

Предсказание стоимости акции

Команда: Гарасев Никита
Тишин Роман

Куратор: Ижеев Сергей

Объект

- В качестве объекта исследования была выбрана банковская отрасль, а именно 4 банка США:
- Bank of America
- Citi Bank
- JP Morgan Bank
- Wells Fargo & Company

Сбор данных


Первоначально были собраны данные с сайта investing.com.

К информации о ценах акций была добавлена мета-информация о безработице, курсе доллара и прочее.

Wells Fargo & Company (WFC)

 Нью-Йорк ▾ Цена в USD · Предупреждение

55,05 -0,54 (-0,97%) ▼

 Закрит · 02/03

 Перед открытием 55,05 0,00 (0,00%) 12:49:37

Обработка данных

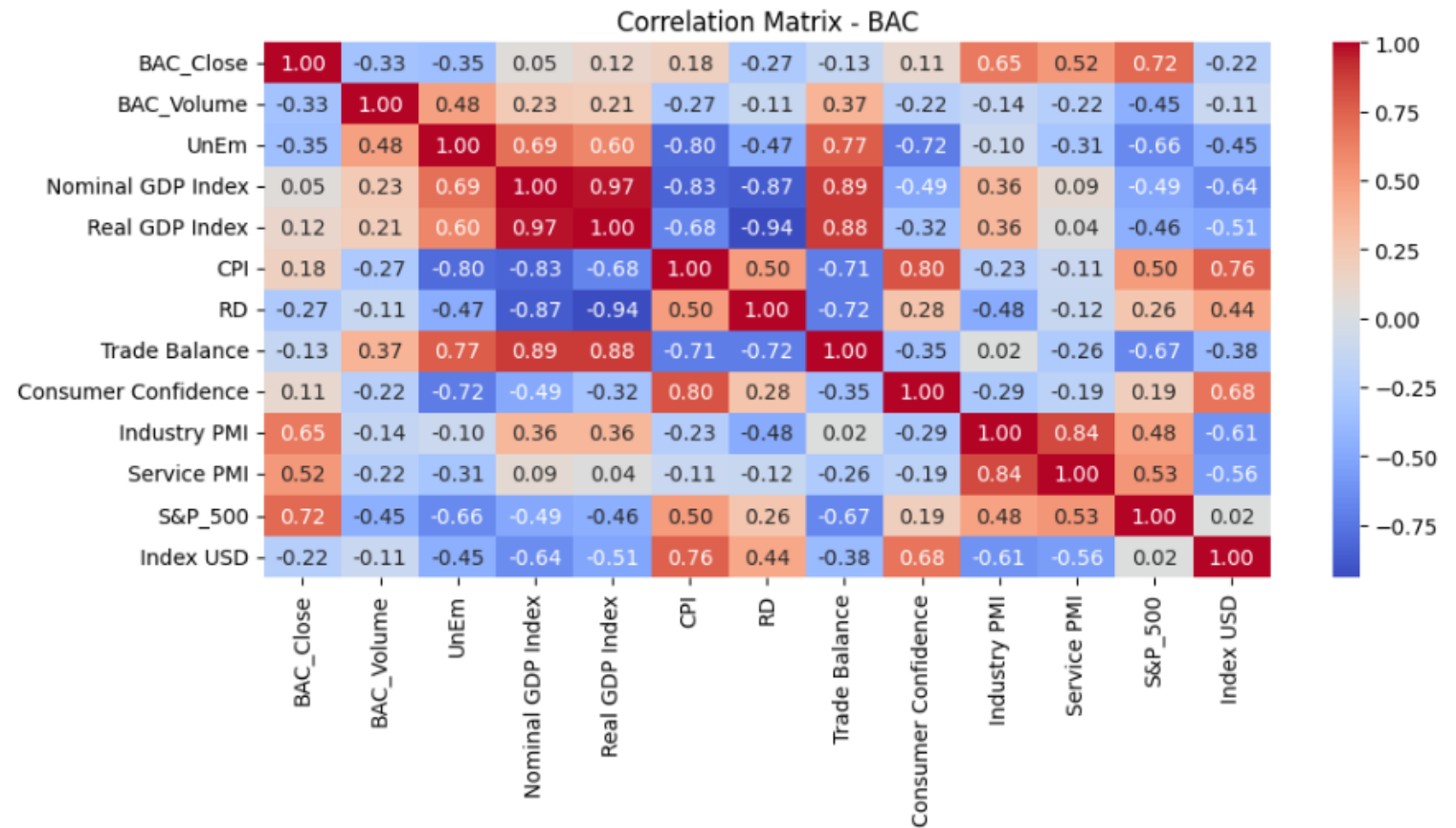
Затем данные были приведены к единому формату, и собраны в один датасет.

