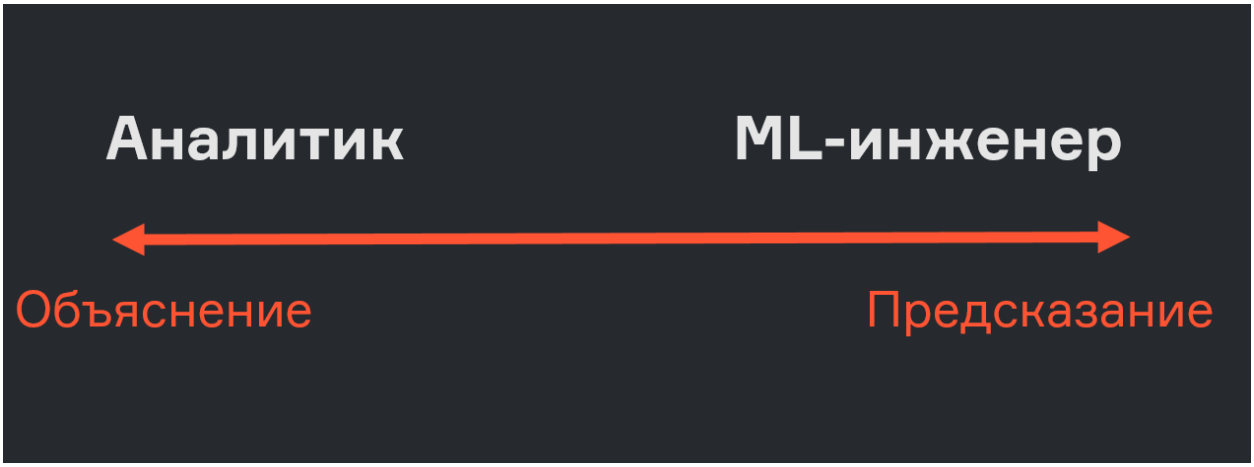
> ПРОГНОЗИРОВАНИЕ МЕТРИК

> Смысл прогнозирования

Как вы могли заметить на протяжении всего курса, аналитики на рабочем месте делают много вещей. Мы и дашборды готовим, и в данных копаемся, и A/B-тесты проводим… Но почему мы это делаем?

Само собой, чтобы улучшить какие-то характеристики нашего продукта и максимизировать прибыль! Но как мы это делаем? В основном, наша задача заключается в анализе данных с целью предоставления **прогнозов** и **предсказаний**.

При всей кажущейся близости между объяснением и предсказанием, существует странное расхождение между ними. И это расхождение ярко проявляется в ролях **аналитика** и **ML-инженера**:



Главная задача аналитика – **находить инсайты в данных**. Для него важнее объяснение.

Главная задача ML-инженера – создать **эффективную модель**. Для него важнее предсказание.

Известно, что самые лучшие предсказательные модели чаще всего являются "черными коробками" – никто не может понять, как модель пришла к такому выводу. И наоборот, часто методы, дающие нам наиболее легко интерпретируемую информацию, крайне далеки от сложности реального мира.

И в попытке взять лучшее из обоих миров рождается **предиктивная аналитика**.

Предсказание или прогноз?

На самом деле это не равнозначные понятия! Предсказанием, например, может быть результат линейной регрессии веса по росту. Прогноз же — это всегда предсказание будущего, и если мы рассматриваем этот вопрос формально, то не сможем избежать *временных рядов*.

Есть ряд компаний, которые решили максимально упростить и стандартизовать процесс построения прогнозных моделей. Мы будем пользоваться наработками компании Uber в виде библиотеки Orbit.

Что ещё почитать?

- Список алгоритмов на разных языках для анализа временных рядов — далеко не полный, но от этого не менее полезный! Основной акцент здесь сделан на детекцию аномалий. В одном из последующих уроков вы будете заниматься именно этой задачей. И хотя для сдачи работы вам придется написать свой алгоритм с нуля, вы можете попробовать готовые решения локально :)
- Книжка на Питоне для прогнозирования временных рядов — ссылка ведёт на её открытую электронную версию. Может быть полезна для общего погружения в теорию временных

рядов и других методов их прогнозирования.

- Если вы не чураетесь контента на R, то вот горячая рекомендация – Forecasting: Principles and Practice. Содержит множество визуализаций, практических примеров и полезных советов, при этом бесплатная.
- Также классика на R — Анализ временных рядов с помощью R авторства Мастицкого. Она может оказаться полезной даже для тех, кто никогда не использовал или не планирует использовать R, поскольку представляет собой хороший обзор возможностей анализа временных рядов.

Отдельно выделим следующие материалы:

- Сигнал и шум. Почему одни прогнозы сбываются, а другие – нет – достаточно известный научпоп о прогнозировании и связанных с ним идеях. Она может быть очень полезна для расширения общего кругозора и для того, чтобы приобрести правильную оптику в отношении прогнозов.
- Choosing prediction over explanation in psychology: Lessons from machine learning – достаточно известная статья в кругах естественнонаучной ветки психологии. В ней рассматривается вышеупомянутое расхождение между объяснением и предсказанием в контексте научных исследований. Эта статья может служить отличным введением в основные концепции машинного обучения.
- Убийственные большие данные. Как математика превратилась в оружие массового поражения – а с этим можно ознакомиться, если интересует этический аспект предсказательных моделей в целом.

