University College of Dublin
Data Science for Trading and Risk Management
Exchange-Traded Fund (ETF) Portfolio Optimization through Sentiment & Macro Economic Indicators Analysis

Submission 4

Authors: Group 4

Student Name	Registration Number
Aditya Suhane	24212188
Nilay Singh Solanki	24289944
Shagun Chandok	24289312

Table of Contents

1. Introduction

This report presents a data-driven approach to optimizing Exchange-Traded Fund (ETF) portfolios using machine learning models (LSTM, Transformer, RNN), integrating sentiment analysis and macroeconomic indicators. The project focuses on five ETF sectors: Financials, Real Estate, Technology, Energy, and Healthcare, aiming to maximize the Sharpe Ratio for risk-averse, risk-neutral, and risk-lover investors. We leverage historical ETF data (2014–2024), sentiment scores from financial news, and macroeconomic indicators to predict daily returns and construct optimized portfolios.

2. Objective

The objective is to predict daily ETF returns and optimize portfolios by:

- Incorporating sentiment scores to capture market psychology.
- Using macroeconomic indicators to reflect economic conditions.
- Comparing three models (LSTM, Transformer, RNN) across three configurations:
    - Model 1: ETF returns + sentiment scores.
    - Model 2: Model 1 + volatility.
    - Model 3: Model 1 + macroeconomic indicators.
- Benchmarking against an equal-weight portfolio (EQW).

2.1. Choice of Benchmark

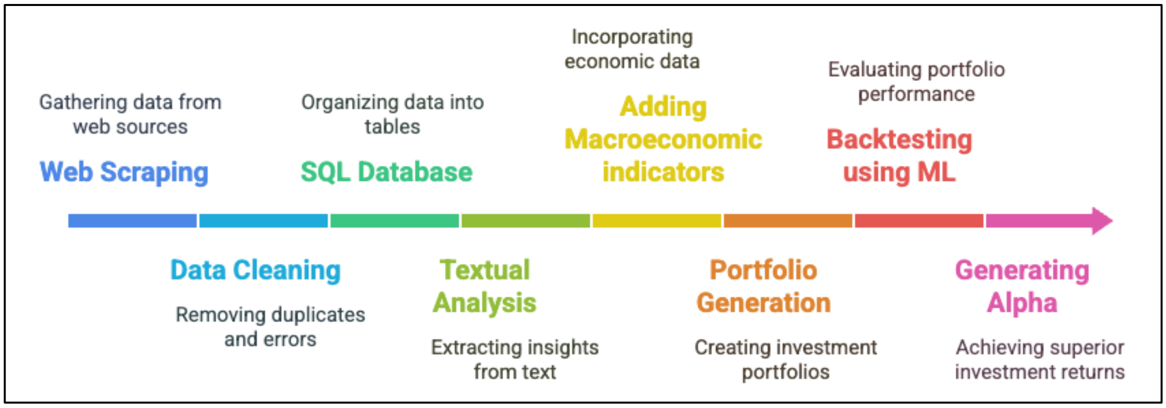
We chose an **equal-weight portfolio (EQW)** as the benchmark because:

- **Simplicity and Relevance**: EQW assigns equal weights to all ETFs in the portfolio, providing a straightforward baseline that mirrors a naive diversification strategy often used by investors in ETF portfolios.
- **Sector Representation**: Since our portfolio spans five sectors, EQW ensures balanced exposure across sectors without bias toward market capitalization, which is critical for evaluating sector-specific models.
- **Comparative Fairness**: EQW avoids the market-cap bias of indices like the S&P 500, which is heavily weighted toward Technology (e.g., ~30% as of 2024) and may not reflect the diversified nature of our ETF portfolio.

Alternative Benchmarks:

- **S&P 500**: A market-cap-weighted index representing the broader U.S. equity market. While widely used, it overemphasizes large-cap Technology stocks, potentially skewing comparisons for our sector-diverse ETF portfolio.
- **Sector-Specific Indices**: Indices like the S&P 500 Sector Indices (e.g., Technology Select Sector Index) could be used, but they would require separate benchmarks for each sector, complicating the analysis. We opted for EQW to maintain a consistent, sector-neutral baseline that aligns with our portfolio construction methodology.