От каждого банка были добавлены: объем и цена закрытия на каждую дату (наши таргеты).

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 965 entries, 0 to 964
Data columns (total 20 columns):
#   Column                Non-Null Count  Dtype
---  -
0   date                  965 non-null   datetime64[ns]
1   WFC_Close             965 non-null   float64
2   WFC_Volume            965 non-null   float64
3   JPM_Close             965 non-null   float64
4   JPM_Volume            965 non-null   float64
5   Citi_Close            965 non-null   float64
6   Citi_Volume           965 non-null   float64
7   BAC_Close             965 non-null   float64
8   BAC_Volume            965 non-null   float64
9   S&P_500               965 non-null   float64
10  Index USD             965 non-null   float64
11  UnEm                  965 non-null   float64
12  Nominal GDP Index     965 non-null   float64
13  Real GDP Index        965 non-null   float64
14  CPI                   965 non-null   float64
15  RD                    965 non-null   float64
16  Trade Balance         965 non-null   float64
17  Consumer Confidence    965 non-null   float64
18  Industry PMI           960 non-null   float64
19  Service PMI           960 non-null   float64
dtypes: datetime64[ns](1), float64(19)
memory usage: 158.3 KB
```

Обработка данных

Были заполнены
пропуски, и
сгенерирована
предварительная
матрица
корреляции
признаков.



Обработка данных

С помощью теста Дики-Фуллера были проверены все данные на стационарность.

Далее все признаки были приведены к стационарным с помощью рекурсивного дифференцирования.

Это необходимо для того, чтобы избавиться от мнимой корреляции, из-за тренда параметров.

```
def data_adf_transform(df):
    black_list = ["date"]

    # пройтись по всем столбцам
    for column in df.columns:
        # пройти мимо столбцов из блек листа
        if column in black_list:
            continue

        print(column)
        i = 1
        # дифференцировать пока не будет стационарным
        while not check_adf_test(df[column]):
            print(i)
            i += 1
            diff_values = df[column].diff()
            diff_values.fillna(diff_values.iloc[1], inplace=True)
            df[column] = diff_values