> Orbit

- Версия, установленная на сервере - 1.1.4.2
- Документация

- Github
- Оригинальная статья, описывающая цель и функционал библиотеки

Всего Orbit поддерживает четыре типа моделей:

- Модель с экспоненциальным сглаживанием (`ETS`) — простейшая прогнозная модель, дающая больший вес недавним значениям и меньший — более старым. Может оценивать только общий уровень и сезонность — тренд и дополнительные регрессоры модель не поддерживает. В Orbit выполняет скорее тренировочную функцию, так как остальные модели круче.
- Модель с локальным и глобальным трендами (`LGT`) — допускает дополнительные регрессоры и тренд. Моделирует как локальные тренды (кратковременные изменения), так и глобальный (характерный для всего набора данных целиком). Работает только с положительными данными.
- Модель с приглушением локального тренда (`DLT`) — похожа на предыдущую, но устроена чуть иначе и допускает отрицательные значения. Также есть параметр "приглушения" (damping), который уменьшает значения локальных трендов и таким образом даёт больше веса глобальным изменениям.
- Ядерная модель с варьирующей во времени регрессией (`KTR`) — более сложная и экспериментальная модель. Позволяет указывать множественную сезонность (например, дневную и недельную), более гибко работает с регрессорами, имеет меньшие вычислительные затраты. Есть более простая версия `KTRLite` с меньшим числом настраиваемых параметров и простейшим алгоритмом оценки. Подробнее об этой модели тут.

Наиболее важные базовые параметры:

- `response_col` — как называется колонка с метрикой, которую мы прогнозируем. По умолчанию `y`, если называется иначе — нужно указать.
- `date_col` — аналогично предыдущему, но это колонка со временем. По умолчанию `ds`.
- `estimator` — алгоритм, по которому считаются параметры модели.
- `seasonality` — величина сезонности. Мы говорим о сезонности в тех случаях, когда наш временной ряд периодически повторяет сам себя в течение какого-то промежутка времени — например, продажи мороженого могут систематически расти летом и падать зимой каждый год. Всегда обращайте внимание на то, в каких единицах времени выражен ваш временной ряд. Так, если нас интересует недельная сезонность и у нас данные с шагом в день, то у этого параметра будет величина `7` (так как в неделе семь дней). Если же у нас шаг в час, то этот же параметр должен иметь значение `7*24` (семь дней, в каждом из которых 24 часа).

- `_sm_input` — степень сглаживания некоторого параметра модели. Чем больше сглаживание, тем меньше деталей в модели (не ловит тонкие изменения), но тем меньше шума.
- `global_trend_option` — какого рода глобальный тренд будет использоваться. Возможные варианты: `"linear"` (линейный тренд — просто прямая линия), `"loglinear"` (логлинейный — очень быстрый рост/падение в начале, после которого тренд тормозится и потенциально выходит на плато), `"logistic"` (тренд, который не может выходить ниже или выше определённых границ, задаваемых параметрами `global_floor` и `global_cap`) и `"flat"` (”плоский” тренд — фактически отсутствие глобального тренда).
- `prediction_percentiles` — границы доверительного интервала нашего прогноза (он же прогнозный интервал). По умолчанию `[0.05, 0.95]` — т.е. 90%-ый прогнозный интервал.
- `damped_factor` — фактор “приглушения”, чем он ниже, тем сильнее убирается тренд. По умолчанию 0.8.
- `_knot_*` — параметры “узлов”, точек “перелома”, после которых параметр меняется.

> Простейший процесс построения модели

Возьмём пример из лекции. Импортируем всё, что нам нужно:

```
import orbit #общий пакет
from orbit.models import DLT #один из вариантов модели
from orbit.diagnostics.plot import plot_predicted_data, plot_predicted_components #для рисования прогноза
```

Определяем параметры модели:

```
dlt = DLT(response_col="CTR", #название колонки с метрикой
          date_col="hour_time", #название колонки с датами-временем
          seasonality=24, #длина периода сезонности
          estimator="stan-map", #алгоритм оценки
          n_bootstrap_draws=1000) #количество сэмплов бутстрапа для доверительного интервала
```

Обучаем модель:

```
dlt.fit(activity) #тут мы указываем данные, на которых строится модель
```

Создаём датафрейм, для которого будем делать прогноз (фактически датафрейм с датами и ковариатами, если такие есть):

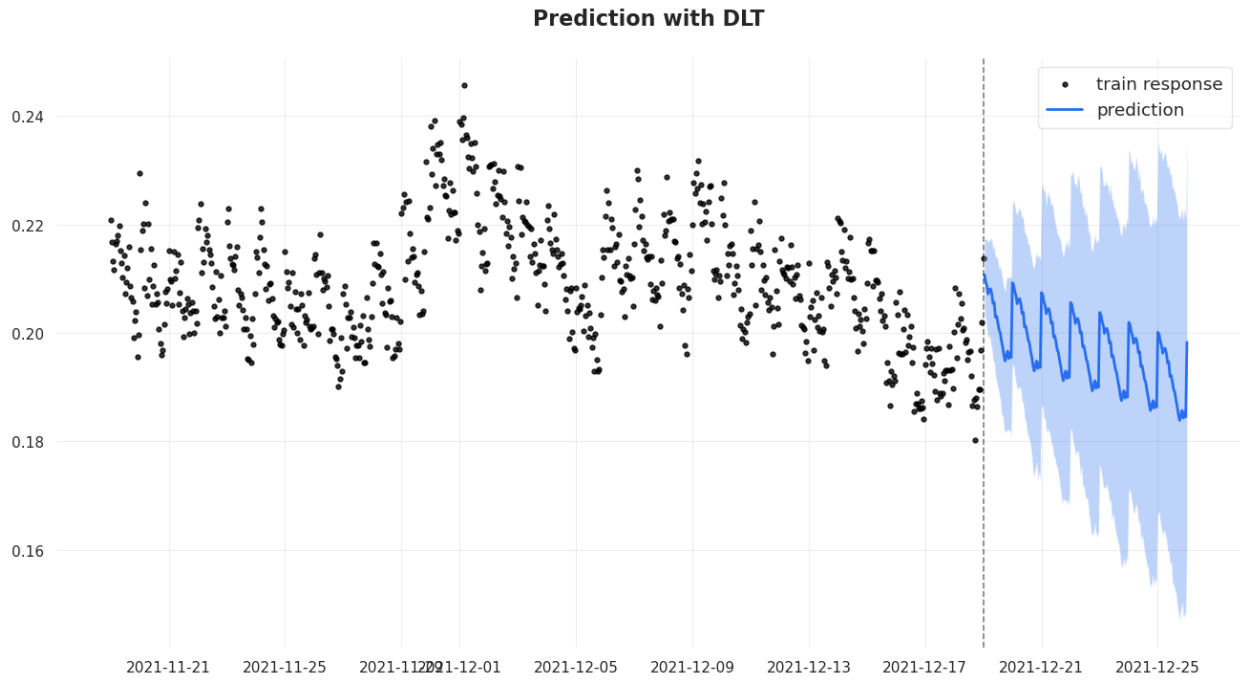
```
future_df = dlt.make_future_df(periods=24*7)
#горизонт будет 7 дней - то есть 7 раз по 24 часа
```

