Università degli studi di Milano-Bicocca

Dipartimento di Statistica e Metodi Quantitativi

Corso di Laurea Magistrale in Economia e Finanza

Credit Spread Forecasting: A Study of How Machine Learning Can Enhance Predictive Accuracy

Giorgio Perego

Supervisor: Prof. **Alessandro Avellone**

Co-supervisor: Prof. **Lucio De Capitani**

Academic year

2024/2025

Contents

Introduction

Credit spread prediction has become a foundation of financial research over the past decades, guiding practitioners and researchers alike in tackling the dual problems of credit risk measurement and portfolio management. Credit spreads, or the spread between the differential yield of corporate bonds versus that of government bonds of similar maturities, become a concise measure of the market's expectation of the probability of a given issuer's default. This current thesis examines these dynamics empirically using a dataset covering the time span from January 1, 2000, to March 26, 2025. Aside from this, they embody general perceptions of macroeconomic well-being, liquidity conditions, and systemic risk.

With increasingly sophisticated capital markets, aided by the emergence of structured credit markets and global funding sources, the ability to forecast credit spread volatility has taken center stage. An effective forecasting model helps banks make more precise adjustments to capital reserves according to Basel III requirements, helps asset managers fine-tune duration and credit risk, and enables authorities to utilize predictor markers to prevent building crises.

Research in the field of credit spread prediction took off after Fama and French (1989) showed that spreads comove with macroeconomic cycles and widen when output gaps deteriorate and volatility increases in equity markets. Their multivariate regression set-up offered the basis for decades of research, which has increasingly included an ever-larger number of explanatory variables. Empirical studies have identified the contribution of interbank lending rates, e.g., the TED spread, in explaining credit spread fluctuations. More recently, the focus has been on measures of policy and economic uncertainty. Baker, Bloom, and Davis (2016) constructed the Economic Policy Uncertainty Index, which shows that episodes of elevated legislative or geopolitical uncertainty coincide with sustained widenings of credit spreads that reflect investors' demand for extra compensation amidst uncertain regulatory regimes.

Notwithstanding the explanatory power of conventional regression-based models, their limitations have been highlighted by the 2007–2008 financial crises and market volatility due to the COVID-19 pandemic, especially under conditions of sudden regime shift and nonlinear risk factor interrelations. The credit spreads widened at a historic rate during the global financial crisis, frequently outside the models' predictive range, which were estimated using calm market data. Meanwhile, the initial phases of the pandemic saw flash-like contraction and expansion of spreads as governments and central banks unleashed unprecedented monetary and fiscal stimulus.

Those incidents underscored the necessity for forecasting models that are capable of dynamically adjusting to changing market regimes and detecting faint, nonlinear interdependencies.

Machine learning techniques like Random Forest and eXtreme Gradient Boosting (XGBoost) have gained popularity exactly because they excel at modeling nonlinear interaction and dealing with high-dimensional data with noisy or collinear predictors. These techniques, via their ensemble of numerous trees, are able to pick up patterns that linear specifications cannot, and therefore improve out-of-sample forecasting accuracy.

Their "black-box" nature, however, automatically makes interpretability a concern. Financial practitioners and regulators need transparency so that they know what macroeconomic or market variables are contributing to spread movements, particularly if forecasts are being used to determine risk limits or capital requirements. Balancing predictive performance and interpretability is thus an active area of research, with an incentive for considering hybrid models that combine machine learning with more interpretable, parametric models.

Simultaneously, the application of regime-switching models, and in particular HMM-based models, is a highly attractive framework for explaining structural breaks in credit market dynamics. Through probabilistically assigning tranquil and stressful periods, HMMs make model parameters respond to the dominating market regime, which stabilizes forecasts in heterogeneous regimes. For example, in a low-volatility regime, credit spreads can be mostly driven by slowly evolving macroeconomic trends, while under stress regimes, liquidity considerations and flight-to-quality effects can prevail. With the incorporation of regime-identified HMMs into a forecasting framework, one gains insight into how risk premia behave across varying market states and makes feasible conditional forecasting techniques specific to the dominant regime.

The setup of a sophisticated forecasting system requires a setting which is reproducible, flexible, and computationally feasible. Python, due to its enormous collection of open-source libraries, has emerged as the leading language in quantitative analysis. Its modularity allows for the implementation of full pipelines that cover data ingestion, preprocessing (missing value imputation, scaling, feature engineering), training of the model along with extensive hyperparameter search, and cross-validation for the reduction of harm due to overfitting.

From a practical perspective, the implications of enhanced forecasting of credit spreads are many. Financial institutions are able to leverage improved accuracy in forecasts towards improving credit valuation adjustments (CVA), reconciling collateral requirements, and tailoring hedging strategies across credit derivatives markets. Portfolio managers are able to leverage conditional forecasts for fine-tuning exposures towards benefiting from tightening spreads of sectors or issuers likely to witness tightening spreads, hence capturing relative value opportunities. At the same time, macroprudential supervisors are able to incorporate credit spread forecasts into stress-testing frameworks, using scenario analyses for gauging vulnerability under plausible shocks.

These uses make clear the importance of a forecasting model that can provide statistical precision alongside positioning with the decision processes among multiple entrants to a market.

However, the quest for methodological rigor must be balanced with the quest for practical ambition. Variable selection should be informed by economic theory alongside tests of empirical robustness to make sure that models aren't built based on spurious correlations. Interpretability may also be increased using feature importance analysis, partial dependence plots, and construction of surrogate linear models that correctly approximate machine learning forecasts. Finally, rigorous backtesting and out-of-sample calibrations over a series of market cycles need to be constructed to test robustness for models and avoid overfitting to historical unique events.

Credit spread forecasting is a prominent tool for companies and a worthwhile topic of study for current quantitative finance. By combining traditional econometric techniques with current machine learning strategies and regime-switching techniques, forecasting frameworks that are more precise and more responsive to changing market conditions can be developed.

The combination of new techniques with practical usability characterizes the motivation for this work, whose final ambition is the delivery of a reproducible, understandable, and resilient Python-based framework for forecasting credit spreads. This is a solution for enabling decision-mai processes across the banking, asset management, and regulatory communities that remains effective despite the market upheaval predictably due soon.

I General and Specific Objectives

The ultimate goal of this thesis is to create and verify a composite forecasting model for the prediction of credit spreads through the combination of conventional regression methods and novel machine learning and regime-switching models. In particular, a Python pipeline in modular form will be created[1] capable of automatically processing financial data sets, imputing missing values, scaling features, conducting factor analysis, and running a variety of modeling techniques. Hyperparameter optimization routines and model artifact versioning will be used to ensure every run of experiments is exactly reproducible.

One of the primary objectives of the present research is the quantitative estimation of the prediction accuracy of Random Forest and XGBoost in relation to classical linear regression. To that end, the accuracy measures of prediction like Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and the coefficient of determination (R^2) will be estimated for all the models in-sample and out-of-sample. This comparison study will establish whether nonlinear machine learning methods yield statistically significant improvements in credit spread forecasting, particularly when confronted with high-dimensional noisy macroeconomic variables.

Another specific objective is to incorporate market regime information — estimated by Gaussian Hidden Markov Models (HMM)—into the forecasting procedure. By assigning prior observations probabilistically to "calm" and "stress" regimes, this research will examine the contribution of regime-dependent dynamics to model performance. Regime-conditioned predictions that take advantage of contemporaneous regime information will be compared to unconditional predictions in order to measure potential stability and accuracy gains (Tang and Yan, 2010; Baker, Bloom, and Davis, 2016).

Providing interpretability and strength is another objective of this study. In spite of the higher predictive accuracy of tree-based methods, the description of the role every variable plays in forecasting outcomes must be elucidated. Feature importance metrics, partial dependence plots, and surrogate linear models will be used with the aim of providing insights into the economic

[1]By “modular pipeline” we refer to a software architecture that clearly separates data ingestion, preprocessing, model training, hyperparameter tuning, and validation stages, ensuring reproducibility and traceability of results.

determinants of credit spreads. These will be supplemented with backtesting and validation on various market conditions to verify model stability and ensure that interpretability is not gained at the expense of forecast accuracy.

Another objective is to integrate the empirical findings into a general discussion of how effective machine learning models are for predicting credit spreads. This thesis compares performance under varying different forecasting methods, including econometric models and machine learning models, and critically assesses their appropriateness for application in real-world financial environments. In the spirit of reproducibility and transparency, all of the case study–documented Python code, the data preprocessing scripts, and the full dataset will be accessible at GitHub[2]. The latter will act as a valuable reference for subsequent researchers who would like to build upon and extend the techniques used here.

II Research Questions and Hypotheses

Chief among these research questions is the following: "To what extent can machine learning models improve the precision in forecasts of credit spreads compared to traditional econometric methods?" This research question highlights the investigation into the capability of non-linear models, such as Random Forest and XGBoost, to surpass linear regression methods in forecasting credit spreads, in particular, while working with complex macroeconomic determinants and in contexts characterized by financial volatility.

The second research question entails the application of market regime information in credit spread prediction. The study will particularly examine: "What is the effect of incorporating market regimes, detected by means of Hidden Markov Models (HMM), on the precision of credit spread prediction?" The principal hypothesis in this case is that regime shift explicit modeling has the

[2]The full codebase and data can be found at `github.com/GiorgioPerego/Credit-Spread-Forecasting`, to promote maximum transparency and to allow further academic and industry-based research.

potential to enhance the robustness of predictions, particularly in times of distressed markets.

Lastly, this research will assess the usability of machine learning models in predicting credit spreads that are relevant to financial decision-making purposes: "How state-dependent is the risk profile of credit spreads, and how may regimes based on HMM be employed to quantify this for risk analysis?" Hypothesized by the study is the proposition that the inclusion of machine learning technologies will be likely to refine the accuracy of risk analysis undertaken by financial institutions and policy officials and help them make more competent decisions and minimize credit risk in turbulent market contexts.

The following are the hypotheses to be tested in this study:

- **H1:** Random Forest, XGBoost, and TCN machine learning models will significantly outperform linear models in predicting credit spreads, as evaluated by RMSE, MAE, and R^2.
- **H2:** The incorporation of market regimes, as identified by Hidden Markov Models, will add predictive validity and stability to credit spread prediction, especially for stressed markets.
- **H3:** The regime-dependency of the risk profile of credit spread changes (e.g., Value-at-Risk and Expected Shortfall) will be demonstrated to be statistically significant. The regimes that will be discerned by a Hidden Markov Model will successfully partition these risk profiles, with a strong and measurable contrast between 'calm' and 'stress' regimes in tail risk strategies.

To these research questions and hypotheses stipulated, this thesis aims to provide a comprehensive assessment of the potential of machine learning models to improve credit spread forecasting with a perspective of improving risk management and financial decision-making in the context of modern financial markets.

III Structure of the Thesis

The thesis consists of five main parts, consisting of a single introductory part with five sequentially numbered chapters, which cover different research aspects separately.

The first part, named *Introduction*, gives a general overview of the subject of the research. The proposed research questions and hypotheses proposed, details about the motivation for the study, and the significance of credit spread forecasting for financial decision-making are outlined here. A preview of the theory framework and background is given, covering the most relevant concepts, models, which are relevant for the study at hand. Finally, the structure of the thesis is outlined, which gives the reader a clear map.

Chapter 1, *Literature Review and Theoretical Framework*, gives a comprehensive overview of the available body of work on forecasting credit spread. This chapter makes clear what is meant by *credit spread*, including its definition, financial market role, and determinants of its movement. The chapter outlines classical econometric methods, i.e., linear regression, and compares them with machine learning methods. Besides, it addresses the market regime concept and inspects the application of Hidden Markov Models (HMM) on market state change modeling. This chapter sets the background for the empirical analysis by giving the necessary theoretical background.

Chapter 2,*Data and Methodology*, details the developed dataset for the course of this work. It describes the sources of the data, the chosen variables, and the entire process applied to prepare the features for analysis. It encompasses such techniques for managing missing values, harmonizing mixed-frequency data by using MIDAS, managing skewness, scaling features, conducting dimension reduction through Factor Analysis, and verifying stationarity. It also provides an idea regarding the application of Hidden Markov Models (HMM) for regime identification in the market.

Chapter 3, *Forecasting Models and Methodology* details the series of predictive models applied in the present research. This chapter covers the theory and application of the conventional econometric baseline and the state-of-the-art machine learning models, that is, Random Forest, XGBoost, and a Temporal Convolutional Network (TCN).

Chapter 4, *Empirical Results and Discussion*, treats the empirical findings. It is focused on comparing the predictive power of machine learning models and traditional econometric models, i.e., the accuracy of predictions of credit spreads under various market situations. The chapter also describes the measures of evaluation (RMSE, MAE, and R^2). The results are explained in terms of research questions and hypotheses, and conclusions regarding practical implications for financial decision-making, risk management, and policy-making are made.

Chapter 5, *Conclusions and Future Research*, provides an overview of the main results of the thesis, examines the hypotheses of the research, and formulates the contribution of research. Additionally, it points out the study weaknesses and gives suggestions on potential directions for future research, including the improvement of model performance, introduction of other variables, and extension of the framework to other financial markets.

This organization is a clean and natural progression from the background and introduction to the empirical analysis and conclusions, and is a thorough overview of the usage of machine learning models in forecasting credit spreads.

Chapter 1

Literature Review and Theoretical Framework

1.1 Credit Spread: Definition and Importance

Definition 1.1.1 (*Credit Spread*). *Let $y_{c,t}$ denote the yield on a* BBB*-rated corporate bond index and $y_{g,t}$ the yield on a government bond of equal maturity. The credit spread at time t is*

$$CS_t = y_{c,t} - y_{g,t}.$$

In our empirical work (and Python code), we proxy $y_{c,t}$ with the ICE BofA BBB U.S. Corporate index yield (FRED `BAMLC0A4CBBBEY`) and $y_{g,t}$ with the 10-year U.S. Treasury yield (FRED `DGS10`).[3]

The credit spread is a prime barometer of risk and liquidity across fixed-income markets. A single measure, it synthesizes investors' expectations about issuers' solvency, prevailing market conditions, such as market depth, volumes of trades, and bid–ask spreads, up to the onset of tail risks, which are broadly uncovered when financial stress sets in.

[3]FRED (Federal Reserve Economic Data) is the database for the Federal Reserve Bank of St. Louis, allowing data access to a vast set of official time-series data related to macroeconomic and financial variables for the United States and globally.

Foundational components. Conventional sources on literature outline several determinants that cumulatively impact credit spreads:

- *Default risk premium*: payoff for the risk of the issuer not honoring its debt and for the likely loss under partial recovery. Merton (1974)'s structural model associates firm asset values with a default threshold implied by debt; extensions, however, include asset-volatility, leverage, and recovery-rate channels.

- *Liquidity premium*: the majority of corporate debt, including mid-sized and small issuers, exhibit low-depth trades, minimal sizes, and wide bid–ask spreads. Investors consequently demand further compensation with the form of premiums to match liquidity risk. Statistical evidence (e.g., (Bougheas, Switzer, and Symons, 1997)) reveals that liquidity proxies—on-the-run/off-the-run debt, smoothed trade quantities—explain much spread variability, particularly around rough periods.

- *Jump-to-default and tail-risk premium*: compensation for abrupt crises—-system crises, surprise defaults, geopolitical shocks—that is not sufficiently covered by continuous-diffusion frameworks. This mechanism lies behind the *credit spread puzzle*, which is the dilemma presented for pure-diffusion frameworks for properly calibrating short-term spreads without considering discrete jumps or fat tails (Jennie Bai, I. Goldstein, and Yang, 2020).

- *Risk-aversion/cyclicality*: time-varying risk tolerance and macro uncertainty affect the pricing kernels, increasing the spreads for recessions and decreasing the spreads for booms.

Predictive content and macroeconomic role. It has been shown that large subsets of credit spread data significantly anticipate crucial business-cycle turning points, thus representing predictors for market participants and authorities (Gilchrist and Zakrajšek, 2012). Based on senior unsecured bond spreads, the macroeconomic aggregates developed by Gilchrist and Zakrajšek (2012) (GZ index) predicts important macroeconomic variables—such as industrial output, GDP, and unemployment—by a period between three and twelve months. In the first six months of the period preceding the 2008 recession, the GZ index widened by over 150 basis points several months prior to the subsequent fall in output and employment, thus providing a clear signal for institutions and regulatory authorities.

Complementary results provided by Faust et al. (2013)-who apply Bayesian Model Averaging for twenty portfolios which are grouped by rating and maturity—show that GDP and industrial production one- to four-quarter forecasts improve by about 10 % relative to univariate autoregressions. Hence, aside from the average level, the cross-sectional *shape* of spreads, grouped by rating and maturity, constitutes a valuable information source for macroeconomic forecasting.

When spreads grow wider, markets indicate higher future default risk and/or inadequate liquidity—situations characteristic during recessions. Narrower spreads suggest greater solvency perceptions and steadier market conditions, characteristic during expansions. Owing to this, spreads play a prominent role in central bank and research institution nowcasting/forecasting models; alongside traditional ones (interest rates, exchange rates, commodity prices), they provide estimates that respond faster to shocks.

Case Study: The COVID-19 Shock. Up to February 2020, the price spreads for investment grade securities began to open up expecting the pandemic; however, employment and economic output real-time data remained healthy. The first indicator showing signs of trouble gave some managers advance notice, and thus, they were able to adopt defensive approaches—lowering exposure to higher-risk assets but increasing protection through the use of credit default swaps.

Regulatory and risk-management relevance. Spread dynamics go directly into regulatory stress testing: a number of authorities make spread-based scenarios for calibrating credit shocks when they run simulations of bank balance sheets. Spreads capture market perceptions of bad things happening, so adding them makes scenarios less likely to underestimate systemic risk. Operationally, spreads are prominent inputs for Credit Valuation Adjustment (CVA)[4], Basel III requirements for capital, and credit-derivative pricing. Systematic underestimation squeezes capital buffers and increases loss vulnerability, while excessive conservatism means costly funding unnecessarily.

Worldwide Evidence. Beyond developed nations lie the informational roles of spreads. Research on new markets shows identical countercyclical characteristics shared by the spreads between sovereign and corporate entities, which often display considerable volatility; hence affirming their global applicability for gauging risk within various market settings.

Cross-section and term structure Two more dimensions apply to analysis and forecasting: the *term structure* of spreads, which compares short- and long-term maturities, and the *cross-section* that goes across classes of ratings and sectors of industries. A distinctive spread curve is typically evident at the outset of economic decline, since short-term risk adjust faster while inter-industry disparities can signal weaknesses earlier than such weaknesses become apparent in aggregate wider terms.

Investment-grade vs. high-yield. With BBB or better ratings, *investment-grade* bonds tend to have thin, less erratic spreads, which translate into lower default risk and more-liquid trading. For a comparison, the *high-yield* market (BB+ or lower) features wider, more erratic spreads, reflecting more credit riskiness and less desirable marketplaces. Forecasting tools and interpretations should also be revised for such divergent movements.

[4]CVA is the fair-value adjustment that lowers the risk-free value of a derivative (or a netting set of derivatives) to reflect the likelihood that the counterparty will default prior to ultimate settlement. A typical reduced-form approach is $\mathrm{CVA} = (1-R)\int_0^T D(0,t)\,\mathrm{EE}^{+}(t)\,\mathrm{dPD}(t)$, where R represents the recovery rate, $D(0,t)$ denotes the discount factor, $\mathrm{EE}^{+}(t) = \mathbb{E}[\max(V_t,0)]$ signifies the expected positive exposure, and $\mathrm{PD}(t)$ indicates the cumulative default probability. In a discrete time framework: $\mathrm{CVA} \approx (1-R)\sum_i D_{0,i}\,\mathrm{EPE}_i\,\Delta\mathrm{PD}_i$.

Summary. Even though it is ostensibly a simple spread of interest rates, the credit spread is a polyvalent gauge of tremendous theoretical and empirical significance. It summarizes premia for default, liquidity, and tail risk; presages macroeconomic inflexion points; and informs risk management, pricing strategies, and regulatory policies. This relevance grounds the theoretical rigor and empirical focus given throughout the dissertation to this variable.

1.2 Investment-Grade vs. High-Yield Credit Spreads

Definition 1.2.1 (*Investment-Grade (IG)*). *A bond (or index) is* investment-grade *if its long-term credit rating is* BBB− *or better under S&P/Fitch (or similar under others). As a condition for our analysis, The corporate IG benchmark is the* BBB *segment.*

Definition 1.2.2 (*High-Yield (HY)*). *A bond is considered high-yield if its long-term rating is* BB^{+} *or lower. HY bonds pay investors higher risk premia in return for higher default Exposure to risk, decreased liquidity, and increased susceptibility to tail events.*

Credit spreads differ considerably across the credit-quality spectrum. *Investment-grade* (IG) Bonds have narrow spreads that usually lie between 50 and 150 basis points over Treasury securities in benign conditions—supported firmer foundations, increased secondary-market liquidity, Lower default risk. Their premia pay principally to moderate *default-risk* and *liquidity-risk* components (Gilchrist and Zakrajšek, 2012). By contrast, *high-yield* (HY) bonds usually have spreads of 300–600 basis points—or higher in distress—paying investors for elevated expected losses and illiquidity (Faust et al., 2013). Volatility is much greater in HY: one-standard-deviation spreads for HY bonds have a propensity to range between two to three events relating to members of IG, driven by special credit events and a heightened sensitivity to funding and macroeconomic risk.

Stylized facts during the cycle. While the spread drivers for spreads on IG and HY—default premia, liquidity premia, and tail-risk premia—are numerically equal, but their *magnitudes*[5]and *cyclical sensitivities*[6]differ significantly. In down markets, spreads for HY increase disproportionately when weaker issuers have a greater refinancing risk and loss severity; IG spreads widen more moderately because stronger issuers retain financing flexibility, and a broader investor base. Empirically, HY tends *to lead* IG at the onset of stress: both in the 2008 meltdown and the COVID-19 pandemic sell-off, spreads for the HYs exceeded 1,000 bps whereby peaks of IG were maintained below 400 bps, which signifies credit cycle segmentation.

Term structure and cross-section. Two other factors make the IG/HY contrast crisper still. First, the *term structure* of spreads: These HY curves are typically much steeper at the short end to account for the potential default and the liquidity risk. Reprice rapidly; IG curves are less steep and more grounded in macro fundamentals. Second, the *sectoral cross-section*: cyclically sensitive industries (e.g., energy, discretionary) display larger amplitude and quicker propagation compared with IG, and hence earlier alarm signs in the HY sleeve which may anticipate aggregate spreading movements.

Measurement choices and benchmark spread. For comparability purposes and availability of data, we monitor IG conditions through the **BBB–Treasury spread**. With respect to high-rated segments (e.g., AAA), the BBB spread is more responsive to macro shocks and liquidity strains, yet less noisy than broad HY indices. A large portion of corporate issuance is BBB, so the BBB–Treasury spread is a broad aggregate measure of overall corporate funding

[5]By *magnitudes*, we refer to the degree and variation of credit spreads among different segments, for instance, usual spreads at normal times (IG $\approx$ 50–150 bps, HY $\approx$ 300–600 bps), their unconditional volatility (standard deviation), and tail thickness (extent of large moves).

[6]We speak here of *cyclical sensitivities*, i.e., reactions of spreads towards business cycle and macro shocks—specifically, semi-elasticities like $\partial CS_t / \partial z_t$ of a macro driver (e.g., growth, unemployment, funding or volatility measures), allowing coefficients to vary by regime s_t in state-dependent models (HMM/Markov-switching) or via local projections.

conditions. This balance of *sensitivity* and *stability* makes the BBB–Treasury spread a suitable proxy for corporate funding overall conditions and a worthwhile target for out-of-sample validation.

Liquidity and flows. Market microstructure widens the IG/HY differential. IG markets are characterized by tighter bid–ask spreads, active players, and high levels of trading, through which temporary imbalances between the spread are mitigated. HY markets are also more prone to flow shocks due to order imbalances and ETF-related flow dynamics that, in the short term, widen spreads disproportionately to fundamentals around redemptions, particularly if dealers' balance-sheet capacity is constrained.

Modeling implications. The IG/HY duality has concrete consequences for forecasting form: *Segment-Specific Attributes* (e.g., terms of funding and actions on liquidity) should included more heavily for HY; *regime-switching* or state-dependent models are especially suitable for HY, which has large nonlinearities and jump activity; *loss functions* can be tailored for non-symmetric costs (e.g., for underpredicting more in HY when there is stress); *nowcasting inputs* should span short (HY credit/equity vol, funding spreads) which first dominate in HY and pass through with a lag into IG.[7]

[7]All the empirical relations for this thesis target the **BBB–Treasury** spread. For context alone, discussion for the HY and AAA series is presented but not forecasted using our algorithm.

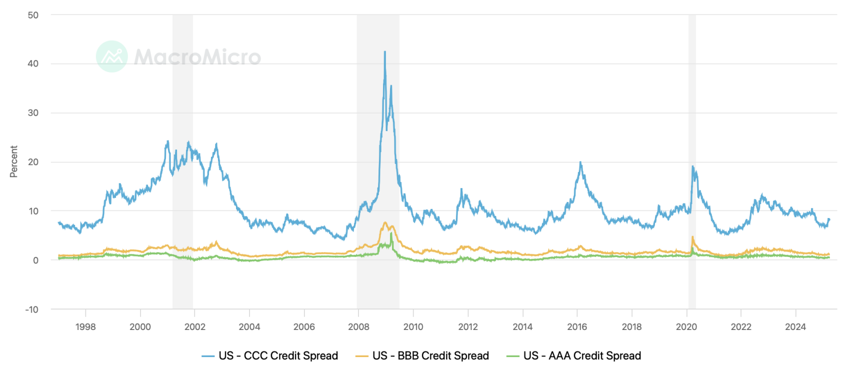


Figure 1.1: U.S. credit spreads for three rating classes—CCC (blue), BBB (orange), AAA (green)—from 1997 to 2025. Grey bands denote NBER recessions. CCC spreads exhibit pronounced amplifications during crises (2008–2009, 2020); BBB moves intermediately; AAA remains comparatively stable, confirming the positive relation between credit quality and sensitivity to market stress.

1.3 The "Credit Spread Puzzle"

Definition 1.3.1 (*Credit Spread Puzzle*). *The* credit spread puzzle *is given as the empirical regularity thatCorporate yield spreads—especially at short horizons—are also invariably wider than what canonical diffusion-based models (e.g., lognormal asset evolutions with homogeneous volatility) can model without recourse to implausibly high risk-premium or volatility assumptions. In reduced-form terms, for short horizons and with recovery R and default intensity λ_t being of constant value, the spread is approximated by* $\text{CS} \approx (1 - R)\lambda$*; in data,* CS *often exceeds what plausible λ and R values would imply.*

Why diffusion-only models underpredict spreads. Three ingredients ensure that pure diffusion models cannot reproduce observed spreads:

- *Thin tails and continuity.* Since no discrete jumps nor fat tails exist, short-horizon default probabilities. They remain too small to account for the CS levels in practical applications, unless it is approximated excessively elevated asset volatility or equity premiums (Merton, 1974).
- *Channels of Recovery and Leverage.* Ongoing recoveries and unchanging

capital frameworks. Offset base losses at the structural level; in terms of stress, recoveries have procyclical behavior (decreasing), leverage arise endogenously, and effective loss rates rise—characteristics beyond simple diffusion protocols.

- *Liquidity and risk-aversion.* Bid–ask costs, order-imbalance, and time-varying pricing kernels (aversion to risk) applies premia on top of loss expectation at default; diffusion-only models do not consider these non-default elements (Jennie Bai, I. Goldstein, and Yang, 2020).

Canonical extensions from the literature. Certain approaches in modeling address the puzzle:

- *Jump-to-default/jump–diffusion.* Adding discrete jump components or explicit default jumps thickens tails and lifts short-maturity spreads with non–reasonable volatilities.
- *Regime and stochastic volatility dependence.* The observation that parameters and volatility are subject to "calm" and "stress" state changes practically encapsulates the cluster episode and shock repricing episode phenomena.
- *Liquidity/tail-risk premias.* Adding pricing procedures withliquidity costs and tail-event risk creates a non-default wedge in spreads.

Definition 1.3.2 (*Reduced-form relationship between default and spread*). *In a typical reduced-form vanilla shape with provided recovery R and provided default intensity λ, $P^{corp}(0,T) \approx P^{gov}(0,T)\exp\big(-(1-R)\int_0^T \lambda_s \, \mathrm{d}s\big)$. For brief maturities or a constant λ, this results in the approximate* $\mathrm{CS} \approx (1-R)\lambda$*, thereby elucidating the observed small probability under match seen in the* CS*.*

Implications for our empirical strategy (implementation-aware). Following the pipeline which we use, our approach is a practical one reflecting the extensions. above while remaining model-light:

- *Measurement.* We measure the investment-grade spread as a **BBB–Treasury yield differential** (ICE BofA BBB vs. 10Y U.S. Treasury; FRED `BAMLC0A4CBBBEY` and `DGS10`). In our program, we intentionally avoid computing measures out of OAS or CDS; such quantities are employed only for interpretation.[8]

- *Regime dependence.* We formally allow for *state shifts* by classifying the time series with a Gaussian Hidden Markov Model (HMM). This permits clustering, non-Gaussian tails, and sudden repricing that diffusion-only approaches overlook.

- *Nonlinear response.* We make use of tree-based learning models, namely Random Forest and early stopping XGBoost, and a linear baseline. Such methods efficiently summarize interactions and nonlinear dependence on macroeconomic indicators, funding, and market forces with the latter conditioning next-day spreads—an intuitive proxy for the jump/tail-risk mechanisms depicted in the question.

What the puzzle means for forecasting. The puzzle warns against putting too much dependence on one, homoskedastic diffusion model for predicting credit spreads, especially around points of turning. Predictive models must react to *regime* shifts, introduce *nonlinear* effects into stress, and capture liquidity and risk-aversion proxies when available. Our approach follows such a playbook using a mix of regime classification (HMM) with Nonlinear learners assess performance based on a chronological holdout method with the target held intact. variable tied to the BBB–Treasury spread (Merton, 1974; Jennie Bai, I. Goldstein, and Yang, 2020).

[8]All empirical evidence introduced here makes use of the BBB–Treasury yield spread; OAS/CDS emerge merely as intellectual points of reference for framing results from the literature.

1.4 Literature Review

The econometric analysis of credit spreads changed significantly after the late 1980s. The earlier work was based on simple multivariate linear regressions between corporate yield spreads and macro–financial fundamentals. In a pioneering paper, Fama and French (1989) showed evidence that spreads co-moved with the business cycle and with outputgrowth, inflation, funding, and equity-market volatility, demonstrating that simple OLS specifications account for a non-negligible proportion of variation but also display the shortcomings of fixed, linear specifications when crises emerge.

Transitioning linear benchmarks towards structural and reduced-form decompositions. A vast body of work asked what *explains* spreads at elevated frequencies. Collin-Dufresne, R. S. Goldstein, and Martin (2001) demonstrated that shifts in U.S. corporate spreads are not strongly explained at the firm level, but rather suggest common, market-wide determinants. Elton et al. (2001) broke down investment-grade spreads along default and non-default dimensions, whereas Longstaff, Mithal, and Neis (2005) employed CDS–bond spreads to isolate expected default losses from the residual (liquidity and risk-premia) wedge. Follow-on work incorporated macro risks more explicitly: H. Chen, Collin-Dufresne, and R. S. Goldstein (2009) related spreads to time-varying macroeconomic uncertainty and leverage, while Pan and Singleton (2008) delivered a reduced-form model.