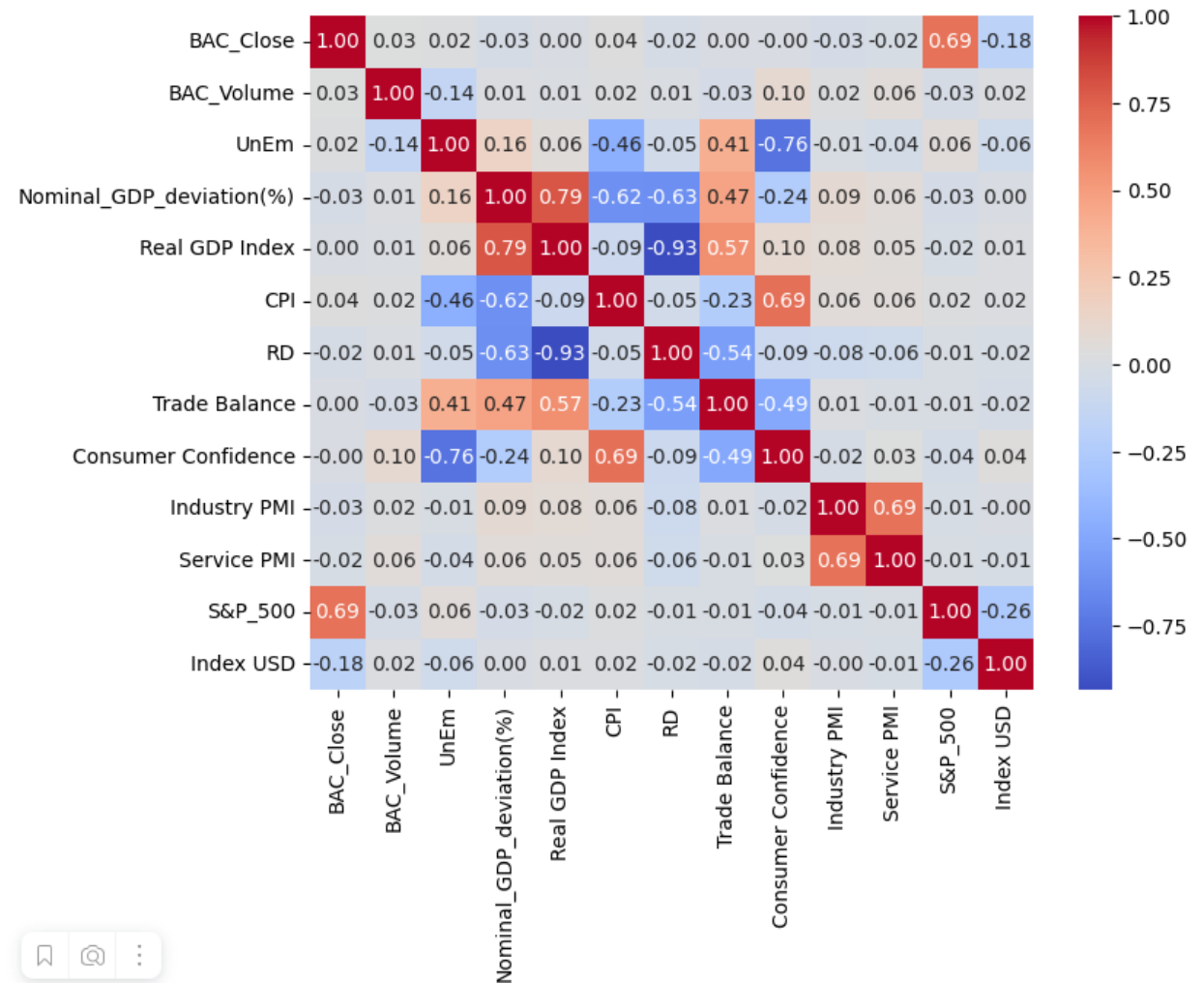
    return df

test_df = data_adf_transform(union_df.copy())
```

Обработка данных

В результате чего матрица корреляции после преобразований стала выглядеть следующим образом.

В ходе дальнейших исследований, было выявлено, что признак "Rate Desicion" является константным, а следовательно никак не повлияет на обучение моделей.



Генерация новых признаков

Для генерации новых признаков была выбрана библиотека `tsfresh`, которая предоставляет несколько генераторов наборов признаков.

```
def _extract_features(df):  
    return extract_features(  
        df,  
        column_id='segment',  
        column_sort='timestamp',  
        default_fc_parameters=EXTRACTION_SETTINGS  
    )
```


Преобразование датасета



В качестве фреймворка для работы с моделями был выбран фреймворк - etna.

Для этого необходимо было преобразовать наш pandas датасет к датасету временных рядов фреймворка etna.

```
def _correct_df(df):
    feature_columns = df.columns.difference(['timestamp',
                                             'WFC_Close',
                                             'JPM_Close',
                                             'Citi_Close',
                                             'BAC_Close']).tolist()

    target_columns = ['WFC_Close', 'JPM_Close', 'Citi_Close', 'BAC_Close']

    df_corrected = pd.melt(df, id_vars=['timestamp'] + feature_columns,
                           value_vars=target_columns, var_name='segment',
                           value_name='target')

    return df_corrected

def _create_ts(df):
    ts_dataset = TSDataset.to_dataset(df)
    ts = TSDataset(ts_dataset, freq='B')

    return ts
```

Обучение наивной модели



В качестве первой модели была выбрана наивная модель.

