ChatGPT

Gogooku3プロジェクトへの改善提案と潜在的課題

コード構造とアーキテクチャの整備

現状評価: リポジトリは最近大規模なモジュール化リファクタリング（v2.0.0移行）を完了しており、`src/gogooku3/`以下にデータ処理・特徴量・モデル・グラフ・学習パイプライン等のパッケージが適切に整理されています [1] [2]。旧来のスクリプトベース構造から脱却し、依存関係管理や設定統合も改善されています。

改善提案:
- **互換レイヤーの整理:** 移行時に導入された**後方互換性モジュール**（`gogooku3.compat`など）や一時的に**隔離された不要コード**について、十分な検証期間を経た後に削除することで、コードの判読性と保守性を向上できます [3]。例えば、安全学習パイプライン内での旧パスfallbackインポート（`src.data.safety.cross_sectional_v2`等）は技術的負債になる可能性があり、移行が完了した現在では整理可能です。
- **コード分岐の統合:** ATFT-GAT-FANモデル関連コードが`src/atft_gat_fan/`配下に存在し、一部スクリプト（例:`scripts/train.py`）で旧パス参照をしています [4]。新構造では`gogooku3.models.atft_gat_fan.py`に実装が統合されているため、**参照パスの更新**や古いスクリプトの削除を行い、コードパスの一貫性を保つべきです。
- **モジュール化の徹底:** 機能追加時には極力**独立モジュール**として実装し、依存関係を明確化する現在の方針を継続してください [5]。例えば、大型モデル改善機能（EMA教師モデルや周波数ドロップアウト等）はフラグでON/OFF可能で既存コードに影響しない構成となっており（後方互換性維持）望ましいです [6] [5]。今後も機能増加による複雑性上昇には注意し、設定管理やドキュメント整備で対応すると良いでしょう。

データパイプラインの前提条件とモジュール性

現状評価: データパイプラインはJ-Quants APIからの**非同期並列取得（150並列）**に始まり、ETL処理・特徴量計算・正規化・ウォークフォワード分割・学習まで一連の流れがモジュール化されています [7] [8]。APIフェッチャはエンドポイント毎に独立し再利用可能で、型変換やバリデーション、エラーハンドリング（自動リトライ）も組み込まれているなど堅牢です [9] [8]。特徴量エンジニアリングでは数百の指標から選抜した約62種のテクニカル指標や信用残高データ統合など、網羅的な特徴量セットを構築しています [10] [11]。

潜在的課題:
- **データ範囲の固定・サバイバーシップバイアス:** 現状、**データカバレッジ98%以上**の銘柄（約632銘柄）のみに分析対象を絞る設計になっています [12]。このフィルタリングは欠損の多い銘柄を除外する利点がある一方で、上場・除外のあった銘柄を排除するためサバイバーシップバイアスを招く可能性があります。実際、最新の実行では5年分のデータで4,220銘柄・248列に拡張されており [13]、従来設定から大幅に銘柄数が増えています。**提案:** フィルタ基準（`MIN_COVERAGE_FRAC`等）を状況に応じて見直し、新規上場銘柄や期間途中の銘柄も含めて分析できるようにすることを検討してください。また、不足データを補完する仕組み（例えば他データ源からの補填や一部指標のフォールバック）を導入し、網羅する銘柄数を増やしてもモデル精度が損なわれにくいようにすると良いでしょう。
- **全量再計算の非効率:** 現在はフル期間のデータセットを一括生成・学習するフローが基本となっています。5年分の**全データパイプライン実行に約27分**を要した報告もあり [14]、毎回フル再計算するのは非効率です。**提案: 増分処理（インクリメンタルアップデート）の実装**です。新しい取引日や四半期決算など増分データ取得時に、その分だけ特徴量計算・追記していく仕組みを導入すれば、日次の更新時間を大幅に短縮できます（現在の課題リストにも「全再生成ではなく増分アップデートを実装」と明記されています [15]）。Dagster

