чтобы применить регуляризацию [2, стр. 191].

## 2.6. Перекрестная проверка

Дла каждой пороговой точки (cutoff) модель будет обучаться на всех данных до нее, а потом будет сделан прогноз в соответствии с горизонтом. Затем полученный прогноз будет сравнен с

фактическими значениями зависимой переменной и будет посчитана метрика качества. Далее модель повторно обучится на всех данных до второй пороговой точки, и процесс будет повторен. Окончательная оценка качества модели будет представлять собой значение качества модели, усредненное по всем пороговым точкам.

Для Prophet рекомендуется использовать как минимум два полных цикла сезонности, поскольку требуется моделировать годовую сезонность.

Пусть есть данные за 3 года, и таким образом, есть как миинимум 2 полных цикла сезонности. Тогда установим размер исходной обучающей выборки равным 2 годам, Руководство магазина хочет прогнозировать на месяц вперед и поэтому устанавливает горизонт, равным 30 дням. Оно планирует запускать модель каждый квартал, поэтому установим период равным 90 дням.

```
df_cv = cross_validation(
    model,
    horizon="90 days",
    period="30 days",
    initial="730 days",
)
```

Мы начинаем обучение с исходного периода (initial) в 2 года, что составляет 730 дней. Устанавливаем горизонт (horizon) равным 90 дней. И наконец, устанавливаем период (period) равным 30 дней, посольку повторно обучаем и оцениваем нашу модель каждые 30 дней.

С вычислительной точки зрения перекрестная проверка – очень затратная операция, и ее можно распараллелить, чтобы ускорить процесс. Все что нужно сделать, – это использовать ключевое слово для `parallel`. Можно выбрать `None`, `"processes"`, `"threads"`, `"dask"`.

```
df_cv = cross_validation(
    model,
    horizon="90 days",
    period="30 days",
    initial="730 days",
    parallel="processes",
)
```

Настройка `parallel="processes"` использует класс `concurrent.futures.ProcessPoolExecutor`, тогда как `parallel="threads"` использует класс `concurrent.futures.ThreadPoolExecutor`. Если есть сомнения, то лучше выбрать `"processes"`.

Для вычисления пяти метрик качества для прогнозов используется функция `performance_metrics()`

```
df_p = performance_metrics(df_cv)
```

Среднеквадратическую ошибку (MSE) нелегко интерпретировать – единицей измерения MSE является квадрат единицы измерения зависимой переменной. Она также чувствительна к выбросам. Однако MSE остается популярной, потому что можно доказать, что MSE равна квадрату смещения плюс дисперсия, поэтому минимизация этого показателя может уменьшить как смещение, так и дисперсию [2, стр. 200].

Средняя абсолютная ошибка (MAE), в отличие от MSE и RMSE, одинаково взвешивает каждую ошибку, она не придает большого значения выборосам или точкам с необычно высокими ошибками.

Охват (coverage) – это посто процентная доля фактичесских значений, которые лежат между прогнозируемыми верхней и нижней границами доверительного интервала. По умолчани границы накрывают 80% данных, поэтому значение охвата должно быть равно 0.8. Если вы обнаружите

значение охвата, которое не равно значению `interval_width`, это означает, что ваша модель плохо откалибрована с точки зрения неопределенности. На практие это означает, что вы вероятно, не можете серьезно относится к полученным доверительным интервалам прогнозов.

Выбор метрики качества для оптимизации вашей модели не является тривиальным выбором. Он может оказать существенное влияние на вашу итоговую модель в зависимости от характеристик данных. С математической точки зрения оптимизация модели по MSE создать модель, предсказывающую значения, близкие к среднему значению ваших данных, а оптимизация по MAE даст прогнозы, близкие к медианному значению данных. Оптимизация по MAPE часто дает заниженные прогнозы.

Однако, если временной ряд является прерывистым, то есть если большинство дат имеют значение 0, то нужно ориентироваться не на медиану, а на среднее. Медиана будет равно 0! В этом случае нужна метрика MSE именно потому, что она чувствительна к выбросам.

Вернемся к кадру данных с результатами перекрестной проверки. Первая строка кадра данных – `"9 days"`. Каждое значение метрики представляет собой скользящее среднее, вычисленное до указанного дня. Функция `performance_metrics()` принимает аргумент `rolling_window`, в котором вы можете изменить ширину окна, но по умолчанию оно равно 0.1. Это представляет собой долю горизонта, которую нужно включить в окно. Поскольку 10% нашего 90-дневного горизонта составляют 9 дней, это значение и будет первой строкой кадра данных.

Параметру `cutoffs` функции `cross_validation()` можно передать пользовательский список дат для пороговых точек.

```
cutoffs = [
    pd.Timestamp(f"{year}-{month}-{day}")
    for year in range(2005, 2019)
    for month in range(1, 13)
    for day in [1, 11, 21]
]

df_cv = cross_validation(
    model,
    horizon="90 days",
    parallel="processes",
    cutoffs=cutoffs,
)

df_p = performance_metrics(df_cv)
```

## 2.7. Обычная оптимизация гиперпараметров по сетке на основе перекрестной проверки

```
param_grid = {
    "changepoint_prior_scale": [0.01, 0.001],
    "seasonality_prior_scale": [1.0, 0.1],
}

# комбинации гиперпараметров
all_params = [
    dict(zip(param_grid.keys(), value))
    for value in itertools.product(*param_grid.values())
]

rmse_values = []
```

```
# создаем пользовательский список пороговых точек
cutoffs = [
    pd.Timestamp(f"{year}-{month}-{day}")
    for year in range(2010, 2019)
    for month in range(1, 13)
    for day in [1, 11, 21]
]

# выполняем перекрестную проверку
for params in all_params:
    model = Prophet(yearly_seasonality=4, **all_params).fit(df)
    df_cv = cross_validation(
        model,
        cutoffs=cutoffs,
        horizon="30 days",
        parllel="processes"
    )
    df_p = performance_metrics(df_cv, rolling_window=1)
    rmse_values.append(df_p["rmse"].values[0])

best_params = all_params[np.argmin(rmse_values)]
```

### 2.7.1. Рекомендации по подбору гиперпараметров

Важные гиперпараметры:

- `changepoint_prior_scale`,
- `seasonality_prior_scale`.

Самым важным гиперпараметром является `changepoint_prior_scale`. Диапазон значений от 0.5 до 0.001 будет приемлемым для большинства задач. Вторым по важности гиперпараметром является `seasonality_prior_scale`. Диапазон – от 10 (по сути, регуляризация отсутствует) до 0.01. Для выбора значения гиперпараметра `seasonality_mode` лучше всего просто изучить график временного ряда и посмотреть, растет ли величина сезонных колебаний вместе с трендом или остается постоянной. Для `changepoint_range` обычно подходит значение по умолчанию 80% (точки изменения тренда определяем в первых 80% обучающих данных). Оно обеспечивает хороший баланс, позволяя тренду измениться там, где это необходимо, но не позволяя ему переориентироваться на последних 20% данных, где ошибки не могут быть исправлены. Параметр `growth` лучше не включать в поиск, значения `"linear"`, `"logistic"`, `"flat"` подбирают, исходя из визуального анализа данных [2, стр. 211].

## 2.8. Байесовская оптимизация гиперпараметров на основе перекрестной проверки

Теперь попробуем баейсовскую оптимизацию гиперпараметров в Prophet

```
param_types = {
    "changepoint_prior_scale": "float",
    "seasonality_prior_scale": "float",
}

bounds = {
    "changepoint_prior_scale": [0.001, 0.5],
    "seasonality_prior_scale": [0.01, 10]
}
```

Затем пишем функию байесовской оптимизации и осуществляем поиск оптимальных значений гиперпараметров

```
def objective(trial):
    params = {}
    for param in [
        "changepoint_prior_scale",
        "seasonality_prior_scale",
    ]:
      params[param] = trial.suggest_uniform(
          param,
          bounds[param][0],
          bounds[param][1],
      )
  m = Prophet(
      yearly_seasonality=4,
      seasonality_mode="additive",
      **params,
  )
  m.fit(df)

  df_cv = cross_validation(
      m,
      initial="730 days",
      period="30 days",
      horizon="90 days",
      parallel="processes",
  )
  df_p = performance_metrics(df_cv, rolling_window=1)

  return df_p["rmse"].values[0]

sampler = RandomSampler(seed=10)
study = optuna.create_study(
    sampler=sampler,
    direction="minimize"
)
study.optimize(lambda trail: objective(trial), n_trails=10)

study.best_params
study.best_value
```

## 3. Библиотека ETNA

В библиотеке ETNA `PerSegment` обозначает, что для каждого сегмента – временного ряда будет построена отдельная модель. `MultiSegment` обозначает, что для всех сегментов – временных рядов будет построена отдна модель.

