*Дата последней сборки документа:*
*5 января 2022 г. в 00:46*

# Заметки по машинному обучению и анализу данных. Том 2

*Подвойский А.О.*

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

## Краткое содержание



## Содержание

# 1. Приемы работы с библиотекой анализа временных рядов ETNA

## 1.1. Перекрестная проверка на временных рядах

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

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


Рис. 1. Перекрестная проверка на временном ряду *расширяющимся* окном

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

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


Рис. 2. Перекрестная проверка *на скользящем* окне

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

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

- Размер тестовой выборки, как правило, определяется горизонтом прогнозирования, а тот в свою очередь определяется бизнес-требования. Если вы предсказываете на 14 дней вперед, то и тестовая выборка должна включать 14 более поздних наблюдений.
- Размер тестовой выборки остается постоянным. Это значит, что метрики качества, полученные в результате вычислений прогнозов каждой обученной модели по тестовому набору, будут последовательны и их можно объединять и сравнивать.
- Размер обучающей выборки не может быть меньше тестовой выборки.
- Если данные содержат сезонность, обучающая выборка должна содержать не менее двух полных сезонных циклов (правило $2L$, где $L$ – количество периодов в полном сезонном цикле, необходимое для инициализации параметров некоторых моделей, например, для вычисления исходного значения тренда в модели тройного экспоненциального сглаживания), учитывая уменьшение длины ряда при выполнении процедур обычного и сезонного дифференциирования.

- Если применяются переменные – лаги, разности на лагах, скользящие статистики, то каждый раз для получения значений в тестовой выборке используются только данных обучающей выборки.

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


Рис. 3. Модфицированная перекрестная проверка расширяющимся окном

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


Рис. 4. Модфицированная перекрестная проверка скользящим окном

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

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

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

---

*Замечание*

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

---

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

## 1.2. CatBoost. Базовая модель с конструированием признаков

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

Для создания признаков можно использовать классы-трансформеры:

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

---

*Замечание*

Ширина скользящего окна не должна превышать горизонт прогнозирования

---

С помощью параметра `in_column` класса-трансформера задаем переменную, которую нужно преобразовать или на основе которой нужно создать признаки (по умолчанию этой переменной будет переменная `target`). С помощью параметра `out_column` (этот параметр есть у всех классов-трансформеров, создающих признаки) можно задать имена генерируемых переменных.

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

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

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

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

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

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

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

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

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

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

```
# создаем экземпляр класса VotingEnsemble
voting_ensemble = VotingEnsemble(pipelines=pipelines,
weights=[1, 2, 4],
n_jobs=4)
# получаем метрики
voting_ensamble_metrics = voting_ensemble.backtest(
    ts=ts,
    metrics=[MAE(), MSE(), SMAPE(), MAPE()],
    n_folds=3,
    aggregate_metrics=True,
```

```
    n_jobs=2
)[0].iloc[:, 1:]
voting_ensamble_metrics.index = ['voting ensemble']
```

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

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

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

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

```
# создаем экземпляр класса StackingEnsemble,
# признаки - прогнозы конвейеров
stacking_ensemble_unfeatured = StackingEnsemble(
    features_to_use='None', pipelines=pipelines,
    n_folds=10, n_jobs=4)
# выполняем стекинг
stacking_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 ensemble']
stacking_ensamble_metrics
```

### 1.3. Пользовательские классы для вычисления скользящих статистик

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

```
# пишем класс MadTransform, вычисляющий скользящие
# средние абсолютные отклонения
class MadTransform(WindowStatisticsTransform):
    """
    MadTransform вычисляет среднее абсолютное отклонение
    (mean absolute deviation - mad) для заданного окна.
    """
    def __init__(
      self,
      in_column: str,
      window: int,
      seasonality: int = 1,
      min_periods: int = 1,
      fillna: float = 0,
      out_column: Optional[str] = None
    ):
```

```
    """
    Параметры
    ----------
    in_column: str
    имя обрабатываемого столбца
    window: int
    ширина окна для агрегирования
    out_column: str, optional
    имя результирующего столбца. Если не задано,
    используем __repr__()
    seasonality: int
    коэффициент сезонности
    min_periods: int
    Минимальное количество наблюдений в окне
    для агрегирования
    fillna: float
    значение для заполнения значений NaN
    """
    self.in_column = in_column
    self.window = window
    self.seasonality = seasonality
    self.min_periods = min_periods
    self.fillna = fillna
    self.out_column = out_column
    super().__init__(
    window=window,
    in_column=in_column,
    seasonality=seasonality,
    min_periods=min_periods,
    out_column=self.out_column
    if self.out_column is not None
    else self.__repr__(),
    fillna=fillna,
  )
  def _aggregate_window(
    self, series: pd.Series
  ) -> float:
    """Вычисляет mad для серии."""
    tmp_series = self._get_required_lags(series)
    return tmp_series.mad(**self.kwargs)
```