によるオーケストレーション環境があるため、スケジュールに合わせて**日次で差分ETL→特徴量計算→モデル再学習**まで自動実行し、研究者は常に最新データで実験できる状態を目指すと良いでしょう。
- **パイプライン処理のスケーラビリティ:** Polarsによる高速処理やlazy評価により、約60万サンプル・145特徴量の処理が1.9秒で完了するなど高い性能が達成されています [16]。しかし一部でpandas-ta等を用いた計算がボトルネックとなっており（全体の中で~12分を占める部分あり）、より長期間・高頻度データを扱う際のスケーラビリティに課題が残ります [17]。**提案: 技術指標計算部の並列化・最適化**を進めてください。例えばマルチコア並列処理やNumbaによる高速化、C++実装の利用、または指標計算自体をPolarsやPyArrowで置き換える検討です。現在のTODOにも「テクニカル指標の並列処理を検討」とあり [18]、優先度高で取り組むと良いでしょう。また、**チェックポイント/レジューム機能**の実装も検討されています [19]。長時間処理が中断した際に途中から再開できれば、計算失敗時の手戻りを減らせます。例えばAPI取得→一時Parquet保存→特徴量計算→別ファイル保存、というようにステージ毎に出力を保持し、途中からリスタート可能にする仕組みです [19]。これは大規模データ処理や長時間学習時の信頼性向上に寄与します。
- **データ品質とエラー耐性:** 取得データに対するバリデーションや品質チェックは一応組み込まれていますが、さらなる強化余地があります。例えばフロー系特徴量のカバレッジが78.2%に留まるとの分析もあり [15]、欠損箇所や異常値への対処が課題です。Great Expectationsによるデータ品質検証スイートは用意されているようなので（ヘルスチェックで使用 [20]）、**データ品質テストの自動実行**や**新規データ到着時の鮮度チェック**（一定期間データ欠如時のアラート）を組み込むと良いでしょう [21]。また、API失敗時のリトライは実装済みですが、連続失敗時にスキップせず異常終了するリスクもあるため、**バックオフリトライやフォールバックデータソース**の検討、失敗部分を記録して後で補完する仕組みもあるとパイプライン全体の堅牢性が増します。

モデル学習のスケーラビリティと信頼性

現状評価: モデル学習基盤はLightGBMによるベースラインと、PyTorch Lightningを用いたATFT-GAT-FANモデルのトレーニングフローで構成されています。PyTorch Lightning + Hydra設定により、モデル定義から学習ループ、ロギングまで標準化され、早期停止やチェックポイント保存も組み込まれています [22] [23]。また**段階学習フェーズ**（PhaseTrainer）に対応しており、大規模モデルを徐々に学習させる工夫もあります [24]。モデル実装面では5.6百万パラメータ規模のTransformer+GATハイブリッドを採用しつつ、**mixed precision**訓練や**勾配チェックポイント**によるメモリ節約、**EMA教師モデル**や**Huber損失**等の安定化策も導入済みです [25]。さらに学習中の**OOM自動検知・復旧**や**W&B/TensorBoard連携**、**異常検知アラート**まで備えており、研究用途としては極めて充実した学習基盤になっています [26]。

改善提案:
- **大規模学習への拡張:** 現状はシングルノード（GPU）での学習を想定していますが、将来的にデータ量やモデル深度が増大した際には**分散学習**や**マルチGPU学習**も視野に入れるべきです。PyTorch Lightningは`Trainer`でdevicesやdistributed_strategyを指定するだけでDDPなどに対応できるため、マルチGPU環境でのスケーリングも検討してください（例えば8GPUでバッチを並列化することでエポック時間短縮が可能）。現在の16GBメモリ前提からモデル・データが増えると単機では限界があるため、**Apache SparkやHorovod**による分散データ並列も将来的な選択肢になります。ただしまずはLightningの機能で容易にスケールアウトできますので、必要に応じ設定項目を追加すると良いでしょう。
- **信頼性と再開性:** 学習中断からの**再開トレーニング**をより確実に行えるようにすることも重要です。LightningのModelCheckpointコールバックにより最新モデルが`.ckpt`に保存されているので、異常終了時にはそれを読み込んで`trainer.fit(..., ckpt_path=...)`で再開できる体制を整えてください。設定としてはHydraの設定項目やCLI引数で`--resume`を受け取り、該当checkpointを自動検出して読み込む処理を追加するイメージです（CIでの長時間学習やスポットインスタンス利用時にも有用です）。現在のスクリプトでも最終モデル保存と設定保存を行っています [27]が、途中経過の復元にも対応できると安心です。加えて、**学習途中の経過モニタリング**を強化することも検討してください。例えば各エポック終了後にモデルを検証し、一定閾値を下回ったら自動で学習打ち切りやパラメータチューニングモードに移行するなど、Long-runningなジョブの無駄を減らす工夫です。
- **ハイパーパラメータ最適化のパイプライン化:** Optunaを用いたチューニングスクリプトが存在しますが（例