ETNA требует определенного формата данных: столбец, которым мы хотим спрогнозировать, должен называться `target`, столбец с временными метками должен называться `timestamp`. Поскольку библиотека может работать с несколькими временными рядами, столбец `segment` также является обязательным. Если мы работаем с одним *временным рядом*, то есть с одним *сегментом*, для столбца `segment` по соглашению используется значение `'main'` (хотя это не приниципально) [2, стр. 248]

```
                    timestamp    target segment
0   2024-01-21 21:59:56.585934  0.982822    main
```

```
1   2024-01-22 21:59:56.585934  0.794492    main
2   2024-01-23 21:59:56.585934 -0.086600    main
3   2024-01-24 21:59:56.585934  1.182513    main
```

Библиотека работает со специальной структурой данных `TSDataset`. Класс `TSDataset` – основной класс библиотеки ETNA для работы с временными рядами. Его главные методы:

- `.to_dataset()` – превращает кадр данных с плоским индексом в кадр данных с мультииндексом,
- `.to_pandas()` – превращает объект TSDataset в датафрейм с плоским индексом (`flatten=True`) или кадр данных с мультииндексом (`flatten=False`),
- `.fit_transform()` – вычисляет и применяет преобразования и конструирование признаков (например, логарифмирует зависимую переменную, прогнозирует тренд и удаляет тренд из зависимой переменной, вычисляет скользящее среднее с шириной окна 20 дней и создает столбец с ним), требует списка экземпляров классов-трансформеров,
- `.inverse_transform()`,
- `.make_future()` – создает тестовый набор / набор новых данных, увеличив обучающий / исторический набор данных на длину горизонта прогнозирования и применив преобразования и конструирование признаков.
- `.train_test_split()` – разбивает на обучающую и тестовую выборку с учетом временной структуры,
- `.describe()` – вывод описательных статистик,
- `.plot()`.

Чтобы получить объект `TSDataset` из кадра данных, нужно обернуть кадр этим классом

```
from etna.datasets.tsdataset import TSDataset
# df -- pd.DataFrame(...)
ts = TSDataset(
    TSDataset.to_dataset(df),  # кадр данных с мультииндексом
    freq="D"
)  # обеъкт TSDataset
```

Все переменные в ETNA делятся на *эндогенные*, или целевые, переменные (собственно прогнозируемые временные ряды, которые получают названия `target`) и экзогенные переменные – все переменные, кроме целевых. Упрощенно говоря, экзогенные переменные будут нашими признаками, помогающими спрогнозировать целевую переменную.

Экзогенные переменные делятся на переменные, которые мы создаем с помощью трансформеров ETNA (внутренние), и переменные, которые мы создаем сомастоятельно и добавляем к признакам, созданным с помощью ETNA (внешние), с помощью параметра `df_exog`.

Экзогенные переменные вне зависимости от того, как они были созданы (в ETNA или самостоятельно), делятся на два типа – экзогенные переменные, значения которых *известны в будущем* (в ENTA из называют *регрессорами*), и экзогенные переменные, значения которых будут *неизвестны в будущем* (*не регрессоры*).

Разбиваем имеющийся временной ряд на обучение и тест

```
train_ts, test_ts = ts.train_test_split(test_size=50)
# или так
train_ts, test_ts = ts.train_test_split(
    train_start="1980-01-01",
    train_end="1980-12-01",
    test_start="1994-01-01",
```

```
    test_end="1994-08-01",
)
```

Формируем набор, для которого требуется получить прогнозы

```
HORIZON = test_ts.df.size

future_ts = train_ts.make_future(
    future_steps=HORIZON,
    tail_steps=model.context_size
)
```

С помощью метода `.forecast()` получаем прогнозы.

### 3.1. Модель SARIMAX

SARIMAX – сезонная авторегрессионная интегрированная модель скользящего среднего с экзогенными переменными. SARIMAX требует *стационарного* временного ряда [2, стр. 272].

Для проверки временного ряда на стационарность применяются тесты на единичные корни, в части *расширенный тесты Дикки-Фуллера*.

```
from statsmodels.tsa.stattools import adfuller
adfuller(ser)
# output
(-9.885645883333893,
3.670284491637315e-17,
2,
247,
{'1%': -3.457105309726321,
  '5%': -2.873313676101283,
  '10%': -2.5730443824681606},
611.8210866094855)
```

Если тренд, сезонность и результаты теста гововрят о нестационарности временного ряда, нужно применить *сезонное дифференцирование*. Однако для этого нужно определить количество периодов для сезонного дифференцирования.

Для этого строим график ACF. Лаги, соответствующие длине полного сезонного цикла ($s$) или кратные ему ($2s$, $3s$ т.д.), будут иметь высокие значения коэффициентов [2, стр. 273].

Первое правило Нау: если график ACF демонстрирует резкое падение коэффициентов и/или имеет отрицательную автокорреляцию в лаге 1, то есть если ряд слегка сверхдифференцирован (при этом на графике PACF обычно происходит затухание коэффициентов экспоненциально или по синусоиде), то нужно рассмотреть добавление MA-члена в модель. Номер лага, в котором начинается резкое падение на графике ACF, – это количество MA-членов [2, стр. 278].

Второе правило Нау: если автокорреляция в сезонный период положительна, рассмотрите возможность добавления SAR-члена в модель. Если автокорреляция в сезонный период отрицательна, рассмотрите возможность добавления SMA-члена в модель.

```
from etna.models import SARIMAXModel

model = SARIMAXmodel(
    order=(0, 1, 1),
    seasonal_order=(0, 1, 1, 12)  # 12 означает, что в данных ГОДОВАЯ сезонность!
)

model.fit(train_ts)
```

```
future_ts = train_ts.make_future(HORIZON)
forecast_ts = model.forecast(future_ts)
```

Перекрестная проверка расширяющимся окном

```
from etna.pipeline import Pipeline

model = ProphetModel()

transform = []

pipeline = Pipeline(
    model=model,
    transform=transfrom,
    horizon=HORIZON,
)
```

Запустить *перекрестную проверку* можно с помощью метода `.backtest()` класса `Pipeline` [2, стр. 289]. Передаем в метод набор данных и указываем следующие параметры:

- `metrics` – список метрик,
- `n_folds` – количество тестовых выборок перекрестной проверки (по умолчанию 5),
- `mode` – режим перекрестной проверки (по умолчанию `"expand"`, можно задать `"expand"` – перекрестную проверку *расширяющимся окном* или `"constant"` – перекрестная проверка *скользящим окном*),
- `aggregate_metrics` – агрегирование метрик по тестовым выборкам перекрестной проверки,
- `n_jobs` – количество ядер процессора для расспараллеливания.

Метод `.backtest()` возвращает три датафрейма – датафрейм с метриками по каждой тестовой выборке перекрестной проверки, датафрейм с прогнозами, датафрейм с временными метками обучающей и тестовой выборок перекрестной проверки.

```
metrics_df, forecast_df, fold_info_df = pipeline.backtest(
    model="expand",
    ts=ts,
    metrics=[smape],
)
```

Класс `MeanTransform` позволяет создать скользящее среднее. Ширину окна обычно берут не меньше горизонта прогнозировани, $w \geqslant \text{horizon}$.

Как правило для градиентного бустинга лучше всего работают скользящие средние, скользящие стандартные отклонения, скользящие средние / медианные абсолютные отклонения [2, стр. 301].

Класс `LagTransform` позволяет создать лаги. Лаги вида $L_{t-k}$ создаем так, чтобы $k$ был не меньше горизонта прогнозирования, $n_{lags} \geqslant \text{horizon}$.

```
from etna.transformers import MeanTransform, LagTransform

mean8 = MeanTransform(
    in_column="target",
    window=8,
    out_column="mean_08",
)

lags = LagTransform(
    in_column="target",
```

```
    lags=range(6, 10),
    out_column="lag",
)

# добавляем новые признаки в выборку
train_ts.fit_transform([mean8, lags])
# формируем набор, для которого нужно получить прогнозы
future_ts = train_ts.make_future(HORIZON, [mean8, lags])
```

Новый способ дает для скользящего среднего с шириной окна, равной горизонту прогнозирования, пропуск в последнем значении валидационной выборки, который заменяется нулем. Лаги, у которых порядок меньше горизонта, в валидационной выборке получают пропуски.

Как только наше скользящее среднее выходит за пределы обучающей выборки, происходит уменьшение ширины окна. При вычислении скользящих средних для тестовой выборки обоими способами мы *мы ни разу не использовали* значения продаж в тестовой выборке.