```
def naive(ts):
    train_ts, test_ts = ts.train_test_split(test_size=HORIZON)

    model = NaiveModel(lag=30)

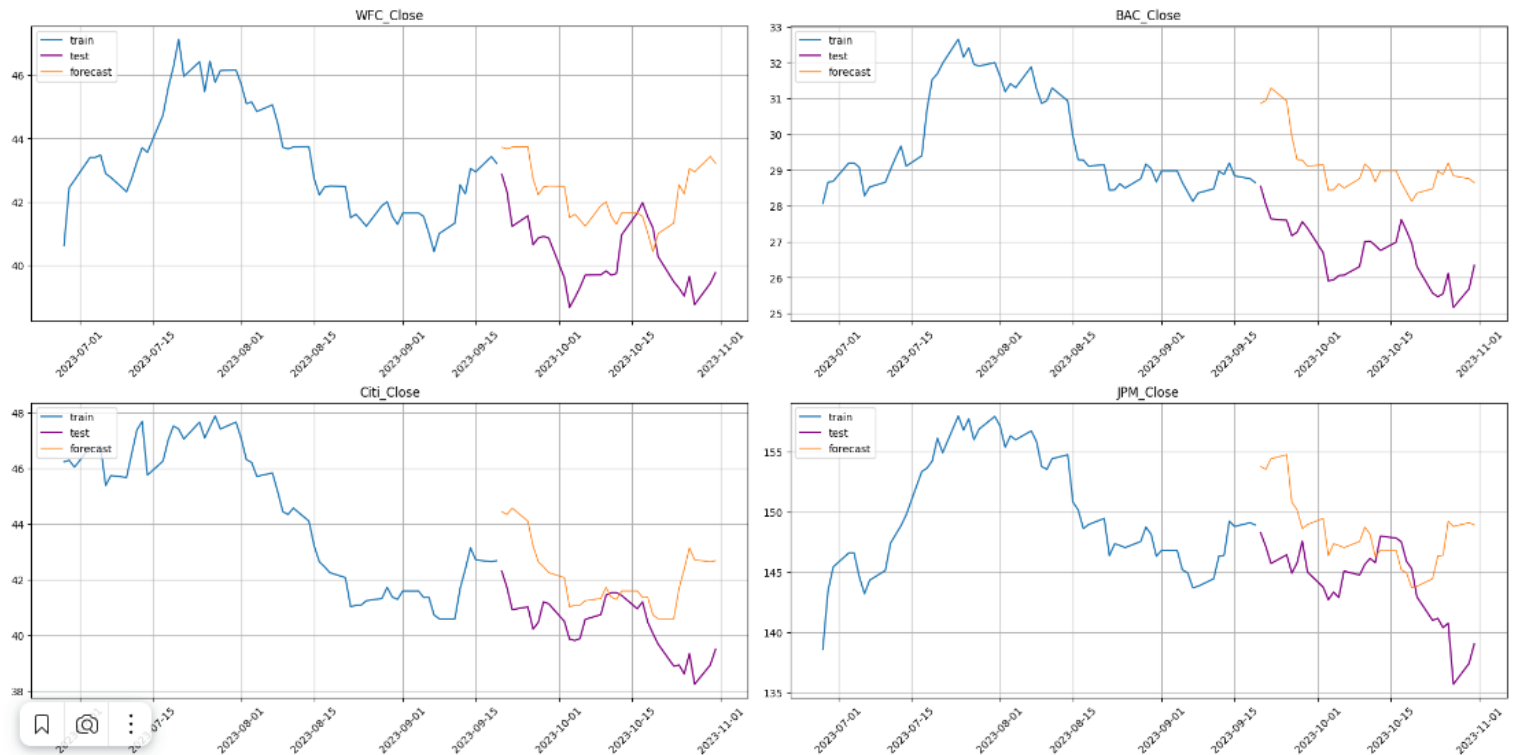
    # Define a pipeline
    pipeline = Pipeline(model=model, horizon=HORIZON)

    metrics_df, forecast_df, fold_info_df = pipeline.backtest(
        ts=ts, metrics=[SMAPE(), MAE(), MAPE()], n_jobs=10
    )
    print(metrics_df.groupby(['segment']).mean())