Прогнозируем:

```
predicted_df = dlt.predict(df=future_df)
```

Если нужно, то рисуем прогноз:

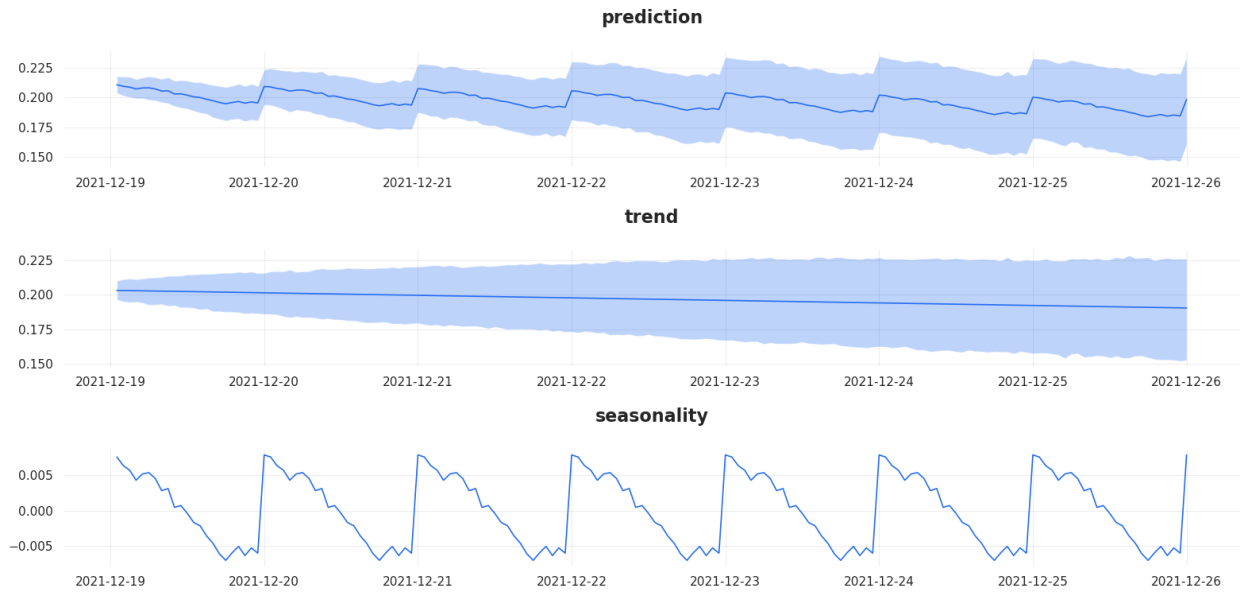
```
_ = plot_predicted_data(activity, #изначальные данные
                        predicted_df, #датафрейм с предсказаниями
                        "hour_time", #колонка со временем
                        'CTR', #колонка со значениями временного ряда
                        title='Prediction with DLT') #заголовок рисунка
```



При желании можно разложить прогноз на несколько компонентов и нарисовать их в отдельности. Возможные варианты для рисования: `"prediction"` (общее предсказание), `"trend"`, `"seasonality"`, `"regressor"` (значения ковариата), `"regression"` (значения коэффициентов регрессии):

```
predicted_df = dlt.predict(df=future_df,
                           decompose=True) #говорит, что нам нужна декомпозиция предсказания

_ = plot_predicted_components(predicted_df,
                              "hour_time",
                              plot_components=['prediction', 'trend', 'seasonality'])
```



> Регрессоры/ковариаты

Во временных рядах информация не обязательно должна браться из прошлых значений показателя. Мы можем использовать другие временные ряды и оценивать их взаимосвязь с нашим рядом — вдруг они хорошо предсказывают его, тем самым улучшая наш прогноз! Такие временные ряды называются **регрессорами или ковариатами**.

Впрочем, есть одно но. Если мы хотим предсказывать что-то вместе с регрессорами, мы должны знать будущие значения этих регрессоров, иначе ничего не заработает! Как можно подойти к этой проблеме?

- Ориентироваться на фиксированные ежегодные события — например, праздники! Такие регрессоры можно создавать в виде бинарной переменной — 1 для наличия события, 0 для отсутствия. Так же можно кодировать известные нам аномалии, чтобы они не так влияли на прогноз.
- Пытаться предсказать самостоятельно другой временной ряд либо взять какие-то официальные прогнозы, если это какой-то невнутренний показатель (например, ВВП).
- Сценарное прогнозирование - "что мы можем ожидать, если переменные поведут себя вот так?". В целом предыдущий вариант вполне можно оформить именно так, но можно подбирать значения хоть вручную.

Задаются регрессоры в виде списка с названием колонок в аргументе `regressor_col`:

```
reg_model = DLT(response_col="CTR",
                date_col="hour_time",
                seasonality=24,
```

```
                estimator="stan-map",
                n_bootstrap_draws=1000,
                regressor_col=["num_posts"] #наша колонка с регрессором! Должна быть списком
               )
```

Параметры регрессоров и регуляризация

Очень часто, чтобы решить проблемы с переобучением, ML-специалисты прибегают к технике регуляризации — дополнительных штрафов к коэффициентам модели, чтобы они не становились слишком большими, особенно если регрессор плохой. Чаще всего это улучшает прогноз модели. Какие у нас есть варианты параметров?

