ChatGPT

指数OHLCデータ統合の実装計画

ステップ1: 指数OHLCデータの取得

- **J-Quants API利用**: まず、J-Quantsの「指数四本値」API（`/indices`エンドポイント）を用いて、統合対象とする各種指数のOHLCデータを取得します。
- API仕様では、指数コード（例: `0000`=TOPIXや`0040`=水産農林業指数など）または日付を指定してデータを取得可能です。コードごとの全期間や、日付レンジ指定もできます（`from`〜`to`） [1]。
- **対象指数の選定**: 既存データセットに追加する指数群を決めます。最低限、全市場指標のTOPIX (コード0000)は既に扱っていますが、それに加えて**33業種別指数**（0040〜0060）、**スタイル指数**（バリュー8100/グロース8200等）、**規模別**（大型=Core30(0028)、小型=Small(002D)など）、**市場区分**（プライム0500、スタンダード0501、グロース市場0502）、**REIT指数**（総合0075とセグメント8501〜8503）などを含めます。
- **Fetcherの拡張**: `JQuantsAsyncFetcher`クラス等に、新たに**複数指数のOHLC取得メソッド**を実装します。例えば`fetch_indices_data(session, start_date, end_date, codes=[...])`のように、指定した指数コードリストをループしてそれぞれのOHLC時系列を取得し、PolarsのDataFrameに統合する形です。既存のTOPIX取得実装（`fetch_topix_data`）を参考にするとよいでしょう [2] [3]。
- **ローカルキャッシュ**: 大量の指数データを毎回API取得すると時間がかかるため、一度取得したらParquetに保存し、次回以降は必要に応じて再利用できるようにします（TOPIXでは`--topix-parquet`指定でオフライン利用可能な設計でした [4]）。同様に例えば`output/indices_history_<日付範囲>.parquet`として保存し、存在すれば読み込むようにすると効率的です。

ステップ2: 指数OHLCデータから特徴量計算（単指数内）

- 取得した全指数のOHLCデータ（各行にDate, Code, Open, High, Low, Closeを含むDataFrame）に対し、**Polars**を用いて各種特徴量を計算します。これは**各指数ごとの時系列**に対して行い、銘柄コード間で混ざりがないよう注意します（Polarsの`over("Code")`機能を活用）。
- **リターン系特徴量**: 前日終値に対する対数リターンや累積リターンを算出します（例: `r_1d = log(C_t / C_{t-1})`、`r_5d = log(C_t / C_{t-5})`など）。日中とオーバーナイトの分離も行い、`r_oc = log(C_t/O_t)`（当日寄与）、`r_co = log(O_t/C_{t-1})`（前日からの窓開け）を計算します。これらはモデルのラベル（将来リターン）作成にも連動するため、**未来データは使わず**過去のみで算出します。各リターンは銘柄別にシフト演算で計算し、Polarsでは`pl.col("Close").shift(1).over("Code")`等で前日終値を取りつつ計算します [5] [6]。
- **ボラティリティ・レンジ系**: OHLCから算出できる各種ボラティリティ指標を追加します。例として：
- **True Range (TR)**: `max(High - Low, |High - C_{prev}|, |Low - C_{prev}|)`を求め、14日指数平滑移動平均で**ATR14**を算出します。
- **パーキンソンのボラ**: $(\ln(High/Low))^2 / (4\ln 2)$ の形で日次分散を推計し（高値・安値のみ利用）、必要なら年率換算します。
- **Garman–Klass**: 高安と始終値を用いた分散推定量、**Rogers–Satchell**: トレンドに頑健な分散推定量等も計算可能です。
- **レンジ比率**: `(High - Low) / Close`はその日の値幅割合として特徴量になります。
これらも各Codeごとに計算し、Polarsでrollingやewmを使って実装します。例えばATRは`df.with_columns(pl.col("TR").ewm_mean(span=14).over("Code").alias("ATR14"))`のように実装できます。
- **トレンド・オシレーター系**: 短期・長期の移動平均を計算し、価格との乖離を特徴づけます。例：20日単純移動平均**SMA20**や50日EMA等を求め、`price_to_SMA20 = Close / SMA20`や、ボリンジャーバ