    pipeline.fit(train_ts)
    forecast_ts = pipeline.forecast()
    plot_forecast(forecast_ts=forecast_ts, test_ts=test_ts, train_ts=train_ts, n_train_samples=60)
```

Результаты наивной модели

Наивная модель показала нормальный результат. К началу предсказываемой области предсказания расположены ближе к тестовым данным.



segment	SMAPE	MAE	MAPE	fold_number
BAC_Close	6.570737	1.935400	6.770824	2.0
Citi_Close	5.141992	2.314933	5.327135	2.0
JPM_Close	5.353490	7.640133	5.294206	2.0
WFC_Close	6.318600	2.653667	6.404027	2.0

Обучение модели AutoARIMA



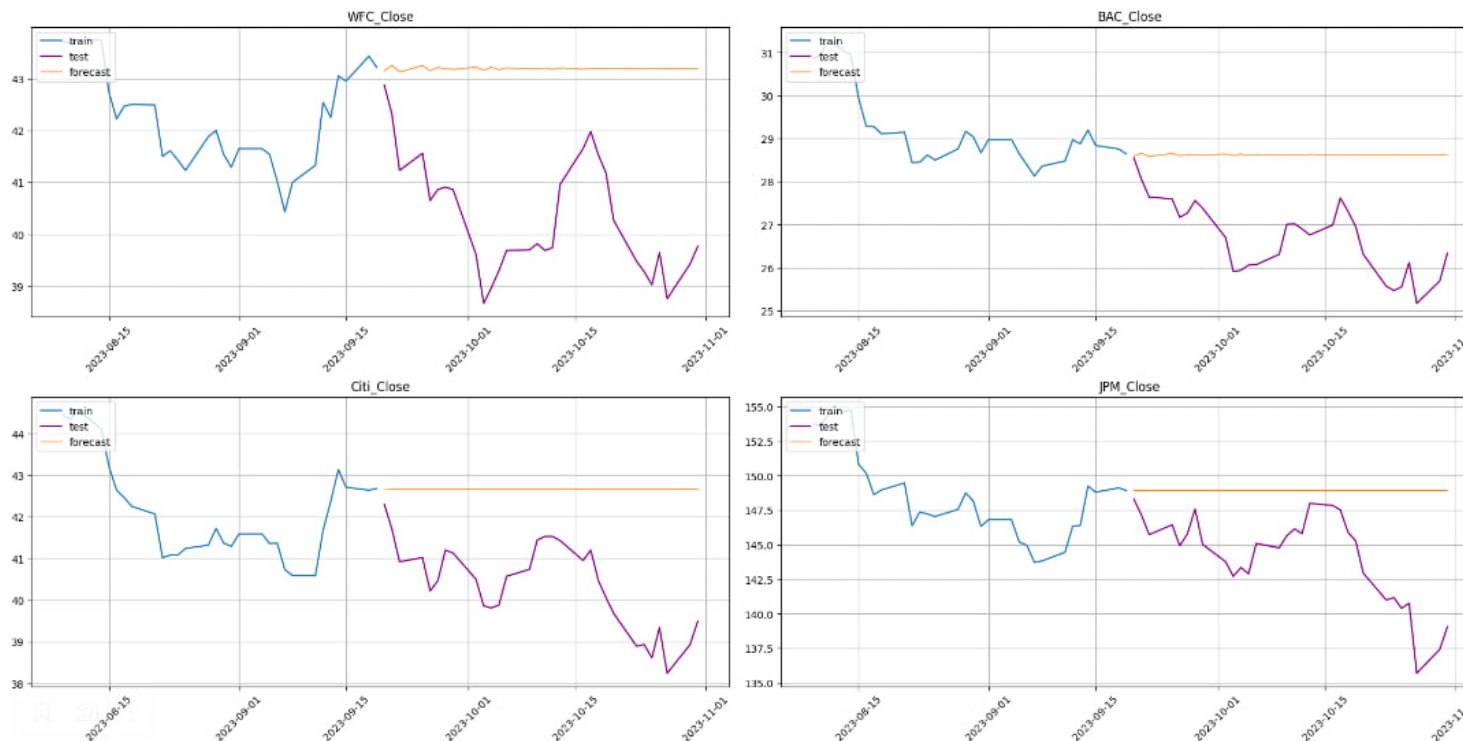
Следующая модель для обучения - AutoARIMAModel