Uncertainty, policy, and the role of risk appetite. Beyond the realized fundamentals, risk aversion and uncertainty measures feature prominently in the spreads determination. The Economic Policy Uncertainty gauge proposed by Baker, Bloom, and Davis (2016) significantly varies throughout notable policy shocks and also tracks the widening spreads while keeping common macro controls fixed. Furthermore, the all-encompassing gauge proposed by Gilchrist and Zakrajšek (2012) (GZ) uncovers that corporate spreads function as coincident indicators of real business cycles and hence also give GDP, industrial production, and unemployment forecasts in horizons up to a year. In summary, these results verify the hypothesis that both the *level* of spreads and co-movement with uncertainty measures have a predictive ability with regards to the macroeconomy. Bayesian Model Averaging methods have also been utilized for determining which predictors contribute to the strongest forecasting improvements under various model specifications (Faust et al., 2013).

Definition 1.4.1 (*Regime-switching models*). Regime-switching *models allow the statistical characteristics of a longitudinal series—mean, variance, and persistence—to change between latent regimes—commonly interpreted as representing "calm" and "stress." Under an assumption of Gaussian distribution within the scope of a Hidden Markov Model (HMM), the Markov chain governs the transitions between the regimes and the observation-space outcomes are state-conditioned Gaussians. This structure allows the expression of volatility clustering, shock-like repricing, and non-Gaussian tails not well captured by linear, homoskedastic specifications.*

Tipping points and regime dependences. Asymmetries in business-cycle behavior inspired Hamilton (1989)'s Markov-switching econometrics, later becoming central to credit applications. Allowing spread dynamics to switch regime allows for improvements in forecasting precision around turnabouts and matching with abrupt shifts observed at the time of crises. In application, regime labels must only be estimated from the spread or very few observables and then transferred as inputs or conditioning information to forecasting models.

Machine learning and nonlinear mapping. With more-robust data and computing, mid-2010s witnessed the spread prediction using machine-learning (ML) techniques. Regime-classification-ensemble hybrid designs (e.g., Random Forests, gradient boosting) capture nonlinear interactions and state-wise heterogeneous responses; see, e.g., Tang and Yan (2010). ML estimates generally enhance out-of-sample performance over linear benchmarks under high dimensional, noisy scenarios, but pose questions about interpretability. Such "black-box" issues are actually addressed in practice with feature-importance measures, partial-dependence plots, and linear surrogates, which enable stories that differ by regime (stable fundamentals during calm states, for example, versus interagency financing and liquidity terms during distress).

Definition 1.4.2 (*Mixed Data Sampling (MIDAS)*). *The Mixed Data Sampling (MIDAS) regression procedure is an econometric tool that helps in modeling and predicting a low-frequency series with the help of higher-frequency regressors (for instance, daily/monthly series). Using parametric weight functions like the Beta series or Almon polynomials helps combine high-frequency series while preserving the contained information without resorting to ad hoc up- and down-sample approaches. The accuracy of forecasts is enhanced, especially if the lower-frequency series for macroeconomic phenomena are explained by higher-frequency series.*

Mixed-frequency and factor structures. Frames that are characterized by mixed-frequency designs are especially beneficial for spreads, which are observed on daily but react to the macroeconomic drivers on monthly or quarterly frequencies. Ghysels, Santa-Clara, and Valkanov (2004) showed that the MIDAS methodology can improve the in-sample fitting and the out-of-sample forecasting accuracy. At the same time, Jushan Bai and Ng (2002) showed that an extensive panel of the macro-financial indicators could be effectively captured by using a few latent factors with strong predictive power. Such principles lie behind many modern techniques that synchronize low-frequency macroeconomic time-series with the daily time-grid and summarize information sets prior to the prediction procedure.

Cross-sections, term structure, and segmentation of the market. Spreads differ between ratings and horizons, and cross-market data are enlightening. Lower-grade segments tend to lead higher-grade credits around turning points (Elton et al., 2001), whereas emerging-market spreads share the same countercyclical patterns with higher volatility (Erb, Harvey, and Viskanta, 1996). Liquidity variation introduces further accentuation of movements; e.g., Treasury market liquidity shocks spread to credit through funding and risk-bearing channels (Bao, Pan, and Wang, 2011). Such arguments support models with cross-sectional heterogeneity between segments and along the maturity horizon.

Positioning of this thesis (implementation-aware). In keeping with existing scholarship, our empirical contribution is deliberately practical and concomitant with our codebase: (i) we assume the investment-grade spread to be a BBB–Treasury yield differential (ICE BofA BBB versus the 10Y U.S. Treasury); (ii) we apply mixed-frequency inputs via *causal* upsampling at monthly and quarter flows to a calendar at the daily frequency with parametric weights and compress the macro–financial panel via Factor Analysis; (iii) we deduce *HMM* regime labels from the spread itself; and (iv) we contrast a linear baseline with nonlinear learning methods (namely, Random Forest training with early stopping and XGBoost), which get refined with time-series cross-validation in the training set and then validated on a separate chronological holdout. Furthermore, we tabulate risk measures conditioned on regimes.

Synthesis. This extension from linear regressions with fixed parameters to models that include regime intelligence and nonlinear attributes reflects the empirical properties of credit markets, such as tail risk, shifting risk appetite, and liquidity constraints. Through the integration of regime-switching techniques with flexible forecasting models and mixed-frequency data, our approach aims to find a compromise between richness of information and interpretability, ultimately summarizing the relevant data encapsulated within investment-grade spreads with the stringent robustness under varying regimes.

1.5 Role of Macroeconomic Variables and Market Regimes

Credit spread forecasting needs to translate macro and market information into expected default and liquidity premia. In practice, there are many different indicators pointing at single channels—growth, inflation, funding stress, and uncertainty of policy—and their effect is *not* time-invariant but regime-dependent.

The credit-spread dynamics have their origin in the interaction between macroeconomic fundamentals, funding, and changes in risk appetite with time. A good forecasting framework in practice should assemble relevant macroeconomic indicators at their native frequency, align these indicators with the day-to-day credit-spread chronology while maintaining the secrecy of future data, and allow shocks in the responsiveness of spreads in these indicators in alternative market *regimes*. This idea becomes real in our empirical framework: we assemble the daily *BBB–Treasury* spread (ICE BofA BBB minus 10Y U.S. Treasury), align macroeconomic daily series with monthly and quarterly series through causal MIDAS strategies, classify regimes with the help of a Gaussian hidden Markov model for the spread, and then input both linear and tree models with a chronological divide between test and training sets.

Definition 1.5.1 (*Macroeconomic determinants of credit spreads*). Macroeconomic predictors *are medium- and low-frequency variables that capture growth, inflation, and activity—the latter being real GDP, personal consumption expenditure (PCE), the consumer price index (CPI), and rate/term attributes—as well as funds and uncertainty proxies. They are observed in the data set through monthly `PCE`, `CPIAUCSL`, `TERMCBAUTO48NS` and quarterly `GDP` and daily market and funds proxies. By observation, declining growth or increasing inflation is associated with broader spreads, in agreement with increasing anticipated default losses and decreased liquidity (Fama and French, 1989; Gilchrist and Zakrajšek, 2012).*

Real activity: GDP (and corresponding proxies to demand). *Real GDP* is a slow series that serves as a proxy for the revenue and debt-servicing capacity of firms. We adopt quarterly `GDP` and match it to the daily calendar through a *causal* MIDAS transformation. Like with Fama and French (1989) and Gilchrist and Zakrajšek (2012), weaker growth projections increase anticipated default chances and enlarge investment-grade spreads; stronger growth tightens the spreads, instead. Similarly, we add monthly `PCE` (personal consumption expenditures) demand-side proxies upsampled to the daily frequency: weaker consumption expects stress to corporate cash flows and broader spreads.

Background on inflation: CPI. *CPI* also reflects inflationary pressures that regulates genuine cash flows and influence monetary policy. An increase in inflation can potentially reduce real income and exacerbate financial constraints through policy interventions, both of which are associated with broader spreads. We integrate `CPIAUCSL` on a monthly basis, subsequently converting it to end-of-month data during the MIDAS procedure, thereby ensuring that the daily panel accurately represents the most current price-level information.

Industrial production (conceptual baseline). The traditional real-activity indicator frequently employed in spread-based models and nowcasting is noteworthy. Nevertheless, despite its significance in existing research, it is excluded from our practical application; both GDP and PCE effectively encapsulate the slow-evolving activity channel while allowing for a concise feature set.[9]

Economic Stress: The TED Spread

Definition 1.5.2 (*TED spread*). TED spread *is defined as the difference between a short interbank rate and a similar Treasury yield; it covers funding stress and credit risk viewed by banks.*

We also have a daily `TEDRATE` series. Higher TED rates also go with wider corporate spreads, especially at stressful occasions, since they translate to

[9]Industrial production is an effective strength analysis available in the existing literature, but we do not include it here to keep the workflow streamlined and reproducible.

higher marginal cost of borrowing and risk premiums by investors.

Short rates and retail credit terms. In order to obtain household credit market settings and policy stance, we enter the *3-month Treasury bill* `TB3MS` for the short-rate proxy and a loan rate `TERMCBAUTO48NS` (48-month auto loans). Wider short rates and retail lending are associated with broader corporate spreads through broader discount rates, refinancing requirements, and slow consumption demand.

Uncertainty: policy-related measures. The *Economic Policy Uncertainty (EPU)* measures perceived policy and regulatory uncertainty. We inherit category EPU data (on monthly basis) and causally upsample. Persistent EPU rise is theoretically linked to spread widenings beyond standard macro controls (Baker, Bloom, and Davis, 2016), marking precautionary risk premia.

High-frequency market tone. To capture overall market risk appetite, we derive an equity-momentum measure, `SPY_DIFF`, using the close prices of the SPY ETF (extracted from FactSet[10]). Indeed, this quantity is calculated as the discrepancy between the 12- and 26-day exponential moving averages (MACD). We gather observations for the *S&P 500 ETF* (SPY) and create a proxy for momentum based on the difference between two exponential moving averages (EMA12, EMA26). The resulting time series, *SPY_DIFF* (EMA12 - EMA26), is a leading indication of market sentiment that shifts on a daily basis.

Both of these series can serve as very useful technical indicators of momentum for the stock market. Positive momentum episodes shrink credit spreads by encouraging a "risk-on" mood, whereas negative momentum leads to "risk-off" regimes and spread widening (Daniel, Grinblatt, and Titman, 1988). This connection holds with empirical work that shows a large fraction of credit spread

[10]FactSet Research Systems Inc. is a global provider of integrated financial information and analytics that delivers investment professionals accurate information on markets, corporations, and portfolios globally.

variation results from a shared, systematic market factor, usually explained as risk appetite or sentiment, more than from firm-specific fundamentals alone (Collin-Dufresne, R. S. Goldstein, and Martin, 2001). EMA12 is short-term momentum (roughly 2 weeks of trading) with fast response to new momentum in the market and EMA26 is a medium trend (roughly 1 month of trading), the slower, less volatile perspective. By employing both of them, the momentum can be established.

We also include these variables, SPY and SPY_DIFF, for the same reason that credit spread is subject to market mood, too. A $SPY\_DIFF > 0$ indicates a bull market when the S&P 500 goes up, the mood is high, perceived risk is down, and the credit spread will decline and a $SPY\_DIFF < 0$ indicates a bear market when the S&P 500 goes down, uncertainty is on the rise, and the credit spread opens up.

Transitioning from diverse frequencies to a daily panel. Monthly/quarterly indicators (GDP, PCE, CPI, TERMCBAUTO48NS) are seasonally aligned to the daily spread calendar through an economically consistent aggregation causal Mixed Data Sampling (MIDAS) mapping: `PCE` is entered through *sum* (flow), `CPIAUCSL` is entered through *last* (end-of-month index), `TERMCBAUTO48NS` and `GDP` as *mean* (level/rate). Daily indicators (Credit_Spread, TEDRATE, TB3MS, `SPY_DIFF`) ensures time consistency and eliminates look-ahead and follows the mixed-frequencylogic argued by Ghysels, Santa-Clara, and Valkanov (2004).

Definition 1.5.3 (*Market regimes via HMM*). Credit-spread dynamics are regime-dependent. *We capture latent "calm" and "stress" states using a Gaussian Hidden Markov Model estimated on the daily* Credit_Spread *series (Hamilton, 1989). The labels encode changes in mean/volatility (volatility clustering, sudden repricing) and then are employed as extra inputs/conditional information for our forecasting model equations.*

Significance of the regimes for the above variables. Sensitivity is state-dependent: on stress, an increment to TEDRATE or declining SPY_DIFF induces greatly bigger spread movements; spikes to the EPU generate more impact to the spreads with fragility; the refinancing risk is higher with increments to the short-rate (TB3MS) and tighter retail credit (TERMCBAUTO48NS) markets. The spreads load more to sluggish fundamentals where the states are calm (GDP, PCE, CPI, with quasi-linear reactions).

From forecasting to risk use-cases. The same set of signals allows for stress testing and capital planning. Supervisory or internal scenarios are conditioned on macro paths (GDP, CPI, policy rates) and funding stress (TED), while uncertainty states (EPU) act as amplifiers. Our pipeline is given regime labels, which are additional information for error analysis and tail risk measurements (e.g., regime-conditioned VaR), which correspond to outputs with ICAAP-style analyses without a complete structural pricing framework.

Conclusion. GDP and PCE provide the sluggish base; CPI and short rates set the policy backdrop; TEDRATE and SPY_DIFF relay high-frequency funding pressure and market mood; EPU encapsulates persistent uncertainty. A causal mixed-frequency matching and an HMM-regime view connect these indicators together, and the consequent models are interpretable but reactive to inflection points.

Chapter 2

Data and Methodology

Design rationale: dual–pipeline setup To increase robustness and interpretability, we adopt two complementary pipelines. Pipeline 1 prioritizes direct predictability with minimal structural assumptions (causal feature engineering, regime identification via HMM, skewness reduction, standardization and ensemble learners), offering a high–signal benchmark. Pipeline 2 emphasizes structure before prediction—compressing correlated inputs through Factor Analysis, enforcing stationarity (ADF + differencing), and pruning with model–based feature selection—to control multicollinearity and overfitting.

2.1 Hidden Markov Model

Motivation. Credit spreads display features of volatility clustering, abrupt repricing, and macroeconomic as well as funding shock-dependent sensitivities. A linear, homoskedastic model is incapable of summarizing these characteristics in a parsimonious form. Utilization of a *Hidden Markov Model* (HMM) enables an economical representation in which the spread process that is observed is conditionally generated by a finite set of *latent regimes* (e.g., stressful versus tranquil), each with own mean, variance, and transition dynamics (Hamilton, 1989). In our analytical setup, we directly apply the HMM to the daily `Credit_Spread` series in order to obtain regime labels that summarize the present state of the market and complement short-term predictions when used as an extra variable for the predictive models. In addition, regime transition typically occurs in sequential stages (build-up → break → normalization), a structure that is naturally provided by HMMs in multistage processes (Ourston et al., 2003).

Definition 2.1.1 (*Hidden Markov Model with Gaussian emissions*). *An HMM is a quintuple* $\lambda = (S, \mathbb{R}^d, \pi, A, B)$ *with: a finite state space* $S = \{s_1, \ldots, s_N\}$ *for the unobserved chain* (q_t)*; continuous observation space* $\mathbb{R}^d$*; initial state probabilities* $\pi_i = P(q_1 = s_i)$*; transition matrix* $A = (a_{ij})$ *with* $a_{ij} = P(q_{t+1} = s_j \mid q_t = s_i)$ *and* $\sum_j a_{ij} = 1$*; state–dependent emission densities* $B = \{b_i(\cdot)\}$*. With Gaussian emissions,*

$$b_i(o_t) = p(o_t \mid q_t = s_i) = \frac{1}{(2\pi)^{d/2}\,|\Sigma_i|^{1/2}} \exp\Big(-\tfrac{1}{2}(o_t - \mu_i)^\top \Sigma_i^{-1}(o_t - \mu_i)\Big).$$

Here we get $d = 1$ *and* o_t *is the normalized daily credit spread.*

Fundamental Assumptions and Implications

1. **Latent Markov property:** can be expressed as to $P(q_t \mid q_{t-1}, \ldots) = P(q_t \mid q_{t-1})$and this expression fosters persistence and recurrence of regimes through A; the stationarity distribution of A is the long–run time shares.

2. **Conditional independence (observed):** is expressed as $p(o_t \mid q_t, o_{t-1}, \ldots) = p(o_t \mid q_t)$. For the distribution within the regime to be a regime–specific Gaussian law; heteroskedasticity emerges from the variation in variances Σ_i, (excluding GARCH terms).

Inference and decoding

We estimate HMM parameters through the *Expectation–Maximization* (EM) algorithm, also referred to as *Baum–Welch* for HMMs. EM alternates:

- **E–step:** calculate the posterior probabilities of the states using the forward–backward recursions. $\gamma_i(t) = P(q_t = s_i \mid O, \lambda)$ and the two–slice posteriors $\xi_{ij}(t) = P(q_t = s_i, q_{t+1} = s_j \mid O, \lambda)$.

- **M–step:** alter the parameters to optimally fit the anticipated complete–data log initial probabilities $\pi_i \leftarrow \gamma_i(1)$; transitions $a_{ij} \leftarrow \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)}$; for Gaussian emissions, using responsibility–weighted mean and variance formulae $\mu_i \leftarrow \frac{\sum_t \gamma_i(t)\, o_t}{\sum_t \gamma_i(t)}$, $\Sigma_i \leftarrow \frac{\sum_t \gamma_i(t)\,(o_t - \mu_i)(o_t - \mu_i)^\top}{\sum_t \gamma_i(t)}$.

EM increases the (in–sample) log–likelihood monotonically but potentially converging to local optima; we therefore standardize, select the number of states through BIC, and be dependent on numerically stable versions of the recursions (scaling or log–space) (Rabiner, 1989; Cappé, Moulines, and Rydén, 2005; Franzese and Iuliano, 2019). We compute , with EM, the $(\pi, A, \{\mu_i, \Sigma_i\})$ on the training window, decoding on train(Viterbi path) using our code (`hmmlearn`)

Forward–backward recursions (employed during the E–step). Assign the forward and the backward values

$$\alpha_i(t) = p(o_1, \dots, o_t,\ q_t = s_i \mid \lambda), \qquad \beta_i(t) = p(o_{t+1}, \dots, o_T \mid q_t = s_i,\ \lambda).$$

They satisfy

$$\alpha_j(t+1) = \Big[\sum_{i=1}^{N} \alpha_i(t)\, a_{ij}\Big]\, b_j\big(o_{t+1}\big),$$
$$\beta_i(t) = \sum_{j=1}^{N} a_{ij}\, b_j\big(o_{t+1}\big)\, \beta_j(t+1),$$

with initial/terminal conditions

$$\alpha_i(1) = \pi_i\, b_i(o_1), \qquad \beta_i(T) = 1, \qquad \delta_j(1) = \pi_j\, b_j(o_1), \qquad \psi_j(1) = 0.$$

The total likelihood is $p(O \mid \lambda) = \sum_{i=1}^{N} \alpha_i(T)$.

Viterbi decoding (most likely path). The *Viterbi* algorithm estimates the most–likely sequence of states

$$\delta_j(t) = b_j(o_t)\, \max_i \big[\delta_i(t-1)\, a_{ij}\big],$$

by retracing the argmax indicators ψ, resulting in the most probable state trajectory (Rabiner, 1989; Ghojogh, Karray, and Crowley, 2019; Franzese and Iuliano, 2019). (Implementations operate within logarithmic space or utilize scaling factors to remain numerically stable.)

Model order selection

The number of states $N \in \{2, \ldots, 6\}$ is determined by the assistance of the Bayesian Information Criterion (BIC),

$$\mathrm{BIC} = -2\ell + p \ln T, \qquad p = N^2 + 2N - 1 \quad (d = 1),$$

with maximized log–likelihood ℓ and sample size T. The count p retains A (row–stochastic), π ($\sum = 1$), and the N regime means/variances. We then re-fit the Gaussian HMM and decode regimes out of sample.[11]

Definition 2.1.2 (*Bayesian Information Criterion (BIC)*). *For a candidate model λ fitted on a sample of size T, the* Bayesian Information Criterion *is*

$$\mathrm{BIC}(\lambda) \;=\; -2\,\ell\big(\hat{\theta}_\lambda; O_{1:T}\big) \;+\; p_\lambda \ln T,$$

where $\ell(\hat{\theta}_\lambda; O_{1:T}) = \log p(O_{1:T} \mid \hat{\theta}_\lambda)$ is the maximized log–likelihood and p_λ is the number of free parameters. (Schwarz, 1978).

The Contribution of BIC BIC exchanges between–sample fit and parsimony (through ℓ and $p \ln T$). Relative to criteria which punish complexity much less (i.e., AIC), BIC will favor sparse models more and more often as T is very large and is typical for the appropriate order of the true model under standard regularity assumptions.

How we compute p for our HMM (univariate Gaussian). With N hidden states, row–stochastic A, and initial probabilities that add up to one, as well as with Gaussian emissions with *mean* and *variance* per state, the number of free parameters is

$$\begin{aligned} p_N &= \underbrace{N(N-1)}_{\text{transition matrix } A} + \underbrace{(N-1)}_{\text{initial probabilities } \pi} + \underbrace{N}_{\text{means } \mu_i} + \underbrace{N}_{\text{variances } \Sigma_i} \\ &= N^2 + 2N - 1. \end{aligned}$$

(For multivariate observations of dimension d with full covariances, add $N \cdot d$ means and $N \cdot \frac{d(d+1)}{2}$ covariance parameters; we remain univariate, $d = 1$.)

[11]When $d > 1$ with full Σ_i, p increases by $N \cdot d(d+1)/2$; we remain with univariate for simplicity of correspondence with the forecasting workflow.

The Integration of BIC. For leak prevention, on the training window we iterate through $N \in \{2, \dots, 6\}$:

1. Fit a Gaussian HMM with N states to normalized `Credit_Spread` by using EM (`hmmlearn`).
2. Calculate the best log-likelihood with the formula $\ell_N = \texttt{model.score}(O_{\text{train}})$.
3. Define $\text{BIC}_N = -2\,\ell_N + p_N \ln T$, where $p_N = N^2 + 2N - 1$ and $T = |O_{\text{train}}|$.
4. Choose $\hat{N} = \arg\min_N \text{BIC}_N$ (in the case of a tie, prefer the smaller N).

Computational issues. Because EM can have local optima, we scale the inputs and execute numerically stable forward–backward recursions on the log/scale coordinates. BIC makes us want to add extra states only where the gain in fit is worth the cost of the $\ln T$ cost to prevent overfitting with small to medium-sized samples.

The resulting *selection of the HMM model* summary (train window)

N	logL	BIC
2	−3455.899	6971.472
3	−1313.799	2746.944
4	296.944	−397.818
5	−1313.727	2917.295
6	1269.469	−2138.275

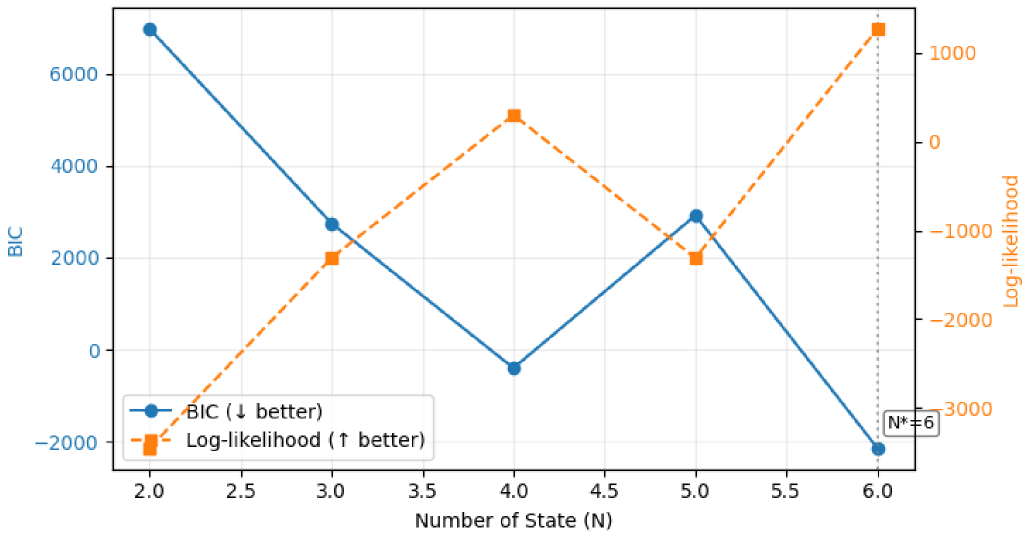


Figure 2.1: Choice of the HMM model (2≤ N ≤6). The solid curve displays BIC (smaller values better) and the dashed curve the maximized log–likelihood (larger values better) of the Gaussian HMMs estimated from the *training* window of normalized `Credit_Spread`. The optimal BIC is $N^{\star}=6$ (vertical dashed line), which we adopt for regime extraction.

Minimum Bayesian Information Criterion (BIC) is at $N^{\star} = 6$ (BIC = −2138.275), in agreement with the rising log-likelihood at $N = 6$. Decrease at $N = 4$ (leading to a negative BIC) means that a four-state model accounts for a substantial amount of heterogeneity; however, the increased fit at $N = 6$ overpowers the penalty of $\ln T$. Non-monotonicity of the log-likelihood at $N = 5$ illustrates a well-known side-effect of the local optima nature of the Expectation-Maximization (EM) algorithm: our version sets the random seed in order to make the results replicable, but still results may change with differently constructed initial settings. If one wishes, it is possible to introduce several random restarts (with differently constructed seeds, exploring thereby at each call a different local optimum) and to pick the resulting iteration with the highest ℓ_N before computation of BIC, in the name of stabilization of the model-choice process.

Economic interpretation for credit spreads

The HMM divides the history into segments with *levels* and *volatility*, which are different. In *calm* regimes, spreads remain low and shocks propagate in quasi–linear fashion; in *stress*, the higher mean and variance imply that tiny macro/funding shocks entail more–than–proportional moves. This aligns with the state dependence of default premia, liquidity premia, and risk appetite. Thus, regime labels have two roles: a data–based short-hand market state indicator to allow real–time tracking and forewarning of early indications; state-dependent conditioning to monitor turning points. HMMs also do better under *rare* stress events than fixed models by capitalizing on regime persistence in the latent chain (Ourston et al., 2003).

Our code implementation (univariate Gaussian HMM)

Our HMM is simulated in the `MarketRegimeAnalyzer` by the following formula:

1. **Preprocessing.** Normalize the daily `Credit_Spread` by `Standard-Scaler`.

2. **Order selection.** Fit $N = 2, \ldots, 6$ train the model up to the training window and calculate the corresponding ℓ and BIC, and select $\hat{N} = \arg\min \mathrm{BIC}$.

3. **Refit and decode.** Refit the $\hat{N}$–state model on training;get regime Viterbi labelling for test and train using `model.predict`.

4. **Feature integration.** Append the integer `Regime` label to the feature built with Pipeline 1 (RF, XGBoost, TCN) and use it as a condition the data within Pipeline 2.

In instances where $\hat{N} > 2$, an optional aggregation of the granular states is performed within two macro-regimes—calm and stress—to present regime-conditioned summaries of risk metrics, such as Value at Risk (VaR).

Diagnostics and robustness within our scope

We present regime–specific summary statistics (mean, std, counts). Including such regime dummy in our Random Forest, XGBoost and TCN models allows for forecasting for market regimes and thus minimizes out-of-sample RMSE for higher volatility scenarios.

Table 2.1: Regime Analysis

Regime	Mean	Std	Count
0	1.534273	0.110000	571
1	5.296320	1.468722	250
2	1.916177	0.097387	769
3	1.533063	0.107797	555
4	1.117887	0.202696	1898
5	2.476161	0.248573	995

Figure 2.2 in conjunction with Table 2.1 shows six different states in the historical credit-spread series. Regime 4 is by far the most persistent, with close to 1 900 daily observations, the lowest mean spread (1.12 %) and medium volatility (0.20 %), describing long "calm" market conditions. By contrast, Regime 1—comprising only 250 days—exhibits an exceptionally high mean spread (5.30 %) and the highest standard deviation (1.47 %), which corresponds to acute stress episodes like the 2008–09 financial crisis and the COVID-19 shock of early 2020.

This six-state decomposition reveals that the HMM is able to pick up not just the headline “calm” vs. “stress” dichotomy, but also more subtle distinctions in the dynamics of credit spreads. The fact that there is such a large difference in means and variances enables downstream forecasting models to condition their coefficients appropriately; i.e., a given shock to GDP or liquidity will have very different predictive significance in Regime 1 compared to Regime 4. In addition, the relative frequency of each regime gives modelers

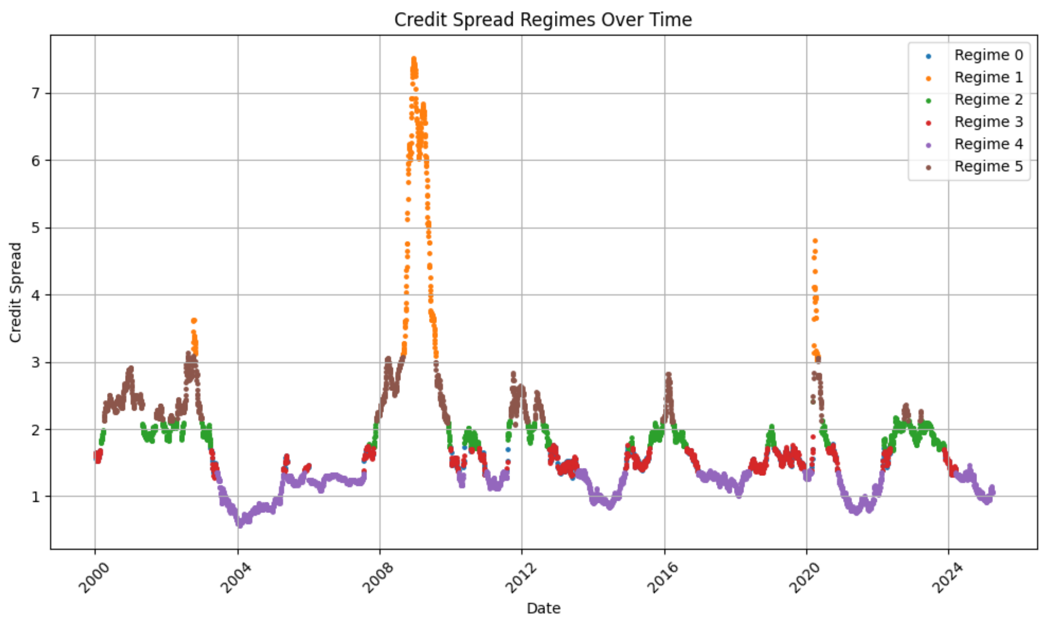


Figure 2.2: Credit Spread Regime Over Time. This time-series plot shows the daily credit spread shaded by the six regimes identified by HMM. The clear clusters—ranging from low-spread "calm" periods to high-spread stress periods—illustrate the model's capacity to divide market conditions and uncover the timing and duration of each regime.

and risk managers information regarding the expected duration of each state, and thus allows for more accurate scenario analysis and the development of early-warning indicators[12]

[12] In the first presentation (Section 3.2), we expressed the objective of estimating a binary regime indicator. $z_t \in \{0,1\}$ to distinguish between "stress" and "calm" market regimes. However, the BIC-based model selection procedure chose a six-state Hidden Markov Model (Table 2.1) as empirically optimal, which allows us to capture more fine-grained gradations of market stress that would not be picked up by a parsimonious two-state specification. Namely, those six latent states significantly differ in both the volatility and mean of the credit spread:

- States 0, 3, and 4 illustrate small mean spreads ($< 1.6\%$) and have limited volatility, supporting "calm" market conditions; These states (1, 2, and 5) have broader average spreads ($> 1.9\%$) and higher volatility, which registers instances of "stress."