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

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

```
# пишем ядро - внутренний класс _LGBMModel,
# внутри - класс LGBMRegressor
class _LGBMModel:
  def __init__(
    self,
    boosting_type='gbdt',
    num_leaves=31,
    max_depth=-1,
    learning_rate=0.1,
    n_estimators=100,
    **kwargs
  ):
```

```
    self.model=LGBMRegressor(
      boosting_type=boosting_type,
      num_leaves=num_leaves,
      max_depth=max_depth,
      learning_rate=learning_rate,
      n_estimators=n_estimators,
      **kwargs
    )
  def fit(self, df: pd.DataFrame):
    features = df.drop(columns=['timestamp', 'target'])
    target = df['target']
    self.model.fit(X=features, y=target)
    return self

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

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

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

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

```
# пишем класс LGBMModelMultiSegment, который строит
# одну модель LGBM для всех сегментов
class LGBMModelMultiSegment(Model):
  def __init__(
    self,
    boosting_type='gbdt',
    num_leaves=31,
```

```
    max_depth=-1,
    learning_rate=0.1,
    n_estimators=100,
    **kwargs
  ):
    self.kwargs = kwargs
    super(LGBMModelMultiSegment, self).__init__()
    self._base_model=_LGBMModel(
      boosting_type=boosting_type,
      num_leaves=num_leaves,
      max_depth=max_depth,
      learning_rate=learning_rate,
      n_estimators=n_estimators,
      **kwargs
    )

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

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

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

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

    return ts
```

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

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

```
            booster=booster,
            max_depth=max_depth,
            learning_rate=learning_rate,
            n_estimators=n_estimators,
            **kwargs,
        )

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

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

Пишем классы `XGBModePerSegment` и `XGBModelMultiSegment`

```
# пишем класс XGBModelPerSegment, который строит
# отдельную модель XGB для каждого сегмента
class XGBModelPerSegment(PerSegmentModel):
  def __init__(
    self,
    booster='gbtree',
    max_depth=3,
    learning_rate=0.1,
    n_estimators=200,
    **kwargs
  ):
  self.kwargs = kwargs
  model = _XGBModel(
    booster=booster,
    max_depth=max_depth,
    learning_rate=learning_rate,
    n_estimators=n_estimators,
    **kwargs
  )
  super(XGBModelPerSegment, self).__init__(
    base_model=model)

# пишем класс XGBModelMultiSegment, который строит
# одну модель XGB для всех сегментов
class XGBModelMultiSegment(Model):
  def __init__(
    self,
    booster='gbtree',
    max_depth=3,
    learning_rate=0.1,
```

```
    n_estimators=100,
    **kwargs
  ):
    self.kwargs = kwargs
    super(XGBModelMultiSegment, self).__init__()
    self._base_model=_XGBModel(
      booster=booster,
      max_depth=max_depth,
      learning_rate=learning_rate,
      n_estimators=n_estimators,
      **kwargs
  )

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

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

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

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

    return ts
```

*Замечание*

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

Таким образом, необходимо создаватвь лаговые переменные так, чтобы они не проникали в тестовый набор. Лаги вида $L_{t-k}$ лучше создавать так, чтобы $k$ был равен или превышал горизонт прогнозирования (рис. 5). Впрочем, допускается создание лагов, у которых порядок будет меньше длины горизонта прогнозирования, но тогда значения зависимой переменной в тестовой выборке нужно заменить на значение NaN. Если лаг и залезет в тест, ему ничего не останется,

как использовать значение NaN, таким образом, в тесте появится значение NaN. В таком случае, чем больше горизонт прогнозирования будет превышать порядок лага, тем больше пропусков будет в тесте.