```
def arima(ts):  
    train_ts, test_ts = ts.train_test_split(test_size=HORIZON)  
  
    autoarima_model = AutoARIMAModel()  
    pipeline = Pipeline(model=autoarima_model, horizon=HORIZON)  
  
    metrics_df, forecast_df, fold_info_df = pipeline.backtest(  
        ts=ts, metrics=[SMAPE(), MAE(), MAPE()], n_jobs=10  
    )  
    print(metrics_df.groupby(['segment']).mean())  
  
    pipeline.fit(train_ts)  
    forecast_ts = pipeline.forecast()  
    plot_forecast(forecast_ts=forecast_ts, test_ts=test_ts, train_ts=train_ts, n_train_samples=60)
```

Результаты модели AutoARIMA



Данный момент остался загадкой. Почему по цифрам все лучше чем у наивной модели, а по графикам мы предсказываем только на 1 день вперед (хотя $\text{lag} = 30$ выставлен).



	SMAPE	MAE	MAPE	fold_number
segment				
BAC_Close	5.791058	1.673995	5.808026	2.0
Citi_Close	4.065619	1.793008	4.133011	2.0
JPM_Close	5.174156	7.413530	5.086601	2.0
WFC_Close	6.743407	2.782365	6.658970	2.0

Обучение модели Prophet

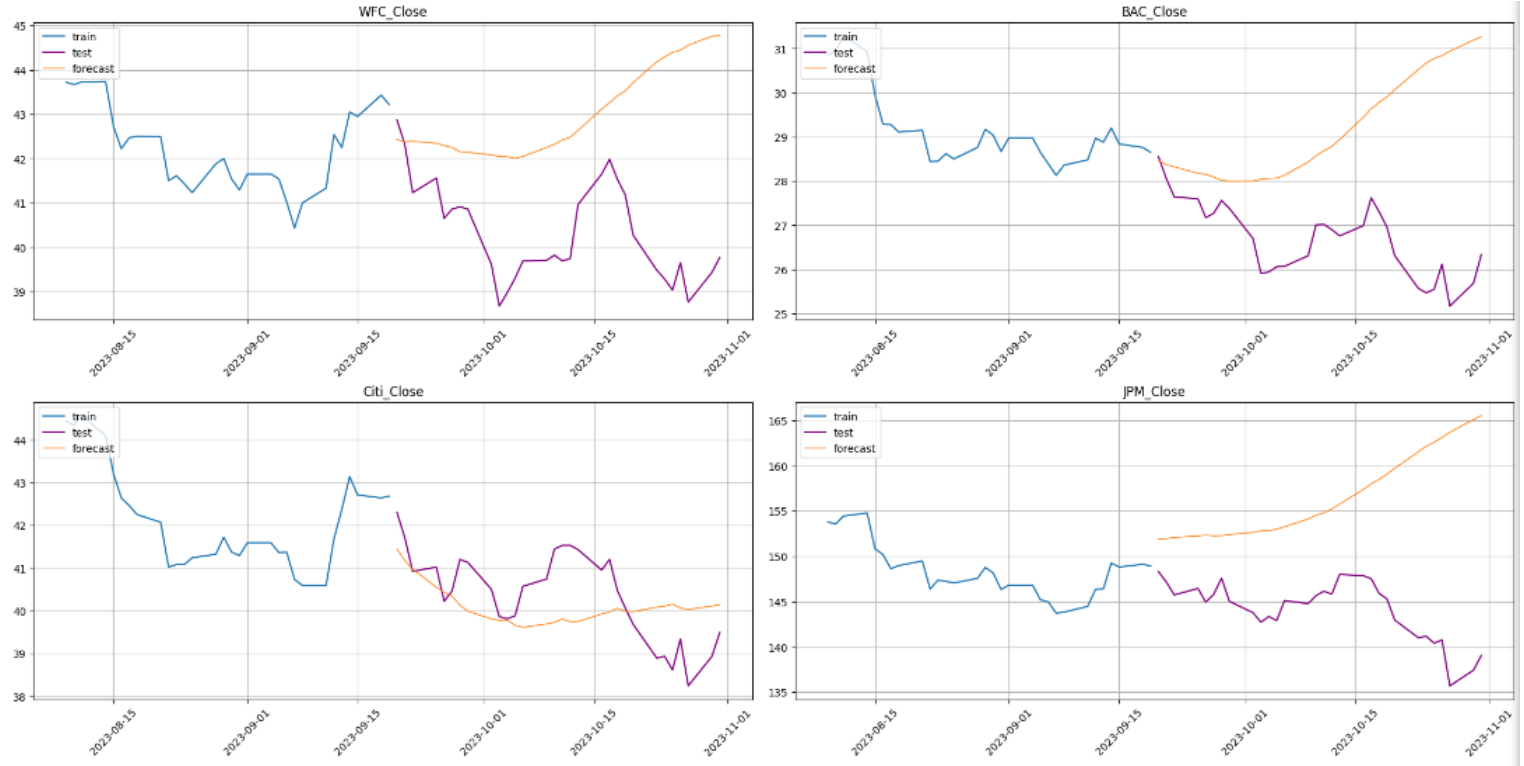


Следующая модель для обучения - ProphetModel