ンド系の**%b**（位置）や幅も算出可能です。また、`golden_cross`のように50日線が200日線を上回ったかのフラグも作れます。

- Polarsでは`rolling_mean`や`rolling_std`をCode別に適用し、例えば`pl.col("Close").rolling_mean(20).over("Code")`で20日平均を計算します。結果を用いて`(Close - SMA20)/STD20`のZスコア化なども行います。
- **ショック・テールリスク系**: 過去60日など長めの窓で**リターンのZスコア**を算出し、極端な値動きかどうかを見る指標を追加します（例: `z_r1d_60 = (r_1d - μ_{60})/σ_{60}`）。また、**半分散**（下方だけの標本分散）や、連続するドローダウン指標（Ulcer Index）等も必要に応じ計算できます。
- **実装ポイント**:
- 計算は**すべて銘柄Code単位**で行い、異なる指数間でローリング計算が混ざらないようにします。これはPolarsで`.over("Code")`指定することで保証します。また、各指数のデータ開始日が異なるため、`rolling_mean`や`ewm_mean`には適切に`min_periods`を指定し、最初の数日に不完全な計算が入るのを防ぎます。
- 具体的な実装例として、Polarsコードの擬似コード:

```
df = df.with_columns([
    pl.col("Close").shift(1).over("Code").alias("C_prev"),
    (pl.col("Close")/pl.col("C_prev")).log().alias("r_1d"),
    (pl.col("Close")/pl.col("Close").shift(5).over("Code")).log().alias("r_5d"),
    # ... 他リターン
    pl.max_horizontal([
        pl.col("High")-pl.col("Low"),
        (pl.col("High")-pl.col("C_prev")).abs(),
        (pl.col("Low")-pl.col("C_prev")).abs()
    ]).alias("TR"),
    pl.col("TR").rolling_mean(14, min_periods=5).over("Code").alias("ATR14"),
    # ... 移動平均・STD計算
    pl.col("Close").rolling_mean(20).over("Code").alias("SMA20"),
    pl.col("Close").rolling_std(20).over("Code").alias("STD20"),
    ((pl.col("Close") - pl.col("SMA20"))/(pl.col("STD20")+1e-12)).alias("z_close_20"),
    # ... その他特徴量
])
```

- **リーク防止**: これら特徴量はすべて時点tまでのデータから計算し、未来の値を参照しない設計です（モデル学習時にはt時点の特徴で$t+1$以降の予測を行う）。例えば学習用ラベルの1日後リターンは別途`shift(-1)`で作りますが、特徴量側はシフトしていません。

ステップ3: 指数間の相対・横断特徴量の付与

単一指数内の特徴量に加え、**複数指数間の関係性**から得られる特徴量も計算・統合します。市場のローテーションや相対的強さを捉えるために、以下のような処理を行います。

- **ベンチマーク相対指標**: 各指数について、あらかじめ定めた**ベンチマーク指数**に対する相対値を算出します。
- 例えば**TOPIX (0000)**を共通ベンチマークとし、すべての指数のリターンやボラとの差分・比率を計算します（TOPIX自身には不要）。セクター指数やスタイル指数など、多くは市場全体を代表するTOPIXと比較することで、相対的な強弱が分かります。
- 実装としては、指数データフレームを自分自身と結合させ、Dateでマッチしつつベンチマークの Code同士を突き合わせます [7]（Polarsでは同日付で結合後、`r_5d - r_5d_bench`等を計算）。例えば

`rel_r_5d = r_5d(index) - r_5d(bench)`、`rel_vol = vol_20d(index) / vol_20d(bench)` の列を追加します。**ベンチマップ**は次のように設定します：

- セクター指数やスタイル指数、規模指数など -> ベンチマークはTOPIX(0000)
- REITセグメント指数(8501等) -> ベンチマークはREIT総合指数(0075)
- マーケット区分指数(0501/0502) -> ベンチマークはプライム市場指数(0500) 等
  コード中では例えば `bench_code` 列を各指数に持たせ、そこに対応するベンチマークのコードを割り当てておき、`df.join(bench_df, on=["Date","bench_code"])` のようにマッチングさせます。