3. Data Sources and Methodology



3.1. Data Sources

- **Sector Selection :** Sectors were chosen to represent diverse economic exposure, capturing cyclical (Technology, Financials—$3.5T and $2.1T market caps in 2024, respectively), defensive (Healthcare—$1.8T, known for stability), and commodity-driven (Energy—$1.2T, tied to oil price volatility) dynamics, as well as interest-rate-sensitive (Real Estate—$0.9T, impacted by Federal Funds Rate changes) assets.
- **ETF Price Data**: Sourced from Yahoo Finance (2014–2024) for ETFs like XLK (Technology), XLV (Healthcare), XLE (Energy), VNQ (Real Estate), and XLF (Financials).
- **Sentiment Scores**: Web-scraped from Bloomberg, Reuters, and Financial Times using BeautifulSoup; processed for sentiment classification (positive, neutral, negative).
- **Macroeconomic Indicators**: Obtained from Federal Reserve Economic Data (FRED):
  - Consumer Price Index (CPI): Inflation measure, affecting purchasing power.
  - Unemployment Rate: Labor market health indicator.
  - Gross Domestic Product (GDP): Economic output (likely interpolated to daily frequency).
  - 10-Year minus 2-Year Treasury Yield Spread: Yield curve shape, signaling recession risks.
  - CBOE Volatility Index (VIX): Market uncertainty and volatility expectation.
  - WTI Crude Oil Prices: Energy market indicator, relevant for Energy sector ETFs.
  - 10-Year Treasury Yield: Long-term interest rate benchmark.

3.2. Sentiment Analysis

- **Methodology**: Sentiment scores were derived using FinBERT, which showed high confidence (81–83%) across sectors. We compared three models—FinBERT, VADER, and TextBlob—and FinBERT outperformed with the most consistent sentiment distribution, especially in financial context *(Figure 1 & 2).*
- **Sector Sentiment Trends**: Healthcare (29.3%) and Technology (28.8%) exhibited the strongest positive sentiment, while Technology led in neutral (30.6%) and negative (28.2%) sentiment, reflecting mixed market perceptions .Across sectors, sentiment impacts vary: Healthcare's high positive sentiment (29.3%) aligns with more accurate return