```
[90]: def prophet(ts):  
    train_ts, test_ts = ts.train_test_split(test_size=HORIZON)  
  
    prophet_model = ProphetModel()  
    pipeline = Pipeline(model=prophet_model, horizon=HORIZON)  
  
    metrics_df, forecast_df, fold_info_df = pipeline.backtest(  
        ts=ts, metrics=[SMAPE(), MAE(), MAPE()], n_jobs=10  
    )  
    print(metrics_df.groupby(['segment']).mean())  
  
    pipeline.fit(train_ts)  
    forecast_ts = pipeline.forecast()  
    plot_forecast(forecast_ts=forecast_ts, test_ts=test_ts, train_ts=train_ts, n_train_samples=30)
```

Результаты модели Prophet

Данная модель без каких либо настроек и трансформаций не показала должный результат, хотя по показателям выглядит не плохо.



segment	SMAPE	MAE	MAPE	fold_number
BAC_Close	7.789928	2.211883	7.646612	2.0
Citi_Close	6.413228	2.944044	6.636153	2.0
JPM_Close	6.301839	9.299269	6.410339	2.0
WFC_Close	6.826897	2.797316	6.633459	2.0

Обучение модели Catboost



Следующая модель для обучения - Catboost. Для обучения этой модели необходимо произвести несколько трансформаций для улучшения качества полученных результатов.

```
def catboost(ts):
    train_ts, test_ts = ts.train_test_split(test_size=HORIZON)

    catboost_model = CatBoostMultiSegmentModel(iterations = 750, depth = 5, learning_rate = 0.001)

    stl = STLTransform(in_column="target", period=30, model="arima")
    anomaly = DensityOutliersTransform(in_column="target", window_size=5, distance_coef=2.5)
    seg = SegmentEncoderTransform()
    lags = LagTransform(in_column="target", lags=list(range(HORIZON, 365)), out_column="lag")

    transforms = [stl, seg, lags, anomaly]
    pipeline = Pipeline(model=catboost_model, transforms=transforms, horizon=HORIZON)

    metrics_df, forecast_df, fold_info_df = pipeline.backtest(
        ts=ts, metrics=[SMAPE(), MAE(), MAPE()], n_jobs=10
    )
    print(metrics_df.groupby(['segment']).mean())

    pipeline.fit(train_ts)
    forecast_ts = pipeline.forecast()
    plot_forecast(forecast_ts=forecast_ts, test_ts=test_ts, train_ts=train_ts, n_train_samples=100)

    return forecast_ts
```


Неудачный опыт

Библиотека `tsfresh` предлагает несколько наборов для генерации признаков, некоторые из которых генерировали по 9 тыс. признаков для нашего датасета для каждого из таргетов. Однако в ходе экспериментов, после многочисленных ошибок, предположительно выявлено, что для обучения модели `catboost` невозможна ситуация, когда число признаков больше числа строк в датасете. Хотелось бы узнать об это раньше и вернуть потраченное время). Пришлось взять набор с минимальным количеством признаков.

Снижение размерности

Для снижения размерности датасета, были убраны все константные или близкие к константе признаки. В дальнейших планах: попробовать набор из большего числа признаков и избавиться от ненужных. Для этого найти хорошо попарно скоррелированные признаки и удалить тот, что оказывает меньший эффект на целевые переменные.

Планы на будущее

Доделать снижение размерности. Затем разработать телеграмм бота, который мог бы предсказывать цену закрытия для каждого из банков на заданный ему день.