えば `run_optuna_atft.py`）、これもLightning + Hydraに統合して再現可能な形にすると一貫性が増します。例えばHydraのSweeperやOptunaプラグインを使って、設定ファイル上で探索範囲を定義し自動探索できる仕組みをCIで回せると、性能向上を系統立てて進められます。
- **ベースライン評価の充実:** SafeTrainingPipeline内ではウォークフォワード分割したうち**最初のフォールドだけ**でLightGBMを訓練しICを算出しています [28] [29]。高速化のための措置ですが、モデル性能評価としては全フォールド平均を取る方が安定します。可能であれば自動でKフォールドすべてでスコア集計するオプションや、少なくとも重要指標（IC, RankIC）の推定分散を算出する仕組みを検討してください。特に本番モデルとの比較や特徴量重要度分析時に、有意な差かどうかを判断する助けになります。
- **追加モデルへの対応:** 現在のモデルアーキテクチャは先進的ですが、将来的に他手法（例えばGraph Neural Network単独モデルやTransformer改良版など）を試す場合も出てくるでしょう。その際、新モデルクラスを追加して簡単にパイプラインに組み込めるよう、抽象クラスやインターフェースを用意しておくと拡張が容易になります。幸い `gogooku3.models` 以下にBaselineとATFTが分かれており、トレーナーもLightning標準に乗せているため、新モデルの `LightningModule` さえ実装すれば共通トレーニングループで動く構造です。今後もこの設計を維持し、**モデルの差し替え試験**を迅速に行えるようにしておいてください。

評価手法とデータバイアスの検証

現状評価: モデルの性能評価指標としては主に情報係数（IC）および順位情報係数（RankIC）が採用されています。ベースラインモデルでも各予測期間（1日後、5日後など）の平均IC/RankICを算出し、コンソールに出力しています [29]。また、モデルドキュメントでは**目標とするシャープレシオ**やPredictionの分散比率なども挙げられており、実運用上のリスク・リターン指標への意識も見られます [30] [31]。ウォークフォワード検証と20日エンバーゴにより、時系列情報リークを防いだ形で汎化性能評価を行う工夫もなされています [32] [33]。