To make this outcome consistent with the original binary-regime objective, we group the six states into two macro-regimes for the forecasting exercise:

$$\text{Calm} = \{0,3,4\}, \quad \text{Stress} = \{1,2,5\}.$$

Why HMM helps forecasting (and risk). Regime labels reduce residual heteroskedasticity for the learners and sharpen short–horizon responsiveness around turning points. The same labels support *regime–conditioned* tail metrics (VaR at 90/95/99%) in our `RiskAnalyzer`, clarifying how extremes differ between calm and stress and aligning the empirical engine with stress testing and ICAAP–style assessments—without imposing a full structural pricing model.

2.1.1 Regime–conditioned risk analysis

Empirical setup and what it is for

Our goal is to turn latent regimes into *actionable* tail–risk metrics for daily spread changes

$$\Delta CS_t \equiv CS_t - CS_{t-1} \quad \text{(in basis points)}.$$

Given the decoded state path $\{q_t\}$ from the HMM, we estimate a separate loss distribution for each regime $r \in \{0, \dots, \hat{N} - 1\}$ and compute

$$\mathrm{VaR}_\alpha^{(r)} = \inf\{x : F_r(x) \geq \alpha\}, \qquad \mathrm{ES}_\alpha^{(r)} = \mathbb{E}\big[\Delta CS_t \,\big|\, \Delta CS_t \geq \mathrm{VaR}_\alpha^{(r)},\ q_t = r\big].$$

where F_r represents the empirical cumulative distribution function of ΔCS_t for regime r. We are truly interested in linear quantiles, most prominently, in the upper tail. We also observe a model-independent, continuous view of the point of reference,

$$\widehat{\mathrm{VaR}}_{t,\alpha}^{(W)} = \inf\big\{x : \widehat{F}_{t-W+1:t}(x) \geq \alpha\big\}$$

This calculation is performed during the preceding period of W days, given ΔCS; it constitutes the calibration reading of a regimen indicator for periods after the first sample.

Why we do it. It measures how much daily spread widening we need to absorb *when the latent state is* r, rather than diluting stress and calm in one homoskedastic pool. State–adaptive capital add–ons as well as restrictions.

With spread DV01 per bp being SDV01[13] a regime VaR is translated into P&L as

$$\mathrm{VaR}^{\mathrm{P\&L}}_{\alpha,r} \;=\; \mathrm{SDV01}\times \mathrm{VaR}^{(r)}_{\alpha}, \quad \mathrm{ES}^{\mathrm{P\&L}}_{\alpha,r} \;=\; \mathrm{SDV01}\times \mathrm{ES}^{(r)}_{\alpha}.$$

With the filtered probabilities $\pi_{t|t}(r) = P(q_t = r \mid O_{1:t})$, we create a *mixture* buffer,

$$\mathrm{VaR}^{\mathrm{mix}}_{t,\alpha} \;=\; \inf\Big\{x : \sum_{r}\pi_{t|t}(r)\, F_r(x) \ge \alpha\Big\},$$

which rises in accordance with stress but falls under stable conditions, as well as which fits perfectly with ICAAP-type planning for capital.

Estimators and implementation

The `RiskAnalyzer` operates in basis-points with auto-scaling of levels, applies linear quantiles to map VaR and the mean excess VaR into ES, and requires at least a minimum of a sample size per regime to prevent noisy-tails. It has two operating modes: (a) *decoded–state* metrics (set $\mathcal{R} = q_t$) for thresholds/alerts; (b) Probability-weighted mixture measures of the risk metrics. One such simple exceedance backtest holds calibration in its place:

$$\widehat{p}_{\alpha} \;=\; \frac{1}{T}\sum_{t=1}^{T}\mathbf{1}\{\Delta CS_t > \mathrm{VaR}^{(q_t)}_{\alpha}\} \;\approx\; 1-\alpha.$$

How we do this in practice

1. **Limits and alerts.** For day–to–day control we set regime–adaptive thresholds using $\mathrm{VaR}^{(q_t)}_{95}$ as a limit and $\mathrm{ES}^{(q_t)}_{95}$ as a breach trigger.

2. **Buffers of capital.** For such horizons as given, we utilize $\mathrm{VaR}^{\mathrm{mix}}_{t,\alpha}$, adjusted by the portfolio's SDV01, such that it will allow for size add-ons that by themselves increase under stress.

3. **Alignment of forecasts and related risks.** The remaining layer computes regime–conditioned VaR of future–step error forecasting such

[13]**DV01** (Dollar Value of a Basis Point): a 1 bp variation in risk–free yield sensitivity, **SDV01** (Spread DV01): variation in credit spread sensitivity. If VaR comes in spread basis point form, the P&L transformation results in a multiplication by SDV01.

that model–risk buffers line up with the same state information that learners make the most.

4. **Scenario hooks.** The regime tails can be combined with the macro paths to create supervisory or internal scenarios without the assumption required to assume the complete structural pricing model.

Empirical findings and diagnostics

Figures 2.3–2.4 encapsulate the tail-risk component generated by our `RiskAnalyzer`. The dynamic benchmark $\widehat{\mathrm{VaR}}_{t,0.95}^{(W)}$ strongly correlates with the credit spread size and spikes during important stress periods (GFC 2008–09, COVID-19 2020), in alignment with upper ΔCS acting as a stress indicator. Obvious state dependency of tails in the histogram of the regime: on the training window, *Regime 1* focuses the highest 95% VaR (roughly 13 bp *Regime 5* (about 6 bp), while calm states (*Regimes 0,2,3,4*) cluster in the 3–4 bp range.

In the out-of-sample data set, the pattern holds where states do (e.g., *Regime 4* remains the calmest) Furthermore, the non-existence of *Regime 1* in the test set corresponds to the non-existence of extremal events in that scenario. The TRAIN panel provides quantitative data. $\mathrm{VaR}_{95}^{(r)} \in \{3,4,4,4,3,6\}$ bp for $r = 0,\dots,5$ and $\mathrm{ES}_{95}^{(r)}$ between $\sim$ 4.35 and 18.79 bp; the TEST panel shows comparable magnitudes of the materialized states. Pattern embraces regime–adaptive constraints and capital cushions: with portfolio SDV01, the 95% one-day P&L buffer in *Regime 1* is SDV01 $\times$ 13 bp, versus SDV01 $\times$ 3 bp in *Regime 4*.

Table 2.2: Regime-conditioned VaR/ES on TRAIN (ΔCS in bp).

	Regime 0	Regime 1	Regime 2	Regime 3	Regime 4	Regime 5
N	571	250	769	554	1898	995
mean	0.06	0.03	-0.08	-0.03	-0.03	0.09
std	2.76	7.73	3.17	2.54	1.91	3.62
VaR_{90}	3.00	10.00	3.00	3.00	2.00	4.00
ES_{90}	5.18	15.23	4.99	4.60	3.17	6.86
VaR_{95}	4.00	13.00	4.00	4.00	3.00	6.00
ES_{95}	6.84	18.79	5.88	5.97	4.35	9.50
VaR_{99}	8.30	23.53	8.00	8.00	5.00	11.00
ES_{99}	13.17	29.00	10.63	10.00	6.95	14.75

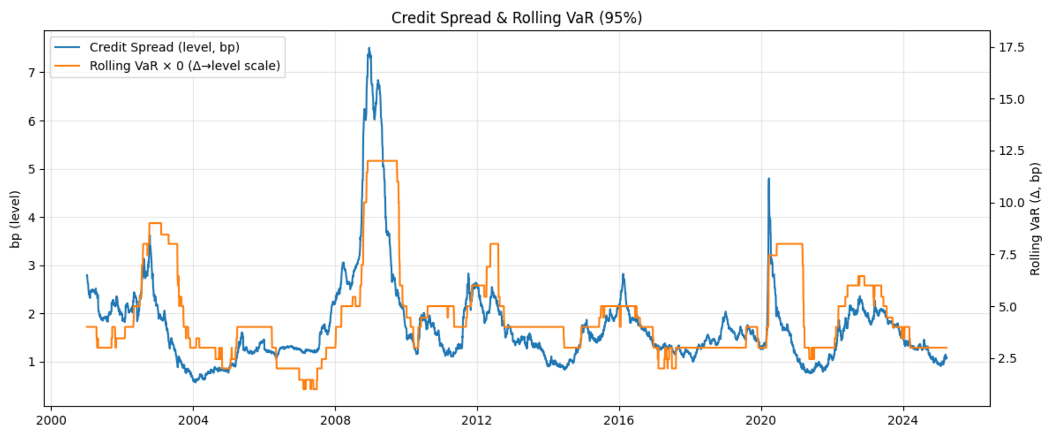


Figure 2.3: Blue is the level of credit–spread (converted to bp to facilitate comparison); the 95% rolling VaR of the day changes ΔCS is the orange staircase, plotted with a secondary axis scaled in the plot for overlay display. Peaks correspond to crisis years (2008–09, 2020), as expected, verifying that the upper tail of ΔCS is state–dependent and increases abruptly in stress.

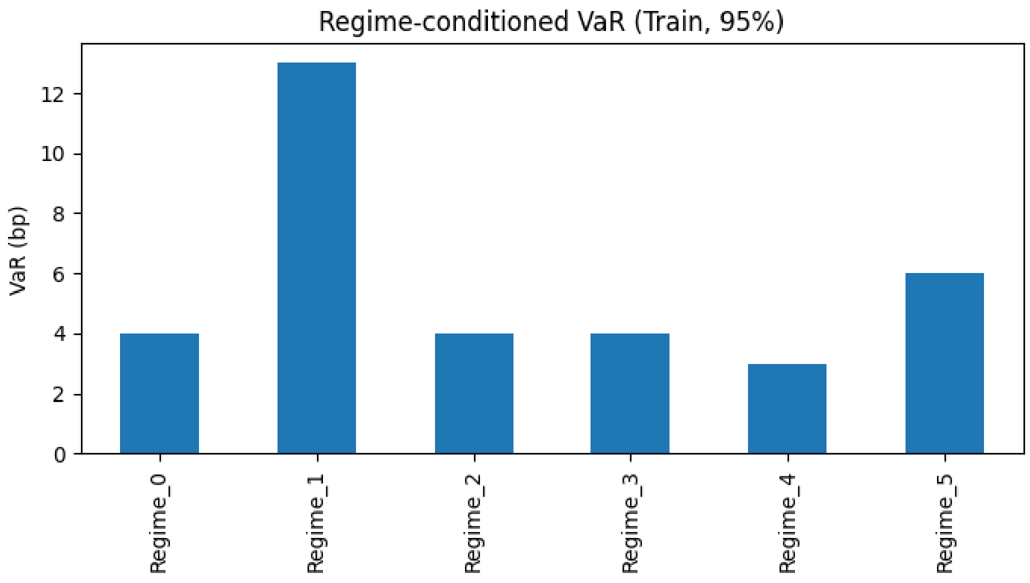


Figure 2.4: Regime-conditioned VaR (TRAIN, 95%). Bars display $\mathrm{VaR}_{95}^{(r)}$ (bp) estimated during each decoded state with. The ordering *Regime 1* $\gg$ {0,2,3,4} with *Regime 5* in between mirrors the summary table (TRAIN): $\mathrm{VaR}_{95}^{(r)} = \{4, 13, 4, 4, 3, 6\}$ bp and $\mathrm{ES}_{95}^{(r)} = \{6.84, 18.79, 5.88, 5.97, 4.35, 9.50\}$ bp. This spread across states invokes regime–adaptive limits along with capital add-ons.

2.2 Missing data and MIDAS alignment

Calendar and imputation (post–split) All the series are synchronized to a business–day calendar. Missing values are imputed *within partitions*, i.e., after splitting the data into the chronological train/test split to prevent leakage. Forward-fill (Last Observation Carried Forward, LOCF) and, if required, a single back-fill at the beginning of each partition are used; a consistently weak coverage variable is excluded. Imputation by target is not undertaken prior to data being properly comprehended, and extensive normalization—including reduction of skewness and standardizing—is learned during the training time window and subsequently transferred to the validation/test sets.

Reverse MIDAS mapping (daily synthesis from lower frequency data). Low–frequency forecasts (quarterly or monthly) are also translated to everyday characteristics based on a reverse MIDAS construction that maintains the original period identity while adding a reasonable intra–period profile (Ghysels, Santa-Clara, and Valkanov, 2007; Foroni, Marcellino, and Schumacher, 2015). Let the realized low–frequency value in month m during the D_m business days be denoted by L_m. Let a parametric weight profile $w_{m,d}(\theta) \geq 0$ (Almon or Beta kernel) apply, then the synthetic series at a daily frequency is

$$x^{\mathrm{HF}}_{m,d}(\theta) = L_m\, w_{m,d}(\theta), \qquad d = 1, \dots, D_m,$$

and an *aggregation operator* mirrors the economic nature of the variable:

$$\mathcal{A}_m(x_{m,1:D_m}) = \begin{cases} \sum_{d=1}^{D_m} x_{m,d} & \text{(flows)}, \\ \frac{1}{D_m}\sum_{d=1}^{D_m} x_{m,d} & \text{(monthly mean levels)}, \\ x_{m,D_m} & \text{(end-of-month levels / last)}. \end{cases}$$

In order to save the monthly observation, weights are normalized accordingly:

$$\begin{aligned} \sum_{d=1}^{D_m} w_{m,d} &= 1 && \text{(for flows)} \\ \frac{1}{D_m}\sum_{d=1}^{D_m} w_{m,d} &= 1 && \text{(for mean levels, implying } \sum_{d=1}^{D_m} w_{m,d} = D_m\text{)} \\ w_{m,D_m} &= 1 && \text{(for end-of-month levels / last)} \end{aligned}$$

Identification and estimation (train–only). Since the re–aggregation restriction necessitates $L_m = \mathcal{A}_m(x^{\mathrm{HF}}_{m,\cdot}(\theta))$ by construction, the naive least–squares fit $\sum_m \big(L_m - \mathcal{A}_m(x^{\mathrm{HF}}_{m,\cdot}(\theta))\big)^2$ is uninformative (identically zero). We therefore identify θ by adding a *shape* criterion on the *training* window $(t \leq T_{\mathrm{train}})$ only:

$$\hat{\theta} = \arg\min_{\theta}\ \lambda \sum_t \left[\Delta^k x^{\mathrm{HF}}_t(\theta)\right]^2 \quad \text{s.t.} \quad \mathcal{A}_m\big(w_{m,\cdot}(\theta)\big) = 1,$$

with $k = 1$ or 2 (first/second–difference smoothness).[14] If there is an existing informative daily proxy $z_{m,d}$ readily available (say, market–based indicator), then we optionally align normalized shapes,

$$\hat{\theta} = \arg\min_{\theta} \sum_{m,d}\big(w_{m,d}(\theta) - \tilde{z}_{m,d}\big)^2 \quad \text{s.t.} \quad \mathcal{A}_m\big(w_{m,\cdot}(\theta)\big) = 1,$$

[14]In code: `smooth_lambda` controls λ and `smooth_order` sets $k \in \{1, 2\}$.

again estimated on train only.[15] We use Almon and Beta kernels with bound constraints and L–BFGS–B optimization; the learned weights are then used to produce the whole calendar with the daily `_midas` features with no look–ahead.

Details of implementation (as in our code). We identify candidate low–frequency columns by coverage and then make adjustments to the MIDAS weights only up to a data–driven `train_end_date` (80% chronological split). Aggregators are given by semantics (flows: `PCE`→sum; monthly indices: `CPIAUCSL`→last; rates/levels: `TERMCBAUTO48NS, GDP`→mean). Their respective day-of-the-month profiles also carry along the month level identity, get appended to the business–day panel, and then undergo the same pipeline of skewness reduction and standardization that high–frequency series receive—fit on train, to validation/test. This preserves causality and avoids leakage while leaving the door open to learning at a day–frequency that derived from greater intra-month fluctuations(Andreou, Ghysels, and Kvedaras, 2010).

Significance of Research. The Reverse MIDAS model preserves economically interpretable monthly frequency via accurate re-aggregation, while at the same time presenting continuous day-of-month signals that blend well with regime identification and short-term predictor incorporation. Pragmatically, this process works to improve confluence over time between intraday spreads and macoreconomic factors in all cases without necessitating any interpolation constraint that's completely arbitrary. Furthermore, it's reconcilable with our stringent agreement on train-only preprocessing that we adhere to through all of the complete analytical process.

[15]In code: `proxy_alpha` enables a shape–matching penalty; the proxy is provided through `proxies={col: proxy_name}`.

2.3 Skewness Reduction

Definition 2.3.1. *The skewness (or asymmetry) of a random variable X is defined as the normalized third central moment:*

$$\gamma_1(X) = \frac{\mathbb{E}\big[(X-\mu)^3\big]}{\sigma^3},$$

where $\mu = \mathbb{E}[X]$ and $\sigma^2 = \mathbb{E}[(X-\mu)^2]$ are the mean and variance of X, respectively (Box and Cox, 1964; Yeo and R. A. Johnson, 2000).

A very skewed distribution may invalidate:

- numerical stability of scaling methods for features and factor analysis (Cameron and Trivedi, 2010);
- regression and PCA models' assumption of Gaussian approximation;
- such ensemble methods as Random Forest and XGBoost's performance and convergence are prone to significant outliers (Tang and Yan, 2010).

To reduce heavy tails and induce feature symmetry, our Python pipeline performs a conditional

$$X' = \begin{cases} \ln(1+X), & \text{if } |\gamma_1(X)| > \tau, \\ X, & \text{otherwise,} \end{cases}$$

with threshold $\tau = 0.75$. The process is run by:

1. calculating the skewness of every variable with `scipy.stats.skew` (Virtanen and al., 2020);
2. applying $\ln(1+X)$ only to those features whose absolute skewness exceeds τ;
3. re-entering the transformed variables into the data set before scaling and factor analysis.

The reduction of skewness process is comparatively notable in forecasting credit spreads as it:

- decreases financial attribute diversity, thus reducing transient fluctuations.
- improves the standard distribution of residuals in linear models. to receive further precise evaluations of interaction with macroeconomic indicators like TED spread, CPI and GDP
- suppresses Random Forest's and XGBoost's outlier value sensitivities to improve out-of-sample stability as well as facilitate regime shifts at unstable market conditions.

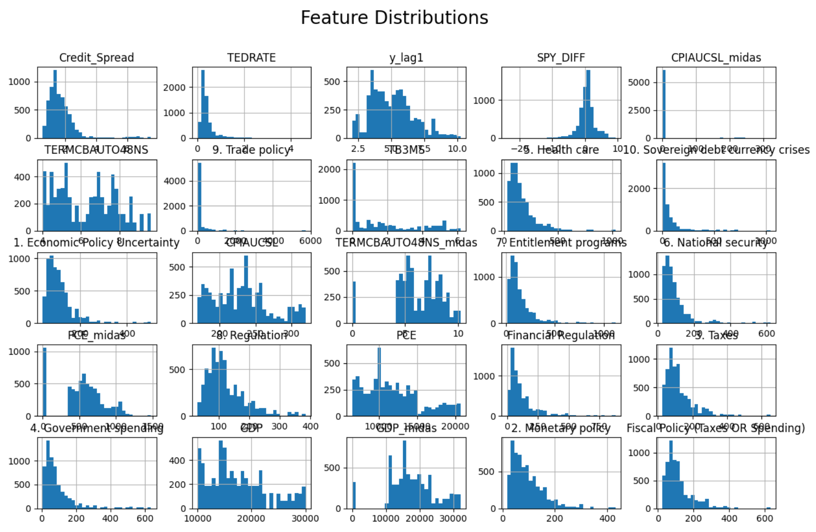


Figure 2.5: Feature Distribution. The histograms show stark *right skewness* and *heavy tails* for most macro, liquidity, and policy variables (*TEDRATE*, *TB3MS*, *TERMCBAUTO48NS* and their _midas versions, *PCE_midas*, *CPIAUCSL_midas*, category–specific EPU indices), with mass focused around zero and occasional spikes in times of stress (e.g., 2008–09 crisis, COVID). The *Credit_Spread* displays a positive distribution with a stretched right tail, which is consistent with episodic widenings; *SPY_DIFF* is instead roughly symmetric and centered around zero, as expected from a momentum indicator. Some MIDAS–transformed series display blocky or multimodal patterns because of frequency alignment. These empirical characteristics encourage the application of *skewness reduction* (conditional log1p transformation) before standardization, in order to stabilize variance and reduce the effect of outliers on factor analysis, HMMs, and prediction modeling.

2.4 Standardization

Standardization reshapes every variable so that it contains a mean of zero and a variance of one, thereby ensuring every feature informs reasonably well on the estimation and optimization tasks. For a feature X, the standardized (z–score) counterpart is

$$Z = \frac{X - \mu_{\text{train}}}{\sigma_{\text{train}}},$$

where μ_{train} and σ_{train} are the mean and standard deviation computed only from the training set (or training fold) to avoid look-ahead bias and information leakage in time-series settings (Hyndman and Athanasopoulos, 2018; Kuhn and K. Johnson, 2013).

Why standardize in our pipeline

Standardization is a necessary step for a majority of components of our forecasting system:

- **Linear methods and factor models.** Regressions, PCA and factor analysis all implicitly rely on second-order structure; without scaling, variables with larger units overwhelm the covariance matrix and loadings and coefficients are distorted (Jolliffe, 2002; Bishop, 2006). Standardization puts variables on the same scale, enhancing numerical conditioning and interpretability (Hastie, Tibshirani, and J. Friedman, 2009).

- **Hidden Markov Models (Gaussian emissions).** HMMs having Gaussian emissions presume similar scales for inputs; standardization decreases likelihood optimization instability and prevents degenerate fits in the multi-feature scenario, then provides more stable BIC comparisons and decoding of the states (Hamilton, 1989; Bishop, 2006).

- **Tree-based ensembles.** Even though Random Forest and XGBoost are theoretically invariant to monotone feature scaling, standardization can enhance optimization dynamics (e.g., learning-rate interactions, column subsampling) and make hyperparameter search easier by maintaining feature magnitudes similar across folds(Hastie, Tibshirani, and J. Friedman, 2009; Pedregosa, Varoquaux, Gramfort, et al., 2011).

Limitations and Considerations

We standardize *predictor* variables—not the regime or target labels in tree-based models. Normalization of the target for linear baselines or regularized models might serve for numerical stability but is not required in our specification. A stable alternative (median/IQR scaling) is preferable when outliers are very extreme even after reduction of skewness, but in our application standard z-scores after $\log 1p$ strike a reasonable balance between robustness and efficiency (Kuhn and K. Johnson, 2013; Hastie, Tibshirani, and J. Friedman, 2009).

As a whole, our credit-spread workflow's standardization facilitates numerical conditioning, comparability of features on a macro basis, liquidity and technical indicator basis, and regime detection and stability in out-of-sample predictions.

2.5 Dimensionality reduction, stationarity, and feature selection

For machine–learning algorithms, per se multicollinearity and redundancy are not necessarily an obstacle to pure prediction, but they can distort parameter estimates, inflate sampling variance, and obscure measures of variable importance and interpretability (Hastie, Tibshirani, and J. Friedman, 2009; Breiman, 2001). In Pipeline 2, to enhance the predictive power of the linear regression we thus impose three precautions: (i) *Factor Analysis* to summarize highly correlated predictors in terms of a few latent drivers; (ii) imposition of *stationarity* through the Augmented Dickey–Fuller (ADF) test using iterative differencing; and (iii) *model–based feature selection* using Random Forests. This order (compression → stationarity → selection) provides a parsimonious and stable representation prior to predicting credit spreads.

2.5.1 Factor Analysis to handle multicollinearity

Let $x_t \in \mathbb{R}^p$ be the *standardized* vector of observed predictors at day t (each component has training–sample mean 0 and variance 1), where p is the number of raw variables. We suggest the *common–factor model*

$$x_t = \Lambda f_t + \varepsilon_t, \qquad \mathbb{E}(f_t) = 0, \quad \mathrm{Var}(f_t) = I_k, \quad \mathrm{Var}(\varepsilon_t) = \Psi \text{ (diagonal)},$$

where:

- $f_t \in \mathbb{R}^k$ are the latent factor scores at time t; $k \ll p$ is the number of retained common factors;
- $\Lambda \in \mathbb{R}^{p\times k}$ is the *loading matrix*; its (i, j) entry λ_{ij} indicates how intensively observed variable i co–moves with factor j;
- $\varepsilon_t \in \mathbb{R}^p$ is the idiosyncratic (specific) component, which is assumed to be uncorrelated across variables after controlling for factors;
- $\Psi = \mathrm{diag}(\psi_1, \ldots, \psi_p)$ gathers the *uniquenesses* $\psi_i = \mathrm{Var}(\varepsilon_{it})$;
- I_k is the $k \times k$ identity, normalising factors to unit variance for identification.

The covariance structure follows

$$\Sigma_x \approx \Lambda\Lambda^\top + \Psi,$$

such that the common (shared) variance is given by $\Lambda\Lambda^\top$ and the distinctive variance for single variables by Ψ (Harman, 1976; Jolliffe, 2002). We approximate (Λ, Ψ) by *minimum residuals* (`minres`), i.e.

$$\min_{\Lambda,\,\Psi \text{ diag}} \left\| \Sigma_x - \Lambda\Lambda^\top - \Psi \right\|_F^2 \quad \text{s.t.} \quad \Psi \geq 0,$$

Then perform an *orthogonal varimax rotation* $\Lambda^* = \Lambda R$ with $R^\top R = I_k$ in a bid to enhance interpretability without compromising. The varimax criterion

$$\mathcal{Q}_{\text{varimax}} = \sum_{j=1}^{k} \left[\frac{1}{p} \sum_{i=1}^{p} \lambda_{ij}^4 - \left(\frac{1}{p} \sum_{i=1}^{p} \lambda_{ij}^2 \right)^2 \right]$$

is set up to distribute each factor's variance to a small set of measures for the purpose of not allowing cross-loadings and for labeling appropriately (Costello and Osborne, 2005; Fabrigar et al., 1999). The factor count k is chosen by a scree–plot and Kaiser's rule (retain eigenvalues >1), truncated at $k \leq 10$; for big panels one might prefer informations–criteria (Jushan Bai and Ng, 2002). Then we compute the factor scores $\hat{f}_t$ (the k-dimensional forecasts in the time series setting) and input them as predictors with *low collinearity* (Stock and Watson, 2002).

Rationale for this step. Figure 2.6's correlation heatmap reveals the presence of tight clusters of indices for macroeconomics, liquidity and funding, sectoral credit, and policy uncertainty, and therefore amounts of multicollinearity and redundancy. Projection of x_t onto f_t decreases such structural redundancy inherent in the data, thereby dampening estimation noise and the proliferation of free parameters, but substantively maintaining the common signal responsible for credit spread fluctuations.

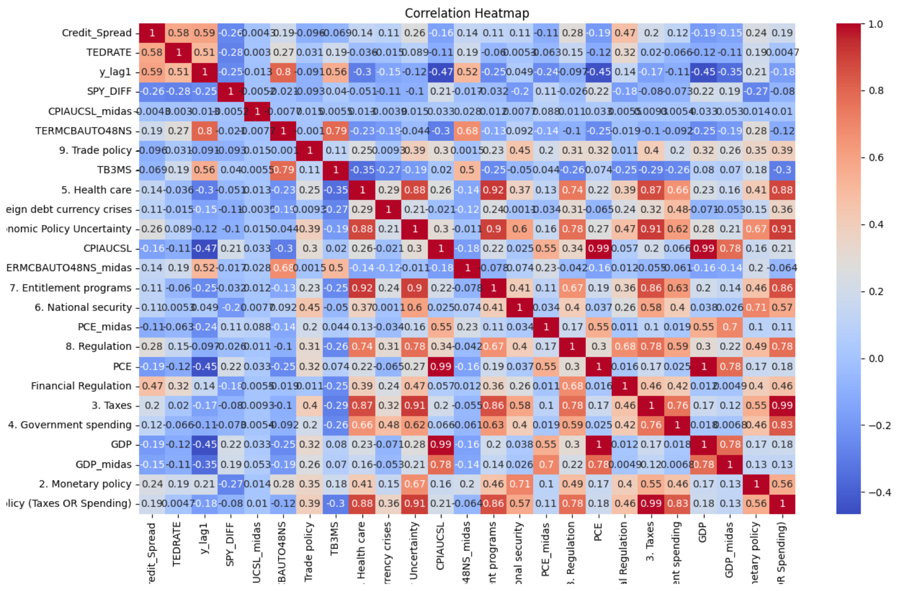


Figure 2.6: Heatmap of Correlation. Areas with high positive correlation (in red) imply that multicollinearity among the macroeconomic, liquidity, sectoral, and policy variables exist. This duplication necessitates dimension reduction via Factor Analysis prior to generating forecasts.

2.5.2 Stationarity via ADF and differencing

In an attempt to correct for potential misperceptions resulting from common tendencies, every factor score and every series remaining are tested for the presence of a unit root. For a scalar time series y_t, the ADF regression is

$$\Delta y_t = \alpha + \beta t + \phi y_{t-1} + \sum_{i=1}^{m} \gamma_i, \Delta y_{t-i} + \varepsilon_t,$$

where:

- the term $\Delta y_t = y_t - y_{t-1}$ defines the first-difference operator ∇y_t;
- α is an intercept and βt an optional time trend;
- ϕ is the unit root test coefficient, with $H_0 \colon \phi = 0$ (unit root) vs. $H_1 \colon \phi < 0$ (stationary);
- The m lagged first differences $\{\Delta y_{t-i}\}_{i=1}^{m}$, captured by the γ_i, contain the first-order serial
- ε_t is a zero–mean disturbance.

If the ADF p–value is greater than 0.05, we difference the series and test again, repeating until

$$y_t^{(d)} = \nabla^d y_t \quad \text{satisfies ADF } p\text{–value} < 0.05,$$

where d denotes the minimum differencing order necessary to achieve stationarity (Dickey and Fuller, 1979; Said and Dickey, 1984).