- `regression_penalty` — вид регуляризации. Варианты: `"fixed ridge"` (полагается на заданные априорные распределения), `"auto_ridge"` (пытается сам подобрать оптимальную регуляризацию), `"lasso"` (полезен в случае, когда у нас много регрессоров — тогда он “убирает” влияние плохих регрессоров).
- `lasso_scale` и `auto_ridge_scale` — настройка для двух последних опций, число от 0 до 1. Чем оно выше, тем "мягче" регуляризация.
- `regressor_beta_prior` — среднее априорного распределения для коэффициента регрессии. По умолчанию 0, но можно задать другое значение, если у нас есть предварительное убеждение о реальном значении этого коэффициента.
- `regressor_sigma_prior` — стандартное отклонение априорного распределения для коэффициента регрессии. Чем больше, тем больше неопределённости в его значении (и тем мягче регуляризация).
- `regressor_sign` — знак коэффициента регрессии. Полезно, если мы изначально знаем направление взаимосвязи. Возможные варианты: `"+"` (может быть только положительным), `"-"` (может быть только отрицательным), `"="` (может иметь любое направление).

> Оценка качества модели

Мы успешно создали нашу первую модель! Но возникает вопрос: насколько эта модель эффективна? Насколько серьезно она может ошибаться?

Вы можете знать, что в машинном обучении это проверяют тремя основными способами:

- Разделением на *тренировочную* (трейн, train) и *тестовую* (тест, test) выборки — на первой мы модель обучаем, на второй проверяем, насколько хорошо она предсказывается
- *Кроссвалидацией* — как прошлый метод, только выборка дробится на трейн и тест несколькими способами

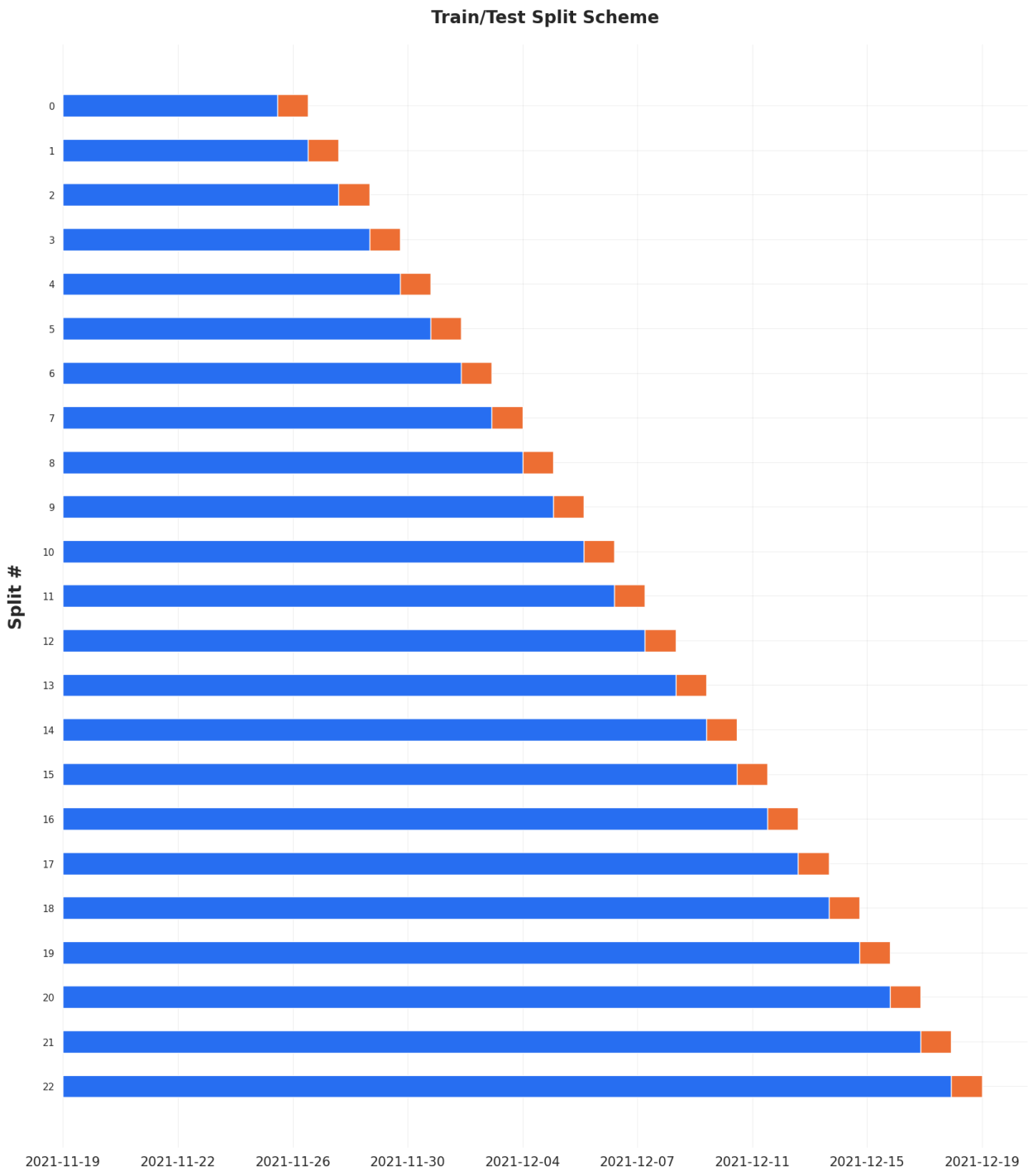
- Комбинация первых двух — выделяем тест, на трейне делаем кроссвалидацию, а потом дополнительно ещё проверяем модель на тесте

Однако в классической версии эти подходы не работают для временных рядов, поскольку наблюдения не являются независимыми, а связаны между собой во времени! В Orbit этот момент реализован через концепцию **бэктестинга**. Фактически это имитация исторических прогнозов — что было бы, если бы мы использовали эту модель в прошлом, насколько сильно мы бы ошибались.