改善提案:
- **評価指標の多様化:** 金融モデルの評価としてIC/RankICは有用ですが、モデルの**経済的有効性**を測る指標も検討してください。例えば、単純な**ポートフォリオシミュレーション**による累積リターンやシャープレシオ測定、あるいはヒット率・トップランキング精度などです。現在MLflowやW&Bでログを取っているようなので、そこにカスタム指標として実装しておくと、モデル改善がどの程度実際の利益に繋がるか定量化できます。さらに、モデルの**安定性指標**（期間別のIC推移、年初来どのくらい性能が変動したか等）も追跡すると良いでしょう。期間ごとのサブグループ分析を行えば、市場レジーム変化（例: ボラティリティ高騰期 vs 安定期）でモデル性能が極端に偏っていないかチェックできます。
- **バイアスリスクの分析:** データやモデルに起因する系統的バイアスの有無を調べることも重要です。例えば、モデルの予測エラーが特定の業種や時価総額レンジに偏っていないかを検証してください。セクター別の性能を比較したり、小型株 vs 大型株でのRankICを分けて計算するなどして、もし偏りが見られる場合は入力特徴量へのセクター情報追加（すでにセクターエンリッチメント機能あり）やモデルのアーキテクチャ上でその偏りに対処する必要があります。幸い最新バージョンではセクターコードをOne-Hotエンコードした特徴量や、セクター内順位（demeanedリターン）等も導入済みです [34]。これらを活用しつつ、**モデル予測の誤差分析**を定期的に行うと良いでしょう。
- **データリーク・ターゲット漏洩の検証:** 時系列データでは厳重に対策されていますが、**ターゲットデータの漏洩**にも注意が必要です。ターゲットとなるリターン変数算出に将来データが混入していないか（例えば株式分割や配当補正ミスによる後出し情報の混入など）を検証してください。現状TODOにも「ターゲット変数の整合性検証（ルックアヘッドなし）」を行うべきとの項目があります [35]。またクロスセクション正規化については、理想的には**学習期間内の統計でテスト期間を標準化**すべきところ、SafeTrainingPipelineでは全体データにfit_transformしている実装上の簡略が見られます [36]。CrossSectionalNormalizerV2はfold毎にfit→transform分離できる設計なので、可能であればウォークフォワード分割と組み合わせて厳密に「各訓練フォールドの統計で対応テストを正規化」する処理に改良すると、より一層リークリスクを低減できます。現状でも各日付ごと独立にZスコア化しているため将来情報の混入は直接無いものの、統計量計算にテスト期間を含めない厳密さが望ましいです。
- **モデル解釈性と監視:** GATによる注意係数やFANのスケーリング係数など、モデル内部の解釈可能な部分も活

用しましょう。例えば重要な特徴量や株間の関連ネットワークを可視化・モニタすることで、モデルが特定の入力に過度に依存していないか、人間が理解できない異常な相関に基づいていないかを把握できます。グラフ構築ステップで生成される**相関ネットワーク**（例:　　50銘柄で266エッジ[37]）も、定期的にスナップショットを取得して重大な変化（エッジ数の極端な増減や中心銘柄の交代など）を検知すると、データ異常やモデル劣化の早期発見につながります。

設定管理と再現性の確保

現状評価: Gogooku3では**Hydraによる設定管理**と**Pydantic設定オブジェクト**を組み合わせており、モデル・データ・学習・ハードウェア設定が体系的に定義されています[38][39]。`.env`ファイルで機密情報や環境変数を扱いつつ、その他のハイパーパラメータは`configs/`以下のyamlや`settings.py`のデフォルト値で一元管理されています。実行時には使用した設定をJSON/YAMLで保存しており、例えば最終モデル保存時にその構成も同時に出力する仕組みがあります[27]。これにより「どの実験でどのパラメータだったか」の再現が容易になっています。乱数シードもグローバルに設定され[40]、Lightningの`deterministic=True`オプションと合わせて実行の再現性に配慮されています。

改善提案:
- **設定と環境のスナップショット:** 現在、`settings.get_settings()`や`save_environment_snapshot()`関数で実行環境（PythonやPyTorchバージョン等）をJSONに保存する機能があります[41]。これを実験実行ごとに自動で記録・管理するとさらに再現性が高まります。例えばMLflowのRunに環境情報タグを付与したり、出力ディレクトリに環境スナップショットを保存する運用です。将来的に環境を再構築する際、この情報を参照して依存ライブラリの正確なバージョンを揃えることができます。
- **依存関係の固定:** `pyproject.toml`で依存パッケージを管理していますが、可能であれば**依存パッケージのバージョンロック**や**ハードウェア依存の明示**を行ってください。特に金融時系列ではライブラリのバージョン差異で微妙な計算結果ズレが生じうるため、モデル精度検証時には同一バージョンであることが重要です。Dockerイメージで環境を固定化しているのであれば、コンテナのタグ付け（例えば`gogooku3-train:20250901`のように日付やGitハッシュを埋め込む）をして、どの実験がどのコンテナで走ったか記録しておくと万全です。
- **設定項目のドキュメント化:** 設定が増えるにつれ、全てのハイパーパラメータの意味やデフォルト値を把握するのが難しくなります。現状Pydanticの`Field`である程度説明されていますが、**主要な設定項目についてはドキュメント（開発ガイドやREADME）に一覧**があると親切です。例えば「`mixed_precision`: TrueにするとAMP自動混合精度訓練を行う（デフォルトTrue）」のように、チーム内で共有できるリファレンスを用意してください。これにより新メンバーも含め、設定を誤解して不適切な実験をするリスクを下げられます。
- **統一された設定インタフェース:** HydraとPydantic設定の併用は柔軟ですが、煩雑さもあります。例えば、現在`main.py`やCLIで`--mode quick`等の引数により一部挙動を変えています[42]。このようなモード切替はHydraのコンフィギュレーション（プロファイル）として定義し、コマンドラインでは`python main.py train mode=quick`のように記述できるようにするのも一案です。いずれにせよ、**コマンドライン引数・環境変数と設定ファイルの優先順位**を明確にしておくことが重要です。現在は`.env`→Pydantic→Hydraという階層でしょうが、ドキュメントに「環境変数で上書き可能な項目」や「モードによる既定値の違い」を明示すると混乱が減ります。