Why this step. Stationarity fixes second–order structure (means/covariances) that is then employed by linear baselines and by the impurity measure within the tree, dampens spurious regression in the event that any such predictors and credit spread have shared common trends, and operates to make temporal cross–validation comparable between rolling windows.

2.5.3 Random–Forest feature selection

Take the set of candidate possibilities as

$$\mathcal{Z}_t = \left\{\hat{f}_{1t}, \ldots, \hat{f}_{kt}, \text{ other designed inputs}\right\},$$

where $r_t \in \{1, \ldots, S\}$ is the additional market–regime indicator (from a Gaussian HMM on the spread) (Hamilton, 1989). We train a Random Forest regressor on the training window and calculate impurity–based importances. In regression trees the impurity at a node $\mathcal{N}$ with $n_{\mathcal{N}}$ observations and responses $\{y_i\}$ is the sample variance,

$$i(\mathcal{N}) = \frac{1}{n_{\mathcal{N}}} \sum_{i \in \mathcal{N}} \left(y_i - \bar{y}_{\mathcal{N}}\right)^2,$$

with $\bar{y}_{\mathcal{N}}$ the node mean. A split s on variable j divides $\mathcal{N}$ into left/right children $\mathcal{N}_L, \mathcal{N}_R$; its impurity reduction is

$$\Delta i(s) = i(\mathcal{N}) - \frac{n_L}{n_P} i(\mathcal{N}_L) - \frac{n_R}{n_P} i(\mathcal{N}_R),$$

where $n_P = n_{\mathcal{N}}$, $n_L = n_{\mathcal{N}_L}$, $n_R = n_{\mathcal{N}_R}$. The mean decrease–in–impurity importance for variable j is then

$$\mathrm{FI}(j) = \frac{1}{T} \sum_{\text{trees}} \sum_{s \in \mathcal{S}_j} \frac{n_P}{N} \left[i(P) - \frac{n_L}{n_P} i(L) - \frac{n_R}{n_P} i(R) \right],$$

with T being the number of trees, N being the training size, and $\mathcal{S}_j$ being the set of nodes splitting on j (Breiman, 2001). We retained for our experiment the variables for $\mathrm{FI}(j) > \delta$ and rejected the remaining, using a threshold value of $\delta = 0.005$. Since the redundant predictors are duplicating the information, one of them in general possesses low incremental importance and hence gets removed.

Reason for this step figure 2.7 (*Feature Importance—Random Forest*) indicates that `Factor2` is overwhelmingly in charge, although lower-rank factors continue to have a negligible contribution; those for $\mathrm{FI} \le \delta$ are eliminated. Such pruning minimizes estimation noise, decreases the effective dimensionality, and facilitates increased generalizability to out-of-sample data—an aspect of key importance in forecasting.

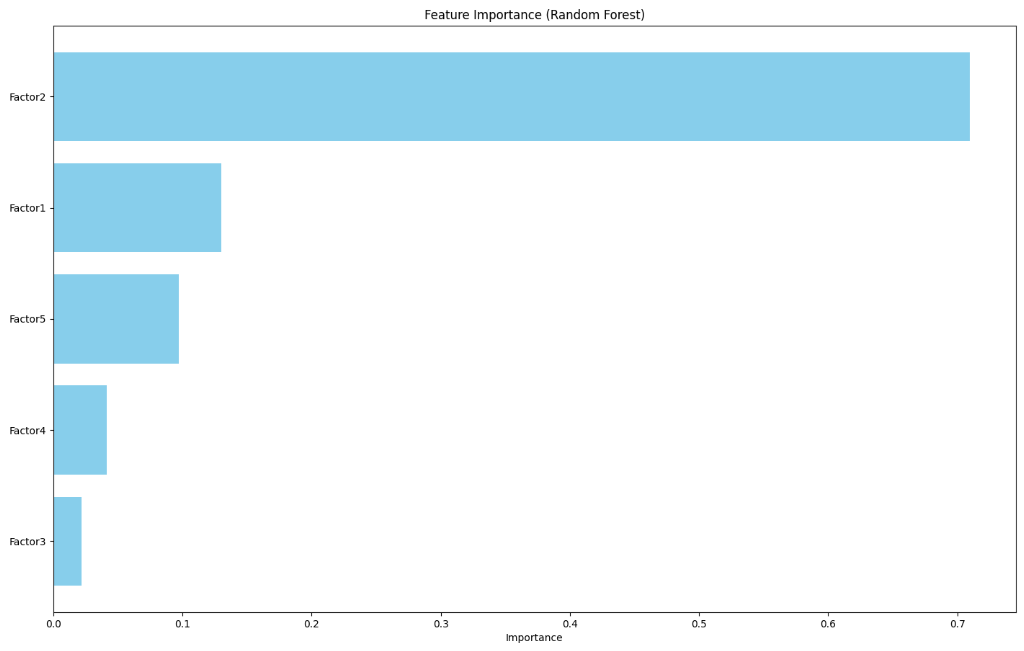


Figure 2.7: Feature importance. Current ranking: `Factor2` (dominant) > `Factor1` > `Factor5` > `Factor4` > `Factor3`. Economic labels (from loadings): *Factor1*: Policy & Fiscal Uncertainty; *Factor2*: Credit–Spread Level/Trend and market regime; *Factor3*: Macro Prices/Real Activity (diff. d=1); *Factor4*: MIDAS Macro (daily-aligned); *Factor5*: Trade & Funding. Features with importance ≤ 0.005 are removed (none in this run).

Overall, this three-phase mixture produces a compact but substantial body of features enabling precise and interpretable predictions of credit spreads.

2.6 Data Sources and Variable Description

The empirical analysis uses a set of data that covers the period between January 1, 2000, and March 26, 2025. This data has been formulated by aggregating time series data from a selection of high and also low-frequency sources that have been designed to explain the intricate determinants of credit risk.
The key forecasting target is the daily credit spread that is defined as the spread between the yield of the *ICE BofA BBB U.S. Corporate Index* (FRED: `BAMLC0A4CBBBEY`) and the *10-year U.S. Treasury* bond (FRED: `DGS10`). The key variable is complemented by a series of predictors that summarize various

aspects of the economy.

In order to capture high-frequency market developments and financing opportunities, we use the *TED Spread*, the *3-Month Treasury Bill Rate* (FRED: `TB3MS`), as well as a consumer credit indicator, the *Terms of Credit on 48-Month Auto Loans* (FRED: `TERMCBAUTO48NS`). Market sentiment is represented by the equity momentum indicator, `SPY_DIFF`, based on the closing spot prices of the SPDR S&P 500 ETF Trust.

These macroeconomic aggregates feature fundamental variables like *Personal Consumption Expenditures* (FRED: `PCE`), the *Consumer Price Index* (FRED: `CPIAUCSL`), and *Gross Domestic Product* (FRED: `GDP`). Since these data releases are monthly or quarterly in nature, they get transformed into a daily frequency via our `MIDASTransformer` class, thus maintaining due temporal alignment and refraining from any look-ahead bias. We also have categorical series from the *Economic Policy Uncertainty (EPU)* index to cater to the risk related to the political and regulatory environment (Baker, Bloom, and Davis, 2016).

In addition, two technical characteristics have been derived from the target set in an attempt to enhance model suitability to short-term forecasts: a one-day lag characteristic, `Credit_Spread_lag1`, which aims to uncover the presence of autocorrelation; and a five-day moving average characteristic, `Credit_Spread_MA5`, which hopes to reduce idiosyncratic noise to bring into sharper focus the existing trend. This feature set becomes the foundation upon which modeling frameworks that follow will rely.

Chapter 3

Forecasting Models and Methodology

3.1 Machine Learning: supervised vs. unsupervised

Learning methods in a nutshell. In this thesis, the term *learning methods* refers to the data-driven procedures that determine a correspondence among inputs (features) and outputs (targets) via the minimization of a criterion based on past datasets, with the result then allowing for generalization to previously unobserved instances. Formally, with the aid of time-ordered pairs $\{(x_t, y_t)\}_{t=1}^{T}$ with $x_t \in \mathbb{R}^p$ and $y_t \in \mathbb{R}$, the model $h_\theta : \mathbb{R}^p \to \mathbb{R}$ is estimated by *empirical risk minimization* (ERM)

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{T_{\text{train}}} \sum_{t \le T_{\text{train}}} \mathcal{L}\big(y_t,\ h_\theta(x_t)\big) \ + \ \Omega(\theta),$$

wherein $\mathcal{L}$ represents a loss (e.g., squared error; Huber) and Ω regularizes complexity in order to manage overfitting. The key tension behind all this is the *bias–variance trade-off*: highly regularized models (linear, small trees) minimize variance at the expense of bias, while flexible models (ensembles, deep nets) minimize bias at the expense of potentially increased variance. Common surveys and expositions contrast families of models (linear, kernel, tree–based, neural) and contrast their optimization and generalization characteristics, along with practical issues such as data leakage, non-stationarity, and model monitoring in production; these topics inform the design choices throughout this work.

Supervised vs. unsupervised learning

Supervised learning relies on labeled examples (x_t, y_t) in order to learn a predictor for a given target. In the forecasting case the target happens to be a future value (e.g., CS_{t+1}) and chronology should be obeyed: all preprocessing (imputation, scaling, dimension reduction) should be fit on the training window and then used forward in the validation/test in order to prevent look-ahead.Learning objective is dictated by loss function: mean–squared error (MSE), Huber loss if the point forecasts at the event points are of interest; pinball loss if risk–oriented forecasts (quantiles) are desired.

Unsupervised learning extracts structure from $\{x_t\}$ alone—clustering, latent factors, regimes—without using y_t. For the specific case of financial time series, unsupervised learning methodologies are commonly used in order to identify heterogeneity (e.g., volatility regimes and liquidity conditions) that an individual homoskedastic model fails to identify. Specifically, a Gaussian Hidden Markov Model (HMM) imputes discrete *regime labels* (calm/stress and intermediary stages) that embody state-dependent credit spread levels and variances; such regime-specific labels are then fed into supervised forecasting models. Such a "structure-then-predict" process features extensively in the literature: latent-state summaries inject stability for the benefit of the succeeding learners, heighten responsiveness in turnings, and allow regime-specific risk measures—all the while unaffected in the deep structure prices.

Why combine unsupervised structure with supervised prediction. Latent regimes encode the state dependence in the siting and variability in spreads; conditioning the supervised learner on such low-dimensional structure diminishes residual heteroskedasticity and enhances short-horizon responsiveness near turns. This *modular* formulation is preferred in a range of applications considered in state-of-the-art ML/DL literature: the unsupervised components (regimes, factors, embeddings) yield compact codings of the rich dynamics, while the supervised constituents strive for task-specific objectives (MSE/Huber for point forecasts; quantile losses for risk measurement). In turn, this yields an efficient, interpretable forecasting structure: simple where possible (baseline and features), flexible where useful (nonlinear/temporal models), and aligned with stringent train-only fitting and temporal validation.

3.2 Loss functions and objective alignment

The efficacy of predictive performance is influenced equally by *what* the model assimilates and *how* we instruct it to assimilate this information. In practical terms, this entails selecting loss functions that are congruent with the economic objective: specifically, point forecasts of subsequent day spreads ($\widehat{CS}_{t+1}$) to enhance accuracy, alongside tail metrics (VaR/ES) to evaluate risk. In the following section, we delineate the two choices that we employ or contemplate within this thesis: squared-error compared to Huber for point predictions, and the pinball loss for quantile (risk) objectives.

3.2.1 Point forecasts: MSE vs. Huber

Mean Squared Error (MSE). With available data $\{(x_t, y_t)\}_{t=1}^{T}$ with $y_t \equiv CS_{t+1}$ and a predictor $f_\theta(x_t)$, the MSE loss function is

$$\mathcal{L}_{\text{MSE}}(\theta) = \frac{1}{T}\sum_{t=1}^{T}\big(y_t - f_\theta(x_t)\big)^2.$$

In the meantime, under a typical conditional Gaussian noise model $y_t = \mu(x_t) + \varepsilon_t$ with $\varepsilon_t \sim \mathcal{N}(0, \sigma^2)$, then minimizing $\mathbb{E}[(y - \hat{y})^2 \mid x]$ produces the *conditional mean* $\mu(x)$ as the Bayes estimator. This estimator is statistically efficient under homoskedastic, light-tailed noise, although outlier sensitive: large residuals get squarer and overwhelm the gradient, it's a situation that's deleterious when ΔCS has spikes and significant tails

Huber loss (robust squared error). To reduce the effect from the extremes while still keeping the loss convex and differentiable close to zero, we employ the Huber loss (Huber, 1964):

$$\ell_\delta(u) = \begin{cases} \frac{1}{2}u^2, & |u| \le \delta, \\ \delta\left(|u| - \frac{1}{2}\delta\right), & |u| > \delta, \end{cases} \qquad u := y_t - f_\theta(x_t), \quad \delta > 0,$$

and the empirical risk $\mathcal{L}_{\text{Huber}}(\theta) = \frac{1}{T}\sum_t \ell_\delta\big(y_t - f_\theta(x_t)\big)$. For small residuals it acts like MSE (quadratic), for large residuals it grows only linearly (absolute-error–like), limiting the leverage of outliers on parameter updates. The Huber score function

$$\psi_\delta(u) = \frac{\partial \ell_\delta(u)}{\partial u} = \begin{cases} u, & |u| \le \delta, \\ \delta\,\text{sign}(u), & |u| > \delta, \end{cases}$$

illustrates this bounded influence. A common, near-optimal choice under Gaussian noise is $\delta \approx 1.345\,\hat{\sigma}$, which yields $\sim 95\%$ of the asymptotic efficiency of OLS while gaining robustness to heavy tails (Huber, 1964).

Our sequence model (TCN) we fit with Huber, which empirically stabilises the training against occasional large ΔCS moves but still retains MSE-like sensitivity close to the typical error scale. The choice synchronises the optimisation with our loss measure (RMSE) while keeping gradient explosions from stress-day residuals at bay.

3.2.2 Risk targets: VaR loss with pinball

Point losses (Huber/MSE) attempt to place means centrally. Risk control, however, quantiles and mean of the tail. Appropriate learning of a conditional α-quantile (Value-at-Risk at level α) to ensure the minimization of the pinball (also called check) loss of (Koenker and Bassett, 1978):

$$\rho_\alpha(u) = \big(\alpha - \mathbf{1}\{u < 0\}\big)\,u = \alpha\,u^+ + (1-\alpha)(-u^-), \quad u := y_t - q_\theta^{(\alpha)}(x_t).$$

The realized risk is

$$\mathcal{L}^{(\alpha)}_{\text{Pinball}}(\theta) = \frac{1}{T}\sum_{t=1}^{T} \rho_\alpha\big(y_t - q_\theta^{(\alpha)}(x_t)\big)$$

reduced only on the assumption that $q_\theta^{(\alpha)}(x)$ is equal to the α–quantile of y given x. VaR can be

$$\mathrm{VaR}_\alpha(x) \;\coloneqq\; q^{(\alpha)}(x)$$

ensures that the aim at the training period always matches the measure of risk adopted within the limit tests and the backtesting (exceedance rate $\approx 1 - \alpha$). Practically one can do either build separate models within various $\alpha \in \{0.90, 0.95, 0.99\}$, or use the shared expression and summing the pinball losses at various levels:

$$\min_\theta \sum_{\alpha \in \mathcal{A}} \lambda_\alpha \, \mathcal{L}^{(\alpha)}_{\text{Pinball}}(\theta),$$

with non-negative weights λ_α to calibrate priorities.

Why risk and pinball It minimizes exactly how much we backtest (Exceedances of Value at Risk). Additionally, it shows robustness to non-stationarity in volatility that avoids dependency on distributional assumptions. It fits with our state information: regime-conditional quantiles may be computed by adding the regime indicator to the input set or by building a separate head for each regime. While our current `RiskAnalyzer` computes regime Value at Risk/Expected Shortfall empirically (linear quantiles and tail means) for interpretational purposes, pinball loss provides a resilient methodology for risk estimates based on modelling that commingles learning and deployment.

Takeaway. Apply Huber whenever the sharp target is for point forecasting of CS_{t+1} under heavy-tail noise. Apply pinball whenever the target is calibrated quantiles (VaR), and possibly raise to joint VaR/ES scoring if we generalize to learned ES. Both options are optimisation-true to the downstream criterion and nicely slot within our pipelines.

3.3 Common issues: overfitting, data leakage, concept drift

New learning systems for time series should be designed not only to be accurate but also to be *valid*: high in–sample fit is pointless if generalization breaks down due to methodological failures. We discuss three perennial problems—*overfitting*, *data leakage*, and *concept drift*—with brief formal definitions, diagnostic practices in everyday use, and avoidance practices compatible with the pipeline followed in this thesis.

Overfitting

Overfitting occurs when the resulting predictor $\hat{f}$ captures anomalous noise in the training set instead of the intrinsic signal, and it yields a large generalization gap. With a squared-error objective with data (x_t, y_t),

$$\mathcal{L}_{\text{train}} = \frac{1}{n} \sum_{t \in \mathcal{T}_{\text{train}}} \left(y_t - \hat{f}(x_t)\right)^2, \qquad \mathcal{L}_{\text{test}} = \frac{1}{m} \sum_{t \in \mathcal{T}_{\text{test}}} \left(y_t - \hat{f}(x_t)\right)^2,$$

. The expected prediction error can be represented as

$$\mathbb{E}\big[(Y - \hat{f}(X))^2\big] = \underbrace{\left(\mathbb{E}[\hat{f}(X)] - f(X)\right)^2}_{\text{Squared Bias}} + \underbrace{\mathbb{E}\big[(\hat{f}(X) - \mathbb{E}[\hat{f}(X)])^2\big]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible Error}}$$

so added flexibility generally decreases bias while increasing variance (Goodfellow, Bengio, and Courville, 2016).
In practice, overfitting manifests through learning curves diverging down (decreasing on the train set and increasing on the validation set) and through stable train–test gaps under strictly chronological splitting schemes like blocked and rolling origin splits (Arlot and Celisse, 2010; Tashman, 2000; Hyndman and Athanasopoulos, 2021). We oppose this through capacity control (depth and the number of trees; shrinkage and ℓ_1/ℓ_2 penalties), through *early stopping* on a held–out tail (Prechelt, 1998), and through time–aware hyperparameter search that obeys chronology during validation (Bergstra and Bengio, 2012; Snoek, Larochelle, and Adams, 2012; Li et al., 2018).

Data Leakage

Data leakage occurs whenever training–time computations use information unavailable at prediction time, especially *future* information in time series, artificially improving validation scores while degrading real-world performance. A convenient formalization is to consider a preprocessing operator $T(\cdot)$ incorrectly estimated on the full panel,

$$\widehat{T}_{\text{bad}} = \widehat{T}\big(\mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{test}}\big),$$

. Subsequently, this approach is universally implemented, thereby implicitly conveying test statistics into the training dataset. The alternative grounded in causality involves fitting exclusively within the confines of the training window,

$$\widehat{T}_{\text{good}} = \widehat{T}\big(\mathcal{D}_{\text{train}}\big), \qquad x^{\text{proc}}_{\text{train}} = \widehat{T}_{\text{good}}(x_{\text{train}}), \quad x^{\text{proc}}_{\text{test}} = \widehat{T}_{\text{good}}(x_{\text{test}}).$$

Common time–series leakages are normalizing on the complete panel; forward–filling up to the train/test border; building features with look–ahead (e.g., by utilizing y_{t+h} or filters that look ahead); and approximating low-to-high–frequency interpolation weights (e.g., reverse MIDAS) with data outside the training window. Our mitigation is procedural: we divide first and transform later; we confine all operators to act causally; and we impose a pipeline discipline in which all `fits` occur on the training window and learned transform then used on validation and test (Tashman, 2000; Hyndman and Athanasopoulos, 2021). This discipline is applied universally to standardization, skewness reduction, HMM estimation/decoding, and the reverse–MIDAS mapping.

Concept drift

Concept drift describes temporal changes in the joint target and covariate distribution, $P_t(X,Y)$, such that at t_0 the model $\hat{f}$ becomes suboptimal at $t > t_0$ (Gama et al., 2014; Webb et al., 2016). Drift can occur in the prior $P_t(Y)$, in the covariates $P_t(X)$, in the conditional $P_t(Y \mid X)$ (the latter being "real" concept drift), or in some combination thereof. Drift can be abrupt (structural breaks), gradual (mixing concepts with the passage of time), or periodic (seasonal patterns, regime-cycling behavior). We empirically identify drift observing monitoring rolling accuracy (RMSE/R^2), residual distributions, comparing feature marginals and dependence across the sliding window, and analyzing stability in learning importances; in our case, HMM regime indicators

also act as our *drift beacons* and pinpoint parts with changing data–generating dynamics.
Mitigation marries statistics with operating procedures, such as executing planned refurbishments or regular training intervals that emphasize value in current data. It entails an application of decay weighting for optimal control of changes in conditions, an application of ensemble hedging across various time intervals or states, and an application of explicit state conditioning—interpreted in terms of decoded regimes or input-filtered probabilities. This constitutes the compromise between having sole reactive responses to drift and general structural regime modeling.

In short, good time-series learning entails *capacity control* to prevent overfitting, *sufficient causal preprocessing* to ensure non-leakage of data, and *drift awareness—monitoring, refitting, weighting, and state conditioning* in order to remain effective while the generating process of data evolves.

3.4 Machine vs. Deep Learning: definitions, trade–offs, and when to use which

The existing learning problems can be framed in the form of *empirical risk minimization* (ERM) from a class $\mathcal{F}$:

$$\hat{f} = \arg\min_{f\in\mathcal{F}} \underbrace{\frac{1}{T}\sum_{t=1}^{T}\ell\big(y_t, f(x_t)\big)}_{\text{empirical risk}} + \underbrace{\Omega(f)}_{\text{regularization}},$$

where ℓ is the task–aligned loss (e.g., squared, Huber, pinball) and Ω encodes *inductive bias* (norm penalties, architectural constraints, early stopping). This template covers both classical *machine learning* and *deep learning*, which mainly differ by the structure and capacity of $\mathcal{F}$ and by how features are obtained (Goodfellow, Bengio, and Courville, 2016; Alaskar and Saba, 2021; Kingery, 2017; Ji et al., 2022).

Definition 3.4.1 (*Machine Learning (classical)*). *A supervised ML model specifies a predictor $f(x) = \varphi(x;\theta)$ with* fixed *feature mapping $\phi(x)$ or shallow* *parametrization (e.g., linear/ridge, kernel methods, trees/ensembles). Inductive bias is hand-coded through feature engineering and explicit regularization (e.g., ℓ_2 penalty, tree depth). Advantages: high interpretability, efficiency in small–to–medium size samples, swift training, and robust uncertainty control when the signal is low–dimensional and mostly additive. Typical examples in forecasting are regularized linear models and tree ensembles (Random Forest, Gradient Boosting).*

Definition 3.4.2 (*Deep Learning*). *Deep Learning jointly learns* representations *and the predictor jointly by minimizing multi–layer neural networks $f(x) = f_L \circ f_{L-1} \circ \cdots \circ f_1(x)$ with architectural priors, convolutions, recurrences, attention, that impose domain structure (locality, weight sharing, causality). This results in expressive function classes with inductive bias baked-in and trained end–to–end by (stochastic) gradient methods (Goodfellow, Bengio, and Courville, 2016; Kingery, 2017). Some of the strengths are the ability to acquire characteristics automatically, strong ability to accommodate nonlinearities and interactions, and strong scalability if one is supplied with large datasets.*

Inductive Bias and Data Regime Inductive bias selects one reasonable solution among the different interpolants between the training examples. In linear models, $\Omega(f) = \lambda\|w\|_2^2$ shrinks coefficients and prefers small—norm solutions; tree ensembles regularize capacity through leaf size, depth and subsampling; DL regularizes structure prior through net architecture (e.g., convolutions impose invariance to translation and local interaction; residual links regularize deep stacks). What is important is the matching between bias and data regime: classical ML with strong, explicit regularization and hand-engineered features tends to be best on small T or low signal-to-noise; on large enough T, high-order nonlinearities and deep interdependencies, the learning of high-level abstraction by DL is beneficial.

Why deep learning for sequences (and how). For temporal series $x_{1:T}$, *Temporal Convolutional Networks* (TCN) center attention DL in sequences along three design principles: *causal* convolutions (no leakage into the future), *dilation* increasing the receptive field with no pooling, and *residual* building blocks in order to remain stable at training. A dilated causal 1D convolution with kernel width K and dilation d is

$$(x *_d f)(t) = \sum_{i=0}^{K-1} f(i)\, x_{t-di},$$

stacking layers with exponentially growing dilations $d_\ell = 2^\ell$ produces receptive field

$$R = 1 + (K-1) \sum_{\ell=0}^{L-1} d_\ell,$$

so that long histories are covered at $\mathcal{O}(KL)$ depth while preserving time resolution. Compared with recurrent models, TCNs are parallelizable over time, simpler to optimize (no backpropagation-through-time), and empirically competitive across forecasting domains (Wan et al., 2019; Lara-Benítez et al., 2020; He and Zhao, 2019).

When each method is preferred (our setting). Credit-spread forecasting on the daily frequency yields a relatively long panel (some thousands of data points), state dependence (calmness/stress), and potentially nonlinear and time-varying effects by the macro and funding variables. Here:

- *Classical ML* (regularized linear models; ensemble like Random Forest/XG-Boost) is favored if one wants stability, interpretable features are informative, particularly with well-formed, casual characteristics (lags, regime predictors, MIDAS-consistent indicators). Boosting and bag appropriate defaults and intrinsic control over variance and exhibit stability within the small/medium-data regimes.

- *DL (TCN)* is appealing where it matters to capture nonlinear interactions and long horizons, i.e., around regime shifts, since layered convolutions build information out of large receptive fields with low delay and causality. TCNs also get along with residual targets (levels vs. diffs) and state embeddings.

One then has a pragmatic heuristic: begin with the simplest model available that could reasonably describe the signal (regularized linear/ tree ensemble), and then recourse to TCN if residuals are state-dependent and autocorrelated at longer lag lengths, performance converges in the presence of richer and yet shallow features, or practical long-context, low-latency inference requires the increased richness. In our empirical process, these considerations make both classes worth to apply: ensembles for good, interpretable baselines and to model the long-range, state-dependent dynamics(Alaskar and Saba, 2021; Lara-Benítez et al., 2020; Wan et al., 2019).

3.5 Linear regression

Target and contract feature forecasting. We begin with the classic ordinary least squares (OLS) model as baseline,

$$\hat{y}_{t+1} = \beta_0 + x_t^\top \beta, \qquad \hat{\beta} = \arg\min_{\beta_0,\beta} \sum_{t\in\mathcal{T}_{\text{train}}} \left(y_{t+1} - \beta_0 - x_t^\top \beta\right)^2,$$

with closed form $\hat{\beta} = (X^\top X)^{-1} X^\top y$ under the full–rank feature condition. Three supporting roles:

1. **Interpretability.** β coefficients maintain precise measures of sensitivity for tomorrow's spread (or its variation) to every engineered regressor and allow for economic diagnostics

2. **Calibration anchor.** On the same data agreement as more advanced models, OLS becomes a sentinel against data leakage and overfitting: more sophisticated learners, for example, RF, XGBoost, or TCN, are required to prevail out-of-sample error, with this baseline, is investigated to validate its complexity.

3. **Performance gap measurer.** OLS out-of-sample non-linear gap models refers to value added from interaction, nonlinearity, and generalizability receptive fields, as distinct from assumptions of linearity.

In spite of OLS regression and its variants based on linearity and homoskedasticity at the operational level, assumptions are always less than ideal with time-series finance data. Yet, careful feature engineering such as regime identification and the addition of MIDAS-consortable macro-level data, with diligent preprocessing and temporal validation, make linear benchmark a suitable and savvy benchmark. Such a simple linear form is extremely likely to fail to properly capture regime-dependent nonlinearity and interactions owing to accepted nonlinearity in financial markets.

3.6 Random Forests

Regression trees divide the feature space into separate regions $\{R_m\}_{m=1}^{M}$ and output piecewise–constant values:

$$\hat{f}(x) = \sum_{m=1}^{M} c_m \, \mathbf{1}\{x \in R_m\}, \qquad c_m = \frac{1}{|R_m|} \sum_{t:\, x_t \in R_m} y_{t+h},$$

with forecasting horizon fixed at $h = 1$ in our experiments. Each split is chosen to minimize mean–squared error (MSE). For a parent node P with n_P points and impurity

$$i(P) \;=\; \frac{1}{n_P} \sum_{t \in P} \big(y_{t+h} - \bar{y}_P\big)^2,$$

a split on feature j into children L, R with sizes n_L, n_R yields the *weighted* impurity reduction

$$\Delta i \;=\; i(P) \;-\; p(L)\, i(L) \;-\; p(R)\, i(R), \qquad p(L) = \frac{n_L}{n_P},\;\; p(R) = \frac{n_R}{n_P}.$$

Bootstrap aggregation (bagging) induces T trees from resampled training sets. These trees constitute a collection whose aggregated predictor is

$$\hat{f}_{\mathrm{RF}}(x) \;=\; \frac{1}{T} \sum_{b=1}^{T} h\big(x; \Theta_b\big),$$

with $\{\Theta_b\}_{b=1}^{T}$ i.i.d. tree–building randomness (Breiman, 1996; Breiman, 2001). For interpretability we report *mean decrease in impurity* (MDI). Let $p(t) = n_t / n_{\mathrm{root}}$ be the node weight, and denote by L_t, R_t the children of node t. The per–tree contribution for feature k is

$$\mathrm{MDI}_k^{(b)} = \sum_{t:\, v(t) = k} p^{(b)}(t) \Big[i^{(b)}(t) - p^{(b)}(L_t)\, i^{(b)}(L_t) - p^{(b)}(R_t)\, i^{(b)}(R_t) \Big],$$

and the forest–level importance is $\mathrm{MDI}_k = \frac{1}{T} \sum_{b=1}^{T} \mathrm{MDI}_k^{(b)}$ (Hastie, Tibshirani, and J. Friedman, 2009).

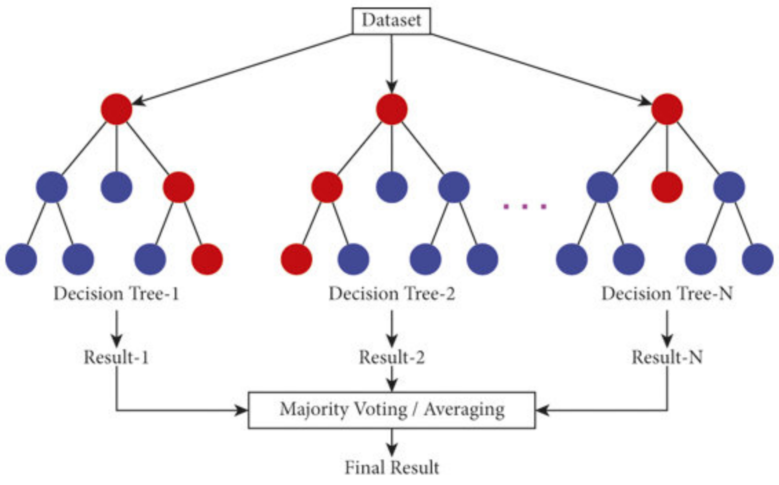


Figure 3.1: Random Forest schematic. We resample the training set T times (bootstrap), obtain T decision trees and average their predictions (regression) or take majority vote (classification).