- **指数ペアのスプレッド特徴**: 2つの関連指数の差から市場環境を表す特徴量を作ります。具体例:
- **バリュー vs グロース**: バリュー指数(例: 8100)とグロース指数(8200)の例えば5日リターン差分を `spread_VG = r_5d(Val) - r_5d(Growth)` として計算します（期間は1日や20日など複数考えられます）。この値が正ならバリュー株優位の地合いを意味します。
- **大型 vs 小型**: 大型株指数と小型株指数のリターンスプレッド（例: TOPIX Core30 (0028) と Small (002D)の差）。これにより資金が大型株に向かっているか小型株に向かっているかが示唆されます。
- **市場区分**: プライム市場 vs スタンダード市場 [8] 、プライム vs グロース市場のリターン差も計算可能です。
- **REITセグメント**: オフィス系REIT vs 住宅系REITなど、REIT内で資金フローがどこに偏っているかを見るスプレッドを算出します。
- これらスプレッドは**日次の単一値シリーズ**として得られ、**全銘柄共通の市場環境特徴量**として扱います。実装上は、指数DataFrameからそれぞれ該当コードの行を同日に取り出し、計算した差分を新たなDataFrame（Dateとスプレッド値）にします。その後で株式データに日付で結合し全銘柄に付与すればよいでしょう。例えば、`spread_VG` 列は全銘柄同日の値が同一になります。
- **横断的な強弱指標**: 同じカテゴリの指数群どうしを**同日内で比較**した指標も有用です。
- 例：**セクター相対強度**として、その日の各業種指数の騰落率をランキングし、各業種が全33業種中何位か（%ランク）を算出します。これにより「今日は〇〇業が市場で最も強かった」のような情報を特徴量化できます [9] （Sector指数DataFrame内で `rank_pct(r_1d) over Date` を計算し、結果を後で株式に結合）。
- **Zスコア**: 各指数の中期モメンタムが同業種内でどれだけ異常かを見るため、例えば**業種指数間で**20日リターンを標準化したスコアを算出します（各Dateで全業種のmom20平均と標準偏差を求め、その日の各業種mom20を引いて割る）。スタイル指数間（バリューvsグロース群、規模別指数群）でも同様です。
- **マーケットブレッド**: 当日、主要指数のうち何割が50日線を上回っているか等も算出できます（指数レベルの「上昇銘柄比率」のようなもの）。例えば全33業種指数のうち終値＞50日平均の割合を `share_above_ma50` として計算すれば、市場全体の上昇トレンド広がりを示す指標となります。
- **相関・β系**: 各指数と市場との**直近相関**や**β値**を算出します。例えば60日窓で各指数のリターンとTOPIXリターンの相関係数、共分散からβ（= Cov/Var）を計算可能です [10] [11] 。ただし個別株ほど厳密に計算しなくても、指数同士は性質がはっきり異なる場合も多いため、必要に応じて実装します（例えば全セクター指数の中でどれが市場と非連動かを見る指標など）。Polarsでのβ実装例は既存の株式×TOPIXクロスでも行われており、60日ローリングの共分散と分散からβ算出、20日版との併用で安定化する処理が参考になります [12] [13] 。

ステップ4: コード差異・イベント対応の調整

- **指数コード体系の統一**: データ取得した指数の中には、途中で名称変更や継続性に注意が必要なものがあります。典型例が**マザーズ指数（コード0070）**で、2023/11/6から「グロース市場250指数」に改称されています【ユーザ提示文】。幸いコード自体は継続して0070が使われているようですが、過去のマザーズ指数時代と連続したものとして扱えるよう、**canonical_code**の概念を設けます。必要ならコードを置換/統合する処理を入れ、（今回の場合はそのまま0070でよいでしょう）。今後もし指数コード体系が変更されても連続性を保てるよう準備します。
- **指数ファミリー定義**: 各指数をカテゴリー別にグループ化しておきます（前述の**family**マップ）。例えば「SECTOR」（0040〜0060全部）、「STYLE_VALUE」（8100等バリュー指数群）、