ドキュメントとナレッジ共有体制

現状評価: 本プロジェクトは非常に丁寧にドキュメント化されています。アーキテクチャ概要、データパイプライン、モデル詳細、開発ガイド、クイックスタート、FAQに至るまで網羅的にMarkdownドキュメントが整備されています。特にクイックスタートガイドではDocker含む環境構築から、ヘルスチェックやサービスUIの確認方法、基本的なパイプライン/学習の実行コマンド、トラブルシューティングまで詳細に記載されており、新規ユーザは10分程度でシステムを動かせる内容です[43][42]。また開発貢献ガイドではリポジトリ構造やコーディング規約、テスト方法、CIフローまで具体例付きで示されており、オンボーディング資料として極めて充実しています[44][45]。

改善提案:
- **最新情報へのアップデート:** ドキュメントは生きたものです。最近の機能追加（セクター特徴量やフロー特徴量拡充など）によってデータセット仕様が変化していますので、**各ドキュメント内の数値や項目を最新に保つ**ようにしましょう。例えばREADMEやデータセット仕様書に記載の「特徴量数145」や「対象銘柄632」等の情報は、最新版では異なるはずです（前述の通り現在は248特徴量・4220銘柄 [13] ）。プロジェクト内で成果を共有する際も古い数字が引用されると混乱を招きます。Changelogや更新日を明記するのも有効です。特に外部公開を視野に入れる場合は英語ドキュメントも必要になりますが、まず内部向けには日本語で最新内容を維持して問題ありません。
- **機能ごとの詳細文書:** すでにアーキテクチャやモデルに関する深堀り文書（例: ATFT改善実装のチームレビュー [46] ）が用意されています。この調子で、各主要コンポーネント（データ取得、特徴量生成、モデル推論、MLOps連携など）について**専門トピック別の詳細資料**を蓄積すると良いでしょう。新人メンバーが特定領域を学習する際に、その部分だけを抜粋して勉強できるようになります。例えば「データ品質管理とバリデーションのガイド」や「特徴量ストア(Feast)の運用手順」などです。現在Great ExpectationsやFeastの使い方が断片的にしか触れられていないようなので、社内Wiki的に追記していくとナレッジの集約につながります。
- **サンプルノートブックの提供:** コードとドキュメントに加え、**Jupyter Notebook形式のチュートリアル**があると実用面で非常に役立ちます。例えば「データセットをロードして特徴量分布を可視化するノートブック」や「学習済みモデルの推論デモ（任意銘柄の予測例）」などです。これらは新規参加者やステークホルダーへのデモとしても使え、プロジェクトの理解促進になります。またノートブック上で小規模データを用いたユニットテスト的な検証（各ステップの出力確認）も可能ですので、開発サイクルの高速化にも寄与します。
- **FAQの充実:** ユーザから寄せられた質問や過去の障害対応事例が蓄積してきたら、FAQに追記しましょう。既にdocs/faq.mdが存在していれば随時更新し、なければ作成を検討してください。例えば「学習途中でメモリ不足になった場合の対処」「特定日のデータ欠損があった際の対応策」「Docker環境で発生しがちなエラー集」など、過去のナレッジをQ&A形式で残すと、同じ問題でつまずく時間をチーム全体で削減できます。