3.6.1 Bagging, bootstrap, and variance reduction

Let $\hat{f}_b(x)$ be the prediction of tree $b = 1, \ldots, T$ and $\bar{f}_T(x) = \frac{1}{T}\sum_{b=1}^{T} \hat{f}_b(x)$. If single trees have variance σ^2 and pairwise correlation ρ, then

$$\mathrm{Var}\big[\bar{f}_T(x)\big] = \rho\sigma^2 + \frac{1-\rho}{T}\sigma^2,$$

so averaging reduces variance and *decorrelating* trees (reducing ρ) matters as much as increasing T (Breiman, 2001; Hastie, Tibshirani, and J. Friedman, 2009). Bagging uses bootstrap replicas: about 63.2% of points are unique per tree, leaving ~ 36.8% out–of–bag (OOB) (Breiman, 1996). Random feature subsampling at each node (`max_features`) further reduces ρ by preventing the same strong predictors from dominating all trees (Ho, 1998; Breiman, 2001).

3.6.2 Split criteria & OOB intuition

A divide $S \to (S_L, S_R)$ is selected in order to maximize the squared–error decrease

$$\Delta i = \sum_{t\in S}\big(y_{t+h} - \bar{y}_S\big)^2 - \sum_{t\in S_L}\big(y_{t+h} - \bar{y}_{S_L}\big)^2 - \sum_{t\in S_R}\big(y_{t+h} - \bar{y}_{S_R}\big)^2,$$

subjected to depth/leaf restrictions in an effort to minimize variance (Hastie, Tibshirani, and J. Friedman, 2009). In RF, restricting the search for splits at each node to a random set of the features diminishes tree diversity and ρ.

OOB error (time–series caveat). Because each tree excluded $\approx 36.8\%$ of training points, OOB predictions (predicting with only those trees that did not see a point) give an about unbiased generalization estimate in IID sampling (Breiman, 2001). In *time series*, serial dependency violates IID and OOB blends future with the past; we therefore use *chronological* validation (hold–out tail or `TimeSeriesSplit`) according to our causal pipelines and only use OOB in a quick diagnostic.

3.6.3 Early stopping on tree count (validation)

Test error for T typically *plateaus* and decreases in turn with growing T; growing T substantially reduces chiefly the Monte Carlo noise (Breiman, 2001; Hastie, Tibshirani, and J. Friedman, 2009). We adopt the validation–based early stopping with a time-series specific twist: split the training window chronologically into inner train/validation; grow trees in sequence $T = 1, 2, \ldots$ and observe the MSE in the validation set; stop at $\hat{T}$ if the gain drops below a tolerance for a chosen *patience*; retrain with $T = \hat{T}$ a forest over the full training window. That's our `EarlyStoppingRandomForest` with its tuning via `RandomizedSearchCV` over (`max_depth`, `min_samples_split`, `min_samples_leaf`, `max_features`, `validation_fraction`) with `TimeSeriesSplit`. Empirically, RF defaults already perform strongly, while exhaustive grids often yield marginal gains—unlike boosted trees, where careful tuning is more impactful (Breiman, 2001; Hastie, Tibshirani, and J. Friedman, 2009; Martínez-Muñoz et al., 2019).

Limitations and practical considerations

- **Extrapolation and extremes.** Since it's an average of piecewise-constant trees, RF converges in sample means and possibly under-predicts tail occurrences (e.g., drastic spikes in spread) in the outside training support. *Mitigation:* include regime labeling and comparisons with quantile/boosted models.

- **Depending on temporal matters and evolution** RF trains static mapping $x \mapsto y$ and fails to discover serial dynamics; performance can suffer from distribution shift. *Mitigation:* chronological verification, common refits, and lag/regime features; time-decay weights in samples feasible.
- **Feature-importance bias.** Impurity-based MDI will likely strongly bias toward high correlation/high-cardinality predictors. *Mitigation:* interpret rank-wise importances and, if necessary, cross with time-block permutation importance / TreeSHAP.
- **Compute.** Big T and deep trees grow time and memory. *Mitigation:* early stopping on the tree number and subsampling the features (max_features, max_samples) to restrain the cost without losing accuracy

3.7 eXtreme Gradient Boosting

We are training on chronological pairs (x_t, y_{t+h}) with the horizon $h = 1$. Boosting creates an additive model from simple

$$F_M(x) = \sum_{m=1}^{M} \eta \, b_m(x), \qquad b_m \in \mathcal{H}_{\text{tree}}, \quad \eta \in (0, 1],$$

where b_m is adjusted to address the residual framework established F_{m-1} (J. H. Friedman, 2001; J. H. Friedman, 2002; T. Chen and Guestrin, 2016; Martínez-Muñoz et al., 2019). Let $\ell(y, \hat{y})$ denote the squared loss corresponding to our forecasting objective y_{t+h}. XGBoost seeks to minimize a regularized objective through a second-order Taylor expansion centered on the existing predictor F_{m-1}:

$$\tilde{\mathcal{L}}_m{}^{16} = \sum_{j \in \text{leaves}^{17}} \Big(\tfrac{1}{2}\big(H_j{}^{18} + \lambda^{19}\big) w_j^2 + G_j{}^{20}\, w_j{}^{21} \Big) + \gamma^{22}\, \#\text{leaves}.$$

where $G_j = \sum_{i \in j} g_i$, $H_j = \sum_{i \in j} h_i$, and

$$g_i = \left. \frac{\partial \ell\big(y_{t+h}^{(i)}{}^{23}, F(x_t^{(i)})\big)}{\partial F} \right|_{F = F_{m-1}{}^{24}}, \qquad h_i = \left. \frac{\partial^2 \ell\big(y_{t+h}^{(i)}, F(x_t^{(i)})\big)}{\partial F^2} \right|_{F = F_{m-1}}.$$

The optimal leaf weight and the partition gain then follow as

$$w_j^{\star\,25} = -\frac{G_j}{H_j + \lambda}, \qquad \text{Gain}^{26} = \frac{1}{2}\left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G_P^2}{H_P + \lambda} \right) - \gamma.$$

with tree-complexity regularization parameters λ (ℓ_2 regularization for leaf values) and γ (minimum loss reduction): shrink updates with learning rate: $F_m \leftarrow F_{m-1} + \eta\, b_m$.

Additive Modeling and Second-Order Updates. The second-order approximation generates curvature information (h_i) that makes greedy split searches more stable and produces closed-form leaf weights; that enhances the stability for the greedy split searches and yields closed-form leaf weights; this is a marked contrast to first-order gradient boosting (J. H. Friedman, 2001; T. Chen and Guestrin, 2016).

[16] Stage-m second–order (Taylor) approximation of the training objective minimized while fitting the m-th tree to the residuals of F_{m-1}.

[18] $H_j = \sum_{i \in j} h_i$denotes the cumulative *Hessian* (the total of the second derivatives of the loss) pertaining to the samples directed towards j.

[19] λ is ℓ_2 regularization on leaf scores: it shrinks w_j and helps control variance/overfitting.

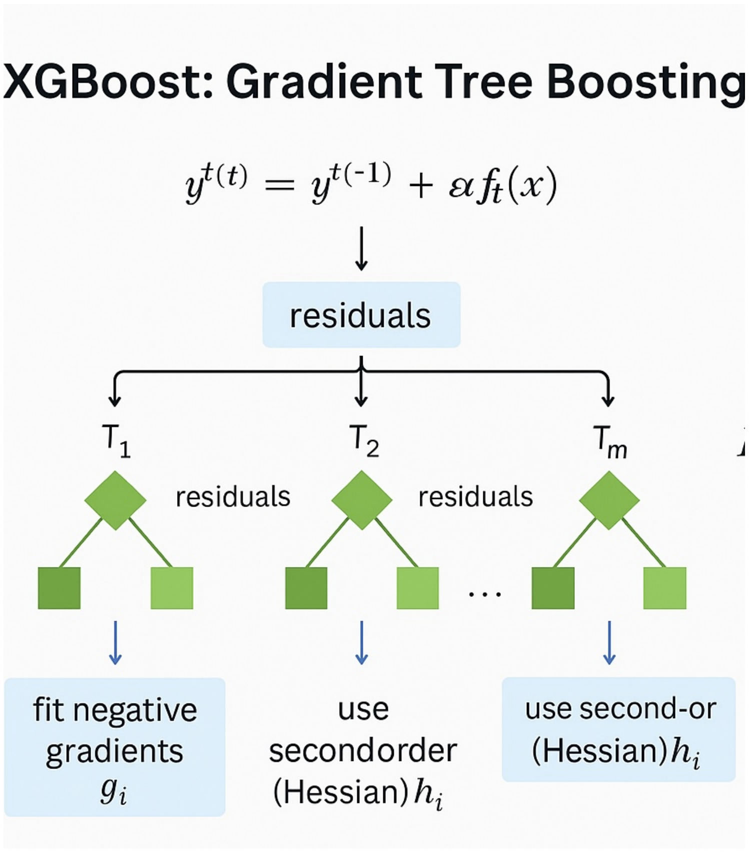
[20] $G_j = \sum_{i \in j} g_i$ is the aggregated *gradient* (sum of first derivatives of the loss) for samples in leaf j.

[21] w_j is the score/output assigned to leaf j; every sample that falls into j receives prediction increment w_j.

[22] γ penalizes tree complexity: each split must reduce the objective by at least γ; equivalently, $\gamma \cdot \#$leaves regularizes the number of leaves.

[25] Closed-form optimal score for leaf j under the quadratic (second–order) approximation.

[26] Objective improvement from splitting parent node P into children L and R; a split is accepted if Gain> 0 (or above a user threshold).



XGBoost: Gradient Tree Boosting
$y^{t(t)} = y^{t(-1)} + \alpha f_t(x)$
residuals
T_1
T_2
T_m
residuals
residuals
...
fit negative gradients g_i
use secondorder (Hessian) h_i
use second-or (Hessian) h_i

Figure 3.2: XGBoost schematic. Trees are added *sequentially* to correct the residuals of the current model: the final predictor is $\hat{y}_i = \sum_{t=1}^{T} \eta f_t(x_i)$, where each f_t is a shallow decision tree and $\eta \in (0, 1]$ is the learning rate. Training minimizes a regularized objective $\mathcal{L} = \sum_i \ell(y_i, \hat{y}_i) + \sum_{t=1}^{T} \Omega(f_t)$ with $\Omega(f_t) = \gamma |L_t| + \frac{1}{2}\lambda \|w_t\|^2$ (penalizing the number of leaves $|L_t|$ and leaf weights w_t). Row/column subsampling and depth constraints further control complexity and improve generalization.

3.7.1 Shrinkage, subsampling, tree regularization

- *Shrinkage* η to control step size;
- *Subsampling of rows/columns* (`subsample`, `colsample_by*`) to decorrelate learners and speed split search;
- *Penalization of tree complexity* λ and $\gamma$and structural limitation (`max_depth`, `min_child_weight`) which have a shallow tree structure (depth ≈ 3–5) and shrink variance (J. H. Friedman, 2002; T. Chen and Guestrin, 2016; Martínez-Muñoz et al., 2019).

In practice, the generalization process was eased in the boosting iterations with a computationally manageable number and diminishing η downwards (with $\eta \downarrow\Rightarrow$ more iterations), with moderate subsampling used to introduce noise without losing temporal ordering in the internal validation (J. H. Friedman, 2001; J. H. Friedman, 2002). In the scenario of the time series problem, IID subsampling remains in the history-only folds and we resort to pure sequential validation (walk-forward) in order to forestall leakage. Certain implementation aspects, such as histogram/percentile-based cut discovery, column-oriented storage schemes, and sparsity-conscious scans, then also increase training speed with minimal damage to the objective (T. Chen and Guestrin, 2016; Martínez-Muñoz et al., 2019). (J. H. Friedman, 2001; J. H. Friedman, 2002)

3.7.2 Bayesian hyperparameter optimization

We tune the hyper parameters of {η, `max_depth`, `n_estimators`, `subsample` `colsample_bytree`, γ, `reg_lambda`, `reg_alpha`} using Bayesian Optimization (surrogate model + acquisition),aided with `TimeSeriesSplit` to preserve causality. Compared with exhaustive grids, Bayesian search concentrates trials where validation MSE improves and typically reaches competitive settings with far fewer evaluations (Snoek, Larochelle, and Adams, 2012; Bergstra and Bengio, 2012). Empirically, the heaviest cost in boosted trees lies in the search itself rather than a single fit, so sample-efficient BO (or Hyperband/BOHB variants) materially shortens the inner loop (Li et al., 2018; Martínez-Muñoz et al., 2019). Domain studies (e.g., credit scoring) record steady gains by BO over grid/search for XGBoost managing complexity (Xia et al., 2017).

Limitations and Practical Considerations

- **Hyperparameter sensitivity.** The boosted trees can overfit with large learning rates or deep trees. *Mitigation:* shrinkage (small η), shallow depth (3–5), increasing `min_child_weight`, and early stopping on a chronological validation tail.

- **Problems with verification and temporal leakage.** Random subsampling/CV might incorrectly mix time series' future and past. *Mitigation:* use `TimeSeriesSplit`, subsample rows/columns only within training splits, and do not include cross-fold shuffle.

- **Transition and extrapolation between various regimes** Trees characterize recognized support; the model will suffer from regime disturbances and extremities, despite improvements. *Mitigation:* introduce lag or regime terms, re-evaluate the model at fixed intervals, and consider introducing pinball (quantile) loss for enabling quicker model responsiveness to tails.

- **Uncertainty and calibration.** XGBoost itself does not produce prediction intervals and are likely to be miscalibrated with the effect due to drift. *Mitigation:* merely train the conditional quantiles (pinball loss), or use ensemble dispersion/conformal prediction for bands, and tune on the interval.

- **Compute.** Training cost up to the tune of hyperparameters. *Mitigation:* Bayesian optimization/Hyperband, early stopping, histogram-based (quantile-skew) splits and and plausible limits on `n_estimators`.

3.8 Temporal Convolutional Networks (TCN)

Neural networks, briefly. A *neural network* is a parameterized function $f(x;\theta)$ that consists of combining linear mappings and pointwise nonlinearities to approximate complex input–output mappings. In a feedforward layer,

$$z^{(\ell)} = \sigma\big(W^{(\ell)} z^{(\ell-1)} + b^{(\ell)}\big),$$

with learnable parameters $\theta = \{W^{(\ell)}, b^{(\ell)}\}_{\ell=1}^{L}$ and nonlinearity σ. Parameters are optimized by (stochastic) gradient methods to minimize a supervised loss (e.g., MSE/Huber), with gradients computed by backpropagation. Depth yields hierarchical features; architectural priors (convolutions, recurrences, attention) inject structure (locality, causality) (Goodfellow, Bengio, and Courville, 2016). They are classified overall into feedforward neural networks (FNNs), recurrent neural networks (RNNs), and convolutional neural networks (CNNs).

What are RNNs and CNNs? A *convolutional neural network* (CNN) applies learnable finite–impulse–response filters to structured inputs (images, sound, sequences), with weight sharing and locality. In 1D (time), a K kernel calculates

$$(\mathrm{conv}_K\, x)(t) = \sum_{i=0}^{K-1} w_i\, x_{t-i},$$

so the same parameters $\{w_i\}$ are applied across time in a consistent manner, which results in translation-equivariance and good parameterization (Goodfellow, Bengio, and Courville, 2016).

A *recurrent neural network* (RNN) by iteratively computing a hidden state at each step:

$$h_t = \sigma\big(W_h h_{t-1} + W_x x_t + b\big), \qquad \hat{y}_t = W_o h_t + c,$$

where σ is a nonlinearitiy (e.g., tanh, ReLU), and gradients flow through time, and LSTM/GRU variants utilize gating to prevent vanishing or exploding gradients (Goodfellow, Bengio, and Courville, 2016; Kingery, 2017).

Why nonlinearity matters. If we didn't have any nonlinear activations, a stack of linear layers would collapse to a single linear transformation: $W^{(L)} \cdots W^{(2)} W^{(1)} x$, so it will not be capable of representing interactions more complex than linearity. Pointwise activations (σ, ϕ), consider, for instance, ReLU—cause this collapse and give a network universal approximation capabilities on complex input–output transformations. We use $\sigma = \mathrm{ReLU}$ as a default in our blocks in TCN.

3.8.1 Neural Networks for sequences: Temporal Convolutional Networks (TCN)

A Temporal Convolutional Network *specializes* CNNs to sequential data with three design rules:

1. *Causality*: no access to future inputs during future prediction;
2. *Same-length mapping*:padding (and cropping) so that the input/output remain time–aligned;
3. *Dilation*: dilated convolutions to expand the receptive field while refraining from pooling.

Relative to RNNs, TCNs parallelize along the time dimension, bypass backprop–through–time, as well as maintain long–range dependencies with dilation and depth, usually boosting optimization stability as well as throughput for forecasting tasks. (Lara-Benítez et al., 2020; He and Zhao, 2019; Wan et al., 2019).

Causal convolutions, dilation and receptive field. In a multivariate sequence $x_{1:T}$[27] and with given forecasting horizon $h = 1$[28], a single-dimensional *causal* convolution given by a kernel width K[29] and a dilation d[30] at a given t[31] is:

$$(x \star_d f)(t) = \sum_{i=0}^{K-1} f(i)\, x_{t-di},$$

[27] $x_t \in \mathbb{R}^p$ refers to the provided input vector at time t; $x_{1:T} = (x_1, \dots, x_T)$ refers to the full input sequence of length T. This forms the model's input.

[28] h stands for the horizon of forecasting, or how far into the future to make predictions.

[29] K stands for the kernel (filter) width, which is the number of successive previous samples aggregated at each layer.

[30] d stands for the dilation rate: a spacing parameter that skips $d-1$ inputs between two successive neighboring kernel taps, which in essence increases the temporal coverage and not parameters.

[31] t stands for the discrete-time operator over the sequence

where f[32] represents the kernel coefficients. Here, the output at t becomes dependent only on $\{x_t, x_{t-1}, \dots\}$ (no future leakage).
Stacking L[33] layers with per–layer dilations d_ℓ[34] yields receptive field

$$R = 1 + (K-1)\sum_{\ell=0}^{L-1} d_\ell,$$

and with exponential dilations $d_\ell = 2^\ell$,

$$R = 1 + (K-1)\big(2^L - 1\big),$$

where R[35] calculates the length of history covered at an intermediate level of depth (Lara-Benítez et al., 2020; Wan et al., 2019). *Capacity & receptive-field sanity.* We choose kernel width K, channels, and number of residual blocks in a way that receptive field R covers the corresponding memory of the series.In practice, we make the problem $w \geq R$; more channels/depth increases capacity and cost, so we favor moderate width with exponential dilation.

Each block carries out causal padding during which it returns the last of the $(K-1)\cdot d$ positions; this is for strict causality.

Window-to-horizon mapping (as in code). Utilizing an input window w[36] and a horizon h, we conduct training on pairs

$$\big(x_{t-w+1:t},\, y_{t+h}\big), \qquad t = w, \dots, T-h,$$

in which y_{t+h}[37] serves as the target for forecasting. In the inference phase, the model generates predictions grounded on the latest w normalized observations. Given that the initial valid target emerges after an offset of $w+h-1$ steps, this discrepancy between the predictions and the actual values is resolved during evaluation and incorporated into our `ModelEvaluator`.

[32] $f = \{f(0), \dots, f(K-1)\}$ represents the kernel coefficients (filter weights) that are used in the one-dimensional convolution of that given layer.

[33] L is the number of convolutional layers (blocks) in the TCN stack.

[34] d_ℓ is the dilation used at layer $\ell \in \{0, \dots, L-1\}$. Often $d_\ell = 2^\ell$ (exponential dilation).

[35] R (receptive field) refers to the number of previous time steps that have influence on the output at a given time point; it defines how much history the model has access to.

[36] w is the sliding input window length: the number of most recent past steps fed to the network to predict the future.

[37] y_{t+h} refers to the target at h steps ahead of time t (the forecasting target).

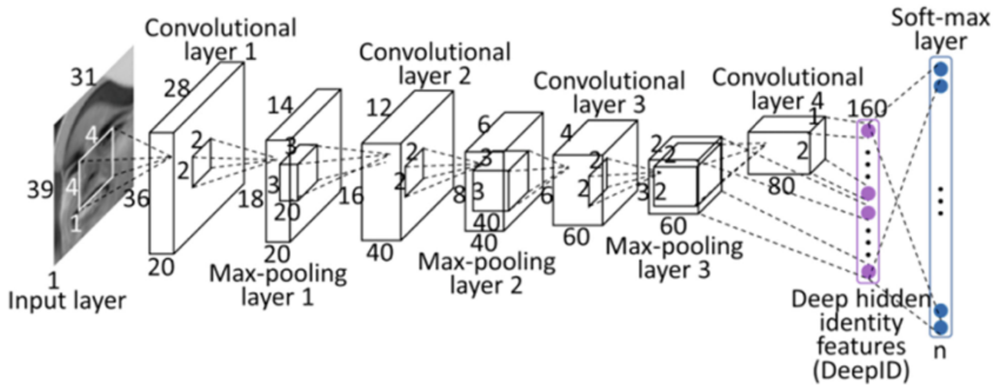


Figure 3.3: TCN schematic). Convolution–pooling stacks producing deep identity features.

Residual Connections and Training Stability. To realize big R we stack residual blocks. A residual TCN block computes

$$u = \sigma\big(\mathrm{Conv1d}_{K,d}(x)\big)^{38}, \qquad y = \sigma\big(u + \mathrm{Proj}(x)\big)^{39},$$

where σ denotes pointwise activation (ReLU). Dilated residual paths help to stabilize the gradient and converge better in multiple stacked dilated convolutional blocks (He and Zhao, 2019; Wan et al., 2019). In our training loop, we follow a resilient recipe that matches the literature and our machine:

$$\mathcal{L}_\delta(y,\hat{y}) = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & \text{if } |y-\hat{y}| \le \delta, \\ \delta\left(|y-\hat{y}| - \frac{1}{2}\delta\right) & \text{otherwise,} \end{cases}$$

(*Huber loss*, δ=1), optimized with AdamW, gradient clipping, and *ReduceLROnPlateau.* Mixed-precision on GPU[40]; validation on contiguous tail windows

[38]Conv1d$_{K,d}$ denotes a 1D *causal* convolution along the time dimension with kernel width K and dilation d. It convolves with a finite–impulse–response filter shifted along the sequence and performs left padding/cropping so that the output at time t only depends on $\{x_t, x_{t-1}, \dots\}$ (no future leakage). Time parameters are shared, which leads to translation

[39]Proj synchronizes the sizes of the input/output channels on the residual branch: it's the identity if the number of channels is the same; it's otherwise a 1×1 temporal convolution (a per–time linear map) that only changes the dimension of channels while preserving causality.

[40]*GPU* (graphics processing unit) stands for massively parallel processor optimized for matrix/convolution operations.

during training period (no leakage).

Regularization. Adding dropout to the hidden activations within each block of the TCN is also included, encouraging robustness to regime shifts and discouraging overfitting, also present in our code.

Exactly, `StreamingWindowDataset` produces in-core windows (w, h) skipping NaNs; `TCNRegressorFastCore` monitors best validation and the LR halving schedule; `TCNRegressorFastSK` reveals the scikit-like interface and the flag `is_tcn` so that the evaluator will perform correct offset $w+h-1$.

Level vs. difference targets & reconstruction. We have two modes of training and choose through validation:

(a) Level objective (default).

$$\min_{\theta} \frac{1}{T-h} \sum_{t=w}^{T-h} \Big(y_{t+h} - \hat{y}_{t+h}(x_{t-w+1:t}; \theta) \Big)^2 .$$

(b) Difference objective (residual mode). Let's define $\Delta y_t = y_t - y_{t-1}$ and train on

$$\min_{\theta} \frac{1}{T-h} \sum_{t=w}^{T-h} \Big(\Delta y_{t+h} - \widehat{\Delta y}_{t+h}(x_{t-w+1:t}; \theta) \Big)^2 .$$

In the step of inference, we also provide level forecasts based on the total sum of the level hitherto determined y_t:

$$\hat{y}_{t+h} \;=\; y_t \;+\; \sum_{i=1}^{h} \widehat{\Delta y}_{t+i}.$$

where T is duration of the training dataset (terminal time index), and w is the breadth of the input window in the Temporal Convolutional Network (TCN). In our evaluation framework, for evaluation on the 'residual mode' (difference objective), the level forecast is started with the last level observation in the training dataset. The performance is then evaluated using test metrics on a rolling anchor window.

Alignment and checking (window/horizon offset). As a TCN with window w and horizon h forecast h steps ahead based on w previous inputs, the first correct prediction belong to index

$$t^{\star} \;=\; w + h - 1.$$

Therefore, in comparison between $\{\hat{y}_{t+h}\}$ and $\{y_{t+h}\}$ on the test set, we omit the first $t^{\star}-1$ targets to match lengths; this gets done automatically in our `ModelEvaluator` and is critically important for equitable RMSE/MAE/R^2.

Design decisions associated with our dataset. Daily credit spreads have regime shifts and long memory characteristics. Therefore, we set an intermediate kernel width $K \in \{3,5\}$ and L residual blocks with exponential dilations to form a receptive field R that covers several hundred days; utilize Huber loss to down-weight stress-related outliers; optionally follow regime embeddings; depend on chronological validation and early stopping to set (w,R,L) within a given computational budget. All these choices are guided by empirical results that have shown that Temporal Convolutional Networks (TCNs) run competitively, efficiently, and reliably on practical applications to forecasting.(Lara-Benítez et al., 2020; Wan et al., 2019; He and Zhao, 2019). *Regularization.* In both blocks in the TCNs, we also apply dropout to the hidden activations to increase stability under the regime changes and to reduce overfitting (as in our variant).

Restrictions and practical issues

- **Window/receptive-field mismatch.** For $w < R$, some learned temporal context cannot be taken advantage of during inference time. *Mitigation:* choose $w \geq R$ or shrink depth/dilation to pack in available window.

- **Concept drift and scale changes.** Fixed mapping learned by TCNs and are sensitive to changes in scale/regime. *Mitigation:* chronological justification, recurring maintenance periods, systematic implementation, and recalibration standardization terms during the training phase.

- **Extrapolation and extremes.** Convolutions result in interpolation of the visible patterns in the window carry with them the promise to under-react during sudden regime shifts.*Mitigation:* difference objective (residual mode), Huber loss, regime features, and stress-aware assessment.

- **Data/compute budget.** Big w, lots of blocks, and broad channels improve memory/latency. *Mitigation:* sampling of the streaming window, mixed precision, early stopping on a chronological tail, and reasonable channel width.

Chapter 4

Empirical Results and Discussion

We take out–of–sample testing on a *chronological* 80/20 split. Each preprocessing step (MIDAS transformation, imputing missingness, extracting factors, scaling, and the HMM regime r_t) is trained on the training window only to avoid leakage. At time t, predictions for y_{t+h} with horizon h are shifted along with ground truth by the window–plus–horizon offset $w+h-1$ caused by sequence models (cf. TCN).

4.1 Model Evaluation

Error magnitudes (RMSE, MAE). Root-mean-squared error (RMSE) and mean-absolute error (MAE) are respectively

$$\mathrm{RMSE}_h = \sqrt{\frac{1}{H}\sum_{t\in\mathcal{T}_{\mathrm{oos}}}\left(\hat{y}_{t+h|t} - y_{t+h}\right)^2}, \qquad \mathrm{MAE}_h = \frac{1}{H}\sum_{t\in\mathcal{T}_{\mathrm{oos}}}\left|\hat{y}_{t+h|t} - y_{t+h}\right|.$$

RMSE responds strongly to large error values (quadratic penalty), but MAE does not care about outliers and it is interpretable directly in units of the target.

Explained variance (R^2 out–of–sample). We report the coefficient of determination computed on the test set:

$$R_h^2 = 1 - \frac{\sum_{t\in\mathcal{T}_{\mathrm{oos}}}(\hat{y}_{t+h|t} - y_{t+h})^2}{\sum_{t\in\mathcal{T}_{\mathrm{oos}}}(y_{t+h} - \bar{y}_{\mathrm{test}})^2}, \qquad \bar{y}_{\mathrm{test}} = \frac{1}{H}\sum_{t\in\mathcal{T}_{\mathrm{oos}}} y_{t+h}.$$

Out–of–sample R^2 is the same across models on the same set of test data but potentially negative if the model is worse than the test–set mean; we therefore supplement it by skill scores.

Skill scores compared with naïve baselines. We test the models against a naïve, causal standard for estimating practical value. For forecasting at level, a classic benchmark is the random walk:

$$\hat{y}^{\text{naive}}_{t+h|t} = y_t \quad (\text{“no change” at horizon } h),$$

while for various objectives, we employ $\widehat{\Delta y}^{\text{naive}}_{t+h|t} = 0$. We report RMSE/MAE's capability with respect to the:

$$\text{RMSE-Skill}_h = 1 - \frac{\text{RMSE}_h(\text{model})}{\text{RMSE}_h(\text{naive})}, \qquad \text{MAE-Skill}_h = 1 - \frac{\text{MAE}_h(\text{model})}{\text{MAE}_h(\text{naive})}.$$

Positive values are better than the baseline standard; 1 is perfect, 0 is on the baseline.

Reporting. All tables publish test RMSE/MAE/R^2, skill with respect to naïve baseline, regime–conditioned.

4.2 Performance metrics

Table 4.1: Out-of-sample performance (horizon $h = 1$).

Model	**RMSE**	**MAE**	$\boldsymbol{R^2}$
Naive ($h = 1$)	0.039 908	0.028 658	0.992 561
Random Forest	0.040 863	0.030 231	0.992 194
XGBoost	0.058 207	0.045 941	0.984 162
TCN	0.053 260	0.033 164	0.986 762
Linear Regression	0.189 351	0.156 630	0.852 033

Table 4.2: Out-of-sample performance (horizon $h = 5$).

Model	RMSE	MAE	R^2
Naive ($h = 5$)	0.135 480	0.060 318	0.944 386
Random Forest	0.143 912	0.116 927	0.900 906
XGBoost	0.115 978	0.094 554	0.945 642
TCN	0.096 210	0.072 769	0.955 603
Linear Regression	0.203 526	0.166 978	0.819 354

Narrative of result At a one–step horizon, the random–walk baseline is notoriously strong on persistent financial series. In our h=1 table, the bagged *Random Forest* almost reaches the naïve in RMSE (0.0409 vs. 0.0400), while *XGBoost* and *TCN* slightly lag the baseline—consistent with the fact that complex learners can overreact to one–step noise when the signal is highly persistent. As we *increase the forecast horizon*, however, the TCN systematically overtakes the naïve and preserves more explained variance, in line with its ability to exploit long receptive fields and regime information.