「STYLE_GROWTH」（8200等グロース指数群）、「REIT_SEG」（8501〜8503）、「MARKET_INDEX」（プライム/スタンダード/グロース市場指数）などです。これにより、同ファミリー内で横断比較する際にコードリストを持たなくても、family属性でフィルタできるようになります。Polars処理では、事前に各Codeにfamily属性を付与する列を用意し、例えば**セクター内順位**計算時に`df.rank().over(["Date","family"])`のように使います（family="SECTOR"のものだけでランク付け）[14]。

- **市場休業・異常日の処理**: 2020/10/01のように**取引停止**となった日は、当日の四本値が前日終値と同じ値でレンジが0になっています。この日は特殊要因なので、学習時に誤解しないよう特徴量側でケアします。具体的には:
- `range_pct`やATR等の**レンジ系指標**が0になるが、それは**極端な低ボラ**を意味するわけではないため、この日のそれら指標はそのままでは使わない方がよいです。対策として、この日だけ適用されるダミーフラグ列（例: `is_halt_20201001`）を設けてモデルに認識させる[5]か、あるいはこの日のレンジ系数値を欠損扱いにするなどが考えられます。ダミーを入れる場合はPolarsで`pl.when(Date=="2020-10-01").then(1).otherwise(0).alias("is_halt_20201001")`のように列追加します。
- **不足データ期間**: 一部指数は途中から算出が開始されたため、データが存在しない期間があります（例えばコード002F TOPIX Small500は2018年から、0503 JPXプライム150は2023年からなど）。これらは取得データ上、開始以前は行自体がないか、Polarsでrolling計算すると初期はNaNになります。**min_periods**指定でNaNは許容し、モデル入力時に欠損値として扱うか、必要に応じて補間せずそのままとします（欠損自体が「存在しなかった指数」情報とも言えます）。
- **データ型とソート**: Date列は適切に`pl.Date`型にキャストし、Codeでソートされた状態でrolling計算することで順序を保証します[15]。これら基本的なデータ整形も抜け漏れないようにします。

ステップ5: 既存パイプラインへの統合

- **パイプラインへのデータ投入**: 上記で取得・加工した指数特徴量データを、既存のMLデータセット構築パイプラインに統合します。大きく分けて:
- **指数データ取得のタイミング**: `run_full_dataset.py`のフローにおいて、TOPIX取得と同様に、株価データを処理する前段階で指数データを取得します。現在フローでは「Step 2」でTOPIXをAPIまたはローカルから取得し、特徴量に組み込んでいます[16][1]。これを拡張し、**複数指数**も必要なら同時取得するようにします。具体的には、TOPIX取得部分でJQuants API認証後、TOPIX以外の指数もまとめて`fetcher.fetch_indices(...)`で取得し、Polars DataFrameにします。
- **特徴量計算の呼び出し**: 取得した`indices_df`に対し、ステップ2・3のロジック（Polars処理）を適用する関数を呼びます。例えば新たに`build_all_index_features(indices_df) -> index_features_df`という関数を実装し、そこで上記のOHLC単体＋相対特徴量を計算します。これにより**Date x Code**で各種特徴量を持つ`index_features_df`が得られます。

- **株式データへの結合**: メインの株式データフレーム（df_base）に対し、指数由来の特徴量を結合します。結合方法は特徴量の種類によって異なります:

    - **市場全体・グローバル特徴**: TOPIX由来のmkt_列や、前述のスプレッド系指標（例: `spread_VG`, `spread_SL`など全銘柄共通の値を持つ特徴）は、Dateで結合*します。既にTOPIXについては`df.join(market_feats, on="Date", how="left")`でmkt_特徴を付与し、その後CrossMarketFeaturesでβやαを計算していました[7]。今回も、スプレッドやブレッドスなどの日付単位シリーズはDateキーで全行に付与できます。Polarsでは`df = df.join(spread_df, on="Date", how="left")`でOKです。
    - **セクター指数特徴**: 業種ごとに異なる値を持つ特徴（その業種のリターンや順位など）は、**Dateと業種キー**で結合します。具体的には、株式ごとの「業種コード/セクターID」をキーに、同じ業種の指数データを紐付けます[17]。