CI/CDと開発プロセスの自動化

現状評価: 開発プロセスには自動化の工夫が取り入れられています。`pre-commit` フックによるコード整形・静的解析（ruff, isort, mypy等）の実行や、プルリクエスト作成前チェックリストでのテスト・Lint合格確認 [45] [47] など、品質を維持する仕組みが整っています。加えてDagsterを用いたジョブスケジューリング・パイプライン管理、Grafana/Prometheusによるモニタリング、MLflowによる実験管理とモデルレジストリと、機械学習基盤(MLOps)としてのCI/CDが構築されています [48] 。Docker Composeで主要サービス（MinIO, ClickHouse, Redis, MLflow, Dagster, Grafana等）をワンコマンド起動できるため、開発から本番運用まで見据えた統合環境が用意されています [49] [50] 。

改善提案:
- **継続的インテグレーションの強化:** 現在は手動の `make test` やPR時のチェックリストに頼っている部分を、**CIパイプライン上で自動化**するとヒューマンエラーをさらに減らせます。GitHub ActionsやGitLab CIなどで、コードプッシュ時に自動でLintとユニットテストを実行しバッジ表示する仕組みを導入してください。特にデータ依存の部分はモックや小規模サンプルデータで代替し、外部API呼び出しはスタブ化することで、ネットワーク不要かつ安定したCIテストを実現できます。エンドツーエンドのパイプライン実行はリソース的に難しくとも、**スモークテスト**（1エポックだけ動かす等）は自動実行可能でしょう [51] 。実際、移行時にもスモークテストやインテグレーションテストが整備されています [52] 。これをCI上で定期的に回すことで、将来の変更による性能退行や不具合を早期検出できます。
- **モデルリリースフローの自動化:** MLflowでのモデル登録やDocker化ができている前提で、学習→評価→モデル登録→デプロイまでをできるだけ**ワンボタン（または一連のCIジョブ）**で完結できるようにしましょう。具体的には、Dagsterのパイプラインにモデル評価結果を判定するステップを入れ、所定の性能指標を満たした場合に自動でMLflowのステージを `Staging` から `Production` に更新する、といった処理です。さらに、そ

のイベントをトリガーにしてAPIサーバ側（もしあれば）へデプロイ信号を送るなどすれば、継続デリバリに近づきます。現在は研究開発フェーズかもしれませんが、将来的にポートフォリオ構築やレコメンドなど実運用にモデルを活用する際、CI/CDパイプラインに組み込むことで人的ミス無く最新モデルをサービスに組み込めます。
- **自動クリーニングとリポジトリ健全性:** 未使用コードの隔離削除プロセス（trashディレクトリへの一時移動→CIテスト確認→最終削除）は非常に慎重かつ有用な手順でした [53] 。今後も定期的にこのような**リポジトリクリーンアップ**を行い、不要ファイルや古い実験アーティファクトを整理すると良いでしょう。また、セキュリティ面ではsecret漏洩防止設定やSAST（静的アプリケーションセキュリティテスト）の実行も検討してください（既に `security/` ディレクトリにLeak　PreventionやSAST設定があるようです）。CIでコード変更時に自動スキャンし、機密情報コミットの防止や脆弱性チェックを行えば、プロジェクトの信頼性が更に高まります。

研究の反復速度とワークフロー効率

現状評価: 本プロジェクトは高速なデータ処理と豊富な自動化により、研究サイクルを短縮する工夫が随所に見られます。例えばPolarsによる高速データロードは対数的に効率化され、大容量データでも秒単位で前処理が完了します [16] 。また**Quickモード**による軽量学習パイプラインも提供されており、本番用フル機能（データ品質チェックや全特徴量計算）ではなく、開発検証用に一部処理をスキップ・縮小したモードが選択できます [42] 。これによりアイデア試行段階では数分で結果を得て、十分手応えがあればフルモードで再実験、というメリハリの効いたアプローチが可能です。さらにMakefileに `smoke` ターゲットが用意され1エポックテストが走るなど、**迅速なフィードバックループ**が考慮されています [51] 。チーム開発面でも、事前に統一されたコードスタイル・テストがあることでレビューにかかる手間も削減されています。