### 3.2. Настройка гиперпараметров модели на валидационной выборке

Можно воспользоваться классом `Pipeline`

```
train_ts, valid_ts = ts.train_test_split(
    train_start="...",
    train_end="...",
    test_start="...",
    test_end="...",
)

lags = LagTransfrom(
    in_column="target",
    lags=range(8, 13),
    out_column="lag",
)

pipeline = Pipeline(
    model=model,
    transforms=[lags, mean8],
    horizon=HORIZON,
)

pipeline.fit_transform(train_ts)
forecast_ts = pipeline.forecast()
```

### 3.3. Настройка гиперпараметров модели с помощью перекрестной проверки (с конструированием признаков)

С целью получения *более надежных оценок* при подборе гиперпараметров выполним перекрестную проверку модели (воспользуемся перекрестной проверкой расширяющимся окном).

Давайте выделим обучающую и тестовую выборки с учетом временной структуры. Тестовая выборка содержит последние 8 дней исходного набора. Сама *перекрестная проверка* запускается на *обучающей выборке*. Обратите внимание, что применение перекрестной проверки *не отменяет откладывание тестовой выборки*. Проверочные выборки перекрестной проверки используются для настройки гиперпараметров, а тестовая выборка, – для итоговой оценки модели. Признаки создаются, а преобразования применяются заново на каждой итерации перекрестной проверки [2, стр. 313].

```
transforms = [lags, mean8]

pipeline = Pipeline(
    model=model,
    transforms=transforms,
    horizon=HORIZON,
)

train_ts, test_ts = ts.train_test_split(
    train_start="1980-01-01",
    train_end="1993-12-01",
    test_start="1994-01-01",
    test_end="1994-08-01",
)

# Перекрестная проверка расширяющимся окном
metrics_df, forecast_df, fold_info_df = pipeline.backtest(
    mode="expand",
    ts=train_ts,
    metrics=[smape],
)

# Среднее значение и стандартное отклонение SMAPE (по фолдам перекрестной проверки)
print(metrics_df["SMAPE"].mean())
print(metrics_df["SMAPE"].std())
```

Ниже приведен пример написания и использования собственной функции для настройки гиперпараметров – набора преобразований / признаков с помощью перекрестной проверки. Функция `etna_cv_optimize()` находит набор переобразований / признаков, дающий наилучшее значение метрики по итогам перекрестной проверки, запущенной на обучающей выборке, и затем обучает конвейер с наилучшим набором на всей обучающей выборке и вычисляет метрику качества на тестовой выборке [2, стр. 315].

```
def etna_cv_optimize(ts, model, horizon, transforms, n_folds, mode, metrics, refit=True,
    n_train_samples=10
):
    train_ts, test_ts = ts.train_test_split(test_size=horizon)
    best_score = np.inf

    for trans in transfrms:
        pipe = Pipeline(
            model=model,
            transforms=trans,
            horizon=horizon,
        )

        df_metrics, _, _ = pipe.backtest(
            mode=mode,
            n_folds=n_folds,
            ts=train_ts,
            metrics=[metrics],
            aggregate_metrics=False,
            joblib_params=dict(verbose=0)
        )

        # вычисляем значение метрики, усредненное по тестовым выборкам
        metrics_mean = df_metrics[metrics.__class__.__name__].mean()
```

```
        # вычисляем стандартное отклонение метрики
        metrics_std = df_metrics[metrics.__class__.__name__].std()

        if metrics_mean < best_score:
            best_score = metrics_mean
            best_params = {"trans": trans}

    if refit:
        pipe = Pipeline(
            model=model,
            transforms=best_params.get("trans"),
            horizon=horizon,
        )

        # обучаем конвейер на всей обучающей выборке
        pipe.fit(train_ts)
        forecast_ts = pipe.forecast()
        print(metrics(y_true=test_ts, y_pred=forecast_ts))

        plot_forecast(
            forecast_ts,
            test_ts,
            train_ts,
            n_train_samples=n_train_samples,
        )
```

Выполняем подбор гиперпараметров – набора преобразований / признаков с помощью перекрестной проверки

```
model = CatboostModelMultiSegment(
    loss_function="MAE",
    n_estimators=600,
    learning_rate=0.05,
    depth=9,
    random_seed=42,
)

etna_cv_optimize(
    ts=ts, model=model, horizon=HORIZON, transforms=[transforms, transforms2],
    n_folds=5, mode="expand", metrics=SMAPE(), refit=True,
)
```

### 3.4. Перекрестная проверка нескольких моделей

В ряде случаев бывает полезно сравнить несколько базовых моделей, не предполагая настройки гиперпараметров. Например, мы много сил вкладываем в модель CatBoost, но вполне возможно, что более простая модель даст лучшее качество прогнозов (ситуация, встречающаяся в прогнозировании рядов сплошь и рядом).

### 3.5. Ансамбли

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

```
from etna.ensemble import VotingEnsemble

pipes = [prophet_pipeline, sma_seasonality_12_pipeline]
```

```
vot_ens = VotingEnsemble(
    pipelines=pipes,
    weights=[0.5, 9],
    n_jobs=2,
)
```

Для получения более надженой оценки качества применим перекрестную проверку

```
vot_ens_metrics, vot_ens_forecasts, _ = vot_ens.backtest(
    ts=ts,
    metrics=[MAE(), MSE(), SMAPE(), MAPE()],
    n_folds=3,
    aggregate_metrics=True,
    n_jobs=2,
)
```

### 3.6. Стекинг

С помощью класса `StackingEnsemble` можно выполнить стекинг. По умолчанию `features_to_use=None` и метамодель может использовать в качестве признаков прогнозы моделей первого уровня (моделей конвейеров). С помощью параметра `n_folds` задаем каличество тестовых выборок перекрестной проверки (используем не для оценки моделей, а для получения прогнозов, которые станут у нас потом признаками) [2, стр. 339].

```
stacking_ensemble_unfeatured = StackingEnsemble(
    features_to_use=None,
    pipelines=pipelines,
    n_folds=10,
    n_jobs=4,
)

stacking_ensamble_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_ensamble_metrics.index = ["stacking ensamble"]
```

### 3.7. Работа с трендом и сезонностью

Исследователям, применяющим градиентный бустинг для прогнозирования временных рядов, следует помнить, что одним из недостатков деревьев и их потомков – случайного леса и градиентного бустинга – можно считать принципиальную неспособность деревьев выполнять *экстраполяцию* при решении задачи регрессии: предсказание для любой комбинации предикторов *всегда будет средним значением в одном из листьев*, то есть *за пределы диапазона значений зависимой переменной в обучающей выборке выйти невозможно*. При работе с временными рядами, содержащими тренд, получится, что на *тестовой* выборке мы выйдем из диапазона значений, известных модели дерева на обучающей выборке, и модель дерева просто будет продолжать *предсказывать последнюю известную точку* (прогнозы будут представлять собой горизонтальную линию). В этом случае можно воспользоваться процедурой удаления тренда с последующим восстановлением [2, стр. 368].

Процедура состоит из следующих шагов:

1. На обучающем массиве признаков, в котором единственный признак – номер периода, и обучающем массиве меток (собственно временном ряды) обучаем линейную модель (напри-

мер, модель линейной регрессии). Прогнозами у нас будут значения тренда для обучающего ряда.

2. Если у нас *сезнность* является *мультипликативной*, то значения обучающего массива меток *делим на значения тренда* для обучающей выборки и получаем обучающий массив меток без тренда. Если у нас *сезонность* является *аддитивной*, то из значений обучающего массива меток *вычитаем значения тренда* для обучающего ряда и получаем обучающий массив меток без тренда.
3. Вычисляем значения тренда для тестового ряда. Обратите внимание, что здесь мы нигде не используем значения тестового ряда, мы помним, что тестовый ряд – это прообраз новых данных, о котором мы ничего не знаем.
4. На обучающем массиве признаков и обучающем массиве меток без тренда обучаем модель градиентного бустинга. Затем получаем прогнозы для *тестового* массива признаков.
5. Наконец, выполняем восстановление тренда. Если у нас была аддитивная сезонность, мы к прогнозам градиентного бустинга для тестового ряда прибавляем значения тренда для тестового ряда (потому что ранее вычитали тренд). Если у нас была мультипликативная сезонность, мы умножаем прогнозы градиентного бустинга для тестового ряда на значения тренда для тестового ряда (потому что ранее делили на тренд).

Данную процедуру можно выполнить с помощью классов `LinearTrendTransform`, `ChangePointsTrendTran` (используется класс `Binseg` библиотеки ruptures) и `TheilSenTrendTransform`[1] (вычисляет линейный тренд с помощью линейной регрессии, в которой выбирается медиана наклонов всех прямых, проходящих через пары точек выборки на плоскости, этот подход называется методом оценочной функции Тейла-Сена).

Часто процедуру детерендинга сочетают с процедурой логарифмирования: сначала выполняют логарифмирование, а затем детрендинг. Логарифмирование позволяет стабилизировать дисперсию временного ряда, особенно это актуально, когда дисперсия возрастает со временем и это может ухудшить прогноз как линейной модели, так и модели на основе деревьев решений. Если выполняется логарифмирование, то прогнозы с восстановленным трендом необходимо экспоненцировать.