- 既存パイプラインでは `listed_info_df` から各銘柄の33業種区分を取得し、`add_sector_features` や `add_sector_series` で**業種別の平均リターン**や**ワンホット**を付与していました [18] [19]。今回はそれを**公式指数ベース**に置き換える形です。
- 実装として、まず `listed_info_df` に各銘柄コード → 対応する業種指数コードのマッピング列を追加します。例えば、33業種コードが「05: 食料品」であれば対応する指数コードは「0043」（東証業種別 食料品）という具合に、ルールに従って割り当てます（業種コード 01→0040, 02→0041,...の対応表を用意）。
- 次に、指数特徴DataFrame（index_features_df）から**業種指数のみ抽出**したDataFrameを用意します（family=="SECTOR"のもの）。これを株式DataFrameに対し、`df.join(sector_index_df, left_on=["Date", "SectorIndexCode"], right_on=["Date", "Code"], how="left")` のように結合し、業種指数の特徴量列（例えばその業種の `r_5d`, `z_close_20`, `cs_rank_r1d_family` 等）を各銘柄の行に持たせます。
- これで各銘柄行に「その銘柄の属する業種指数」の昨日までの状態（リターンやボラ、業種内での位置など）が付与され、個別株のパフォーマンスを説明する外生変数として利用できます。 [20] にあるように、例えば `rel_strength_5d`（5日相対強度）は従来マーケットに対するものでしたが、業種版として**その銘柄の5日リターン - 業種5日リターン**も同様に計算可能です。これは結合後に `pl.col("returns_5d") - pl.col("sector_r_5d")` のように列演算して新規特徴量にしても良いでしょう（既存コードでは `add_relative_to_sector` で類似計算をしていました [21]）。
- **スタイル/サイズ指数特徴**: もし銘柄ごとに「バリュー株/グロース株」や「大型/小型」の属性が取れるなら、同様にマッピングしてスタイル指数特徴を結合します。例えば、ユーザが企業財務指標等から銘柄をスタイル分類しているか、市場インデックスの採用銘柄リストがあれば、それを利用します。全銘柄一律にバリューorグロースに分かれるわけではない場合、スタイル指数は**市場全体の雰囲気指標**としてスプレッドで与えるだけでも十分です（全銘柄共通特徴としてすでに `spread_VG` を付与する方法）。
- **コード識別情報**: なお、株式と指数が混在するDataFrameにはしない想定ですが、仮に**指数自体も予測対象**に含めるなら（例: インデックス先物予測など）、同じテーブルに統合しCode列で区別する方法もあります。その場合、モデル入力時に株式と指数で分布が異なる特徴も混在するため、`Code` を表すカテゴリ埋め込みを使うなど対策が必要です。ただ、通常は指数は指数用データセットとして別ファイルにするか、株式データとは結合せず**日付キーで参照**するだけにとどめます。今回は株式データセットに指数由来特徴を結合するだけなので、指数コード自体は出力しません（紛らわしい場合、結合後にIndex側のCode列はドロップします）。

- **ビルダーへの組み込み**: 上記処理をパイプラインのビルダークラス（`MLDatasetBuilder` 等）にメソッドとして実装します。例えば、新しく `add_index_features(df, index_feat_df, listed_info_df)` を作り、そこで

- グローバル（日付単位）特徴の結合、
- 業種別特徴の結合、
- （必要ならスタイル別特徴の結合）、

を行うようにします。TOPIXの `add_topix_features` や `add_sector_series` 処理と類似した箇所にフックします。現在 `add_topix_features` はCrossMarketFeaturesGeneratorを使い、市場指数とのβやαを計算していました [22] [23]。今回の指数統合ではβ計算はひとまずTOPIXに対して各銘柄で行えば十分なので、クロス指標は既存のまま（TOPIXで）維持し、それ以外の指数特徴は単純結合のみ実施します（複雑な演算は前ステップまでにindex_features_df内で済ませている前提）。