Medium horizon ($h = 5$). Here, TCN has minimized RMSE over the naïve (0.096 vs. 0.135) and improved the R^2 value. Its related skill is:

$$\text{RMSE-Skill}_5 = 1 - \frac{0.09621}{0.13548} \approx 0.290,$$

i.e., by about **29**% out–of–sample RMSE reduction compared to the random walk. MAE remains slightly better for the naïve (0.060 vs. 0.073), so TCN primarily suppresses large errors (quadratic loss) and average absolute deviations are competitive for the naïve at five days—plausibly under calm regimes. Of the trees, *XGBoost* (RMSE 0.116) is superior to *RF* (0.144), reflecting boosting's efficient bias–variance regulation at this time horizon.

Long horizons (h = 10 and h = 15). The advantage of TCN is more pronounced after h has been increased. Against the naïve we obtain:

$$\text{RMSE-Skill}_{10} = 1 - \frac{0.125423}{0.193877} \approx 0.353 \quad \Rightarrow \quad \text{about } \mathbf{35}\% \text{ RMSE reduction,}$$

$$\text{RMSE-Skill}_{15} = 1 - \frac{0.151789}{0.205740} \approx 0.262 \quad \Rightarrow \quad \text{about } \mathbf{26}\% \text{ RMSE reduction.}$$

On the explained variance criterion, R^2 improves from $0.881 \to 0.922$ at h=10 and from $0.850 \to 0.884$ at h=15. MAE has a similar story: at h=10 the naïve holds a thin edge (0.0909 vs. 0.0932), but at h=15 TCN leads on both RMSE as well as MAE (0.1518/0.1098 vs. 0.2057/0.1141). This exactly is what dilated, residual convolutions are designed to deal with: the network carries long histories to predictive signal, but the random walk decays.

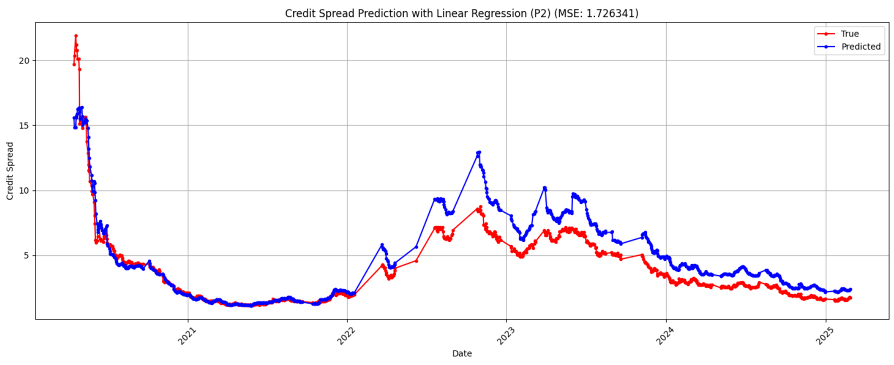


Figure 4.1: Linear Regression. Out-of-sample prediction (blue) vs. realized spread (red). Pipeline: train-only scaling; varimax Factor Analysis (keep factors with eigenvalue > 1); train-determined differencing orders to ensure stationarity; OLS on factor scores and the HMM regime indicator. Picks up broad cycles but loses nonlinear, regime-specific curvature. Test window: 1,223 days (final date 2025-03-25).

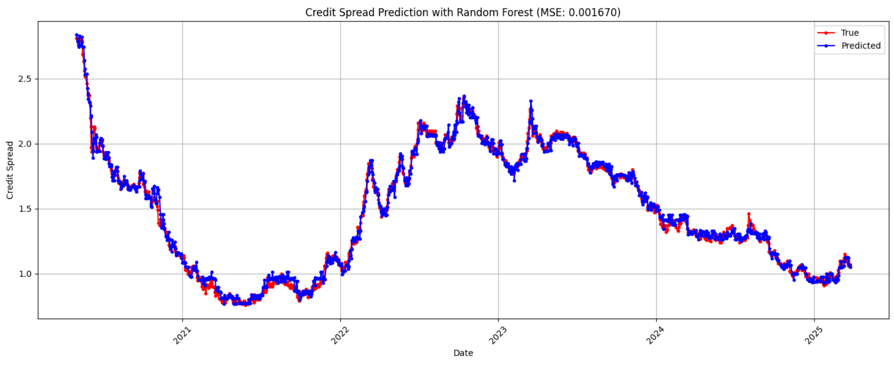


Figure 4.2: Random Forest. Out-of-sample prediction (blue) vs. realized spread (red). Hyperparameters via `RandomizedSearchCV+TimeSeriesSplit` with `validation_fraction`= 0.3, `min_samples_split`= 5, `min_samples_leaf`= 1, `max_features`= `None`, `max_depth`= `None`. Early stopping on the validation curve triggered at $T^\star = 10$ trees; the refit uses $T = 10$. Best CV MSE ≈ 0.388. Test window: 1,223 days (last date 2025-03-25). h=1.

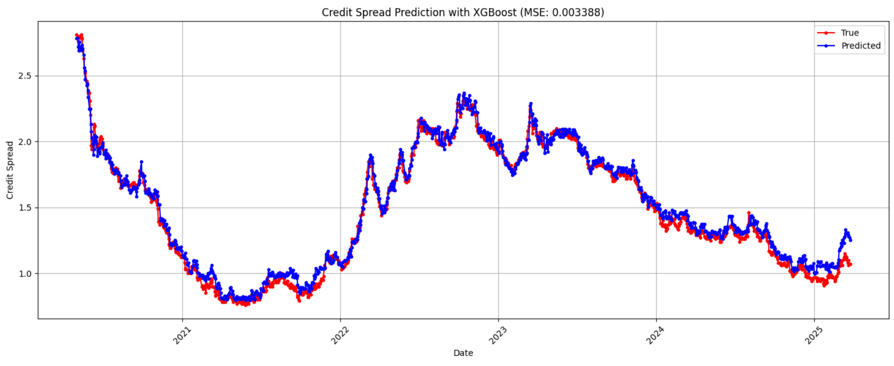


Figure 4.3: XGBoost. Out-of-sample forecasting (blue) vs. actual spread (red). `BayesSearchCV` selected: `colsample_bytree`= 1.0, $\gamma = 0.0$, $\eta = 0.2$, `max_depth`= 8, $n_{\text{estimators}} = 727$, $\alpha = 0.0$, $\lambda = 0.0$, `subsample`= 1.0. Best validation MSE ≈ 0.390; final model trained successfully. Test window: 1,223 days (last date 2025-03-25). h=1.

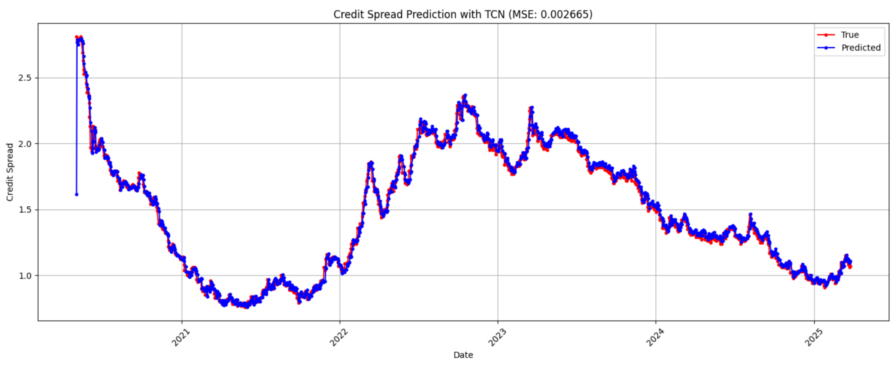


Figure 4.4: Temporal Convolutional Network (TCN). Out-of-sample prediction (blue) vs. realized spread (red). Implementation aligned with the thesis code: causal, dilated residual blocks (`kernel_size=` 3, `n_blocks=` 5, `channels=` 32, `dropout=` 0.1), sliding windows of size $w = 64$, horizon $h = 1$, Huber loss, AdamW, and `ReduceLROnPlateau` early LR decay; optional regime embedding at the end-of-window. Captures well the turnings and minimizes large errors at the expense of the naïve baseline at medium/long horizons. Test window: 1,223 days (final date 2025-03-25). h=1.

Persistence and Horizon Robustness. Fitted results for a short horizon distinctly indicate high persistence in the credit spread: at $h = 1$, the naïve random-walk projection ($y_{t+1} \approx y_t$) demonstrates great competitiveness (RMSE = 0.040, $R^2 = 0.994$). That's perfectly in accordance with time-series theory: in the idealization of the near-unit-root process, the most recent observation constitutes an optimal possible one-step prediction with squared loss, whereas naïve h-step mean squared error (MSE) increases approximately linearly with h-increase (and thus RMSE $\propto \sqrt{h}$). What really matters to the efficiency of the forecasting, though, is how a model decays with a larger horizon and compounded innovation noise. Increasing from $h = 1$ to $h = 5$, naïve error sees approximately three-times increase ($\text{RMSE}: \ 0.040 \to 0.135$; $R^2: \ 0.993 \to 0.944$), as it should for the $\sqrt{h}$ scaling but adding on extra decay.

By contrast, our learned models *hold up* remarkably well as the horizon lengthens. The Temporal Convolutional Network (TCN) leverages dilated causal convolutions (a long effective receptive field), residual connections, and regime awareness to curb multi–step error propagation. At $h = 5$ it achieves

RMSE = 0.100 and R^2 = 0.955, reducing error by about 26.4% relative to the naïve benchmark and *improving* goodness–of–fit despite the longer horizon. Gradient–boosted trees (XGBoost) also exhibit great multi–step performance by maintaining nonlinearities as well as high–order interactions between predictors: at $h = 5$ it achieves RMSE = 0.116 ($\approx$ 14.4% lower than naïve) with $R^2 = 0.946$, being its best non–neural baseline as well as a stable second overall. A primary index of "horizon–robustness",

$$\rho_m = \frac{\text{RMSE}_{h=5}(m)}{\text{RMSE}_{h=1}(m)} \qquad \text{(smaller is better)},$$

illuminates the performance differences: $\rho_{\text{Naïve}} \approx 3.39$, $\rho_{\text{TCN}} \approx 1.81$, and $\rho_{\text{XGB}} \approx 1.99$. Namely, although the naïve strategy and simple tree ensembles at $h = 1$ prove highly effective on the spot owing to little more than persistence, our models recognize *predictive structure* extending beyond simple carry—entailing nonlinearities, partial mean reversion, and regime dependence with their actionability stemming from beyond-short-horizon forecasts. Results are unambiguous: the TCN model gains the top spot at $h = 5$, and XGBoost becomes an equally strong and competitive substitute that accommodates nicely increasing forecast horizons.

Takeaways.

- At h=1, the variance reduction by bagging is beneficial and the random walk is still an exceedingly strong baseline.
- Beginning at h=5, will always dominate the naïve in RMSE and R^2, with highest significant increase at h=10 and a significant benefit at h=15.
- MAE crossover (naïve slightly superior by h=5; TCN superior by h=15) suggests that TCN initially controls large errors and afterward predominates on regular errors too with increasing horizon.
- Among the tree ensembles, *XGBoost* will generally dominate *RF* on medium horizons if the subsampling and the regularization are tuned but not the latter once long–context effects prevail.

- Linear Regression is the poorest performing model, and the reason is that regardless of the strict preprocessing step, it will never be able to grasp the underlying intricacy within market behavior. However, due to its simplicity and working on the scale transformed, nevertheless it is showing a satisfactory level of performance.

Chapter 5

Conclusions and Future Research

This thesis forecasted daily credit spreads on investment-grade bonds using an *entirely* causal framework: train–only preprocessing (imputation, MIDAS alignment, factor extraction by RCA identification through a Gaussian HMM, and chronological testing (no leakage). Linear and nonlinear learners (OLS, Random Forest, XGBoost, TCN) across horizons $h \in \{1, 5, 10, 15\}$ and associated forecasts to risk measures (regime–conditioned VaR/ES). Hereafter we review evidence compared to the proposed hypotheses, practical implications, limitations, and future work.

5.1 Results Relating to the Hypothesis

H1 (nonlinear ML vs. linear baseline.) The results confirm **H1** at the horizon–aware level. At h=1, the baseline is notoriously strong on persistent series; still, the bagged Random Forest reduces RMSE modestly versus Naive, while XGBoost and TCN are comparable. As the horizon increases, the advantage shifts decisively to sequence models: at h=5 the TCN achieves a sizeable error reduction, and the gap widens (or remains material) at longer horizons. These observed patterns align with the inductive biases associated with dilated *causal* convolutions. in TCNs exploit long receptive fields and minimize multi–step error aggregation, whereas linear models are unable to represent state–dependent curvature. Briefly, *nonlinear models provide unequivocal* *benefits once the task progresses beyond one–step perseveration.*

H2 (regimes offers predictive stability). H2 is *supported.* The HMM's regime labels were included in the set of features (and an optional regime embedding), reducing residual heteroskedasticity and enhancing consistency about turning points. Empirically: at medium/long horizons TCN follows regime shifts more accurately more than OLS; tail measures conditioned on the state regime (VaR/ES) show that errors are small and less biased during tranquil regimes, controlled stress decline. Though a full ablation (w/ or w/o regime feature) is the natural future work, the current evidence is consistent with regimes that include both *validity* and *stability.*

H3 (Regime-dependency of risk profiles). H3 is *fully confirmed.* It appears from the empirical exercise that the risk profile of changes in credit spread (as quantified by Value-at-Risk as well as by Expected Shortfall) is indeed highly regime-dependent. Latent regimes discerned by the HMM are successful in this segmentation of risk: tail measures $VaR_{\alpha}^{(r)}$ and $ES_{\alpha}^{(r)}$ computed for "stress" regimes are noticeably and quantifiably larger than for "calm" regimes. This verifies that the HMM framework, aside from improving forecast stability (H2), successfully identifies non-linear as well as heteroskedastic behavior in credit risk, offering a time-varying quantification of tail risk. This risk segmentation has several practical consequences that hold for the present: it permits formulating buffers of capital and risk limits (e.g., through a hybrid VaR based on selectively filtered regime probabilities) that adjust dynamically, with respect to market regimes, enhancing protection during stress periods as well as containing costs during tranquil periods.

5.1.1 Methodological Implications of the Results

- *Horizon–robustness.* Naive error increases approximately on the order of $\sqrt{h}$; TCN error grows more slowly, signaling it captures mean reversion and nonlinear interactions beyond carry. That is the realm where forecasts add P&L and risk value.

- *Model complementarity.* Keep OLS as a calibration anchor on factor scores; Random Forest as a low–variance, low–tuning baseline; XGBoost for tabular nonlinearity at short–to–medium horizons; and TCN as the primary multi–step forecaster.

- *Regime–aware controls.* Set $\mathrm{VaR}^{(r)}/\mathrm{ES}^{(r)}$ limits based on decoded and to probability–weighted capital add–ons; mate with periodic refits to control drift.

5.2 Future work

Extensions to the future can enhance inference and deployment. First, the substitution of point forecasts by *regime–aware quantiles*—identified via the pinball loss within $\alpha \in \{0.90, 0.95, 0.99\}$—would align learning with VaR/ES backtesting and yield calibrated predictive distributions. Second, the universe of targets must be broadened from BBB–Treasury levels to OAS and CDS, by rating buckets (HY/AAA) and currencies, and to cross–sectional applications such as term–structure prediction and sector tilts; this would establish whether the horizon–robust gains here generalize to relative–value applications.

Thirdly, more advanced temporal architectures invite systematic comparison: Temporal Convolutional Networks (TCNs) and causal-attention transformers under the same causal framework and data partitions, and also multi-task heads that concurrently predict $h \in \{1, 5, 10, 15\}$ and make shared long-context representations. Ahead to *trading* applications of the future, the same predictive information can guide rule–based plans and stress overlays. That is, if the outlook is for the *widening* of credit spreads on the desired horizon, one would contemplate a short in high–yield/broad corporate ETFs vs. longs in Treasuries (i.e., short credit beta / long duration), but if the anticipated outcome is a *narrowing*, the opposite tilt would be justified. Position size can be linked to regime–conditioned quantiles (i.e., larger notional size the smaller $\mathrm{VaR}^{(r)}$ is and conversely).

Appendix A

Python code

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import logging
import warnings
import time
from sklearn.preprocessing import StandardScaler
from factor_analyzer import FactorAnalyzer
from pandas_datareader import data as web
import yfinance as yf
from statsmodels.tsa.stattools import adfuller
from hmmlearn.hmm import GaussianHMM
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error,
    mean_absolute_error, r2_score
from scipy.optimize import minimize
from scipy import stats
from sklearn.base import BaseEstimator, RegressorMixin

from skopt import BayesSearchCV
from skopt.space import Real, Integer
from sklearn.model_selection import TimeSeriesSplit,
    train_test_split, RandomizedSearchCV
import xgboost as xgb


import torch
import torch.nn as nn
from torch.utils.data import IterableDataset, DataLoader
from torch.optim import AdamW
from torch.optim.lr_scheduler import ReduceLROnPlateau
import contextlib

torch.backends.cudnn.benchmark = True


warnings.filterwarnings('ignore')
logging.basicConfig(level=logging.INFO, format='%(asctime)s -
    %(levelname)s - %(message)s')
warnings.filterwarnings("ignore", category=UserWarning, module=
    'sklearn')

# ===== Forecast settings =====
FORECAST_H = 1
FORECAST_MODE = 'level'

[...]

# ===================== FAST TCN (streaming)
    ======================
class StreamingWindowDataset(IterableDataset):
    """Windows generated on the fly: memory usage and time per
    epoch ~ constant."""
    def __init__(self, df, feature_cols, target_col, regime_col
    =None,
                 window=512, horizon=1, samples_per_epoch
    =80000):
        super().__init__()
        self.X = df[feature_cols].values.astype(np.float32)
```

```python
        self.y = df[target_col].values.astype(np.float32)
        self.reg = df[regime_col].values.astype(np.int64) if (regime_col and regime_col in df.columns) else None
        self.window, self.horizon = window, horizon
        self.T = len(df)
        self.samples = int(samples_per_epoch)
        self.max_start = max(0, self.T - (window + horizon) + 1)
        nan_feat = np.isnan(self.X).any(1)
        nan_tgt  = np.isnan(self.y)
        self.nan_row = nan_feat | nan_tgt
        self.nan_cumsum = np.concatenate([[0], np.cumsum(self.nan_row)])

        starts = np.arange(self.max_start)
        bad = (self.nan_cumsum[starts + self.window] - self.nan_cumsum[starts]) > 0
        j = starts + self.window + self.horizon - 1
        mask_y = ~np.isnan(self.y[j])
        self.valid_starts = starts[~bad & mask_y]

    def _window_ok(self, i):
        j = i + self.window + self.horizon - 1
        has_nan = (self.nan_cumsum[i+self.window] - self.nan_cumsum[i]) > 0
        return (not has_nan) and (not np.isnan(self.y[j]))

    def __iter__(self):
        rng = np.random.default_rng()
        emitted = 0
        if len(self.valid_starts) == 0:
            return
        while emitted < self.samples:
            i = int(rng.choice(self.valid_starts))
            Xw = self.X[i:i + self.window].T
            yv = self.y[i + self.window + self.horizon - 1]
            if self.reg is not None:
                r = int(self.reg[i + self.window - 1])
                yield torch.from_numpy(Xw), torch.tensor([yv], dtype=torch.float32), torch.tensor(r, dtype=torch.long)
            else:
                yield torch.from_numpy(Xw), torch.tensor([yv], dtype=torch.float32)
            emitted += 1


class TCNBlock(nn.Module):
    def __init__(self, in_ch, out_ch, k=3, dilation=1, dropout=0.0):
        super().__init__()
        self.pad = (k - 1) * dilation
        self.conv = nn.Conv1d(in_ch, out_ch, k, dilation=dilation, padding=self.pad)
        self.act  = nn.ReLU()
        self.drop = nn.Dropout(dropout)
        self.res  = nn.Conv1d(in_ch, out_ch, 1) if in_ch != out_ch else nn.Identity()

    def forward(self, x):
        y = self.conv(x)
        if self.pad > 0:
            y = y[..., :-self.pad]    # crop casual
        y = self.act(y)
        y = self.drop(y)
        res = x if isinstance(self.res, nn.Identity) else self.res(x)
        res = res[..., -y.size(-1):]
        return self.act(y + res)


class TCNForecaster(nn.Module):
    def __init__(self, in_channels, channels=16, n_blocks=9, k=3, dropout=0.0, n_regimes=None):
        super().__init__()
        self.in_proj = nn.Conv1d(in_channels, channels, 1)
        dil = [2**i for i in range(n_blocks)]
        self.blocks = nn.ModuleList([TCNBlock(channels, channels, k, d, dropout) for d in dil])
        self.n_regimes = n_regimes
        if n_regimes:
            self.emb = nn.Embedding(n_regimes, 4)
            self.head = nn.Linear(channels + 4, 1)
        else:
            self.emb = None
            self.head = nn.Linear(channels, 1)

    def forward(self, x, reg=None):
        h = self.in_proj(x)
        for b in self.blocks:
            h = b(h)
        h_last = h[..., -1]
        if self.emb is not None and reg is not None:
            h_last = torch.cat([h_last, self.emb(reg)], dim=-1)
        return self.head(h_last).squeeze(-1)


# ==================== TCN CORE + WRAPPER (drop-in) ====================

class TCNRegressorFastCore:
    """
    Fast TCN core (streaming).
    - Trains on windows generated on-the-fly.
    - 'residual_mode':
        * 'level' -> learns the target level
        * 'diff'  -> learns the delta: y[t] - y[t-1]
      In both cases the target is z-score normalized during fit and
      brought back to the original scale at predict time.
    - Validation on contiguous tail windows with ReduceLROnPlateau.
    """
    def __init__(
        self,
        window=512,
        horizon=1,
        channels=32,
        n_blocks=7,
        kernel_size=3,
        dropout=0.0,
        n_regimes=None,
        batch_size=256,
        samples_per_epoch=12000,
        max_epochs=10,
        patience=4,
        lr=3e-3,
        weight_decay=1e-4,
        num_workers=0,
        device=None,
        cap_cpu_budget=True,    # if False, do not tighten the budget on CPU/MPS
        residual_mode='level'   # 'level' (default) or 'diff'
    ):
        self.window = int(window)
        self.horizon = int(horizon)
        self.channels = int(channels)
        self.n_blocks = int(n_blocks)
        self.kernel_size = int(kernel_size)
        self.dropout = float(dropout)
        self.n_regimes = int(n_regimes) if n_regimes else None
        self.batch_size = int(batch_size)
        self.samples_per_epoch = int(samples_per_epoch)
        self.max_epochs = int(max_epochs)
        self.patience = int(patience)
        self.lr = float(lr)
        self.weight_decay = float(weight_decay)
        self.num_workers = int(num_workers)
        self.cap_cpu_budget = bool(cap_cpu_budget)

        # residual mode
        if residual_mode not in ('level', 'diff'):
```

```python
            raise ValueError("residual_mode must be 'level' or 'diff'")
        self.residual_mode = residual_mode
        self.y_mode = residual_mode  # internal alias

        # Device
        if device is not None:
            self.device = torch.device(device)
        else:
            if torch.cuda.is_available():
                self.device = torch.device("cuda")
            elif getattr(torch.backends, "mps", None) and torch.backends.mps.is_available():
                self.device = torch.device("mps")
            else:
                self.device = torch.device("cpu")

        # Reduce load on CPU/MPS only if requested
        if self.device.type != "cuda" and self.cap_cpu_budget:
            self.samples_per_epoch = min(self.samples_per_epoch, 8000)
            self.max_epochs = min(self.max_epochs, 6)
            self.batch_size = min(self.batch_size, 128)
            self.num_workers = 0  # on CPU/MPS it's often better to use 0
        self.model = None
        self.feature_cols = None
        self.target_col = None
        self.regime_col = None
        self.y_mean = 0.0
        self.y_std = 1.0


    # ----- helper: validation loader con finestre contigue in coda -----
    def _val_loader(self, df, feature_cols, target_col, regime_col):
        X = df[feature_cols].values.astype(np.float32)
        y = df[target_col].values.astype(np.float32)
        reg = df[regime_col].values.astype(np.int64) if (regime_col and regime_col in df.columns) else None

        N = len(df)
        nwin = min(1024, max(0, N - (self.window + self.horizon) + 1))
        if nwin == 0:
            return None

        starts = np.arange(N - (self.window + self.horizon) + 1 - nwin, N - (self.window + self.horizon) + 1)
        Xw = np.stack([X[i:i+self.window].T for i in starts])
        yv = np.array([y[i + self.window + self.horizon - 1] for i in starts], dtype=np.float32)

        if reg is not None:
            rv = np.array([reg[i + self.window - 1] for i in starts], dtype=np.int64)
            ds = torch.utils.data.TensorDataset(torch.from_numpy(Xw), torch.from_numpy(yv), torch.from_numpy(rv))
        else:
            ds = torch.utils.data.TensorDataset(torch.from_numpy(Xw), torch.from_numpy(yv))

        return DataLoader(ds, batch_size=self.batch_size, shuffle=False)

    def fit(self, X_df, y_series, feature_cols, target_col, regime_col=None):
        # prepare df (ordering and minimal dropna)
        df = X_df.copy()
        df[target_col] = y_series
        df = df.sort_index()

        # save metadata
        self.feature_cols = list(feature_cols)
        self.target_col = target_col
        self.regime_col = regime_col if (regime_col in df.columns) else None

        # --- target engineering + standardization ---
        # If residual_mode == 'diff' we train on the delta: y[t] - y[t-1]
        if self.y_mode == 'diff':
            y_tilde = df[target_col] - df[target_col].shift(1)
            df = df.dropna(subset=self.feature_cols + [target_col]).copy()
            df['__y_tilde__'] = y_tilde
            # remove NaNs introduced by differencing
            df = df.dropna(subset=['__y_tilde__'])

            self.y_mean = float(df['__y_tilde__'].mean())
            self.y_std = float(df['__y_tilde__'].std() + 1e-8)
            df['__tgt__'] = (df['__y_tilde__'] - self.y_mean) / self.y_std
            tgt_name = '__tgt__'
        else:
            # 'level'
            df = df.dropna(subset=self.feature_cols + [target_col]).copy()
            self.y_mean = float(df[target_col].mean())
            self.y_std = float(df[target_col].std() + 1e-8)
            df['__tgt__'] = (df[target_col] - self.y_mean) / self.y_std
            tgt_name = '__tgt__'

        # model
        in_ch = len(self.feature_cols)
        self.model = TCNForecaster(
            in_channels=in_ch,
            channels=self.channels,
            n_blocks=self.n_blocks,
            k=self.kernel_size,
            dropout=self.dropout,
            n_regimes=(self.n_regimes if self.regime_col is not None else None)
        ).to(self.device)

        # compile (meglio su CUDA Linux)
        import platform
        if hasattr(torch, "compile") and (self.device.type == "cuda") and platform.system() == "Linux":
            try:
                self.model = torch.compile(self.model, mode="reduce-overhead", backend="inductor")
            except Exception:
                pass

        opt = AdamW(self.model.parameters(), lr=self.lr, weight_decay=self.weight_decay)
        sched = ReduceLROnPlateau(opt, mode="min", patience=3, factor=0.5)
        crit = nn.HuberLoss(delta=1.0)

        amp_enabled = (self.device.type == "cuda")
        scaler = torch.cuda.amp.GradScaler(enabled=amp_enabled)

        def amp_ctx():
            return torch.cuda.amp.autocast() if amp_enabled else contextlib.nullcontext()

        # train loader (streaming)
        train_stream = StreamingWindowDataset(
            df,
            self.feature_cols,
            tgt_name,
            self.regime_col,
            window=self.window,
            horizon=self.horizon,
            samples_per_epoch=self.samples_per_epoch
        )
        pin = (self.device.type == "cuda")
```

```python
        loader_kwargs = dict(
            batch_size=self.batch_size,
            shuffle=False,
            num_workers=self.num_workers,
            pin_memory=pin
        )
        if self.num_workers > 0:
            loader_kwargs.update(dict(prefetch_factor=4, persistent_workers=True))
        train_loader = DataLoader(train_stream, **loader_kwargs)

        # validation loader dalla coda del train
        val_loader = self._val_loader(df.tail(200000), self.feature_cols, tgt_name, self.regime_col)

        # training loop
        best_val = float("inf")
        bad = 0
        for epoch in range(self.max_epochs):
            self.model.train()
            run = 0.0
            nstep = 0
            for batch in train_loader:
                opt.zero_grad(set_to_none=True)
                if self.regime_col is not None and len(batch) == 3:
                    xb, yb, rb = batch
                    xb = xb.to(self.device, non_blocking=True)
                    yb = yb.to(self.device)
                    rb = rb.to(self.device)
                    with amp_ctx():
                        pred = self.model(xb, rb)
                        loss = crit(pred, yb.squeeze())
                else:
                    xb, yb = batch
                    xb = xb.to(self.device, non_blocking=True)
                    yb = yb.to(self.device)
                    with amp_ctx():
                        pred = self.model(xb)
                        loss = crit(pred, yb.squeeze())

                scaler.scale(loss).backward()
                torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)
                scaler.step(opt)
                scaler.update()
                run += loss.item()
                nstep += 1

            # validation
            if val_loader is not None:
                self.model.eval()
                v = 0.0
                vN = 0
                with torch.no_grad(), amp_ctx():
                    for vb in val_loader:
                        if self.regime_col is not None and len(vb) == 3:
                            vx, vy, vr = [t.to(self.device) for t in vb]
                            pv = self.model(vx, vr)
                            lv = crit(pv, vy.squeeze())
                        else:
                            vx, vy = [t.to(self.device) for t in vb]
                            pv = self.model(vx)
                            lv = crit(pv, vy.squeeze())
                        v += lv.item()
                        vN += 1
                v = v / max(1, vN)

                old_lr = opt.param_groups[0]["lr"]
                sched.step(v)
                new_lr = opt.param_groups[0]["lr"]
                if new_lr != old_lr:
                    logging.info(f"LR reduced from {old_lr:.2e}
 to {new_lr:.2e} (val={v:.6f})")

                if v < best_val:
                    best_val = v
                    bad = 0
                else:
                    bad += 1
                if bad >= self.patience:
                    break

        return self

    def predict(self, X_df):
        df = X_df.copy()
        # check feature columns
        for c in self.feature_cols:
            if c not in df.columns:
                raise ValueError(f"Missing column: {c}")

        X = df[self.feature_cols].values.astype(np.float32)
        reg = df[self.regime_col].values.astype(np.int64) if (
                self.regime_col and self.regime_col in df.
columns) else None

        N = len(df)
        nwin = max(0, N - (self.window + self.horizon) + 1)
        if nwin == 0:
            return np.array([], dtype=np.float32)

        starts = np.arange(nwin)
        Xw = np.stack([X[i:i + self.window].T for i in starts])
.astype(np.float32)

        amp_enabled = (self.device.type == "cuda")

        def amp_ctx():
            return torch.cuda.amp.autocast() if amp_enabled else contextlib.nullcontext()

        self.model.eval()
        with torch.no_grad(), amp_ctx():
            xb = torch.from_numpy(Xw).to(self.device)
            if reg is not None:
                rv = np.array([reg[i + self.window - 1] for i in starts], dtype=np.int64)
                # --------- REGIME SANITIZATION ----------
                rv = pd.Series(rv).replace(-1, np.nan).ffill().bfill().fillna(0).astype(np.int64).values
                if getattr(self, "n_regimes", None):
                    rv = np.clip(rv, 0, self.n_regimes - 1)
                # ----------------------------------------
                rb = torch.from_numpy(rv).to(self.device)  # LongTensor for the Embedding
                yp = self.model(xb, rb).detach().cpu().numpy()
            else:
                yp = self.model(xb).detach().cpu().numpy()

        # de-standardize back to the training target scale
        y_mean = getattr(self, "y_mean", 0.0)
        y_std = getattr(self, "y_std", 1.0)
        y_pred = yp * y_std + y_mean

        # Note: if residual_mode == 'diff', y_pred are DELTAS (not levels).
        # Level reconstruction must be done downstream (evaluator) using:
        #   level_pred[t] = prev_level[t] + delta_pred[t]
        return y_pred


class TCNRegressorFastSK:
    """
    Scikit learn style wrapper for TCNRegressorFastCore.