Библиотека ETNA предлагает инструменты для моделирования сезонности. Сезонная компонента – регулярные изменения уровня ряда с постоянным периодом. Этим периодом может быть час, день, неделя, месяц. кварател, год. Сезонность может быть *аддитивной* (амплитуда сезонных колебаний остается неизменной) и *мультипликативной* (амплитуда сезонных колебаний варьирует). Сезонность всегда фиксирована и известна.

С помощью класса `FourierTransform` подбираем порядок (параметр `order`), по сути количество компонентов (членов) ряда Фурье с периодом, соответствующим обнаруженной годовой сезонности, то есть с периодом 365.24 (параметр `period`).

Признаки времени в рамках 24-часового суточного цикла можно создать с помощью параметров класса `TimeFlagsTransform`. Признаки дат в рамках годового цикла можно создать с помощью параметров `DateFlagsTransform`.

```
train_raif_ts, valid_raif_ts = raif_ts.train_test_split(
    train_start="2008-01-02",
    train_end="2015-07-04",
    test_start="2015-07-05",
    test_end="2015-10-02",
```

[1] Используется класс `TheilSenRegressor` библиотеки scikit-learn

```
)

# для логарифмирования зависимой переменной
log = LogTransform(in_column="target")

# для детрендинга
detrend = LinearTrendTransform(in_column="target")

# порядок лага должен быть неменьше горизонта прогнозирования
lags = LagTransform(
    in_column="target",
    lags=list(range(90, 271, 30)),
    out_column="lag",
)

# скользящие средние
mean90 = MeanTransform(
    in_column="target",
    window=90,
    out_column="mean90",
)

mean150 = MeanTransform(
    in_column="target",
    window=150,
    out_column="mean150",
)

mean210 = MeanTransform(
    in_column="target",
    window=210,
    out_column="mean210",
)

# признаки на основе дат
d_flags = DateFlagsTransform(
    day_number_in_week=True,
    day_number_in_week=True,
    month_number_in_year=True,
    out_column="datetime",
)

# признаки для праздников
holidays = HolidayTransform(
    iso_code="RUS",
    out_column="RUS_holidays",
)

fourier_year = FourierTransform(
    period=365.24,
    order=3,
    out_column="fourier_year",
)

# список преобразований/признаков
trans = [log, detrend, lags, mean90, mean150, mean210, d_flags, holidays, fourier_year]

train_raif_ts.fit_transform(trans)

model = CatBoostMultiSegmentModel()
```

```
model.fit(train_raif_ts)

future_ts = train_raif_ts.make_future(raif_HORIZON, trans)
forecast_ts = model.forecast(future_ts)
forecast_ts.inverse_transform(trans)
smape(y_true=valid_raif_ts, y_pred=forecast_ts)
```

### 3.8. Обработка выбросов

С помощью функций `get_anomalies_median()`, `get_anomalies_density()` и `get_anomalies_hist()` мы можем попробовать разные способы обработки выбросов. Они используются только для разведочного анализа.

Согласно медианного способа выбросы – это все точки, отклоняющиеся от медианы более чем на `alpha * std`, то есть $median \pm \alpha \cdot \sigma$. Параметры:

- $\alpha$ (`alpha`) – порог (по умолчанию 3),
- $\sigma$ (`sigma`) – выборочная дисперсия в окне размером `window_size`,
- `window_size` – размер окна или количество наблюдений (по умолчанию 10).

Согласно способу обнаружения выбросов на основе плотности для каждой точки ряда мы строим все окна размера `window_size`, содержащие эту точку, и если какое-либо окно из окон содержит по крайней мере `n_neighbors`, которые находятся ближе, чем `distance_coef * std(series)`, до интересующей точки в соответствии с `distance_func`, то интересующая нас точка не является выбросом:

- `window_size` – размер окна (по умолчанию 15),
- `n_neighbors` – минимальное количество соседей точки, необходимое для того, чтобы точка не была объявлена выбросом (по умолчанию 3),
- `distance_coef` – множитель для стандартного отклонения, который формирует пороговое значение расстояния, при котором мы считаем точки близко расположенными друг к другу (по умолчанию 3),
- `distance_func` – функция расстояния, которой по умолчанию является абсолютная разница между двумя значениями.

Согласно способу обнаружения выбросов на основе гистограммы выбросы – это точки, удаление которых приводти *к гистограмме с меньшей ошибкой аппроксимации*, даже если количество бинов меньше, чем количество выбросов.

В библиотеке ETNA процесс обработки выбросов состоит из двух этапов:

- Заменяемы выбросы, обнаруженные с помощью определенного метода, на значения NaN, используя экземпляр соответствующего класса `OutliersTransform`,
- Импутируем значения NaN с помощью класса `TimeSeriesImputerTransform`.

```
best_params = {
    "window_size": 60,
    "alpha": 2.35,
}

outliers_remover = MedianOutliersTransform(
    in_column="target",
    **best_params,
)

outliers_imputer = TimeSeriesImputerTransform(
```

```
    in_column="target",
    strategy="running_mean",
    window=30,
)
outlier_ts.fit_transform([
    outliers_remover,  # заменяем выбросы на NaN
    outliers_imputer,  # импутируем пропуски
])
```

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 `MedianOutliersTransform` для замены выбросов, обнаруженных в соответствии с медианным методом, на значения NaN,
- класс `TimeSeriesImputerTransform` для импутации значений NaN в соответствии с выбранной стратегией,
- класс `LogTransform` для логарифмирования и экспоненцирования переменной (логарифмирование позволяет сгладить негативное влияние выбросов объективной природы, помогает выделить тренд),
- класс `LinearTrendTransform` для:
  - *прогнозирования* тренда,
  - *удаления* тренда из данных и
  - *добавления* тренда к прогнозам (это необходимо для деревьев решений и для ансамбля деревьев решений, *не умеющих экстраполировать*)
- класс `TrendTransform` для добавления тренда в качестве признака,
- класс `SegmentEncoderTransform` для кодирования меток сегментов целочисленными значениями в лексиграфическом порядке (LabelEncoding),
- класс `LagTransform` для генерации лагов,
- класс `DateFlagsTransform` для генерации признаков на основе дат – порядковый дня недели, порядковый номер дня месяца, порядковый номер месяца в году,
- класс `HolidayTransform` для генерации праздников на основе дат,
- класс `MeanTransform` для вычисления скользящего среднего по заданному окну,
- класс `FourierTransform` для генерации компонент ряда Фурье.

```
mult_outliers_remover = MedianOutliersTransform(
    in_column="target",
    window_size=150,
    alpha=15
)
mult_outliers_imputer = TimeSeriesImputerTransform(
    in_column="target",
    strategy="forward_fill",
)
mult_log = LogTransform(in_column="target")
mult_linear_trend = LinearTrendTransform(in_column="target")
mult_trend_feat = TrendTransform(
    in_column = "target",
    change_points_model=RupturesChangePointsModel(
        change_points_model=Binseg(model="ar"),
```

```
        n_bkps=5,
    ),
    out_column="trend_feat",
)
mult_seg = SegmentEncoderTransform()
mult_lags = LagTransfrom(
    in_column="target",
    lags=list(range(7, 211, 7)),
    out_column="lag",
)
mult_mean_7 = MeanTransform(
    in_column="target",
    window=7,
    out_column="mean7",
)
mult_mean14 = MeanTransform(
    in_column="target",
    window=14,
    out_column="mean14",
)
mult_mean30 = MeanTransform(
    in_column="target",
    window=30,
    out_column="mean30",
)
mult_d_flags = DateFlagsTransform(
    day_number_in_week=True,
    day_number_in_month=True,
    is_weekend_in_year=True,
    is_weekend=True,
    out_column="datetime",
)
mult_d_flags2 = DateFlagsTransform(
    is_weekend=True,
    out_column="datetime2",
)
mult_holidays = HolidayTransform(
    iso_code="RUS",
    out_column="RUS_holidays",
)
# для моделирования годовой сезонности
mult_fourier_year = FourierTransform(
    period=365.24,
    order=3,
    out_column="fourier_year",
)
# для моделирования недельной сезонности
mult_fourier_week = FourierTransform(
    period=7,
    order=2,
    out_column="fourier_week",
)

mult_ctsbst_preprocess = [
    mult_outliers_remover,
    mult_outliers_imputer,
    mult_log,
    mult_linear_trend,
    mult_trend_feat,
    mult_seg,
```

```
    mult_lags,
    mult_mean7,
    mult_mean14,
    mult_mean30,
    mult_d_flags,
    mult_d_flags2,
    mult_holidays,
    mult_fourier_year,
    mult_fourier_week,
]

ctbst_model = CatBoostMultiSegemntModel()

# выполняем подбор гиперпараметра -- набора преобразований/признаков с помощью перекрестной пров
    ерки модели CatBoost
etna_cv_optimize(
    ts=mult_ts,
    model=ctbst_model,
    horizon=mult_HORIZON,
    transforms=[
        mult_ctbst_preprocess,
        mult_ctbst_preprocess2,  # без детрендинга и добавления тренда в качестве фичи
    ],
    n_folds=12,
    mode="expand",
    metrics=SMAPE(),
    refit=True,
)
```