改善提案:
- **データ/実験管理の活用:** 現在MLflowで各実験のパラメータやメトリクスを記録していますが、研究速度向上のためにはそのログを積極的に活用してください。例えば過去の類似実験結果を簡単に検索・比較できるようタグ付けを工夫したり、重要なメトリクス（例えばRankIC@5d）の推移をダッシュボード表示することで、チーム全員が進捗を把握しやすくするといったことです。また、大量の実験を効率よく回すため、**並列実験**や**事前スクリーニング**も有効です。現在Optuna HPOなどありますが、例えば「特徴量をカテゴリ毎に追加した場合の影響」を同時並行で検証するようなスクリプトを用意し、一晩で複数条件を試すなどすれば、人的待ち時間を減らせます。
- **コラボレーションとナレッジ共有:** 研究の高速化にはチーム全員の知見共有が欠かせません。定期的に**モデル精度改善会議**や**障害共有会**を開き、そこで得た知見をドキュメントに反映する習慣をつけましょう。既にドキュメントは豊富ですが、口頭で議論したベストプラクティス（例：「メモリ16GBでは銘柄数を4000までに抑えるのが安定」等）は暗黙知になりがちです。これらをIssueやWikiに記録し、次の実験計画に活かすことで無駄なトライ&エラーを減らせます。
- **環境の標準化と実験ポータビリティ:** Dockerを用いた統一環境により、「動かない」という問題はかなり減っていると思われます。さらに研究者各自が計算リソースを有効活用できるよう、例えばDockerイメージを使って社内GPUサーバやクラウド上で簡単にジョブを投げられる仕組みを用意すると良いでしょう。具体的にはDagsterやGitHub　ActionsからKubernetesジョブを起動する、あるいはSLURMクラスター用のJobスクリプトをテンプレ化する等です。環境差異を気にせずどこでも実験できると、思いついたアイデアをすぐ形にして検証するスピードが上がります。
- **ユーザフィードバックの循環:** 最後に、研究開発のスピードを保つには**ユーザ（最終成果の利用者）からのフィードバック**も大事です。モデル予測の実適用先（例えば社内の運用チームや他プロダクト）がある場合、定期的にモデルの有用性や要望を聞き出し、それを研究の優先順位に反映してください。結果として無駄な方向に時間を使うことが減り、プロジェクト全体の効率が上がるはずです。現在は主に自己完結型の研究プロジェクトと推察しますが、将来的に成果物をデプロイ・活用していく段階では、このループを早めに回し始めることが肝要です。

以上、Gogooku3コードベースと開発プロセス全般について包括的にレビューし、改善候補を提案しました。現在すでに高水準の設計・実装・文書化がなされていますが、上記のようなポイントに継続的に取り組むことで、プロジェクトのメンテナビリティ・拡張性・生産性はさらに向上できると考えられます。今後の発展を期待しております。

Sources: 1 2 3 4 5 7 8 12 13 14 15 18 19 20 21 22 23 25 26 28 29 32 33 30 31 36 27 42 44 45 47 49 53

---

1 52 MIGRATION.md
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/MIGRATION.md

2 44 45 47 48 49 51 contributing.md
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/docs/development/contributing.md

3 quarantine_manifest_2025-09-13.txt
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/archive/quarantine_manifest_2025-09-13.txt

4 22 23 24 27 38 train.py
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/scripts/train.py

5 6 25 26 31 46 ATFT_IMPROVEMENTS_TEAM_REVIEW.md
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/docs/ml/atft/ATFT_IMPROVEMENTS_TEAM_REVIEW.md

7 8 9 10 11 12 16 data-pipeline.md
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/docs/architecture/data-pipeline.md

13 14 15 17 18 19 21 34 35 TODO.md
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/TODO.md

20 42 43 50 QUICK_START.md
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/QUICK_START.md

28 29 36 safe_training_pipeline.py
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/src/gogooku3/training/safe_training_pipeline.py

30 37 model-training.md
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/docs/ml/model-training.md

32 33 split.py
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/src/gogooku3/training/split.py

39 40 41 settings.py
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/src/utils/settings.py

53 CLEAN_UP.md
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/CLEAN_UP.md