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

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

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

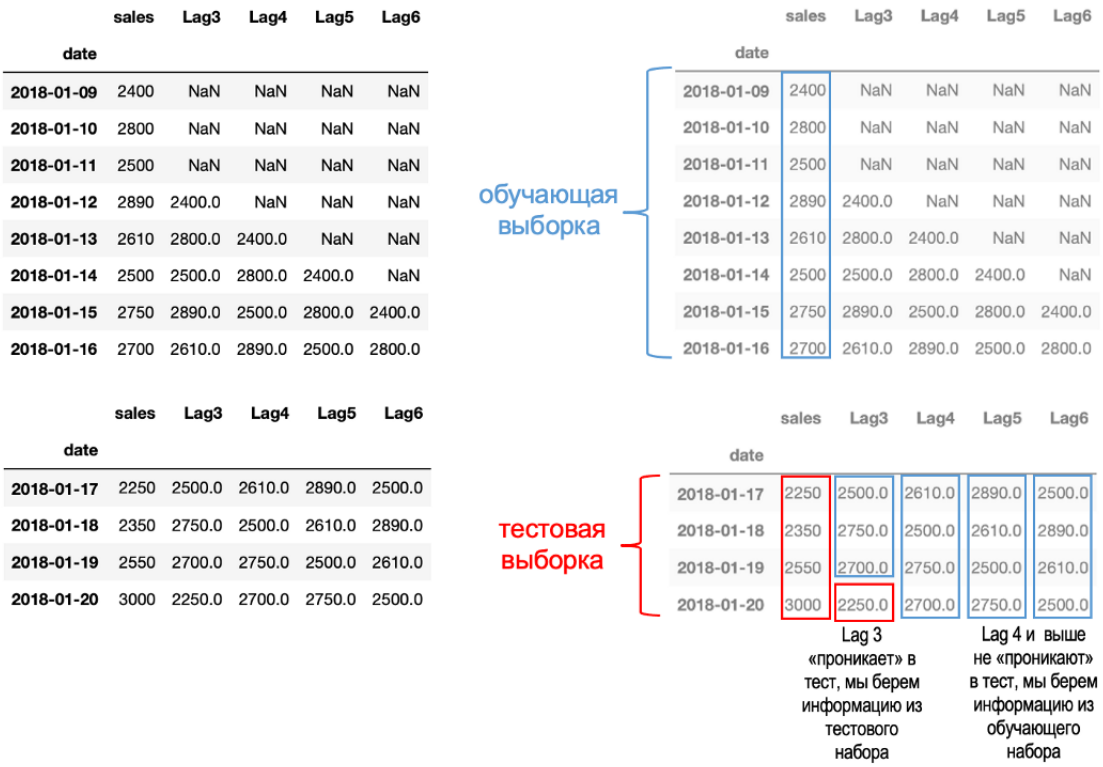


Рис. 5. Лаги, у которых порядок равен горизонту прогнозирования или превышает его, не используют тестовую выборку

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

```
# создаем экземпляр класса LagTransform для генерации лагов,
# с помощью in_column задаем переменную, на основе которой
# генерируем лаги, мы будем генерировать лаги порядка от 8 до 23,
# порядок лагов не должен быть меньше длины горизонта
lags = LagTransform(in_column='target',
        lags=list(range(8, 24, 1)),
        out_column='lag')
# создаем экземпляр класса MeanTransform для вычисления
# скользящего среднего по заданному окну
mean8 = MeanTransform(in_column='target',
        window=8,
        out_column='mean8')
```

```
# создаем экземпляр класса MadTransform для вычисления
# среднего абсолютного отклонения по заданному окну
mad8 = MadTransform(in_column='target',
        window=8,
        out_column='mad8')
# добавляем лаги, mean8, mad8 в обучающую выборку
train_ts.fit_transform([lags, mean8, mad8])
# создаем экземпляр класса LGBMModelPerSegment
model = LGBMModelPerSegment()
# обучаем модель
model.fit(train_ts)
# формируем тестовый набор
future_ts = train_ts.make_future(HORIZON)
# получаем прогнозы
forecast_ts = model.forecast(future_ts)
# оцениваем качество прогнозов
smape(y_true=test_ts, y_pred=forecast_ts)
```