Конвейер можно дополнить этапом стандартизации *всех признаков*

```
standardscaler = StandardScalerTransform(in_column=None)
```

### 3.9. Оптимизация гиперпараметров с помощью Optuna

Пишем функцию инициализации логгера

```
def init_logger(
    config: dict,
    project: str = "wandb-sweeps",
    tags: t.Optional[t.List[str]] = ["test", "sweeps"]
):
    tsloggerloggers = []
    wblogger = WandbLogger(
        project=project,
        tags=tags,
        config=config,
    )

    tslogger.add(wblogger)
```

Пишем функцию, загружающую данные

```
def dataloader(
    file_path: Path,
    freq: str = "D",
) -> TSDataset:
    df = pd.read_csv(file_path)
    df = TSDataset.to_dataset(df)
    ts = TSDataset(df, freq)
```

```
    return ts
```

Теперь пишем целевую функцию Optuna

```
def objective(
    trial: optuna.Trial,
    metric_name: str,
    ts: TSDataset,
    horizon: str,
    lags: int,
    seed: int,
):
    set_seed(seed)

    pipeline = Pipeline(
        # подбираем гиперпараметры модели
        model=CatBoostMultiSegmentModel(
            iterations=trial.suggest_int("iterations", 10, 100),
            depth=trail.suggest_int("depth", 1, 12),
        ),
        # и подбираем пространство признаков
        transforms=[
            StandardScalerTransform("target"),
            SegmentEncoderTransform(),
            LagTransform(
                in_column="target",
                lags=list(range(horizon, horizon + trial.suggest_int("lags", 1, lags)))
            )
        ],
        horizon=horizon,
    )

  init_logger(pipeline.to_dict())

  metrics, _, _ = pipeline.backtest(  # CV
      ts=ts, metrics=[MAE(), SMAPE(), Sign(), MSE()]
  )

  return metrics[metric_name].mean()
```

Создаем приложение для запуска

```


@app.command()
def run_optuna(
    horizon: int = 14,
    metric_name: str = "MAE",
    storage: str = "sqlite:///optuna.db",
    study_name: t.Optional[str] = None,
    n_trials: int = 5,
    file_path: Path = "Data/example_dataset.csv",
    direction: str = "minimize",
    freq: str = "D",
    lags: int = 24,
    seed: int = 11,
):
    ts = dataloader(file_path, freq=freq)
    study = optuna.create_study(
        storage=storage,
        study_name=study_name,
```

```
        sampler=optuna.samplers.TPESampler(multivariate=True, group=True),
        load_if_exists=True,
        direction=direction,
    )

    study.optimize(
        parital(objective, metric_name, ts, horizon, lags, seed),  # NB!
        n_trials,
    )

if __name__ == "__main__":
    typer.run_optuna)
```

### 3.10. Задача Райффайезн Банк

Будем логарифмировать и стандартизировать зависимую переменную, из нее будем вычитать тренд, делать LabelEncoding для кодирования меток сегментов, создавать календарные признаки (порядковый номер дня в году/неделе/месяце, порядковый номер недели в месяце/году, порядковый номер месяца в году, порядковый номер сезона, индикатор выходного дня), формировать лаги, скользящие средние и скользящие разности между максимальным и минимальным значениями на основе прологарифмированной зависимой переменной с удаленным трендом.

```
# для логарифмирования зависимой переменной
log = LogTransform(in_column="target")

# для стандартизации зависимой переменной
scaler = StandardScalerTransform(in_column="target")

# для прогнозирования тренда, удаления тренда из данных и добавления тренда к прогнозам
detrend = LinearTrendTransform(in_column="target")

# для кодирования меток сегментов целочисленными значениями в лексикографическом порядке
seg = SegmentEncoderTransform()

# для генерации лагов
lags = LagTransform(
    in_column="target",
    lags=list(range(90, 361, 30)),
    out_column="lag",
)

# для генерации признаков на основе дат
d_flags = DateFlagsTransform(
    day_number_in_year=True,
    day_number_in_week=True,
    week_number_in_month=True,
    week_number_in_year=True,
    month_number_in_year=True,
    season_number=True,
    is_weekend=True,
    out_column="datetime",
)

# для вычисления среднего по заданному окну
mean90 = MeanTransform(
    in_column="target",
    window=90,
```

```
    out_column="mean90",
)

mean180 = MeanTransform(
    in_column="target",
    window=180,
    out_column="mean180",
)

mean270 = MeanTransform(
    in_column="target",
    window=270,
    out_column="mean270",
)

# для вычисления разности по заданному окну
minmax270 = MinMaxDifferenceTransform(
    in_column="target",
    window=270,
    out_column="minmax270",
)
```

Для большей надежности оценим качество модели с помощью перекрестной проверки

```
ctbst_model = CatBoostMultiSegmentModel(
    iterations=500,
    learning_rate=0.25,
    depth=5,
)

preprocess = [log, scaler, detrend, seg, lags, d_flags, mean90, mean180, mean270, minmax270]

pipe = Pipeline(
    model=ctbst_model,
    transforms=preprocess,
    horzon=HORIZON,
)

metrics_df, forecast_df, fold_info_df = pipe.backtest(
    mode="expand",
    n_folds=4,
    ts=ts,
    metrics=[smape],
    aggregate_metrics=True,
    joblib_params=dict(backend="loky")
)

# среднее значение по сегментам
mean_smape = metrics_df["SMAPE"].mean()
```

Если качество модели устраивает, обучаем на всем историческом наборе [2, стр. 429].

### 3.10.1. Оптимизация гиперпараметров модели с выделением отдельной тестовой выборки

Выполним оптимизацию гиперпараметров с выделением отдельной тестовой выборки, на которой будем тестировать модель с найденными наилучшими гиперпараметрами.

Пишем целевую функцию Optuna. Мы будем настраиваеть верхнюю границу порядка лага, шаг изменения порядка лага и ширину окна для скользящего среднего. Таким образом, настра-

иваем *не только гиперпараметры модели* машинного обучения, но и *гиперпараметры, отвечающие за конструирование признаков*. Для оценки качества модели используется перекрестная проверка расширяющимся / скользящим окном

```
def objective(trial, ts, horizon, metrics, metric_name, n_folds, depth, lag_upper_bound,
    lag_step, window_size, print_configurations, print_metrics, seed):
    set_seed(seed)

    # для логарифмирования зависимой переменной
    log = LogTransform(in_column="target")

    # для стандартизации зависимой переменной
    scaler = StandardScalerTransform("target")

    # для прогнозирования тренда, удаления тренда из данных и добавления тренда к прогнозам
    detrend = LinearTrendTransform(in_column="target")

    # для кодирования меток сегментов целочисленными значениями
    seg = SegmentEncoderTransform()

    # для генерации лагов
    lag = LagTransform(
        in_column="target",
        lags=list(
            range(
                horizon,
                trial.suggest_int("lag_upper_bound", 100, lag_upper_bound),
                trail.suggest_int("lag_step", 5, lag_step)
            )
        )
    )

    # для генерации скользящих средних
    mean = MeanTransform(
        in_column="target",
        window=trail.suggest_int("window_size", horizon, window_size)
    )

    # для генерации признаков на основе дат
    d_flags = DateFlagsTransform(
        day_number_int_week=True,
        day_number_int_month=True,
        month_number_in_year=True,
        season_number=True,
        is_weekend=True,
        out_column="datetime",
    )

    transforms = [log, scaler, detrend, seg, lag, mean, d_flags]

    pipeline = Pipeline(
        model = CatBoostMultiSegmentModel(
            iterations=500,
            depth=trial.suggest_int("depth", 3, depth)
        ),
        transforms=transforms,
        horizon=horizon,
    )
```

```
    df_metrics, _, _ = pipeline.backtest(ts=ts, metrics=[metrics], n_folds=n_folds,
    aggregate_metrics=True)

    return df_metrics[metric_name].mean()
```

Пишем функцию, создающую обучающую и тестовую наборы. На вход она принимает исторический набор эндогенных переменных и исторический набор экзогенных переменных. *Эндогенные* переменные – это собственно прогнозируемые временные ряды, а *экзогенные* переменные – любые дополнительные данные, которые помогают нам прогнозировать интересующие нас ряды.