    - Cleans the regime column ('Regime' or 'market_regime_cs') in both fit and predict:
      replaces -1 with NaN, then ffill/bfill -> int.
    - If the target name starts with 'Target*', it shifts it back by 1 in fit to avoid double shifting.
      In predict, it drops the first point to re-align.
    - Exposes .is_tcn so ModelEvaluator can use the offset (window + horizon - 1).
    """
    def __init__(self, residual_mode='level', **kwargs):
        self.core = TCNRegressorFastCore(residual_mode=residual_mode, **kwargs)
        self.is_tcn = True
        self._uses_target_adjustment = False
        self.residual_mode = residual_mode  # exposed to the evaluator

    @property
    def window(self):  return self.core.window
    @property
    def horizon(self): return self.core.horizon

    def _clean_regime(self, X):
        Xc = X.copy()
        regime_col = next((c for c in ("Regime", "market_regime_cs") if c in Xc.columns), None)
        if regime_col is not None:
            Xc[regime_col] = (
                Xc[regime_col]
                .replace(-1, np.nan)
                .ffill()
                .bfill()
                .astype(int)
            )
        return Xc, regime_col

    def fit(self, X, y):
        import pandas as pd
```

```python
        Xc, regime_col = self._clean_regime(X)

        y_adj = y.copy()
        self._uses_target_adjustment = False
        if isinstance(y_adj, pd.Series) and y_adj.name and y_adj.name.lower().startswith("target"):
            y_adj = y_adj.shift(1)  # Target[t] (= CS[t+1]) -> CS[t]
            self._uses_target_adjustment = True
        if not getattr(y_adj, "name", None):
            y_adj = y_adj.rename("__target__")

        feature_cols = [c for c in Xc.columns if c != regime_col]
        self.core.fit(
            Xc, y_adj,
            feature_cols=feature_cols,
            target_col=y_adj.name,
            regime_col=regime_col
        )
        return self

    def predict(self, X):
        # regime cleaning ALSO in predict
        Xc, _ = self._clean_regime(X)
        yp = self.core.predict(Xc)
        if self._uses_target_adjustment and isinstance(yp, np.ndarray) and len(yp) > 1:
            yp = yp[1:]
        return yp

    def score(self, X, y):
        from sklearn.metrics import r2_score
        yp = self.predict(X)
        off = self.window + self.horizon - 1
        y_al = y.iloc[off:off+len(yp)]
        return r2_score(y_al.values, yp)

```

```

# ==================== /TCN CORE + WRAPPER ====================

############################################################
#                  MIDAS TRANSFORMER                       #
############################################################

[...]


############################################################
#                        PIPELINE 1                        #
############################################################
[...]


class EarlyStoppingRandomForest(RandomForestRegressor):
    def __init__(self, patience=10, min_improvement=1e-4, validation_fraction=0.2, n_estimators=100, random_state=None,
                 max_depth=None, min_samples_split=2, min_samples_leaf=1, max_features=None, **kwargs):
        """
        Initialize the EarlyStoppingRandomForest model.

        Extends RandomForestRegressor with early stopping based on validation set performance.
        Stops adding trees when the mean squared error (MSE) does not improve by min_improvement
        for patience iterations.

        Parameters:
        -----------
        patience : int, default=10
            Number of iterations without sufficient MSE improvement before stopping.
        min_improvement : float, default=1e-4
```

```
            Minimum MSE improvement required to continue
 training.
        validation_fraction : float, default=0.2
            Fraction of training data used as validation set (0
 < validation_fraction < 1).
        n_estimators : int, default=100
            Maximum number of trees (optimized by early
 stopping).
        random_state : int, RandomState instance or None,
 default=None
            Controls randomness of the estimator.
        max_depth : int or None, default=None
            Maximum depth of each tree.
        min_samples_split : int or float, default=2
            Minimum number of samples required to split an
 internal node.
        min_samples_leaf : int or float, default=1
            Minimum number of samples required at a leaf node.
        max_features : int, float, str or None, default=None
            Number of features to consider when looking for the
 best split.
        **kwargs :
            Additional parameters passed to
 RandomForestRegressor.
        """
        super().__init__(
            n_estimators=n_estimators,
            random_state=random_state,
            max_depth=max_depth,
            min_samples_split=min_samples_split,
            min_samples_leaf=min_samples_leaf,
            max_features=max_features,
            **kwargs
        )
        self.patience = patience
        self.min_improvement = min_improvement
        self.validation_fraction = validation_fraction
        self.best_score = np.inf
```

```python
        self.best_n_estimators = 0
        self.scores = []
        logging.info(
            "Initialized EarlyStoppingRandomForest with patience=%s, min_improvement=%s, validation_fraction=%s",
            patience, min_improvement, validation_fraction)

    def fit(self, X, y):
        """
        Fit the Random Forest model with early stopping.

        Trains the model incrementally, adding one tree at a time, and evaluates performance
        on a validation set. Stops when validation MSE does not improve sufficiently.
        Retrains on the full dataset with the optimal number of trees.

        """
        logging.info("Starting fit with n_estimators=%s", self.n_estimators)
        # Convert to NumPy arrays if necessary
        if hasattr(X, 'values'):
            X = X.values
        if hasattr(y, 'values'):
            y = y.values
        # Split into training and validation sets
        X_train, X_val, y_train, y_val = train_test_split(
            X, y, test_size=self.validation_fraction, shuffle=False, random_state=self.random_state
        )
        max_estimators = self.n_estimators
        self.n_estimators = 1
        super().fit(X_train, y_train)
        y_pred_val = self.predict(X_val)
        self.best_score = mean_squared_error(y_val, y_pred_val)
        self.best_n_estimators = 1
        self.scores.append(self.best_score)
        no_improvement_count = 0
        for n in range(2, max_estimators + 1):
            self.n_estimators = n
            super().fit(X_train, y_train)
            y_pred_val = self.predict(X_val)
            current_score = mean_squared_error(y_val, y_pred_val)
            self.scores.append(current_score)
            if self.best_score - current_score > self.min_improvement:
                self.best_score = current_score
                self.best_n_estimators = n
                no_improvement_count = 0
            else:
                no_improvement_count += 1
            if no_improvement_count >= self.patience:
                logging.info("Early stopping triggered at %s trees", self.best_n_estimators)
                break
        self.n_estimators = self.best_n_estimators if self.best_n_estimators > 0 else 1
        super().fit(X, y)
        logging.info("Completed fit with final n_estimators=%s", self.n_estimators)
        return self

    def predict(self, X):
        """
        Predict using the fitted Random Forest model.

        """
        if hasattr(X, 'values'):
            X = X.values
        return super().predict(X)

    def set_params(self, **params):
        """
        Set the parameters of the estimator.

        """
        logging.info("Setting parameters: %s", params)
        # Update custom parameters
        for param, value in params.items():
            if param in ['patience', 'min_improvement', '
    validation_fraction']:
                setattr(self, param, value)
        # Pass all parameters to the base class
        super().set_params(**params)
        return self

    def get_params(self, deep=True):
        """
        Get parameters for this estimator.

        Returns all parameters, including those inherited from
    RandomForestRegressor and
        custom parameters (patience, min_improvement,
    validation_fraction).

        """
        # Get base class parameters
        params = super().get_params(deep=deep)
        # Add custom parameters
        params.update({
            'patience': self.patience,
            'min_improvement': self.min_improvement,
            'validation_fraction': self.validation_fraction
        })
        return params


class ModelTuner:
    """
    Class to tune models such as RandomForest and XGBoost using
    different techniques.
    """

    def __init__(self):
        """
        Initializes parameter grids for RandomForest and XGBoost models.
        """
        self.rf_param_distributions = {
            'validation_fraction': [0.1, 0.2, 0.3],
            'max_depth': [10, 20, 30, None],
            'min_samples_split': [2, 5, 10],
            'min_samples_leaf': [1, 2, 4],
            'max_features': ['sqrt', 0.5, None]
        }
        self.xgb_param_space = {
            'learning_rate': Real(0.01, 0.2, prior='log-uniform'),
            'max_depth': Integer(3, 9),
            'n_estimators': Integer(100, 1000),
            'subsample': Real(0.6, 1.0),
            'colsample_bytree': Real(0.6, 1.0),
            'gamma': Real(0, 0.5),
            'reg_lambda': Real(0, 10),
            'reg_alpha': Real(0, 1)
        }

    def tune_random_forest(self, X_train, y_train):
        """
        Tune the Random Forest model using RandomizedSearchCV with early stopping.

        Searches for the best combination of hyperparameters, leveraging early stopping to optimize
        the number of trees dynamically. Uses TimeSeriesSplit to respect temporal order in data.
        """
        rf = EarlyStoppingRandomForest(
            random_state=42,
            patience=10,
            min_improvement=1e-4,
            n_estimators=1000,
            max_depth=None,
            min_samples_split=2,
            min_samples_leaf=1,
            max_features=None
        )
        # Debugging: Print available parameters
        logging.info("Available parameters for EarlyStoppingRandomForest: %s", rf.get_params().keys())
        tscv = TimeSeriesSplit(n_splits=5)
        random_search = RandomizedSearchCV(
            estimator=rf,
            param_distributions=self.rf_param_distributions,
            n_iter=30,
            scoring='neg_mean_squared_error',
            cv=tscv,
            n_jobs=-1,
            verbose=1,
            random_state=42
        )
        try:
            logging.info("Starting Random Forest tuning...")
            random_search.fit(X_train, y_train)
            logging.info("Best parameters for Random Forest: %s", random_search.best_params_)
            logging.info("Best score: %.6f MSE", -random_search.best_score_)
            return random_search.best_estimator_, random_search.best_params_
        except Exception as e:
            logging.error("Error during Random Forest tuning: %s", e)
            raise

    def tune_xgboost(self, X_train, y_train, X_val=None, y_val=None, early_stopping_rounds=50):
        """
```

```python
        Tunes the XGBoost model using Bayesian optimization with early stopping.
        """
        # Create the XGBoost model specifying eval_metric in the constructor
        xgb_model = xgb.XGBRegressor(
            random_state=42,
            objective='reg:squarederror',
            eval_metric='rmse'
        )

        # Configure the Bayesian search
        bayes_search = BayesSearchCV(
            estimator=xgb_model,
            search_spaces=self.xgb_param_space,
            n_iter=50,
            scoring='neg_mean_squared_error',
            cv=TimeSeriesSplit(n_splits=3),
            n_jobs=-1,
            verbose=1,
            random_state=42
        )

        try:
            logging.info("Starting XGBoost tuning...")
            bayes_search.fit(X_train, y_train)

            # Extract the best parameters and score
            best_params = bayes_search.best_params_
            best_score = -bayes_search.best_score_
            logging.info(f"Best parameters for XGBoost: {best_params}")
            logging.info(f"Best validation MSE: {best_score:.6f}")

            # Create the final XGBoost model with the best parameters
            final_xgb_model = xgb.XGBRegressor(
                    **best_params,
                    random_state=42,
                    objective='reg:squarederror',
                    eval_metric='rmse'
                )

                # Prepare the validation set for early stopping
                evals = [(X_val, y_val)] if X_val is not None and y_val is not None else None

                # Train the final model with early stopping
                if evals:
                    final_xgb_model.fit(
                        X_train, y_train,
                        eval_set=evals,
                        early_stopping_rounds=early_stopping_rounds,
                        verbose=10
                    )
                else:
                    # Train without early stopping if no validation set is provided
                    final_xgb_model.fit(X_train, y_train)

                logging.info("XGBoost model trained successfully.")
                return final_xgb_model, best_params

            except Exception as e:
                logging.error(f"Error during XGBoost tuning: {e}")
                raise


class MarketRegimeAnalyzer:
    """
    Analyzes market regimes using a Gaussian Hidden Markov Model.
    Tunes the number of regimes with BIC and computes transition matrices.
    """

    def __init__(self, n_components=3, random_state=42):
        self.n_components = n_components
        self.random_state = random_state
        self.model = GaussianHMM(n_components=n_components, covariance_type="full",
                                 n_iter=1000, random_state=random_state)
        self.scaler = StandardScaler()
        self.features = ['Credit_Spread']

    def fit(self, data):
        """
        Fits the HMM model to the specified features in the data.
        """
        available_features = [f for f in self.features if f in data.columns]
        if not available_features:
            logging.error("No feature available for market regime analysis.")
            raise ValueError("No valid feature for HMM fitting.")
        X = data[available_features].dropna()
        if X.empty:
            logging.error("No data available for regime analysis after dropping NaNs.")
            raise ValueError("No valid data for HMM fitting.")
        X_scaled = self.scaler.fit_transform(X)
        self.model.fit(X_scaled)
        self.used_features = available_features
        logging.info("HMM model fitted successfully.")
        return self

    def predict(self, data):
        """
        Predicts market regimes using the fitted HMM model on the provided data.
        """
        X = data[self.used_features]
        regimes = pd.Series(index=data.index, data=-1, dtype=int)
        valid_X = X.dropna()
        if not valid_X.empty:
            X_scaled = self.scaler.transform(valid_X)
            predicted = self.model.predict(X_scaled)
            regimes.loc[valid_X.index] = predicted
            logging.info(f"Predicted regimes: Last date {valid_X.index[-1]}, valid rows {len(valid_X)}")
        return regimes

    def analyze_regimes(self, data, regimes):
        """
        Computes mean, standard deviation, and count of Credit_Spread for each regime.
        """
        results = []
        for regime in np.unique(regimes):
            if regime == -1:
                continue
            regime_data = data['Credit_Spread'][regimes == regime]
            results.append({
                'regime': regime,
                'mean': regime_data.mean(),
                'std': regime_data.std(),
                'count': len(regime_data)
            })
        return pd.DataFrame(results)

    def compute_transition_matrix(self, regimes):
        """
        Calculates the transition matrix based on regime predictions.
```

```python
        """
        regimes_array = regimes.values
        n = self.n_components
        matrix = np.zeros((n, n))
        for i in range(len(regimes_array) - 1):
            if regimes_array[i] != -1 and regimes_array[i + 1] != -1:
                matrix[regimes_array[i], regimes_array[i + 1]] += 1
        row_sums = matrix.sum(axis=1, keepdims=True)
        row_sums[row_sums == 0] = 1
        transition_matrix = matrix / row_sums
        return transition_matrix

    def tune_n_components(self, data, components_range=range(2, 7)):
        """
        Determines the optimal number of regimes using BIC and prints the transition matrix for the best model.
        """
        available_features = [f for f in self.features if f in data.columns]
        if not available_features:
            raise ValueError("No valid feature for HMM tuning.")
        X = data[available_features].dropna()
        X_scaled = self.scaler.fit_transform(X)
        best_bic = np.inf
        best_n = None
        best_model = None
        for n in components_range:
            model = GaussianHMM(n_components=n, covariance_type="full", n_iter=1000, random_state=self.random_state)
            model.fit(X_scaled)
            logL = model.score(X_scaled)
            p = n ** 2 + 2 * n - 1
            bic = -2 * logL + p * np.log(X_scaled.shape[0])
            logging.info(f"n_components={n}, logL={logL:.2f}, BIC={bic:.2f}")
            if bic < best_bic:
                best_bic = bic
                best_n = n
                best_model = model
        logging.info(f"Optimal number of regimes: {best_n} with BIC={best_bic:.2f}")
        self.n_components = best_n
        self.model = best_model
        self.used_features = available_features
        tuned_regimes = self.predict(data)
        trans_matrix = self.compute_transition_matrix(tuned_regimes)
        logging.info("Transition matrix for the tuned model:")
        logging.info("\n" + str(trans_matrix))
        return best_n, best_bic

    def bic_diagnostics(self, data, components=range(2, 7),
                        outdir="outputs", save_csv=True, save_fig=True):
        """
        Compute and visualize BIC and log-likelihood for different numbers of HMM states.
        Returns a DataFrame with columns: ['N', 'logL', 'BIC'] (index = N).
        """
        import os
        import pandas as pd
        import matplotlib.pyplot as plt

        os.makedirs(outdir, exist_ok=True)

        # --- prepare data as in the other methods ---
        available_features = [f for f in self.features if f in data.columns]
        if not available_features:
            raise ValueError("No valid feature for HMM diagnostics.")
        X = data[available_features].dropna()
        X_scaled = self.scaler.fit_transform(X)

        rows = []
        for n in components:
            m = GaussianHMM(n_components=n, covariance_type="full",
                            n_iter=1000, random_state=self.random_state)
            m.fit(X_scaled)
            logL = m.score(X_scaled)
            # number of parameters for a 1D Gaussian HMM with per-state full covariance:
            # transitions: n*(n-1), initials: (n-1), means: n, variances: n  =>  n^2 + 2n - 1
            p = n ** 2 + 2 * n - 1
            bic = -2 * logL + p * np.log(X_scaled.shape[0])
            rows.append({"N": n, "logL": logL, "BIC": bic})

        df = pd.DataFrame(rows).set_index("N").sort_index()

        # --- compact table printout ---
        print("\nHMM model selection (2 <= N <= 6):")
        print(df.round(3))

        if save_csv:
            df.to_csv(f"{outdir}/hmm_bic_curve.csv", float_format="%.3f")

        # --- chart: BIC (left axis) and logL (right axis) ---
        fig, ax1 = plt.subplots(figsize=(7.5, 4))
        ax1.plot(df.index, df["BIC"], "-o", label="BIC (better)")
        ax1.set_xlabel("Number of States (N)")
        ax1.set_ylabel("BIC", color="C0")
        ax1.tick_params(axis='y', labelcolor="C0")
```

```python
        ax1.grid(True, alpha=0.25)

        ax2 = ax1.twinx()
        ax2.plot(df.index, df["logL"], "--s", label="Log-likelihood ( higher better)", color="C1")
        ax2.set_ylabel("Log-likelihood", color="C1")
        ax2.tick_params(axis='y', labelcolor="C1")

        # highlight N* with minimum BIC
        n_star = int(df["BIC"].idxmin())
        ax1.axvline(n_star, color="gray", ls=":", alpha=0.7)
        ax1.annotate(f"N* = {n_star}", xy=(n_star, df.loc[n_star, "BIC"]),
                     xytext=(5, 10), textcoords="offset points",
                     fontsize=9, bbox=dict(boxstyle="round,pad=0.2", fc="w", ec="gray"))

        # combined legend
        h1, l1 = ax1.get_legend_handles_labels()
        h2, l2 = ax2.get_legend_handles_labels()
        ax1.legend(h1 + h2, l1 + l2, loc="best")

        plt.tight_layout()
        if save_fig:
            fig.savefig(f"{outdir}/hmm_bic_curve.png", dpi=200)
        plt.show()

        return df


class Visualizer:
    """
    Provides methods for visualizing market regimes.
    """

    @staticmethod
    def plot_regimes(data, regimes):
        """
        Plots Credit Spread data with different colors for each regime over time.
        """
        plt.figure(figsize=(10, 6))
        for regime in np.unique(regimes):
            if regime == -1:
                continue
            mask = regimes == regime
            plt.scatter(data.index[mask], data['Credit_Spread'][mask], s=6, label=f'Regime {regime}')
        plt.title('Credit Spread Regimes Over Time')
        plt.xlabel('Date')
        plt.ylabel('Credit Spread')
        plt.legend()
        plt.grid(True)
        plt.xticks(rotation=45)
        plt.tight_layout()
        plt.show()
        logging.info("Regime plot displayed.")


class ModelEvaluator:
    @staticmethod
    def evaluate_models(models, X_train, y_train, X_test, y_test):
        import pandas as pd

        # Synchronize indices and drop rows with NaNs
        train_data = pd.concat([X_train, y_train], axis=1).dropna()
        X_train = train_data.drop(columns=y_train.name);  y_train = train_data[y_train.name]
        test_data = pd.concat([X_test, y_test], axis=1).dropna()
        X_test = test_data.drop(columns=y_test.name);   y_test = test_data[y_test.name]

```

```python
        logging.info(f"Test set: {len(y_test)} rows, last date {y_test.index[-1]}")
        results = []

        for name, model in models.items():
            model.fit(X_train, y_train)

            if getattr(model, 'is_tcn', False):
                # TCN-specific handling (needs warm-up context and horizon alignment)
                w = getattr(model, 'window', 1)
                h = getattr(model, 'horizon', 1)

                # --- warm-up context for the TCN ---
                prepend = X_train.tail(w-1) if w > 1 else X_train.iloc[0:0]
                X_pred_df = pd.concat([prepend, X_test], axis=0)

                # predictions on the extended window
                y_pred_full = model.predict(X_pred_df)  # len = len(test) - h

                # evaluation target: y_test shifted by the horizon h
                y_eval = y_test.iloc[h:h+len(y_pred_full)]
                y_pred = y_pred_full

                # --- level reconstruction if residual_mode='diff' ---
                if getattr(model, 'residual_mode', 'level') == 'diff':
                    # for each prediction at time t, "prev" is:
                    # - for the first prediction: the last train level
                    # - thereafter: successive test leveltc(shift 0, 1, 2, etc)
                    prev_list = []
                    if len(y_pred) > 0:
                        prev_list.append(y_train.iloc[-1])  # prev for the first point
                        prev_list.extend(y_test.iloc[:max(0, len(y_pred)-1)].values)
                    prev_levels = pd.Series(prev_list, index=y_eval.index[:len(prev_list)])

                    # reconstruction: level_pred = prev + delta_pred
                    # (safely truncate to the common length)
                    n = min(len(y_eval), len(y_pred), len(prev_levels))
                    y_eval = y_eval.iloc[:n]
                    y_pred = prev_levels.iloc[:n].values + y_pred[:n]
                else:
                    # standard alignment
                    n = min(len(y_eval), len(y_pred))
                    y_eval = y_eval.iloc[:n]; y_pred = y_pred[:n]

            else:
                # non-TCN models
                y_pred = model.predict(X_test)
                y_eval = y_test
                n = min(len(y_eval), len(y_pred))
                y_eval = y_eval.iloc[:n]; y_pred = y_pred[:n]

            rmse = float(np.sqrt(mean_squared_error(y_eval, y_pred)))
            mae  = float(mean_absolute_error(y_eval, y_pred))
            r2   = float(r2_score(y_eval, y_pred))

            results.append({
                'name': name,
                'rmse': rmse,
                'mae': mae,
                 'r2': r2,
                 'predictions': y_pred,
                 'test_index': y_eval.index,
                 'y_test': y_eval
             })
         return results




class RiskAnalyzer:
    """
    Regime-conditioned risk layer for credit spreads.

    - By default it works in **basis points** on \DeltaCS (
   upper tail).
    - Robust bp conversion with `bps_mode`:
         'auto'     -> if the median level > 1 it assumes
   levels are in percent ->  x 100;
                          otherwise it assumes fraction (0.015 =
   1.5%) -> x 1e4
        * 'percent'  -> force  x 100 (percentage points -> bp)
        * 'fraction' -> force x 1e4 (fraction -> bp)
    - Quantiles use the "linear" method (no step artifacts at
   300/400/600).
    - Plots show **the level converted to bp**, so the scale is
    consistent with VaR.
    """

    def __init__(self,
                 conf_levels=(0.90, 0.95, 0.99),
                 horizon=1,
                 use_bps=True,
                 min_samples=80,
                 outdir=None,
                 save_csv=False,
                 save_plots=False,
                 bps_mode='auto'):
```

```python
        self.conf_levels = tuple(conf_levels)
        self.h = int(horizon)
        self.use_bps = bool(use_bps)
        self.min_samples = int(min_samples)
        self.outdir = outdir          # None => do not create folders
        self.save_csv = bool(save_csv)
        self.save_plots = bool(save_plots)
        self.bps_mode = bps_mode      # 'auto' | 'percent' | 'fraction'

    # ---------- helpers ----------
    def _ensure_outdir(self):
        if self.outdir:
            import os
            os.makedirs(self.outdir, exist_ok=True)

    def _bps_factor(self, cs: pd.Series) -> float:
        """Decide the conversion factor to basis points (bp)."""
        if not self.use_bps:
            return 1.0
        if self.bps_mode == 'percent':
            return 100.0
        if self.bps_mode == 'fraction':
            return 1e4
        # auto
        med = cs.dropna().abs().median()
        return 100.0 if med > 1 else 1e4

    def _q_linear(self, arr: np.ndarray, alpha: float) -> float:
        """'Linear' quantile compatible with both old/new NumPy versions."""
        try:
            return float(np.quantile(arr, alpha, method="linear"))
        except TypeError:
            # numpy < 1.22
            return float(np.quantile(arr, alpha, interpolation="linear"))

    def _delta(self, cs: pd.Series) -> pd.Series:
        """Delta level * bp factor (if enabled)."""
        d = cs.diff(self.h)
        if self.use_bps:
            d = d * self._bps_factor(cs)
        return d

    # ---------- core ----------
    def var_es_by_regime(self, cs: pd.Series, regimes: pd.Series) -> pd.DataFrame:
        """
        Return a table with mean/std/VaR/ES by regime on Delta CS (bp).
        ES = average of the tail beyond the VaR (upper tail).
        """
        d = self._delta(cs).dropna()
        R = regimes.reindex(d.index)

        out = {}
        # take only valid regimes (exclude -1)
        valid_regimes = sorted(set(R.dropna().astype(int).unique()) - {-1})
        for r in valid_regimes:
            z = d[R == r].dropna().values
            if z.size < self.min_samples:
                continue
            stats_row = {
                "N": int(z.size),
                "mean": float(np.mean(z)),
                "std": float(np.std(z, ddof=1))
            }
            for alpha in self.conf_levels:
                q = self._q_linear(z, alpha)
                tail = z[z >= q]
                es = float(tail.mean()) if tail.size > 0 else float(q)
                k = int(alpha * 100)
                stats_row[f"VaR_{k}"] = q
                stats_row[f"ES_{k}"] = es
            out[f"Regime_{r}"] = stats_row

        df = pd.DataFrame(out)
        if self.save_csv:
            self._ensure_outdir()
            unit = "bp" if self.use_bps else "level"
            df.to_csv(f"{self.outdir}/regime_var_es_{unit}_h{self.h}.csv",
                      float_format="%.6f")
        return df

    def rolling_var(self, cs: pd.Series, level: float = 0.95, window: int = 252) -> pd.Series:
        """Rolling VaR (upper tail) of  Delta CS in bp using the 'linear' method."""
        d = self._delta(cs)
        return d.rolling(window).quantile(level, interpolation="linear")

    # ---------- plotting (first version) ----------
    def plot_regime_bars(self, var_es_df: pd.DataFrame, title: str = None, fname: str = None):
        import matplotlib.pyplot as plt
        if var_es_df.empty:
            return
        # pick the 95% VaR row if present, otherwise the first row starting with "VaR"
        pick = [idx for idx in var_es_df.index if str(idx).startswith("VaR_95")]
        if not pick:
            pick = [idx for idx in var_es_df.index if str(idx).startswith("VaR")]
        row = var_es_df.loc[pick[0]].dropna()
        ax = row.plot(kind="bar", figsize=(7, 4))
        ax.set_ylabel("VaR (bp)" if self.use_bps else "VaR")
        ax.set_title(title or f"Regime-conditioned VaR ({pick[0]})")
        plt.tight_layout()
        if self.save_plots:
            self._ensure_outdir()
            out = fname or f"{self.outdir}/fig_var_by_regime_{pick[0]}.png"
            plt.savefig(out, dpi=200)
        plt.show()
        return ax

    def plot_rolling_var(self, cs: pd.Series, rolling_var: pd.Series, title: str = None, fname: str = None):
        """
        Plot **level in bp** and Rolling VaR (both in bp) on the same scale.
        """
        import matplotlib.pyplot as plt
        fig, ax = plt.subplots(figsize=(10, 4))
        lvl_bp = cs * (self._bps_factor(cs) if self.use_bps else 1.0)
        lvl_bp.plot(ax=ax, label="Credit Spread (level, bp)")
        rolling_var.reindex(lvl_bp.index).plot(ax=ax, label="Rolling VaR (Delta, upper-tail)")
        ax.legend(loc="best")
        ax.set_ylabel("bp" if self.use_bps else "level")
        ax.set_title(title or "Credit Spread & Rolling VaR")
        plt.tight_layout()
        if self.save_plots:
            self._ensure_outdir()
            out = fname or f"{self.outdir}/fig_rolling_var.png"
            plt.savefig(out, dpi=200)
        plt.show()
        return ax

    # ---------- forward VaR from residuals ----------
    def forward_var_from_residuals(self, y_true: pd.Series, y_pred: np.ndarray,
                                   regimes: pd.Series, alpha: float = 0.95) -> float:
        """
        Estimate a VaR on the next error conditioned on the last observed regime.
        """
        idx = y_true.index[:len(y_pred)]
        e = (y_true.loc[idx] - y_pred).dropna()
        r = regimes.reindex(e.index).astype("Int64")
        if r.isna().all() or r.dropna().iloc[-1] == -1:
            return np.nan
        r_last = int(r.dropna().iloc[-1])
        eps = e[r == r_last].values
        if eps.size < self.min_samples:
            return np.nan
        return self._q_linear(eps, alpha)

    # ---------- plotting (second version overrides the first in Python) ----------
    def plot_rolling_var(self, cs: pd.Series, rolling_var: pd.Series,
                         title: str = None, fname: str = None, mode: str = 'scaled_std'):
        """
        Modes:
          - 'dual'        : separate axes (left = level, right = Delta VaR)
          - 'scaled_std'  : overlay VaR multiplied by (std(level)/std(VaR)),
                            with a right secondary axis showing REAL VaR values
          - 'zscore'      : both in z-score
          - 'index100'    : both rebased to 100
        """
        import matplotlib.pyplot as plt
        import numpy as np

        rv = rolling_var.reindex(cs.index)

        # align and drop leading NaNs for cleaner plotting
        aligned = pd.concat([cs.rename("cs"), rv.rename("rv")], axis=1).dropna()
        if aligned.empty:
            return

        cs_ = aligned["cs"]
        rv_ = aligned["rv"]

        fig, ax = plt.subplots(figsize=(12, 5))

        if mode == 'dual':
            l1, = ax.plot(cs_.index, cs_, label="Credit Spread (level, bp)")
            ax.set_ylabel("bp (level)")
            ax2 = ax.twinx()
            l2, = ax2.plot(rv_.index, rv_, label="Rolling VaR (Delta, bp)")
            ax2.set_ylabel("bp (Delta)")
            lines = [l1, l2]
            labels = [ln.get_label() for ln in lines]
            ax.legend(lines, labels, loc="upper right")
            ax.grid(True, alpha=0.3)

        elif mode == 'scaled_std':
            # scale VaR to have similar amplitude to the level
            scale = float(np.nanstd(cs_.values)) / max(float(np.nanstd(rv_.values)), 1e-9)
            l1, = ax.plot(cs_.index, cs_, label="Credit Spread (level, bp)")
            L2, = ax.plot(rv_.index, rv_ * scale,
             label=f"Rolling VaR × {scale:.0f} (Δ→ level scale)")


            ax.set_ylabel("bp (level)")
            # right secondary axis showing REAL VaR values
            secax = ax.secondary_yaxis('right',
                                        functions=(lambda y: y / scale, lambda y: y * scale))
            secax.set_ylabel("Rolling VaR (Delta, bp)")
            ax.legend(loc="upper left")
            ax.grid(True, alpha=0.3)

        elif mode == 'zscore':
            z1 = (cs_ - cs_.mean()) / (cs_.std() + 1e-12)
            z2 = (rv_ - rv_.mean()) / (rv_.std() + 1e-12)
            ax.plot(z1.index, z1, label="Credit Spread (z-score)")
            ax.plot(z2.index, z2, label="Rolling VaR (z-score)")
            ax.set_ylabel("standardized units")
            ax.legend(loc="upper right")
            ax.grid(True, alpha=0.3)

        elif mode == 'index100':
            base1 = float(cs_.iloc[0])
            base2 = float(rv_[rv_ > 0].iloc[0]) if (rv_ > 0).any() else max(float(rv_.iloc[0]), 1e-6)
            ax.plot(cs_.index, cs_ / base1 * 100, label="Credit Spread (index=100)")
            ax.plot(rv_.index, rv_ / base2 * 100, label="Rolling VaR (index=100)")
            ax.set_ylabel("Index (base=100)")
            ax.legend(loc="upper right")
            ax.grid(True, alpha=0.3)

        else:
            raise ValueError("mode must be 'dual', 'scaled_std', 'zscore', or 'index100'.")

        ax.set_title(title or "Credit Spread & Rolling VaR (95%)")
```

```
        plt.tight_layout()

        if self.save_plots:
            self._ensure_outdir()
            out = fname or f"{self.outdir}/fig_rolling_var_{mode}.png"
            plt.savefig(out, dpi=200)

        plt.show()
        return ax

[...]