### 1.4. Работа с несколькими временными рядами

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

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

df = TSDataset.to_dataset(original_df)
```

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

Нам понадобятся следующие классы-трансформеры:

- класс `LogTransform` для логарифмирования и экспонецирования перерменной (логарифмирование позволяет сгладить негативное влияние выборосов объективной природы, помогает выделить тренд),
- `LinearTrendTransform` для прогнозирования тренда, удаления тренда из данных и добавления тренда к прогнозам (это необходимо для деревьев решений и для ансамблей деревьев решений, *не умещющих экстраполировать*),
- `LagTransform` для генерации лагов,
- `DateFlagsTransform` для генерации признаков на основе дат – порядковый номер дня недели, порядковый номер дня месяца, порядковый номер недели в месяце и пр.,
- `MeanTransform` для вычисления скользящего среднего по заданному окну.

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

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

# создаем экземпляр класса LinearTrendTransform
# для прогнозирования тренда, удаления тренда из
# данных и добавления тренда к прогнозам
```

```
trend = LinearTrendTransform(in_column='target')

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

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

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

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

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

```
# разбиваем набор на обучающую и тестовую выборки
# с учетом временной структуры
train_ts, test_ts = ts.train_test_split(
    train_start="2019-01-01",
    train_end="2019-11-30",
    test_start="2019-12-01",
    test_end="2019-12-31",
)

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

Задаем явно горизонт в 31 день, обучаем модель CatBoost, оцениваем качество прогнозов и визуализируем прогнозы. Кроме того, не забываем выполнить обратные преобразования (добавление тренда, экспонецирование зависимой переменной) с помощью метода `.inverse_transform()`

для обучающего набора для правильной визуализации значений зависимой переменной в обучающей выборке.

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

# получаем прогнозы
forecast_ts = model.forecast(future_ts)

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

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

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

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

```
pipe = Pipeline(
    model=model,
    transform=[
        log,
        trend,
        seg,
        lags,
        d_flags,
        mean30,
    ],
    horizon=HORIZON,
)
metrics, forecast, info = pipe.backtest(ts, [smape], aggregate_metrics=True)
```

Ансамбль бустингов

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

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

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

```
)

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

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

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

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

## 2. Генерация признаков и кодирование категориальных признаков

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

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

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

Еще категории можно кодировать по разным признакам.

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

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

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

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

```
from numpy.linalg import svd

def code_factor(data, cat_feature, cat_feature2):
    """
    Кодирование признака на основе другого признака
    """
    ct = pd.crosstab(data[cat_feature], data[cat_feature2])
```

```
    u, _, _ = svd(ct.values)
    coder = dict(zip(ct.index, u[:, 0]))  # берем только первый сингулярный вектор

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

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

---

*Замечание*

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

---

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

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

где $\Lambda$ – диагональная матрица.

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

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

где $r$ – ранг матрицы $A$.

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

$$\lambda_1 \geqslant \lambda_2 \geqslant \ldots \geqslant \lambda_n > 0.$$

Столбцы матриц $U$ и $V$ называются соответсвенно *левыми* и *правыми сингулярными векторами*, а значения диагонали матрицы $\Lambda$ – *сингулярными числами*.

Эквивалентная запись сингулярного разложения

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

Например, матрица

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

имеет сингулярное разложение

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

Легко увидеть, что матрицы $U$ и $V$ ортогональны,

$$U^TU = UU^T = I, \quad V^TV = VV^T = I,$$

и сумма квадратов значений их столбцов равна единице.

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

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

где $r = \min(m, n)$.