Здесь под эндогенными переменными подразумеваются данные продаж, а под экзогенными переменными – данные рекламной активности. Кроме того, помним, что набор *экзогенных* переменных в формате ETNA *охватывает историческую часть и прогнозируемый период* [2, стр. 435].

```
def train_test_dataloader(
    internal_hist_path,
    external_hist_pah,
    horizon=90,
    freq="D",
    verbose=False,
):
    hist_sales = pd.read_csv(
        internal_hist_path,
        index_col=["date"],
        parse_dates=["date"],
    )

    hist_advert = pd.read_csv(
        external_hist_path,
        parse_dates=["date"],
    )

    hist_sales.interpolate(method="linear")

    train_sales = hist_sales.iloc[:-horizon]
    train_sales = train_sales[train_sales.index >= "2013-01-01"]
    hist_advert = hist_advert[hist_advert["date"] >= "2013-01-01"].reset_index(drop=True)

    train_sales["timestamp"] = train_sales.index
    train_sales = train_sales.reset_index(drop=True)

    train_sales_melt = train_sales.melt(
        id_vars="timestamp",
        var_name="segment",
        value_name="target",
    )

    train_ts_format = TSDataset.to_dataset(train_sales_melt)
    hist_advert.rename(columns={"date": "timestamp"}, inplace=True)
    hist_advert_melt = hist_advert.melt(
        id_vars="timestamp",
        var_name="segment",
        value_name="advert",
    )

    hist_advert_melt["quater"] = hist_advert_metl["timestamp"].dt.quater
    hist_advert_melt["quater_start"] = hist_advert_melt["timestamp"].dt.is_quater_start
    hist_advert_melt["quater_end"] = hist_advert_metl["timestamp"].dt.is_quater_end
```

```
    hist_advert_melt["month_name"] = hist_advert_melt["timestamp"].dt.strftime("%b")
    hist_advert_melt["month_name"] = hist_advert_melt["month_name"].astype("category")

    # переводим набор с экзогенными переменными (рекламная активность, календарные признаки) в ф
    ормат ETNA
    hist_regressors_ts_format = TSDataset.to_dataset(hist_advert_melt)

    # создаем итоговый обучающий набор как объект TSDataset
    train_ts = TSDataset(
        train_ts_format, freq=freq,
        df_exog=hist_regressors_ts_format, known_future="all"
    )

    # формируем тестовый набор
    test_sales = hist_sales.iloc[-horizon:]
    test_sales["timestamp"] = test_sales.index
    test_sales = test_sales.reset_index(drop=True)

    test_sales_melt = test_sales.metl(
        id_vars="timestamp",
        var_name="segment",
        value_name="target",
    )

    test_ts_format = TSDataset.to_dataset(test_sales_melt)
    test_ts = TSDataset(test_ts_format, freq=freq)

    return train_ts, test_ts
```

Можно выполнить оптимизацию гиперпараметров *на всем историческом наборе без выделения тестовой выборки*. Хотя данную схему часто используют в условиях нехватки данных для полноценного тестирования, она является менее предпочтительной стратегией проверки, поскольку настройка гиперпараметров и выбор наилучшей модели происходят с помощью одних и тех же тестовых выборок перекрестной проверки [2, стр. 468].

Если качество модели нас устраивает, то обучаем модель с найденным значением гиперпараметра `depth` *на всех исторических данных*.

## 3.11. Задача Store Sales – Time Series Forecasting

### 3.11.1. Добавление экзогенных переменных – регрессоров

В терминологии ETNA зависимая переменная *Объем продаж* будет *эндогенной* переменной, а все признаки, кроме признака даты, будут *экзогенными* переменными.

Экзогенные переменные, значения которых известны в будущем, называют *регрессорами*.

Создаем объединенный набор, указав с помощью значения `"all"` параметра `known_future`, что значение всех экзогенных переменных известны в будущем.

```
ts = TSDataset(
    df=TSDataset.to_dataset(train),
    df_exog=TSDataset.to_dataset(train_exog),
    freq="D",
    known_future="all",
)
```

Теперь создаем список преобразований

```
transforms = [
    MinMaxScaler(in_column="target"),
    LagTransform(
        in_column="target",
        lags=list(range(HORIZON, HORIZON + num_lags)),
        out_column="lag",
    ),
    DateFlagsTransform(
        day_number_in_week=True,
        day_number_in_month=True,
        is_weekend=True,
        out_column="datetime",
    ),
    MeanTransform(
        in_column="target",
        window=32,
        out_column="mean32",
    )
]
```

В качестве модели возьмем модель градиентного бустинга LightGBM (гиперпараметры были подобраны, исходя из априорных знаний). Для более надежной оценки качества модели воспользуемся перекрестной проверкой скользящим окном

```
model = LGBMMultiSegmentModel(
    n_estimators=400,
    learning_rate=0.08,
    min_data_in_leaf=80,
    subsample=0.6,
)

pipeline = Pipeline(
    model=model,
    horizon=HORIZON,
    transforms=transforms,
)

# запускаем перекрестную проверку расширяющимся окном
metrics_df, forecast_df, fold_info_df = pipeline.backtest(
    ts=ts, metrics=[MSE()], n_folds=3,
)
```

Если качество модели нас устраивает, то обучаем модель *на всех исторических данных!!!*.

```
model = LGBMultiSegmentModel(
    n_estimators=400,
    learning_rate=0.08,
    min_data_in_leaf=80,
    subsample=0.6,
)

pipeline = Pipeline(
    model=model,
    horizon=HORIZON,
    transforms=transforms,
)

# обучаем на ВСЕМ ИСТОРИЧЕСКОМ НАБОРЕ
pipeline.fit(ts=ts)
```

### 3.11.2. Добавление экзогенных переменных – регрессоров и экзогенных переменных – не регрессоров

Используем как экзогенные переменные – регрессоры («идентификационный номер магазина», «продуктовая группа»), так и зкзогенную переменную – не регрессор (переменная «ежедневная цена не нефть» из набора oil.csv). Экзогенная переменная «ежедневная цена на нефть» не является регрессором, поскольку мы не знаем, какой будет цена на нефть в будущем.

Выполняем переиндексацию, потому что данные о ценах на нефть не содержат выходных дней, а наши данные о продажах включают выходные дни, неизбежно появятся пропуски, которые нужно будет импутировать.

Теперь к набору с экзогенными переменными – регрессорами «идентификационный номер магазина» и «продуктовая группа» присоединяется набор с экзогенной переменной – не регрессором «ежедневная цена на нефть».

Для прогнозируемого периода значения регрессоров известны, а у не регрессоров стоят значения NaN.

Теперь на основе полученных наборов создаем объединенный набор, указав с помощью параметра `known_future` экзогенные переменные, значения которых известны в будущем.

```
ts = TSDataset(
    df=TSDataset.to_dataset(train),
    df_exog=TSDataset.to_dataset(train_exog),
    freq="D", known_future=["regressor_store_nbr", "regressor_family"],
)
```

Создаем список преобразований/признаков. Список процедур включает стандартизацию зависимой переменной, создание лагов, скользящего среднего и некоторых календарных праздников. Кроме того, мы добавили импутацию пропусков в переменной dcoilwtcio, генерацию переменной dcoilwtico с лагом 1, вычисление скользящего среднего на основе этой переменной с последующим удалением исходной переменной dcoilwtico и переменной dcoilwtico, взятой с лагом 1.

```
num_lags = 50

transforms = [
    MinMaxScalerTransform(in_column="target"),
    LagTransform(
        in_column="target",
        lags=list(range(HORIZON, HORIZON + num_lags)),
        out_column="lag"
    ),
    DateFlagsTransform(
        day_number_in_week=True,
        day_number_in_month=True,
        is_weekend=True,
        out_column="datetime",
    ),
    MeanTransform(
        in_column="target",
        window=32,
        out_column="mean32",
    ),
    TimeSeriesImputerTransform(
        in_column="dcoilwtico",
        strategy="forward_fill",
        window=16,
    ),
```

```
    LagTransform(
        in_column="dcoilwtico",
        lags=[1],
        out_column="dcoilwtico_lag",
    ),
    MeanTransform(
        in_column="dcoilwtico_lag_1",
        window=16,
        out_column="dcoilwtico_mean16_on_lag_1",
    ),
    FilterFeaturesTransform(
        exclude=["dcoilwtico", "dcoilwtico_lag_1"]
    )
]
```

### 3.12. Отбор признаков

Библиотека ETNA позволяет выполнить отбор признаков.

```
select_ts = TSDataset(
    df=df,
    freq="D",
    df_exog=regressor_df,
    known_future="all",
)
```