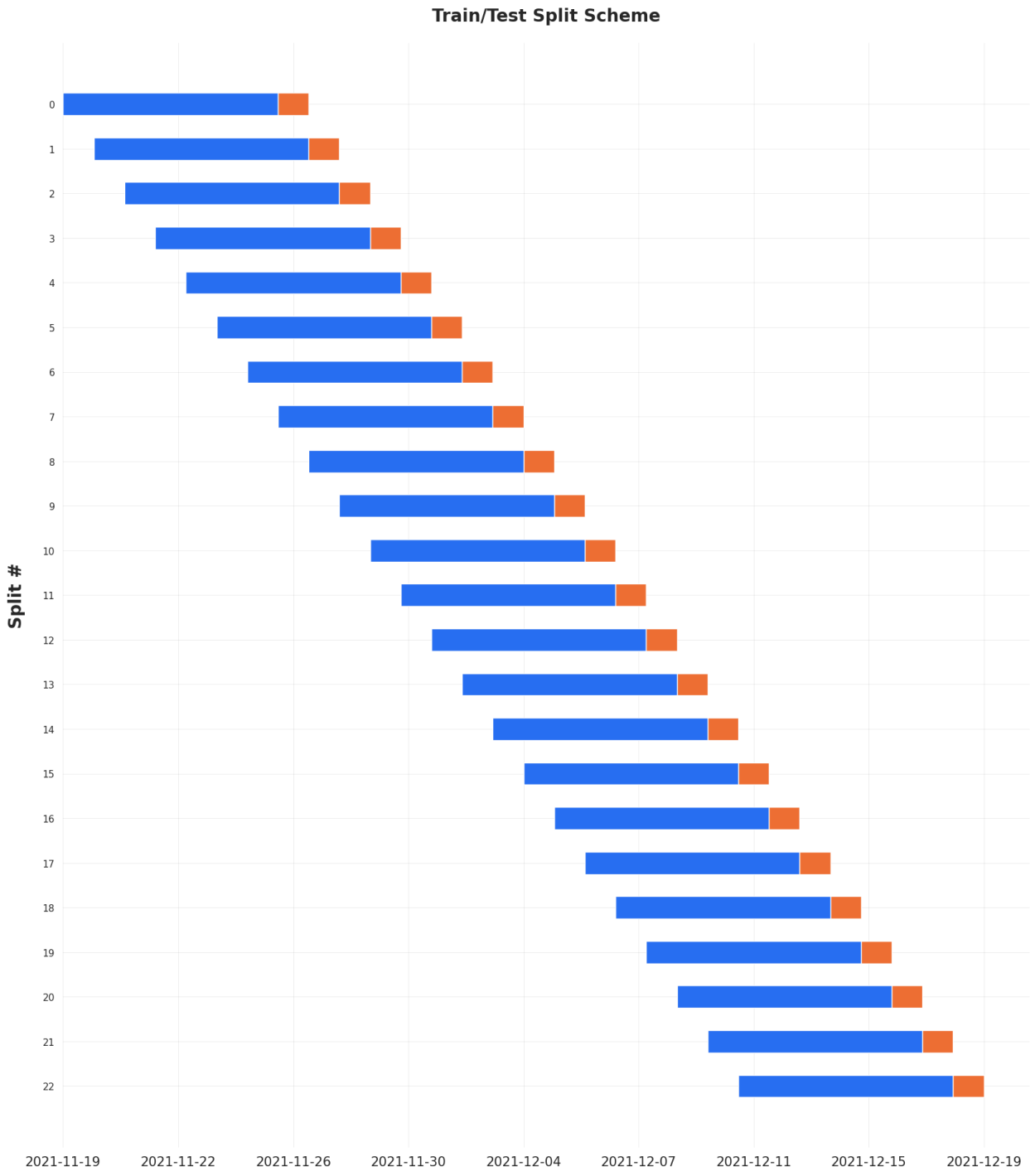
```
from orbit.diagnostics.backtest import BackTester #основной класс для бэктестинга

BackTester(
    model, #наша моделька
    df, #датафрейм для обучения
    min_train_len, #самый маленький кусочек данных, с которого начинается обучение
    incremental_len, #на сколько происходит прирост тренировочных данных
    forecast_len, #горизонт прогнозирования
    window_type #тип окошка
)
```

Расширяющееся (`"expanding"`) окно начинает с небольшого размера тренировочных данных и постепенно его увеличивает, пока не кончатся данные. Так можно оценить, как меняется точность предсказания с наращиванием количества информации. Такой вариант окна лучше всего подходит, если данные не слишком волатильны и в далеком прошлом есть важная информация, полезная для предсказания в будущем:



В скользящем (`"rolling"`) окне размер тренировочных данных не меняется - он сдвигается во времени. Такой вариант полезен, если наши данные волатильны и основная информация о предсказании находится в ближайшем прошлом:



Оценка результата проводится с помощью двух методов:

```
bt_exp.fit_predict() #обучаем
bt_exp.score() #выводим метрики
```

Но это ещё не всё: мы можем так подбирать оптимальные параметры для нашей модели! Делается это через grid search — перебор возможных параметров модели и проверку, какой из них выдаёт минимальное значение функционала ошибки.

```
from orbit.utils.params_tuning import grid_search_orbit #для подбора оптимальных параметров

grid_search_orbit(param_grid, #наши параметр и значения, которые мы у этих параметров тестируем
                  model,
                  df,
                  eval_method, #метод проверки модели - бэктестинг или BIC
                  min_train_len,
                  incremental_len,
                  forecast_len,
                  metrics, #какие метрики качества модели мы смотрим
                  criteria, #ищем минимальное или максимальное значение показателя?
                  verbose) #печатать или не печатать промежуточные результаты
```

Информационные критерии

Если вы когда-нибудь работали с регрессионными моделями, то могли видеть аббревиатуры AIC/BIC. Это так называемые информационные критерии — мера относительного качества модели. Их задача — отобрать модель с максимальной предсказательной способностью и минимумом предикторов.