*Сингулярное разложение и собственные числа матрицы*

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

Так как матрицы $U$ и $V$ ортогональные, то

$$AA^T = U\Lambda \underbrace{V^TV}_{=I}\Lambda U^T = U\Lambda^2U^T,$$
$$A^TA = V\Lambda \underbrace{U^TU}_{=I}\Lambda V^T = V\Lambda^2V^T.$$

Умножая оба выражения справа соответсвенно на $U$ и $V$, получаем

$$AA^TU = U\Lambda^2,$$
$$A^TAV = V\Lambda^2.$$

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

*SVD и норма матриц*

Евклидова норма

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

Норма Фробениуса

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

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

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

*Нахождение псевдообратной матрицы с помощью SVD*

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

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

$$A^+A = I_n,$$
$$AA^+ = I_m,$$
$$A^+AA^+ = A^+,$$
$$AA^+A = A.$$

Пусть найдено разложение матрицы $A$ вида

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

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

Тогда матрица

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

является для матрицы $A$ *псевдообратной*.

*Усеченное SVD при обращении матриц*

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

Тогда *усеченная псевдообратная матрица* $A_s^+$

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

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

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

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

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

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

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

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

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

На практике хорошо работает смесь подходов!

## 3. Перестановочная важность признаков и важность признаков по Шепли

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

Статья Дьяконова про интерпретацию черных ящиков

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

Плохой метод – чем чаще выбирался признак, тем лучше. Дело в том, что признак может действительно часто выбираться, но на более низких уровнях дерева (дальше от корня). Другими словами, признак выбирается часто, но используется для построения небольших «уточняющих» разбиений (рис. 6). Как правило, все наоборот. Если признак выбирается часто, значит модель не может по каким-то причинам сразу получить от него нужную информацию.

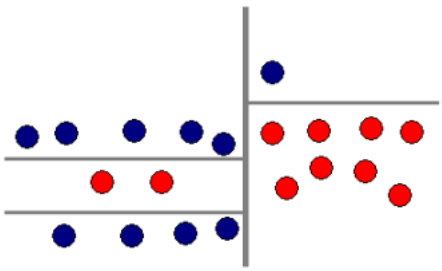

Рис. 6. К вопросу о важности признака по частоте его выбора. Признак по оси $x$ выбирается только один раз, а признак по оси $y$ выбирается три раза, но очевидно, что первый признак лучше справляется с задачей

Нельзя отбрасывать признаки по порогу.

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

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

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

*Замечание*

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

Идея перестановочной важности признаков: признак важный, если его перетасовка снижает качество. Можно вычислять перестановочную важность признаков на *обучающем поднаборе данных* (вроде как бы можно, но лучше использовать отложенную выборку PFI-holdout), на *отложенном контроле* (тестовый поднабор данных PFI-holdout, рис. 7) и на любой схеме *валидации* (надежнее использовать валидацию).

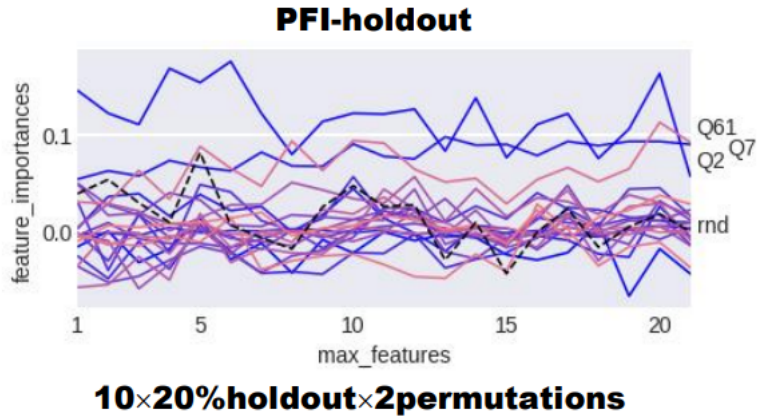


Рис. 7. Перестановочная важность признаков, вычисленная на отложенной выборке

Перестановочная важность скоррелиррованных признаков может размазываться между ними.

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

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

*Замечание*

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

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

***Важность по Шепли*** $i$-ого признака вычисляется следующим образом

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

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

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

Замечания по методам оценки важности признаков:

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

Советы Дьяконова:

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

# 4. Приемы работы с библиотекой Catboost

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

## 4.1. Установка CatBoost

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

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

## 4.2. Ключевые особенности пакета

## 4.3. Параметры

*Замечание*

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

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

В случае дисбаланса классов будет полезным настраивать

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

при этом не нужно настраивать эти параметры одновременно.