Создаем список преобразований/признаков без отобора прзнаков

```
scaler = StandardSalerTransform(in_column="target")

lags = LagTransform(
    in_column="target",
    lags=list(range(90, 360, 20)),
    out_column="lag",
)

# для моделирования годовой сезонности с периодом 365.24
fourier_year = FourierTransform(
    period=365.24,
    order=3,
    out_column="fourier_year",
)

mean90 = MeanTransform(
    in_column="target",
    window=90,
    out_column="mean90",
)

mean180 = MeanTransform(
  in_column="target",
  window=180,
  out_column="mean180",
)

mean270 = MeanTransform(
  in_column="target",
  window=270,
  out_column="mean270",
)
```

```
mean360 = MeanTransform(
  in_column="target",
  window=360,
  out_column="mean360",
)

d_flags = DateFlagsTransform(
    day_number_in_year=True,
    ...
)

one_hot = OneHotEncoderTransform(
    in_column="month",
    out_column="one_hot",
)

drop = FilterFeaturesTransform(exclude=["month"])

preprocess = [scaler, lags, fourier_year, mean90, ..., d_flags, one_hot, drop]
```

Теперь формируем список преобразований/признаков с отбором признаков

```
forest = RandomForestRegressor(
    max_depth=4,
    n_estimators=10,
    random_state=42,
)
```

Теперь создаем экземпляр класса `TreeFeatureSelectionTransform` для отбора признаков с помощью важностей признаков, вычисленных моделью случайного леса

```
forest_select = TreeFeatureSelectionTransform(top_k=20, model=forest)

preprocess_select = preprocess + [forest_select]
```

### 3.12.1. Кластеризация временных рядов

Библиотека ETNA позволяет выполнить кластеризацию временных рядов. Для объединения рядов в кластеры сначала нужно задать функцию расстояния между наблюдениями. Наиболее часто используемой мерой расстояния между наблюдениями рядов является евклидово расстояние и расстояние на основе динамической трансформации временной шкалы (dynamic time warping – DTW).

В библиотеке ETNA процедура вычисления расстояний реализована в классах `EuclideanDistrance` и `DTWDistance`, их можно вычислить для двух объектов `pd.Series`, проиндексированных по временным меткам.

```
x1 = pd.Series(
    data=[0, 0, 1, 2, 3, 4, 5, 6, 7, 8],
    index=pd.date_range("2020-01-01", periods=10)
)

x2 = pd.Series(
    data=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    index=pd.date_range("2020-01-02", periods=10)
)
```

Расчет расстояний в случае разных временных меток в классах `EuclideanDistance` и `DTWDistance` может выполняться в двух режимах:

- `trim_series=True` – вычисляем расстояние только по общей, совпадающей части рядов (общая часть определяется по индексу временных меток), несовпадающие части рядов отсекаем (выполняется тримминг), отсюда и название параметра,
- `trim_seires=False` – вычисляем расстояние по всему ряду, игнорируя временные метки (эквивалентно отбрасыванию индекса временных меток и использованию индекса целочисленных значений, как в обычном массиве).

В библиотеке ETNA в классах `EuclideanClustering` и `DTWClustering` реализована иерархическая кластеризация.

```
from etna.clustering EuclideanClustering, DTWClustering

model = EuclideanClustering()
model.build_distance_matrix(ts=ts)
```

Пример иерархической кластеризации на основе DTW-расстояний

```
dtw_model = DTWClustering()
# вычисляем матрицу DTW-расстояний
dtw_model.build_distance_matrix(ts=ts)
dtw_model.build_clustering_algo(n_clusters=3, linkage="average")

segment2cluster = dtw_model.fit_predict()
segment2cluster
# output
{
    "market_1": 0,
    "market_2": 2,
    ...
}
```

### 3.12.2. Классификация временных рядов

Библиотека ETNA позволяет выполнить классификацию временных рядов. Каждое наблюдение – это временной ряд.

Необработанные значения временных рядов обычно не являются лучшими признаками для классификатора. Длина ряда, как правило, намного больше, чем количество наблюдений в наборе данных, и в этом случае классификаторы работают плохо [2, стр. 592].

Библиотека предлагает два способа извлечения признаков, способных работать с временными рядами различной длины. В первом способе используется класс `TSFreshFeatureExtractor`, извлекающий признаки с помощью функции `extract_featuers()` библиотеки tsfresh.

```
from tsfresh.feature_extractoin.settings import MinimalFCParameters
from etna.experimental.classification.feature_extraction import TSFreshFeatureExtractor

tsfresh_feature_extractor = TSFreshFeatureEXtractor(
    default_fc_parameters=MinimalFCParameters(),
    fill_na_value=-100,
)
```

Во втором способе используется класс `WEASELFeatureExtractor`, извлекающий признаки с помощью алгоритма WEASEL https://arxiv.org/abs/1701.07681. Алгоритм имеет большой список параметров, наиболее важными из которых являются:

- `padding_value` – значение для заполнения ряда в тестовом наборе, так чтобы он соответствовал кратчайшему ряду в обучающем наборе,
- `word_size`, `n_bins` – размер слова и размер алфавита для аппроксимации ряда (сильно влияет на качество),
- `window_sizes` – раземеры скользящих окон,
- `window_steps` – шаги окон,
- `ch2_threshold` – порог отбора признаков (чем выше порог, тем меньше признаков отбирается).

```
from etna.experimental.classification import TimeSeriesBinaryClassifier
from sklearn.linear_model import LogisticRegression
from etna.experimental.classification.feature_extraction import WEASELFeatureExtractor

weasel_feature_extractor = WEASELFeatureExtractor(
    padding_value=-10,
    word_size=4,
    n_bins=4,
    window_size=[0.2, 0.3, 0.5, 0.7, 0.9],
    window_steps=[0.1, 0.15, 0.25, 0.35, 0.45],
    chi2_threshold=2
)

model = LogisticRegression(max_iter=1000)
clf = TimeSeriesBinaryClassifier(
    feature_extractor=tsfresh_feature_extractor,
    classifier=model,
)

mask = np.zeros(len(X_train))
for fold_idx, (train_index, test_index) in enumerate(KFold(n_splits=5).split(X_train)):
    mask[test_index] = fold_idx

metrics = clf.masked_crossval_score(
    x=X_train,
    y=y_train,
    mask=mask,
)
```

### 3.13. Анализ прогнозируемости

Анализ прогнозируемости, производимый с помощью класса `PredictabilityAnalyzer`, помогает выполнить некую предварительную проверку на наборе данных. Он может помочь выявить «плохие» сегменты, которые следует обрабатывать отдельно.

*Авторегрессионные компоненты* полезны для выявления оставшихся трендов, которые не объясняются сезонностью, ростом, событиями и точками изменения тренда [2, стр. 646].

Мы требуем, чтобы наборы данных содержали *обучающие данные не менее чем за два года*, чтобы модели могли точно оценивать *годовые сезонности* [2, стр. 649].

Замечания:

- Качество прогнозов моделей измеряется в течение года, чтобы гарантировать, что кумулятивно тестовые выборки представляют собой реальные данные по своим временным характеристикам, например с точки зрения сезонности, праздников и т.д.

- Тестовые выборки полностью рандомизированы с точки зрения временных признаков. Если, работая с ежедневными данными, вы установите «periods between splits» любое число, кратное 7, это приведет к тому, обучающие и тестовые наборы будут всегда заканчиваться в один и тот же день недели. Такое отсутствие рандомизации приведет к смещенной оценке качества прогноза. Аналогично установка значения, кратного 30, приведет к той же проблеме для дня месяца. Промежуток в 25 дней гарантирует отсутствие подобных мешающих факторов.

Признаки авторегрессии (лаги) очень полезны в *краткосрочном* прогнозировании, но могут быть опасны для использования в *долгосрочном* прогнозировании [2, стр. 666].

# 4. Библиотека sktime

## 4.1. Сжатая сводка

Библиотека sktime поддерживает построение сложных прогнозных моделей, состоящих из более простых моделей:

- редукция – превращение временного ряда в таблицы (массив признаков и массив меток) с помощью модели редукции; на основе этих таблиц можно строить регрессионную модель из библиотеки sktime или из библиотеки scikit-learn,
- настройка гиперпараметров модели редукции и/или прогнозной модели (например, настраиваем размер окна и/или количество ближайших соседей),
- построение конвейера, в рамках которого объединяем модели преобрвазования данных с прогнозной моделью (например, устранение тренда и сезонности с помощью моделей удаления тренда и сезонности, а затем построение прогнозной модели типа градиентного бустинга),
- автоматический выбор модели, при этом можно перебирать не только гиперпараметры, но и целые прогнозные модели,
- автоматическая настройка моделей преобразования данных внутри конвейера,
- ансамблирование прогнозных моделей путем простого объединения и с помощью весов.

Прогнозирование временного ряда часто решается с помощью регрессионной модели. Этот подход иногда называют *редукцией*, потому что мы сводим задачу прогнозирования к более простой, но связанной с ней задаче табличной регрессии. Это позволяет применить к задаче прогнозирования любой алгоритм ререссии.