Мы можем использовать их, чтобы выбрать лучшую модель! Чем меньше (W)BIC — тем лучше модель.

```
model.get_bic()

#если мы используем алгоритм MCMC, то нужен дополнительный шаг

model.fit_wbic() #обучаем модель таким образом, чтобы можно было рассчитать WBIC
model.get_wbic()
```

Дополнительно: проверка MCMC

Если вы используете алгоритм MCMC для оценки модели, то важно проверять, насколько хорошо он "сошёлся". Это можно делать графически для разных параметров - главное смотреть на глобальные показатели!

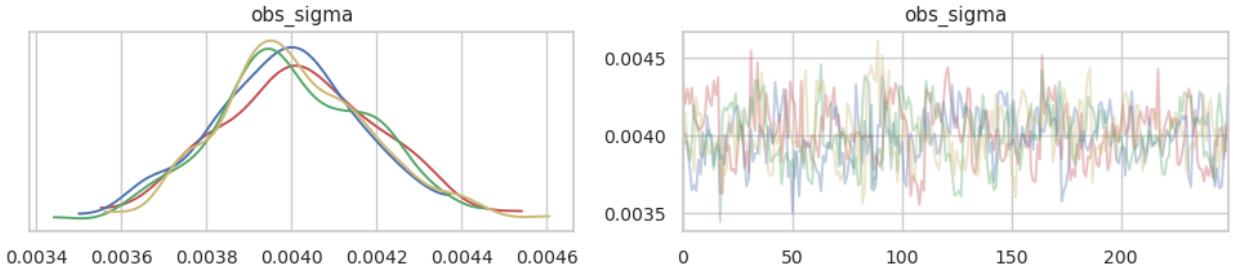
Есть очень много разных вариантов проверок, реализованных в библиотеке Arviz. Самая важная их них — *сходимость цепей*. Визуально она должна выглядеть как 4 временных ряда, скачущих вокруг примерно одного значения. Это значит, что разные инстанции алгоритма пришли к одному и тому же выводу в плане параметров!

```
import arviz as az

params = model.get_posterior_samples(permute=False, relabel=True) #достаём информацию о параметрах

params.keys() #их названия - можно залезть в оригинальный код на Stan, чтобы вычислить их значение

az.plot_trace(params, #словарь с параметрами
              var_names) #названия параметров, на которые мы смотрим
```



> Метрики

Есть много метрик, по которым можно оценить качество прогноза. В Orbit по умолчанию реализованы следующие (смотреть `orbit.diagnostics.metrics`):

- Mean Squared Error (`mse`) - средний квадрат ошибок:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- Mean Absolute Error (`mae`) - средняя абсолютная ошибка. По сравнению с предыдущей метрикой даёт меньше веса аномально неверным предсказаниям (например, выбросам в данных):

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- Root Mean Squared Scaled Error (`rmsse`) - похожа на MSE, но с дополнительным приколом. Она показывает, насколько наша модель хуже, чем если бы мы просто гадали на основе предыдущих значений. Подробнее о том, почему это хорошая метрика - тут.

$$\sqrt{\frac{1}{h}\frac{\sum_{i=n+1}^{n+h}(y_i - \hat{y}_i)^2}{\frac{1}{n-1}\sum_{i=2}^{n}(y_i - y_{i-1})^2}}$$

- Mean Absolute Percentage Error (`mape`) - похожа на MAE, но выражена в долях от истинного значения. Легко интерпретируется, но имеет большой недостаток - метрика предубеждена в сторону моделей, которые “недопредсказывают”, и наказывает модели с “перепредсказанием”.

$$\frac{1}{n}\sum_{i=1}^{n}|\frac{y_i-\hat{y}_i}{y_i}|$$

- Symmetric Mean Absolute Percentage Error (`smape`) - вариант коррекции предыдущей метрики, чтобы недопредсказания имели тот же вес, что и перепредсказания.

$$\frac{1}{n}\sum_{i=1}^{n}\frac{|y_i-\hat{y}_i|}{|y_i|+|\hat{y}_i|}$$

- Weighted Mean Absolute Percentage Error (`wmape`) - взвешенный вариант MAPE. Каждому наблюдению дополнительно даётся определённый вес, по которому оценивается “вклад” конкретного наблюдения - в Orbit больший вес даётся наиболее большим по модулю наблюдениям относительно временного ряда.

$$\frac{1}{n}\frac{\sum_{i=1}^{n}w_i*|y_i-\hat{y}_i|}{\sum_{i=1}^{n}w_i*|y_i|}$$

- Weighted Symmetric Mean Absolute Percentage Error (`wsmape`) - два предыдущих пункта в одном.

$$\frac{2}{n}\frac{\sum_{i=1}^{n}w_i*|y_i-\hat{y}_i|}{\sum_{i=1}^{n}w_i*(|y_i|+|\hat{y}_i|)}$$

Некоторые размышления о выборе той или иной метрики можно взять тут.

> Основные алгоритмы байесовского вывода

Как правило, прямой аналитический вывод в реальной практике практически невозможен, когда дело доходит до байесовских методов. Поэтому существует ряд алгоритмических методов, с помощью которых можно рассчитать параметры модели! Какие у нас есть варианты?