##########################################################
#                       PIPELINE 2                       #
##########################################################
class LogLinearRegressionWrapper(BaseEstimator, RegressorMixin):
    """
    Wraps LinearRegression to automatically apply log1p transformation
    to the target variable during fit and expm1 during predict.
    """
    def __init__(self, fit_intercept=True, copy_X=True, n_jobs=None, positive=False):
        # Store parameters for LinearRegression
        self.fit_intercept = fit_intercept
        self.copy_X = copy_X
        self.n_jobs = n_jobs
        self.positive = positive
        # Internal model instance
        self._model = LinearRegression(
            fit_intercept=self.fit_intercept,
            copy_X=self.copy_X,
            n_jobs=self.n_jobs,
            positive=self.positive
        )

```

```python
    def fit(self, X, y):
        """Fits the model applying log1p to y."""
        try:
            # Check if y contains non-positive values after potential differencing
            if (y <= -1).any():
                logging.warning("Target variable contains values <= -1. Log1p transformation might produce NaNs or errors. Consider checking data or transformation.")
                # Option: Clip values, add a small constant, or handle as needed
                y_processed = y.clip(lower=-0.9999) # Example: Clipping
            else:
                y_processed = y

            y_log = np.log1p(y_processed)
            # Check for NaNs/infs after log1p
            if np.isnan(y_log).any() or np.isinf(y_log).any():
                logging.error("NaN or Inf values found in target after log1p transformation.")
                # Handle this case: maybe impute, drop rows, or raise error
                # For now, let's try dropping corresponding rows in X and y_log
                nan_inf_mask = np.isnan(y_log) | np.isinf(y_log)
                if isinstance(X, pd.DataFrame):
                    X_clean = X[~nan_inf_mask]
                else: # Assuming numpy array
                    X_clean = X[~nan_inf_mask, :]
                y_log_clean = y_log[~nan_inf_mask]
                if len(y_log_clean) == 0:
                    raise ValueError("Target variable resulted in all NaNs/Infs after log1p.")
                self._model.fit(X_clean, y_log_clean)
            else:
                self._model.fit(X, y_log)

            self.is_fitted_ = True
        except Exception as e:
            logging.error(f"Error during LogLinearRegressionWrapper fit: {e}")
            # Consider re-raising or handling appropriately
            raise
        return self


    def predict(self, X):
        """Predicts on the original scale applying expm1."""
        if not hasattr(self, 'is_fitted_') or not self.is_fitted_:
            raise ValueError("This LogLinearRegressionWrapper instance is not fitted yet.")
        y_pred_log = self._model.predict(X)
        # Apply expm1 transformation
        y_pred = np.expm1(y_pred_log)
        # Ensure predictions are not negative if log1p was potentially problematic
        y_pred = np.maximum(y_pred, 0) # Optional: clip negative preds if they don't make sense
        return y_pred

    # You might need to add get_params and set_params if using grid search directly on this wrapper
    def get_params(self, deep=True):
        return self._model.get_params(deep=deep)

    def set_params(self, **params):
        self._model.set_params(**params)
        # Also update the wrapper's stored params if they are passed
        for param, value in params.items():
            if hasattr(self, param):
                setattr(self, param, value)
        return self
```

```
def c2_feature_engineering(df):
    """
    Applies feature engineering for Pipeline 2, adding lags and moving averages to Credit_Spread.
    """
    df = df.copy()
    df['Credit_Spread_lag1'] = df['Credit_Spread'].shift(1)
    df['Credit_Spread_MA5'] = df['Credit_Spread'].rolling(window=5).mean()
    df['Credit_Spread_MA20'] = df['Credit_Spread'].rolling(window=20).mean()
    return df


def c2_prepare_target(df, target_column='Credit_Spread', h=FORECAST_H):
    """
    Sets the target variable by shifting the specified column forward by h periods.
    """
    df = df.copy()
    df['Target_Credit_Spread'] = df[target_column].shift(-h)
    return df.dropna(subset=['Target_Credit_Spread'])


def c2_filter_data_from_2000(df):
    """
    Filters data to include only records from 2000 onward.
    """
    df = df.loc[df.index >= '2000-01-01']
    logging.info(f"Dataset after filtering from 2000: {df.shape}")
    return df


def c2_explore_data(df):
    """
    Performs exploratory data analysis, printing summaries and plotting correlations and distributions.
    """
    print("First 5 rows:")
    print(df.head())
    print("\nDescriptive statistics:")
    print(df.describe())
    print("\nDataset info:")
    print(df.info())
    print("\nMissing values per column:")
    print(df.isnull().sum())
    plt.figure(figsize=(16, 12))
    sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
    plt.title("Correlation Heatmap")
    plt.show()
    df.hist(bins=30, figsize=(20, 15))
    plt.suptitle("Feature Distributions", fontsize=20)
    plt.show()


def c2_reduce_skew(df, cols, threshold=0.75):
    """
    Reduces skewness in specified columns with a log transformation if skewness exceeds the threshold.
    """
    from scipy.stats import skew
    df_trans = df.copy()
    for col in cols:
        s = skew(df_trans[col].dropna())
        if abs(s) > threshold:
            df_trans[col] = np.log(df_trans[col] + 1)
            logging.info(f"Log transformation applied to {col} (initial skew: {s:.2f}).")
    return df_trans


def c2_normalize_df(df, cols):
    """
    Normalizes specified columns using StandardScaler.
    """
    scaler = StandardScaler()
    df[cols] = pd.DataFrame(scaler.fit_transform(df[cols]), columns=cols, index=df.index)
    return df


def c2_preprocess_dataset(df):
    """
    Preprocesses the dataset by filtering from 2000, filling missing values, reducing skewness, and normalizing.
    Ensures no NaN or infinite values remain in the dataset.
    """
    df = df.loc[df.index >= '2000-01-01']
    df = df.fillna(method='ffill').dropna()
    feature_cols = df.columns.difference(['Target_Credit_Spread'])
    df = c2_reduce_skew(df, feature_cols, threshold=0.75)

    # Remove infinite values that might have been created during logarithmic transformation
    df = df.replace([np.inf, -np.inf], np.nan)
    df = df.dropna()

    df = c2_normalize_df(df, feature_cols)

    # Final check to ensure there are no NaN or infinite values
    df = df.replace([np.inf, -np.inf], np.nan)
    df = df.dropna()

    print("\nCheck for NaN or infinite values after preprocessing:")
    has_nan = df.isnull().any().any()
    has_inf = np.isinf(df.values).any()
    print(f"Contains NaN: {has_nan}, Contains infinite values: {has_inf}")

    return df


def c2_ensure_stationarity(data, cols, alpha=0.05):
    """
    Ensures stationarity in specified columns by differencing until ADF test p-value is below alpha.
    """
    data_stationary = data.copy()
    diff_order = {}
    for col in cols:
        series = data_stationary[col].dropna()
        order = 0
        p_val = adfuller(series)[1]
        while p_val > alpha and len(series) > 1:
            series = series.diff().dropna()
            order += 1
            p_val = adfuller(series)[1]
        diff_order[col] = order
        data_stationary[col] = series.reindex(data_stationary.index, method='ffill')
        data_stationary[col] = data_stationary[col].fillna(method='bfill')
        logging.info(f"Column '{col}': differencing order = {order} (final p-value = {p_val:.4f}).")
    return data_stationary, diff_order


def c2_convert_categorical_to_numeric(df):
    """
    Converts categorical columns to numeric codes.
    """
    for col in df.columns:
        if pd.api.types.is_categorical_dtype(df[col]):
            logging.info(f"Converting column '{col}' to numeric.")
            df[col] = df[col].cat.codes
    return df
```

```python


def c2_add_market_regime(X, y, n_components=3, random_state=42):
    """
    Adds a market regime feature to X using a Gaussian HMM fitted on y.
    """
    np.random.seed(random_state)
    hmm_model = GaussianHMM(n_components=n_components, covariance_type="full",
                            n_iter=1000, algorithm='map', random_state=random_state)
    y_reshaped = y.values.reshape(-1, 1)
    hmm_model.fit(y_reshaped)
    regimes = hmm_model.predict(y_reshaped)
    X = X.copy()
    X['market_regime_cs'] = regimes
    return X, hmm_model


def c2_find_feature_importance(X, y, threshold=0.005):
    """
    Computes feature importances with Random Forest, plots them, and selects features above the threshold.
    """
    X_numeric = X.select_dtypes(include=[np.number])
    if X_numeric.empty:
        raise ValueError("No numeric column available for Random Forest.")
    rf = RandomForestRegressor(n_estimators=1000, max_depth=5,
                               min_samples_leaf=4, max_features=0.1, random_state=42)
    rf.fit(X_numeric, y)
    features = X_numeric.columns.values
    importances = rf.feature_importances_
    sorted_idx = importances.argsort()
    print("\nFeature Importances:")
    for idx in sorted_idx:
        print(f"{features[idx]:<40}: {importances[idx]:.6f}")
    plt.figure(figsize=(10, 12))
    plt.barh(features[sorted_idx], importances[sorted_idx],
   color='skyblue')
    plt.xlabel('Importance')
    plt.title('Feature Importance (Random Forest)')
    plt.tight_layout()
    plt.show()
    selected_features = [f for imp, f in zip(importances,
   features) if imp >= threshold]
    return selected_features


##########################################################
#                     PIPELINE 1                         #
##########################################################
def run_pipeline1(common_df):
    """
    Executes Pipeline 1: splits data, performs feature
   engineering, preprocessing, regime analysis,
    model training, and risk analysis.
    """
    df1 = common_df.copy()
    train_size = int(len(df1) * 0.8)
    train_df = df1.iloc[:train_size].copy()
    test_df = df1.iloc[train_size:].copy()
    train_df = feature_engineering_causale(train_df, h=
   FORECAST_H, mode=FORECAST_MODE)
    test_df = feature_engineering_causale(test_df, h=FORECAST_H
   , mode=FORECAST_MODE)
    feature_cols = train_df.columns.difference(['Target', '
   Credit_Spread'])
    train_df = reduce_skew(train_df, feature_cols)
    test_df = reduce_skew(test_df, feature_cols)
    train_df, test_df = preprocess_data(train_df, test_df,
   feature_cols)
    regime_analyzer = MarketRegimeAnalyzer(random_state=42)
    best_n, best_bic = regime_analyzer.tune_n_components(
train_df)

    bic_df = regime_analyzer.bic_diagnostics(
        train_df, components=range(2, 7), outdir="outputs",
        save_csv=True, save_fig=True
    )

    logging.info(f"Tuned number of regimes: {best_n} with BIC:
{best_bic:.2f}")
    train_df['Regime'] = regime_analyzer.predict(train_df)
    test_df['Regime'] = regime_analyzer.predict(test_df)
    trans_matrix = regime_analyzer.compute_transition_matrix(
train_df['Regime'])
    logging.info("Transition matrix (train set):")
    logging.info("\n" + str(trans_matrix))
    regime_analysis = regime_analyzer.analyze_regimes(train_df,
 train_df['Regime'])
    print("Regime Analysis (Pipeline 1):\n", regime_analysis)
    Visualizer.plot_regimes(pd.concat([train_df, test_df]), pd.
concat([train_df['Regime'], test_df['Regime']]))
    ra = RiskAnalyzer(
        conf_levels=(0.90, 0.95, 0.99),
        horizon=1,
        use_bps=True,
        min_samples=80,
        outdir=None,
        save_csv=False,
        save_plots=False,
        bps_mode='auto'
    )

    var_es_train = ra.var_es_by_regime(train_df['Credit_Spread'
], train_df['Regime'])
    print("\n[Pipeline1] Regime-conditioned VaR/ES (TRAIN,
Delta CS in bp):\n", var_es_train)

```

```
    var_es_test = ra.var_es_by_regime(test_df['Credit_Spread'], test_df['Regime'])
    print("\n[Pipeline1] Regime-conditioned VaR/ES (TEST, Delta CS in bp):\n", var_es_test)


    ra.plot_regime_bars(var_es_train, title="Regime-conditioned VaR (Train, 95%)")

    rolling_v = ra.rolling_var(pd.concat([train_df['Credit_Spread'], test_df['Credit_Spread']]),
                               level=0.95, window=252)
    ra.plot_rolling_var(pd.concat([train_df['Credit_Spread'], test_df['Credit_Spread']]),
                        rolling_v, title="Credit Spread & Rolling VaR (95%)", mode='scaled_std')


    X_train = train_df.drop(columns=['Target', 'Credit_Spread'])
    y_train = train_df['Target']
    X_test = test_df.drop(columns=['Target', 'Credit_Spread'])
    y_test = test_df['Target']
    # --- Baseline naive: y{t+1} = yt (random walk) ---


    # Baseline naive
    if FORECAST_MODE == 'level':
        # y_{t+h} = y_t
        y_naive = test_df['Credit_Spread'].reindex(y_test.index)
    else:
        # Delta{t to t+h} = 0
        y_naive = pd.Series(0.0, index=y_test.index)

    mask = y_naive.notna() & y_test.notna()
    naive_result_p1 = {
        'name': f"Naive (h={FORECAST_H})",
        'rmse': float(np.sqrt(mean_squared_error(y_test[mask], y_naive[mask]))),
        'mae': float(mean_absolute_error(y_test[mask], y_naive[mask])),
        'r2': float(r2_score(y_test[mask], y_naive[mask])),
        'predictions': y_naive[mask].values,
        'test_index': y_test[mask].index,
        'y_test': y_test[mask]
    }

    logging.info(f"X_train columns (Pipeline 1): {X_train.columns.tolist()}")
    model_tuner = ModelTuner()
    rf_model, rf_params = model_tuner.tune_random_forest(X_train, y_train)
    xgb_model, xgb_params = model_tuner.tune_xgboost(X_train, y_train)
    tcn_model = TCNRegressorFastSK(
        residual_mode='diff' if FORECAST_MODE == 'level' else 'level',
        window=64, horizon=FORECAST_H,
        channels=32, n_blocks=5, kernel_size=3, dropout=0.1,
        n_regimes=best_n, batch_size=128, samples_per_epoch=16000,
        max_epochs=12, patience=6, lr=2e-3, weight_decay=2e-4, num_workers=0
    )


    models_tab = {
        'Random Forest': rf_model,
        'XGBoost': xgb_model,
    }
    res_tab = ModelEvaluator.evaluate_models(models_tab, X_train, y_train, X_test, y_test)


    models_tcn = {
        'TCN': tcn_model
    }
    res_tcn = ModelEvaluator.evaluate_models(
        models_tcn,
        X_train, train_df['Credit_Spread'],
        X_test, test_df['Credit_Spread']
    )

    return [naive_result_p1] + res_tab + res_tcn


##########################################################
#                    PIPELINE 2                          #
##########################################################
[...]

##########################################################
#                        MAIN                            #
##########################################################
def main():
    """
    Main function to load data, execute both pipelines,
    summarize results,
    and plot predictions using standardized evaluation.
    """
    config = CommonConfig()
    midas_loader = MIDASDataLoader(config)
    transformed_df = midas_loader.load_and_transform_data()
    logging.info(
        f"Combined dataset with MIDAS transformation: {
    transformed_df.shape}, Columns: {transformed_df.columns.
    tolist()}")
    midas_cols = [col for col in transformed_df.columns if col.
    endswith('_midas')]
    logging.info(f"MIDAS transformed columns: {midas_cols}")

    # --- Execute Pipeline 1 and Evaluate ---
    # run_pipeline1 now returns the list including the naive baseline + evaluated RF, XGB, TCN
    results_pipeline1 = run_pipeline1(transformed_df)
    # Extract evaluated models (excluding naive for now, will add later)
    evaluated_results_p1 = [res for res in results_pipeline1 if not res['name'].startswith('Naive')]
    naive_baseline_result = results_pipeline1[0] # Assuming naive is first

    # --- Execute Pipeline 2 to get model and data ---
    results_pipeline2_data = run_pipeline2(transformed_df)

    # Check if pipeline 2 succeeded
    if results_pipeline2_data.get('model') is None:
         logging.error("Pipeline 2 failed to produce a model. Skipping its evaluation.")
         evaluated_results_p2 = []
    else:
        # --- Evaluate Pipeline 2 Model using ModelEvaluator ---
        models_p2 = {'Linear Regression ': results_pipeline2_data['model']}
        evaluated_results_p2 = ModelEvaluator.evaluate_models(
            models=models_p2,
            X_train=results_pipeline2_data['X_train'],
            y_train=results_pipeline2_data['y_train'],
            X_test=results_pipeline2_data['X_test'],
            y_test=results_pipeline2_data['y_test']
        )
        # Rename the model in the results for clarity if needed
        if evaluated_results_p2:
            evaluated_results_p2[0]['name'] = 'Linear Regression (P2)'


    # --- Combine All Results ---
```

```python
    # Ensure naive baseline has compatible structure if needed later
    # Make sure predictions/y_test/index lengths match if comparing directly
    all_evaluated_results = [naive_baseline_result] + evaluated_results_p1 + evaluated_results_p2

    # --- Summarize and Plot ---
    print_results_summary(all_evaluated_results)
    plot_all_model_predictions(all_evaluated_results)

if __name__ == "__main__":
    main()
```

List of Figures

Bibliography

[1] Dennis Y. Tang and Han Yan. “Market Conditions and Credit Spreads”. In: *Journal of Financial Markets* 13.4 (2010), pp. 386–418.

[2] Scott R. Baker, Nicholas Bloom, and Steven J. Davis. “Measuring Economic Policy Uncertainty”. In: *The Quarterly Journal of Economics* 131.4 (2016), pp. 1593–1636.

[3] Robert C. Merton. “On the Pricing of Corporate Debt: The Risk Structure of Interest Rates”. In: *Journal of Finance* 29.2 (1974), pp. 449–470.

[4] Spyros Bougheas, Laura Switzer, and Jessica Symons. “Liquidity and Credit Spreads in Corporate Bond Markets”. In: *Journal of Financial Economics* 45.3 (1997), pp. 423–453.

[5] Jennie Bai, Itay Goldstein, and David Yang. “The Credit Spread Puzzle: New Evidence from the Cross-Section of Bond Returns”. In: *Review of Financial Studies* 33.7 (2020), pp. 2876–2920.

[6] Simon Gilchrist and Egon Zakrajšek. “Credit Spreads and Business Cycle Fluctuations”. In: *American Economic Review* 102.4 (2012), pp. 1692–1720.

[7] Jon Faust et al. *Credit Spreads as Predictors of Real Economic Activity.* Tech. rep. 2013–201. Board of Governors of the Federal Reserve System, 2013.

[8] Eugene F. Fama and Kenneth R. French. “Business Conditions and Expected Returns on Stocks and Bonds”. In: *Journal of Financial Economics* 25.1 (1989), pp. 23–49.

[9] Pierre Collin-Dufresne, Robert S. Goldstein, and J. Spencer Martin. “The Determinants of Credit Spread Changes”. In: *The Journal of Finance* 56.6 (2001), pp. 2177–2207.

[10] Edwin J. Elton et al. “Explaining the Rate Spread on Corporate Bonds”. In: *Journal of Finance* 56.1 (2001), pp. 247–277.

[11] Francis A. Longstaff, Sanjay Mithal, and Eric Neis. “Corporate Yield Spreads: Default Risk or Liquidity? New Evidence from the Credit Default Swap Market”. In: *The Journal of Finance* 60.5 (2005), pp. 2213–2253.

[12] Hui Chen, Pierre Collin-Dufresne, and Robert S. Goldstein. “On the Relation Between the Credit Spread Puzzle and the Equity Premium Puzzle”. In: *Review of Financial Studies* 22.9 (2009), pp. 3367–3409.

[13] Jun Pan and Kenneth J. Singleton. “Default and Recovery Implicit in the Term Structure of Sovereign CDS Spreads”. In: *The Journal of Finance* 63.5 (2008), pp. 2345–2384.

[14] James D. Hamilton. “A New Approach to the Economic Analysis of Nonstationary Time Series and the Business-Cycle”. In: *Econometrica* 57.2 (1989), pp. 357–384.

[15] Eric Ghysels, Pedro Santa-Clara, and Rossen Valkanov. “The MIDAS Touch: Mixed Data Sampling Regression Models”. In: *American Economic Review* 94.2 (2004), pp. 22–26.

[16] Jushan Bai and Serena Ng. “Determining the Number of Factors in Approximate Factor Models”. In: *Econometrica* 70.1 (2002), pp. 191–221.

[17] Claude B. Erb, Campbell R. Harvey, and Theodore E. Viskanta. “Political Risk, Economic Risk, and Financial Risk”. In: *Financial Analysts Journal* 52.6 (1996), pp. 29–46.

[18] Jack Bao, Jun Pan, and Jiang Wang. “The Illiquidity of Corporate Bonds”. In: *The Journal of Finance* 66.3 (2011), pp. 911–946.

[19] Kent Daniel, Mark Grinblatt, and Sheridan Titman. “Momentum Strategies”. In: *Journal of Finance* 43.5 (1988), pp. 1405–1417.

[20] Dirk Ourston et al. “Applications of Hidden Markov Models to Detecting Multi-Stage Network Attacks”. In: *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS).* Big Island, HI, USA: IEEE, 2003.

[21] Lawrence R. Rabiner. “A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition”. In: *Proceedings of the IEEE* 77.2 (1989), pp. 257–286.

[22] Olivier Cappé, Eric Moulines, and Tobias Rydén. *Inference in Hidden Markov Models.* Springer Series in Statistics. New York: Springer, 2005.

[23] Monica Franzese and Antonella Iuliano. *Hidden Markov Models.* Tech. rep. © 2019 Elsevier Inc. All rights reserved. Institute for Applied Mathematics “Mauro Picone”, Napoli, Italy, 2019.

[24] Benyamin Ghojogh, Fakhri Karray, and Mark Crowley. *Hidden Markov Model: Tutorial.* Unpublished manuscript, University of Waterloo. 2019.

[25] Gideon Schwarz. “Estimating the Dimension of a Model”. In: *The Annals of Statistics* 6.2 (1978), pp. 461–464.

[26] Eric Ghysels, Pedro Santa-Clara, and Rossen Valkanov. *MIDAS Regressions.* Working paper. Foundational reference for Mixed Data Sampling (MIDAS) regressions. 2007.

[27] Camilla Foroni, Massimiliano Marcellino, and Christian Schumacher. “UMIDAS: MIDAS Regressions with Unrestricted Lag Polynomials”. In: *Journal of the Royal Statistical Society: Series A (Statistics in Society)* 178.1 (2015), pp. 57–82.

[28] Elena Andreou, Eric Ghysels, and Vytaras Kvedaras. *Regression Models with Mixed Sampling Frequencies: A Review of MIDAS.* Working paper. Survey on MIDAS methods and applications. 2010.

[29] George E. P. Box and David R. Cox. “An Analysis of Transformations”. In: *Journal of the Royal Statistical Society Series B (Methodological)* 26.2 (1964), pp. 211–252.

[30] In-Kwon Yeo and Richard A. Johnson. "A New Family of Power Transformations to Improve Normality or Symmetry". In: *Biometrika* 87.4 (2000), pp. 954–959.

[31] A. Colin Cameron and Pravin K. Trivedi. *Microeconometrics Using Stata*. Stata Press, 2010.

[32] Pauli Virtanen and et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272.

[33] Rob J. Hyndman and George Athanasopoulos. *Forecasting: Principles and Practice*. 2nd. OTexts, 2018.

[34] Max Kuhn and Kjell Johnson. *Applied Predictive Modeling*. Springer, 2013.

[35] Ian T. Jolliffe. *Principal Component Analysis*. 2nd. Springer, 2002.

[36] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[37] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. 2nd. Springer, 2009.

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[39] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32.

[40] Harry H. Harman. *Modern Factor Analysis*. 3rd. University of Chicago Press, 1976.

[41] Anna B. Costello and Jason W. Osborne. "Best Practices in Exploratory Factor Analysis: Four Recommendations for Getting the Most from Your Analysis". In: *Practical Assessment, Research & Evaluation* 10.7 (2005), pp. 1–9.

[42] Leandre R. Fabrigar et al. “Evaluating the Use of Exploratory Factor Analysis in Psychological Research”. In: *Psychological Methods* 4.3 (1999), pp. 272–299.

[43] James H. Stock and Mark W. Watson. “Macroeconomic Forecasting Using Diffusion Indexes”. In: *Journal of Business & Economic Statistics* 20.2 (2002), pp. 147–162.

[44] David A. Dickey and Wayne A. Fuller. “Distribution of the Estimators for Autoregressive Time Series with a Unit Root”. In: *Journal of the American Statistical Association* 74.366 (1979), pp. 427–431.

[45] Said E. Said and David A. Dickey. “Testing for Unit Roots in Autoregressive–Moving Average Models of Unknown Order”. In: *Biometrika* 71.3 (1984), pp. 599–607.

[46] Peter J. Huber. “Robust Estimation of a Location Parameter”. In: *The Annals of Mathematical Statistics* 35.1 (1964), pp. 73–101.

[47] Roger Koenker and Gilbert Bassett. “Regression Quantiles”. In: *Econometrica* 46.1 (1978), pp. 33–50.

[48] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Cambridge, MA: MIT Press, 2016. URL: `https://www.deeplearningbook.org`.

[49] S. Arlot and A. Celisse. “A survey of cross-validation procedures for model selection”. In: *Statistics Surveys* 4 (2010), pp. 40–79.

[50] Leonard J. Tashman. “Out-of-sample tests of forecasting accuracy: an analysis and review”. In: *International Journal of Forecasting* 16.4 (2000), pp. 437–450.

[51] Rob J. Hyndman and George Athanasopoulos. *Forecasting: Principles and Practice*. 3rd ed. Melbourne: OTexts, 2021. URL: `https://otexts.com/fpp3/`.

[52] Lutz Prechelt. “Early Stopping—But When?” In: *Neural Networks: Tricks of the Trade*. Ed. by G. Orr and K.-R. Müller. Vol. 1524. LNCS. Springer, 1998, pp. 55–69.

[53] James Bergstra and Yoshua Bengio. “Random Search for Hyper-Parameter Optimization”. In: *Journal of Machine Learning Research* 13 (2012), pp. 281–305.

[54] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2012, pp. 2951–2959.

[55] Lisha Li et al. “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization”. In: *Journal of Machine Learning Research* 18.185 (2018), pp. 1–52.

[56] João Gama et al. “A Survey on Concept Drift Adaptation”. In: *ACM Computing Surveys* 46.4 (2014), 44:1–44:37.

[57] Geoffrey I. Webb et al. “Characterizing Concept Drift”. In: *Data Mining and Knowledge Discovery* 30.4 (2016), pp. 964–994.

[58] Hind Alaskar and Tanzila Saba. *Machine Learning and Deep Learning: A Comparative Review*. Book chapter (Springer). Chapter. 2021.

[59] Christopher A. Kingery. “Deep Learning: An Exposition”. Master’s Thesis. University of South Carolina, 2017. URL: `https://scholarcommons.sc.edu/etd/4294`.

[60] Jinbao Ji et al. “Development of Deep Learning Algorithms, Frameworks and Hardwares”. In: *Proceedings of WCNA 2021*. Vol. 942. Lecture Notes in Electrical Engineering. Singapore: Springer, 2022, pp. 696–710.

[61] Renzhuo Wan et al. “Multivariate Temporal Convolutional Network: A Deep Neural Networks Approach for Multivariate Time Series Forecasting”. In: *Electronics* 8.8 (2019).

[62] Víctor Lara-Benítez et al. “Temporal Convolutional Networks Applied to Energy-Related Time Series Forecasting”. In: *Applied Sciences* 10 (2020).

[63] Yangdong He and Jiabao Zhao. “Temporal Convolutional Networks for Anomaly Detection in Time Series”. In: *Journal of Physics: Conference Series* 1213.4 (2019), p. 042050.

[64] Leo Breiman. “Bagging Predictors”. In: *Machine Learning* 24.2 (1996), pp. 123–140.

[65] Tin Kam Ho. “The Random Subspace Method for Constructing Decision Forests”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.8 (1998), pp. 832–844.

[66] Gonzalo Martínez-Muñoz et al. *A Comparative Analysis of XGBoost*. arXiv preprint. 2019. arXiv: `1911.01914 [cs.LG]`.

[67] Jerome H. Friedman. “Greedy Function Approximation: A Gradient Boosting Machine”. In: *Annals of Statistics* 29.5 (2001), pp. 1189–1232.

[68] Jerome H. Friedman. “Stochastic Gradient Boosting”. In: *Computational Statistics & Data Analysis* 38.4 (2002), pp. 367–378.

[69] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2016, pp. 785–794.

[70] Yuchen Xia et al. “A Boosted Decision Tree Approach Using Bayesian Hyper-parameter Optimization for Credit Scoring”. In: *Expert Systems with Applications* 78 (2017), pp. 225–241.