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

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

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

Общие параметры:

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

## 4.4. Классификатор CatBoostClassfier

Класс `CatBoostClassifier`

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

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

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

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

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

## 4.5. Регрессор CatBoostRegressor

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

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

## 4.6. Функции потерь и метрики качества

### 4.6.1. Для классификации

Для мультиклассификации https://catboost.ai/en/docs/concepts/loss-functions-multiclassificat

*Функции потерь*

- LogLoss

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

- CrossEntropy

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

*Метрики качества*

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

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

- BalancedErrorRate

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

- AUC,
- BrierScore,
- HingeLoss,
- HammingLoss

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

- Kappa

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

- LogLikelihoodOfPrediction.

### 4.6.2. Для регрессии

*Метрики качества, которые могут играть роль функции потерь*

- MultiRMSE (в случае мультирегрессии)

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

- MAE

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

- MAPE

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

- Poisson

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

- Quantile (большие значения $\alpha$ сильнее штрафуют за заниженные прогнозы)

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

- RMSE

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

- LogLinQuantile,
- Lq

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

- Huber

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

- Excpectile

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

- Tweedie

$$\frac{\sum\limits_{i=1}^{N}\Big(\frac{e^{a_i(2-\lambda)}}{2-\lambda}-t_i\frac{e^{a_i(1-\lambda)}}{1-\lambda}\Big)w_i}{\sum\limits_{i=1}^{N}w_i},$$

где $\lambda$ – значение параметра дисперсии мощности,

*Метрики качества*

- SMAPE

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

- R2 (коэффициент детерминации)

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

- MSLE (среднеквадратическая логарифмическая ошибка)