Если кратко: важна скорость — берите `stan-map`. Важна точность параметров и доверительных интервалов — берите `stan-mcmc`. Хочется чего-то промежуточного —

берите `pyro-svi`. Но проверяйте, какие из трёх опций доступны для выбранной вами модели — у разных моделей в Orbit разный набор возможных алгоритмов!

Maximum A Posteriori (MAP)

Байесовский аналог метода максимального правдоподобия, дополнительно учитывающий априорное распределение оцениваемого параметра. Фактически оценивает только наиболее вероятное значение этого параметра, без доверительного интервала, что отчасти компенсируется методом бутстрапа.

Этот метод обычно обладает невысокой точностью и не всегда учитывает полную изменчивость данных. Его преимущество заключается в быстрой скорости, что делает его подходящим для исследовательской фазы создания прогнозных моделей и для прогнозов в реальном времени.

Особых дополнительных параметров нет, однако можно задать `n_bootstrap_draws` — количество итераций бутстрапа.

Markov Chain Monte Carlo (MCMC)

"Золотой стандарт" байесовского вывода с теоретически гарантированным достижением "истинного" параметра за n-ое число итераций. Впрочем, нет никакой гарантии, что это произойдёт быстро, особенно для очень сложных моделей.

Есть много вариантов алгоритма с разной вычислительной скоростью и разными допущениями. В Orbit используется адаптивный алгоритм под названием **No-U-Turn Sampler (NUTS)**, который является разновидностью алгоритма Hamiltionian Monte-Carlo. О других вариантах можно почитать тут, а тут можно даже посмотреть в действии. Также можно ознакомиться с общей схемой работы MCMC на примере простейшего варианта этого алгоритма - тут, тут и тут.

Наиболее важные параметры:

- `chains` — количество цепей (по умолчанию 4). Фактически алгоритм запускается 4 раза, после чего берётся средний результат. Это даёт нам хорошую диагностику качества модели — если четыре “цепи” сошлись на одном диапазоне параметров, значит, алгоритм отработал хорошо.
- `cores` — количество ядер для параллельных вычислений (по умолчанию берётся возможный максимум). Это нужно для параллельного выполнения четырех вышеупомянутых цепей, а значит — экономии времени.
- `num_warmup` — количество итераций "разогрева". В этом промежутке цепь "настраивается" на оптимальные параметры, с которыми она уже будет оценивать значения параметра.

- `num_sample` — количество итераций сэмплирования. Можно интерпретировать в том же ключе, что и итерации бутстрапа - чем их больше, тем более точный результат и тем большее разрешение он имеет.

Variational Inference (VI)

Весьма популярный класс методов байесовского вывода среди специалистов по машинному обучению. По сути, это задача оптимизации — подбирается некоторое "суррогатное" распределение, которое должно быть похоже на апостериорное, после чего параметры модели "подгоняются" под оптимум. Общее описание подхода можно прочитать тут. В Orbit реализован вариант под названием Stochastic Variational Inference.

Поскольку вместо "настоящего" апостериорного распределения используется суррогатное, нет гарантий достижения глобального оптимума. В результате точность вариационного вывода может быть ниже, чем у MCMC, но скорость выполнения будет значительно выше.

Наиболее важные параметры:

- `num_steps` — количество итераций алгоритма до достижения оптимального решения. Можно увеличить, если оптимальное решение не нашлось.
- `learning_rate` — скорость обучения. Чем выше значение этого параметра, тем быстрее происходит обучение, но при этом возрастает риск пропуска оптимального решения. Если значение `learning_rate` снизить, то алгоритм будет работать медленнее, увеличится вероятность того, что за определенное количество итераций он не успеет дойти до оптимума. Зато повышается вероятность обнаружения этого оптимума. Поэтому важно балансировать этот параметр между скоростью и сходимостью.
- `learning_rate_total_decay` — понижение скорости обучения между первым и последним шагом алгоритма. По умолчанию значение скорости обучения остается неизменным.

> CausalImpact

A/Б-тесты — это, конечно, полезный инструмент, но не все вещи можно проверить с их помощью. Это может быть обусловлено следующими причинами:

1. Интересующая нас переменная не подчиняется прямой манипуляции.
2. Управлять переменной дорого или неэффективно.
3. Управлять ею неэтично.

Что же делать? Существует широкий набор различных методик, объединенных общим названием "**Causal Inference**" (причинно-следственный вывод). Один из подходов очень прост: мы предсказываем временной ряд, а затем смотрим, насколько предсказания расходятся с реальными показателями после некоторого события.

Наиболее удобно такая техника реализована в пакете под названием *CausalImpact*.