Приведение к регрессии работает следующим образом: сначала нам нужно преобразовать данные в требуемый табличный формат. Мы делаем это с помощью модели редукции, разбив обучающий временной ряд на окна фиксированной длины и состыковав друг с другом. Так мы получаем массив признаков. Каждое наблюдение, которое следует за последним наблюдением каждого окна, становится значением массива меток. Затем мы можем использовать эти массивы для обучения любой модели машинного обучения с учителем, решающей задачу регрессии [2, стр. 705].

С помощью класса `ForecastingGridSearchCV` мы можем настраивать гиперпараметры модели редукции и прогнозной модели-регрессора. Сейчас с его помощью мы попробуем настроить длину окна для модели редукции и количество деревьев для модели случайного леса – прогнозной модели-регрессора.

```
from sktime.forecasting.model_selection import ForecastingGridSeachCV
from sklearn.ensemble import RandomForecastRegressor
```

```
regressor = RandomForestRegressor(n_estimators=50, random_state=42)
forecaster = make_reduction(regressor, window_length=15, strategy="recursive")
param_grid = {
    "window_length": [7, 12, 15],
    "estimator__n_estimators": [50, 100, 150],
}

cv = SlidingWindowSplitter(
    initial_window=int(len(y_train) * 0.8),
    window_length=20,
)

# мы ищем оптимальное значение гиперпараметров модели редукции и модели-регрессора, которые даду
    т наименьшее усредненное значение sMAPE по результатам проверки скользящим окном
gscv = ForecastingGridSearchCV(
    forecaster,
    strategy="refit",
    cv=cv,
    param_grid=param_grid,
    scoring=mape,
)
gscv.fit(y_train)
y_pred = gscv.predict(fh)

# смотрим наилучшие гиперпараметры
gscv.best_params_
```

Вышеописанный метод редукции не принимал во внимание сезонность и тренды, но мы можем легко удалить тренд и сезонность в данных. *Удаление тренда и сезонности может быть полезно при построении моделей на основе деревьев, учитывая их неспособность к экстраполяции* [2, стр. 709].

```
from sktime.forecating.trend import PolynomialTrendForecaster
from sktime.transformations.series.detrend import Deseasonalizer, Detrender
```

Сналача надо с помощью класса `PolynomialTrendForecaster` создать модель, предсказывающую тренд, а затем с помощью класса `Detrender` создать модель-детрендер, в которую мы передаем модель, предсказывающую тренд. Прогнозами у нас будут значения предсказанного тренда, а при вычитании тренда из фактических значений ряда (или деления фактических значений ряда на тренд) мы получаем остатки.

```
# модель, предсказывающая тренд
forecaster = PolynomialTrendForecaster(degree=1)
transformer = Detrender(forecaster=forecaster)
# прогнозируем и удаляем тренд, в итоге получаем остатки
yt = transformer.fit_transform(y_train)
forcaster = PolynomialTrendForecaster(degree=1)
fh_ins = -np.arange(len(y_train))
y_pred = forecaster.fit(y_train).predict(fh=fh_ins)
```

Можно пойти еще дальше. Мы можем создать конвейер, в рамках которого удалим тренд и сезонность, а затем построим композитную прогнозную модель, которая, например, будет состоять из модели редукции на основе рекурсивной стратегии многошагового прогнозирования и модели случайного леса.

```
forecaster = TransformedTargetForecaster([
    ("deseasonalize", Deseasonalizer(model="multiplicative", sp=12)),
    ("detrend", Detrender(forecaster=PolynomialTrendForecaster(degree=1))),
    ("forecast", make_reduction(regressor, scitype="tabular-regressor", window_length=15,
    strategy="recursive"))]
)

# обучаем модель, по сути, удаляем тренд, сезонность и обучаем композитную модель
forcaster.fit(y_train)
y_pred = forecaster.predict(fh)
```

Класс `MultiplexForecaster` в связке с классом `ForecastingGridSearchCV` позволяет автоматически отобрать наилучшую прогнозую модель. Сам по себе `MultiplexForecaster` принимает на вход разные прогнозные модели sktime. `ForecastingGridSearchCV` позволяет `MultiplexForecaster` переключаться между разными прогнозными моделями, чтобы найти модель, дающую наилучшее качество для имеющихся обучающих данных.

```
from sktime.forecasting.compose import MultiplexForecaster

forecaster = MultiplexForecaster(
    forecasters=[
        ("arima", ARIMA(
            order=(1, 1, 0),
            seasonality_order=(0, 1, 0, 12),
            suppress_warnings=True)),
        ("ets", ExponentialSmoothing(trend="add", sp=12))]
)

cv = SlidingWindowSpliter(
    initial_window=int(len(y_train) * 0.5),
    window_length=30,
)
forecaster_param_grid = {"selected_forecaster": ["arima", "ets"]}
gscv = ForecastingGridSearchCV(
    forecaster,
    cv=cv,
    param_grid=forecaster_param_grid,
)
gscv.fit(y_train)
y_pred = gscv.predict(fh)
```

С помощью класса `EnsembleForecaster` можно выполнить простое ансамблирование прогнозных моделей. С помощью параметра `forecasters` задаем список 2-кортежей, первый элемент кортежа – название этапа, второй элемент кортежа – экземпляр класса, в котором реализована прогнозная модель sktime. С помощью параметра `aggfunc` задаем способ агрегирования прогнозов моделей, доступны значения `"mean"`, `"median"`, `"min"` и `"max"`.

```
from sktime.forecasting.compose import EnsembleForecaster

holt = ExponentialSmoothing(trend="add", seasonal="mul", sp=12)
arima = ARIMA(order=(1, 1, 0), seasonal_order=(0, 1, 0, 12), suppress_warning=True)
forecaster = EnsembleForecaster([("holt", holt), ("arima", arima)])
forecaster.fit(y_train)
y_pred = forecaster.predict(fh)
```

Библиотека sktime выполняет автоматический отбор компонентов конвейера внутри него самого с помощью класса `OptionalPassthrough`.

Например, мы хотим выяснить, даст ли преимущество для нашей прогнозной модели стандартизация – класс `StandardScaler` из sklearn или нет, и передаем его классу `OptionalPassthrough`.

В следующем примере мы попробуем выполнить / не выполнить удаление сезонности, выполнить / не выполнить стандартизацию (с центрированием / без центрирования) в сочетании с тремя разными стратегиями прогнозирования для модели наивного прогноза внутри конвейера – таким образом получается 24 комбинации.

```
from sktime.transformations.compose import OptionalPassthrough
from sktime.transformations.series.adapt import TabularToSeriesAdaptor
from sklearn.preprocessing import StandardScaler

# создаем конвейер из трансформеров и прогнозной модели
pipe = TransformedTargetForecaster(
    steps=[
        ("deseasonalizer", OptionalPassthrough(Deseasonalizer())),
        ("scaler", OptionalPassthrough(
            TabulerToSeriesAdaptor(StandardScaler()))),
        ("forecast", NaiveForecaster())
    ]
)

cv = SlidingWindowSplitter(
    initial_window=60,
    window_length=24,
    start_with_window=True,
    step_length=24,
)

parm_grid = {
    "deseasonalizer__passthrough": [True, False],
    "scaler__transformer__transformer__with_mean": [True, False],
    "scaler__passthrough": [True, False],
    "forecaster__strategy": ["drift", "mean", "last"],
}

gscv = ForecastingGridSearchCV(
    forecaster=pipe,
    param_grid=param_grid,
    cv=cv,
    n_jobs=-1,
)

gscv.fit(y_train)
y_pred = gscv.predict(fh)
```

Ключевая идея, лежащая в основе библиотеки sktime, – *редукция*, когда мы сводим задачу прогнозирования временного ряда к более простой табличной задаче регрессии. Это позволяет применить к задаче прогнозирования временного ряда любой алгоритм регрессии [2, стр. 725].

Библиотека sktime поддерживает следующие стратегии многошагового прогнозирования:

- прямая стратегия многошагового прогнозирования,
- рекурсивная стратегия многошагового прогнозирования: здесь принцип *скользящего окна* заключается в том, что мы каждый раз добавляем в массив `last` прогноз модели, сдвигаем значения на один шаг вперед и формируем новый массив признаков `X`; количество итераций определяется длиной горизонта; в данном случае у нас будет 12 итераций, мы получим 12 прогнозов с помощью *одной и той же модели*,

- гибридная стратегия многошагового прогнозирования: здесь принцип *расширяющего окна* заключается в том, что мы каждый раз добавляем в массив `X_full` прогноз модели и формируем новый массив `X`; количество итераций определяется длиной горизонта прогнозирования (`fh`),
- стратегия со множеством выходов.

## Список литературы


1. *Маккинли У.* Python и анализ данных, 2015. – 482 с.
2. *Груздев А.* Прогнозирование временных рядов с помощью Facebook, Prophet, ETNA, sktime и LinkedIn Greykite: Строим, настраиваем, улучшаем модели прогнозирования временных рядов с помощью специальных библиотек. – М.: ДМК Пресс, 2023. – 780 с.