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

- MedianAbsoluteError

$$median(|t_1-a_1|,\ldots,|t_N-a_N|)$$

# 5. Приемы работы с библиотеками Gym и Ecole

## 5.1. Gym

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

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

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

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

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

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

```
import gym

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

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

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

## 5.2. Ecole

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

Полезный ресурс по математической оптимизации https://scipbook.readthedocs.io/en/latest/

### 5.2.1. Observations

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

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

## 6. Графовые нейронные сети

Полезные ресурсы Distill

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

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

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

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

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

В классических графовых сетях изменение состояния проихсходит по очень простой формуле

$$h_v^{(k)} = \sigma\Big(W_0^{(k)} h_v^{(k-1)} + W_1^{(k)} \sum_{u\in N(v)} h_u^{(k-1)} + b^{(k)}\Big),$$

т.е. агрегация здесь – обычное суммирование, а при обновлении последнее состояние и результат агреграции пропускаются через линейные слои (домножаются на матрицы), потом их линейная комбинация со смещением пропускается через функцию активации.

Чаще агрегация происходит суммированием состояний соседей, хотя можно использовать усреднение (Neighborhood Normalization)

$$m_{N(v)}^{(k)} = \frac{\sum\limits_{u\in N(v)} h_v^{(k-1)}}{|N(v)|}$$

или

$$m_{N(v)}^{(k)} = \sum_{u\in N(v)} \frac{h_v^{(k-1)}}{\sqrt{|N(v)|\,|N(u)|}}.$$

Послдений вариант усреднения используется в *графовых сверточных нейронных сетях* (Graph Convolutional Networks, GCNs). Важно правильно выбрать агрегацию, поскольку она может терять некоторую ценную информацию, например, степень вершины при обычном усреднении.

Еще одна из популярных агреграций (Set pooling)

$$m_{N(v)}^{(k)} = f_\theta\Big(\sum_{u\in N(v)} g_\varphi(h_u^{(k-1)})\Big)$$

функции $f$ и $g$ реализуются многослойными персептронами. Кстати, доказано, что сеть с такой агрегацией является универсальным аппроксиматором.

Также отметим Janossy pooling

$$m_{N(v)}^{(k)} = f_\theta\Big(\frac{1}{|\Pi|}\sum_{\pi\in\Pi} g_\varphi(h_{\pi_1})^{(k-1)}, \ldots, h_{\pi_{|N(v)|}}^{(k-1)}\Big),$$

в котором суммирование производится по всем перестановкам (в теории, а на практике – по нескольким случайным).

В GIN (Graph Isomorphism Network) используется довольно простая формула для адапатации состояний (и агрегация здесь – обычное суммирование)

$$h_v^{(k)} = f_\theta\Big((1+\varepsilon^{(k)})h_v^{(k+1)} + \sum_{u\in N(v)} h_u^{(k-1)}\Big).$$

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

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

Сводка по графовым нейронным сетям

- На вход сети подается граф, каждая вершина которого имеет признаковое описание. Это описание можно считать начальным состоянием вершины,
- Могут быть слои, которые независимо модифицируют представления (для каждой вершины его представление пропускается через небольшую нейронку),
- Могут быть слои, которые модифицируют представления всех вершин, учитывая представления вершин-соседей,
- Могут быть слои, упрощающие граф (например, уменьшающие число вершин),
- Могут быть слои, получающие представление графа (вектор фиксированной длины) по текущему графу с представлениями вершин.

Обычно такие сети используются для задач

- предсказания на уровне вершин, например, классификация вершин,
- предсказание на уровне пар вершин / ребер, например, в задаче предсказания вероятности появления ребра в графе,

- предсказания на уровне графа, например, для классификации графов,
- для детектирования сообществ или предсказания отношений между вершинами графа.