- 実装時は既存の列名との衝突に注意します。例えば業種指数のリターンを株式側に付与する際、その列名 `r_5d` だと株式自身の5日リターン列と区別がつかないため、**接頭辞**を付けます。`sect_r_5d` や `sector_index_r_5d` などとリネームするか、Polarsの結合時に `suffix` オプションを利用します。事前に指数特徴計算時に列名を「指標カテゴリ付き」にしておくのも有効です（例: Codeが業種指数なら列名に `sect_` を付けて計算）。こうすることで、異なる家族の指数特徴同士や株式側既存特徴とコンフリクトせず、後から選択的に使いやすくなります。

- **カレンダー要因の付加**: （もし未実施であれば）曜日や月末などのフラグを追加します。ユーザ提示の仕様には曜日One-Hotや祝日明けフラグなどが含まれていました。パイプラインではすでに `enable_advanced_features` でカレンダー特徴を付ける仕組みがあるかもしれません。無い場合は、Dateから `weekday` を取り `dow` として0（月曜）〜4（金曜）を数値特徴量に、`is_month_end`（月末営業日か）をboolで持たせるなどします [24]。これらは指数・株式共通に影響するので、どのDataFrameで付与してもよいですが、一括して株式データに付与して構いません。

ステップ6: 動作確認と検証

- 実装後、**小規模な日付範囲**でパイプラインを実行し、期待通りに統合できているか検証します。例えば直近1年分程度で `run_full_dataset.py --enable-advanced-features --enable-sector-cs`（必要なら新設のフラグも指定）を動かし、出力のParquetを確認します。
- **指数特徴テーブルの検証**: 中間生成される `index_features_df`（または統合後のDataFrame）について、主要な列を確認します。Date, Codeごとにリターンやatrなどが正しく計算されているか、値がおかしなスケールになっていないか見ます。特にZスコアやランク系は-3〜+3程度に収まるはずですし、ATRやvolは%表記（0.x）か年率換算か単位系統一をチェックします。
- **結合結果の検証**: 株式データに指数特徴が付与された結果、例えばある日のある銘柄行について
  - TOPIX由来の `mkt_*` 列があり、値がその日のTOPIX特徴量と合致する（既存実装で検証済みのはず）。
  - 新規追加した `spread_VG` 等が全銘柄同日に同じ値を持つ（例: 全ての9/14付の行でspread_VG = X）。
  - 業種指数特徴（例えば `sect_r_5d`）がその銘柄の属する業種の指数5日リターン値になっている。他業種の銘柄とは異なる値を持つことを確認します。元の指数データと突き合わせて正確ならOKです。
  - 業種相対ランク `sect_cs_rank_1d`（仮）を付けたなら、同じ業種の銘柄はすべて同じ値を共有するはずなので、数銘柄分を目視で追い、一致を確認します（異なる業種なら当然別値）。これは全銘柄に対して**業種指数の強さを伝達**できている証拠です。
- **リークがないか**: 例えば**本日終値から計算した翌日リターン**など未来情報が混入していないかをチェックします。リターン系特徴は全て当日までの過去データで計算しており、モデルの教師目的変数（feat_ret_）は別途シフトで入れているため、ここでは問題ないはずです [25]。念のため、出力データの末尾何行か（最新日付付近）において、未来のはずの列がNULLで終わっていること（例えばfeat_ret_5dが直近4日はNULLになる等）を確認します。指数特徴についても、当日分まで*で計算しており翌日を見ていないことはロジック上担保されています。
- **特異日の確認**: 2020-10-01のデータ行について、`is_halt_20201001` が1になっているか、またその日のレンジ系特徴量が学習から除外されているかを確認します。例えば当日のrange_pctやATRが異常に低い値になっていれば、それがそのまま特徴に入っていないか（NaNにしたか）を見ます。問題があれば該当日の該当特徴をNaNにクリップするなど修正します。
- **コード連続性**: 0070番の指数データについて、2022年以降もしっかり値が入っているか確認します（改称時に抜けていないか）。Polars上はコードが同じなので特に問題なく連続しているはずです。念のため2023-11-06前後の日付の値動きに飛びがないか目視します。
- **パフォーマンス**: 全銘柄5年分のデータに対し、新たに指数データ（数十系列分）を結合することで処理時間やメモリが多少増加します。Polarsは高速ですが、特にRolling計算で60日以上の窓を多数やると負荷がかかります。必要に応じてウィンドウ幅や計算する指標の数を調整します（例えば高度なGKやRSボラまで全て使う必要がなければ絞る）。開発時にはメモリ使用状況もチェックし、問題あれば分割処理（例えばセクター指数とスタイル指数で別々に計算して結合）するなど対処します。