- Пакет
- Ноутбук с демонстрацией
- Статья с описанием алгоритма

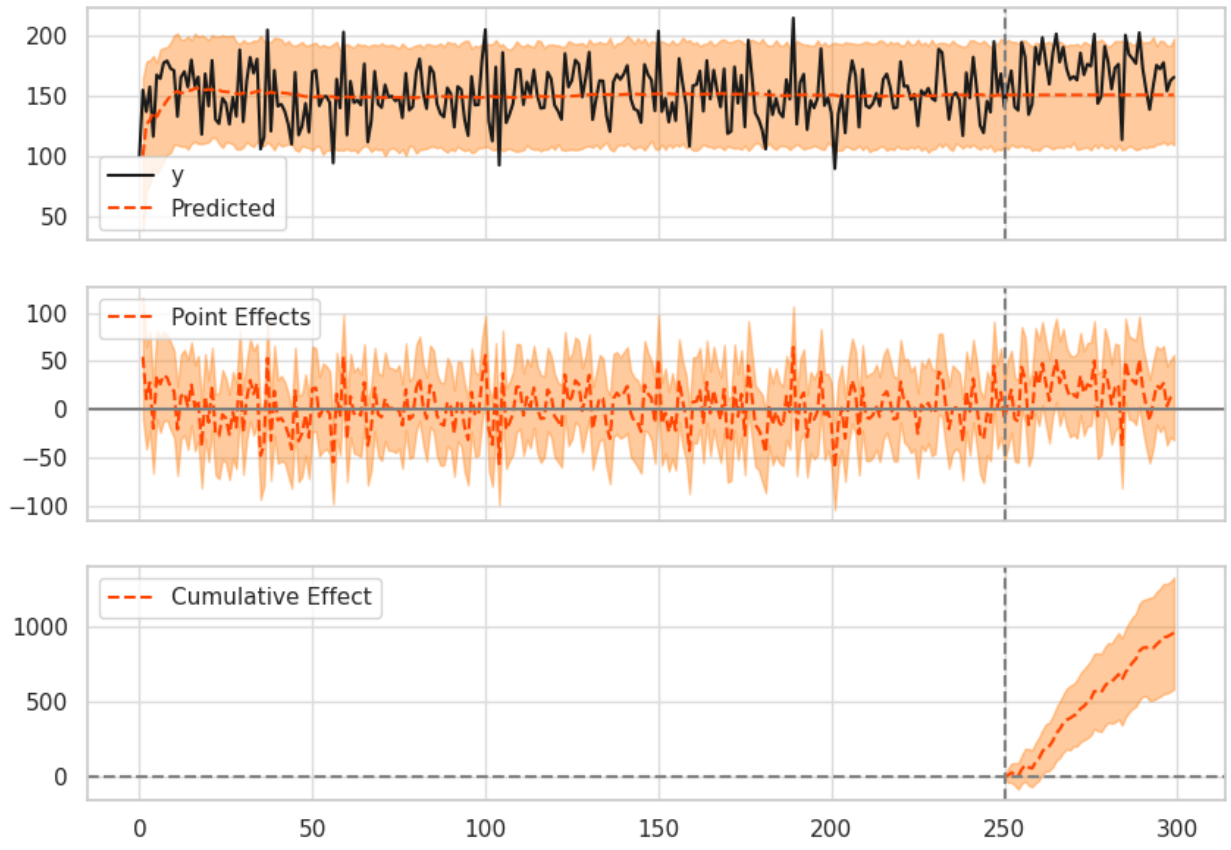
Само использование довольно простое:

```
from causalimpact import CausalImpact

impact = CausalImpact(data, #данные
                      pre_period, #период до события
                      post_period, #период после события
                      model, #можно задать свою модель, а не пользоваться моделью по умолчанию
                      model_args, #аргументы модели - см. docstring функции
                      alpha) #уровень значимости
```

Результат можно нарисовать — он покажет реальные и предсказанные значения временного ряда (1 ряд), отклонения реальных значений от предсказанных (2 ряд) и накопленные отличия реальных значений от предсказания (3 ряд):

```
impact.plot()
```



Можно вывести отчёт о величине и статзначимости эффекта:

```
print(impact.summary())
```

```
Posterior Inference {Causal Impact}
                          Average            Cumulative
Actual                    169.83             8321.84
Prediction (s.d.)         150.3 (3.95)       7364.64 (193.67)
95% CI                    [142.52, 158.01]   [6983.5, 7742.69]

Absolute effect (s.d.)    19.53 (3.95)       957.21 (193.67)
95% CI                    [11.82, 27.31]     [579.16, 1338.34]

Relative effect (s.d.)    13.0% (2.63%)      13.0% (2.63%)
95% CI                    [7.86%, 18.17%]    [7.86%, 18.17%]

Posterior tail-area probability p: 0.0
Posterior prob. of a causal effect: 100.0%

For more details run the command: print(impact.summary('report'))
```

Можно даже в виде текстового отчёта:

```
print(impact.summary('report'))
```

Наиболее важное допущение метода — **ковариаты не должны быть подвержены воздействию**. Таким образом мы максимально удостоверяемся, что у нас получилось предсказание "как если бы ничего не случилось". Впрочем, если вы хотите проверить, изменилась ли какая-то метрика без учёта прироста другой метрики (например, выросла ли выручка вне зависимости от увеличения числа клиентов), то такой ковариат добавить можно.

Кастомная модель

По умолчанию модель CausalImpact предполагает, что временной ряд держится на каком-то постоянном уровне. Опционально в данные можно (и нужно) добавлять регрессоры, также в `model_args` можно задать сезонность. Но если нужно что-то ещё, то нам необходимо добавлять свою кастомную модель, построенную через TensorFlow Probability. Конкретно нас интересует модуль sts, где можно построить так называемую структурную модель временного ряда.

Если мы хотим построить модель, которая учитывает не только общий уровень, но и тренд, нам нужно всего лишь сделать это:

```
import tensorflow_probability as tfp

#нам достаточно указать временной ряд до события
#он сам рассчитает необходимые параметры
```

```
trend_component = tfp.sts.LocalLinearTrend(observed_time_series)
```

Эту модель уже можно использовать внутри функции `CausalImpact()`.

Если нам хочется объединить в единую модель несколько компонентов (тренд, сезонность, регрессию, ещё что-нибудь), то нам пригодится дополнительно функция `tfp.sts.Sum()`:

```
#на вход идёт список со всеми компонентами
#полный список есть в документации модуля

custom_model = tfp.sts.Sum([comp_1, comp_2], observed_time_series)
```

Дополнительно:

- Пример использования `tfp.sts` в документации
- Пример использования `tfp.sts` на просторах интернета

> Дополнительно

Если вы не знакомы с баейсовской статистикой и методологией причинно-следственного вывода, то можете не понять ряд деталей и терминов в этом уроке. Вы можете заполнить пробелы с помощью наших постов от преподавателя этого урока:

- Про формулу Байеса в целом
- Про общую идею байесовского вывода — в конце список полезной литературы
- Про причинно-следственный вывод и DAG-и — в конце тоже список полезной литературы по тематике причинно-следственного вывода