## 7. Отбор признаков с библиотекой BoostARoota

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

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

## 8. Классический и байесовский бутстреп

Бутстреп является универсальным инструментом для оценки статистической точности.

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

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

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

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

## 9. HDI

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

## 10. Площадь по ROC-кривой

Построение ROC-кривой происходит следующим образом (рис. 8):

1. Сначала сортируем все наблюдения по убыванию спрогнозированной вероятности положительного класса,

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

Значение вероятности положительного класса, при котором ROC-кривая находится на минимальном расстоянии от верхнего левого угла – точки с координатами (0, 1), дает наибольшую правильность классификации. В данному случае (рис. 9) будет 0.72.

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

| № | фактический класс | спрогнозированная вероятность положительного класса |
|---|---|---|
| 20 | P | 0,92 |
| 19 | P | 0,9 |
| 18 | P | 0,88 |
| 12 | N | 0,85 |
| 17 | P | 0,82 |
| 16 | P | 0,79 |
| 11 | N | 0,75 |
| 15 | P | 0,73 |
| 14 | P | 0,72 |
| 10 | N | 0,7 |
| 9 | N | 0,6 |
| 8 | N | 0,59 |
| 7 | N | 0,58 |
| 6 | N | 0,53 |
| 13 | P | 0,52 |
| 5 | N | 0,4 |
| 4 | N | 0,33 |
| 3 | N | 0,32 |
| 2 | N | 0,24 |
| 1 | N | 0,18 |

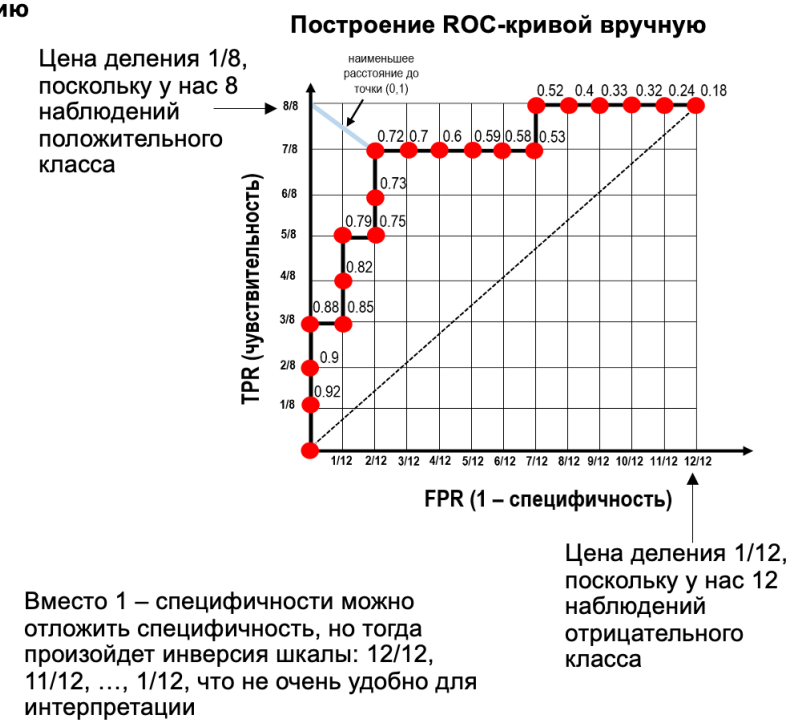


Рис. 8. Построение ROC-кривой

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

---

*Замечание*

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

---

Однако недостаток такой интепретации заключается в том, что мы пренебрегаем часто встречающейся ситуацией равенства вероятностей. Поэтому правильнее будет сказать, что ROC-AUC

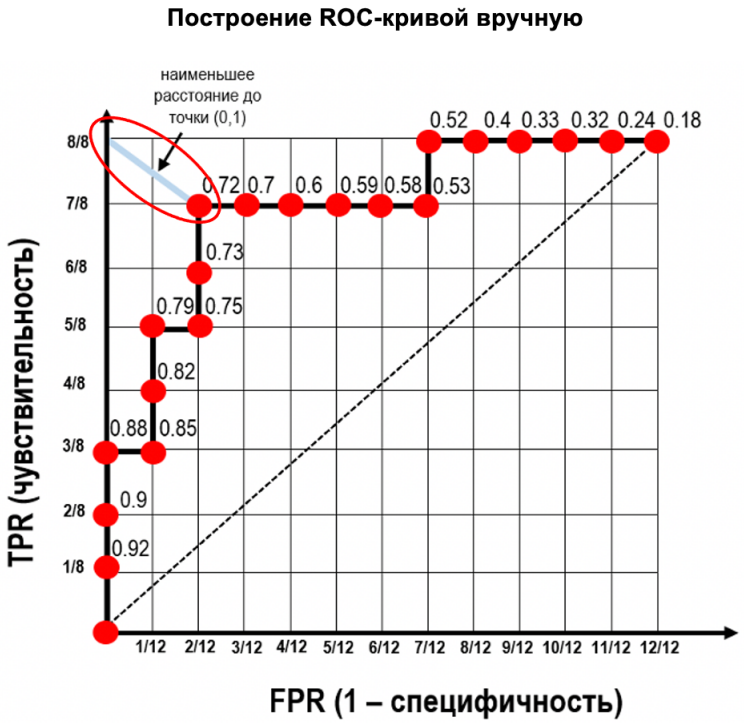


Рис. 9. ROC-кривая. Порог отсечения 0.72

равен доле пар вида (экземпляр положительного класса, экземпляр отрицательного класса), которые алгоритм верно упорядочил в соответствии с формулой

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

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

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

Если говорить более точно, мы берем наблюдение положительного класса под номером 20 и каждый раз образовываем пару с наблюдением отрицательного класса (рис. 10), у нас 12 пар, 12 раз наблюдение полжительного класса под номером 20 было проранжировано выше наблюдений отрицательного класса 12, 11, 10 и т.д. Записываем число 12 напротив наблюдения 20.

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

*Замечание*

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

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

## 11. Приемы работы с Gurobi

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

Чтобы запустить Gurobi в интерактвином режиме, следует в командной оболочке набрать `gurobi`

Сессия GUROBI

```
gurobi> m = read("./ikp_milp_problem.lp")
gurobi> m.optimize()
gurobi> vars = m.getVars()
gurobi> help(m)
# вывести 2-картежи целочисленных переменных с отличным от нуля значением
gurobi> [(var.varName, var.x) for var in vars if (var.x > 0) and (var.vType == "I")]
gurobi> m.write("res.sol")  # записать решение
```

## Список иллюстраций

**Отсортированные спрогнозированные вероятности положительного класса**

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

Рис. 10. Расчет ROC-AUC по формуле (1)

## Список литературы


1. *Лутц М.* Изучаем Python, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 1280 с.
2. *Бизли Д.* Python. Подробный справочник. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 864 с.

**Отсортированные спрогнозированные вероятности положительного класса**

**случай равенства вероятностей**

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

**Считаем количество отрицательных ниже каждого наблюдения положи-тельного класса**

Рис. 11. Расчет ROC-AUC по формуле (1) для случая равных вероятностей принадлежности экземпляра положительному классу