ステップ7: ドキュメンテーションと今後の拡張

- **仕様書の更新**: 新たに追加された特徴量について、社内ドキュメント（例えば `docs/DATASET.md` や特徴量一覧）に追記します。それぞれの列名の意味（例: `sect_r_5d`: 銘柄属する業種指数の5日リターン、`spread_VG`: バリューvsグロース5日リターンスプレッド 等）を明示し、チーム内で共有します。

- **利用方法の周知**: 今回統合した特徴量群により、**短期予測精度向上**や**レジーム検出**に役立つことを説明します [26] 。例えば、高ボラ期間を示すmkt_high_volフラグや、業種スプレッドの顕著化をモデルが捉えれば、リスクオン/オフやセクターローテーションを予測に織り込めるようになります。
- **フラグ管理**: パイプライン実行時に、新指数特徴を**任意で無効化**できるフラグが必要か検討します。現状`enable_sector_cs`（セクター横断特徴）というPhase2フラグが存在しているようなので [27] 、これを今回の統合に活かし、当該フラグON時に指数データを結合するよう条件分岐させます。将来的には常時使う前提ならデフォルト有効でも構いません。
- **次の段階**: 今後、指数データ統合を発展させて
- **セクター別特徴の高度化**: 今回は指数値そのものを用いましたが、必要に応じて業種ごとの株価分布から算出するオリジナル指標（例: 業種内でのPBR分位など）も併用検討します。ただし、市場指数だけでかなりの情報が得られるためまずは十分でしょう。
- **モデルへの組み込み**: モデル学習時には、**銘柄固有の癖**を捉えるために**Codeのembedding**を使うことも推奨されています（ユーザ提示のアイデアより）。これはデータ統合というよりモデリング部分ですが、特徴量として銘柄IDを入れる余地も含めデータセット設計してあります。例えば`Code`列をカテゴリー変数として残しておき、PyTorchやLightGBMで扱えるようにします。また**target_scale_code20**のような銘柄ボラに基づく重み付け変数も指数データでは計算していましたが（値が大きいほどボラ小）、株式にも応用して損失関数で使用可能です。こうした**データセット内の工夫**もドキュメント化します。

以上のステップを踏むことで、既存データセットに各種指数OHLC由来の豊富な特徴量を統合できます。これは**価格データだけ**を用いて多ホライズン予測や市場レジーム判定を行う上で強力な情報源となり、モデルの精度向上や分析の深みを増すことが期待できます [26] 。実装にあたっては`run_full_dataset.py`や関連ソースを適宜参照しつつ、上記計画に沿って開発を進めてください。

---

[1] [2] [3] [5] [17] [18] [19] [21] [24] [27] full_dataset.py
https://github.com/wer-inc/gogooku3/blob/44498856ccdadab02c35bb8708c5252ffa3d1d6e/src/pipeline/full_dataset.py

[4] [16] run_full_dataset.py
https://github.com/wer-inc/gogooku3/blob/44498856ccdadab02c35bb8708c5252ffa3d1d6e/scripts/pipelines/run_full_dataset.py

[6] [7] [14] [15] [22] [25] ml_dataset_builder.py
https://github.com/wer-inc/gogooku3/blob/44498856ccdadab02c35bb8708c5252ffa3d1d6e/scripts/data/ml_dataset_builder.py

[8] [20] [26] MARKET_FEATURES_IMPLEMENTATION.md
https://github.com/wer-inc/gogooku3/blob/44498856ccdadab02c35bb8708c5252ffa3d1d6e/docs/ml/MARKET_FEATURES_IMPLEMENTATION.md

[9] [10] [11] [12] [13] [23] market_features.py
https://github.com/wer-inc/gogooku3/blob/44498856ccdadab02c35bb8708c5252ffa3d1d6e/src/features/market_features.py