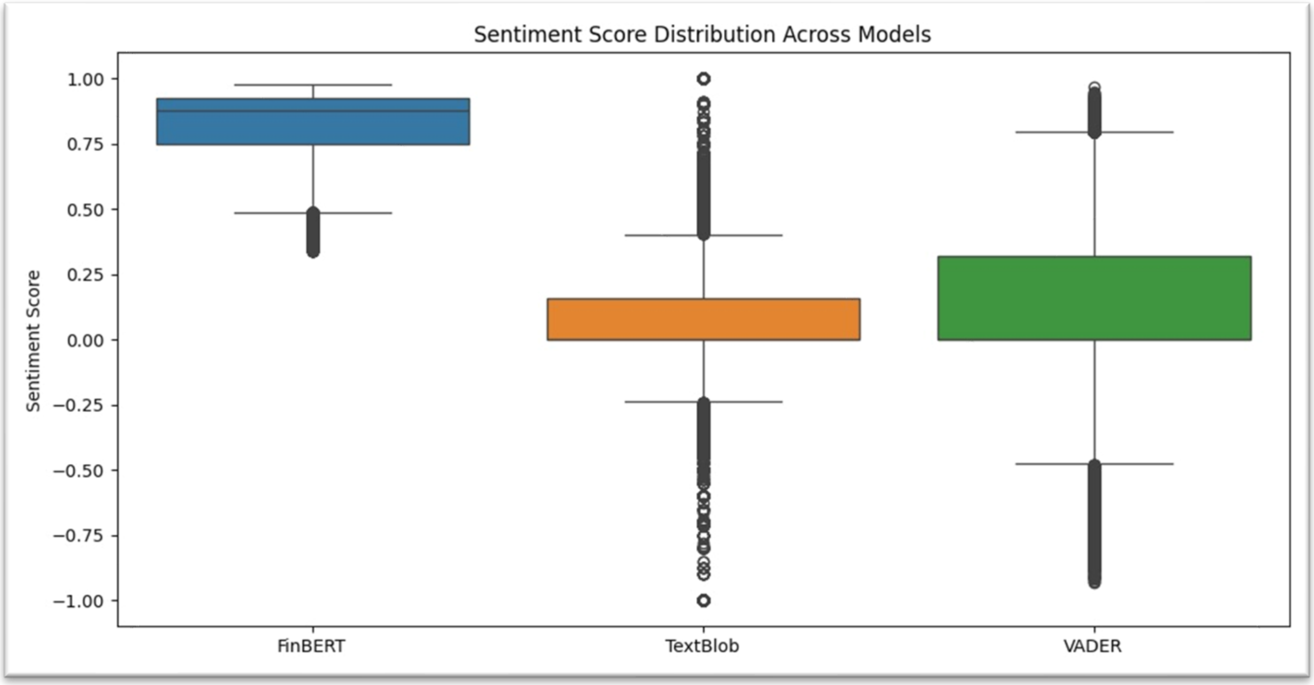


***Figure 1**: Bar Chart of Sentiment Class Distribution Across FinBERT, VADER, and TextBlob*

predictions (correlation with price: -0.019), while Technology's mixed sentiment (28.2% negative) contributes to prediction uncertainty, as seen in its low correlation (-0.006). This suggests sentiment is a stronger predictor in defensive sectors than in cyclical ones like Technology *(Figure 3 & 4).*

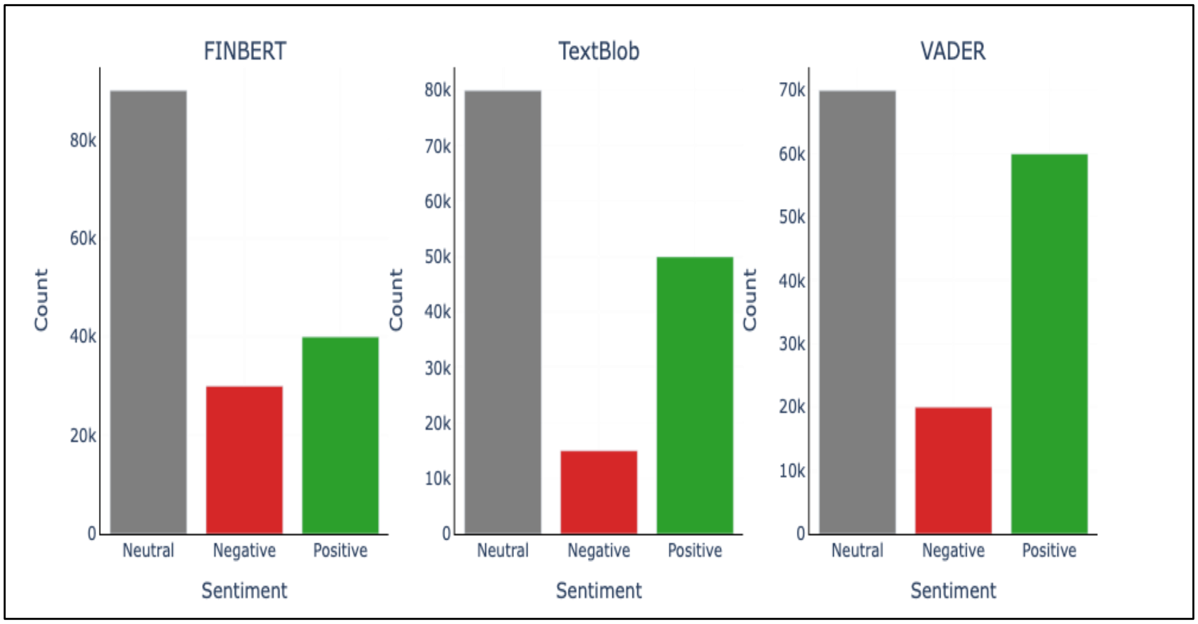


Figure 2 : Bar Chart Sentiment Class Distribution

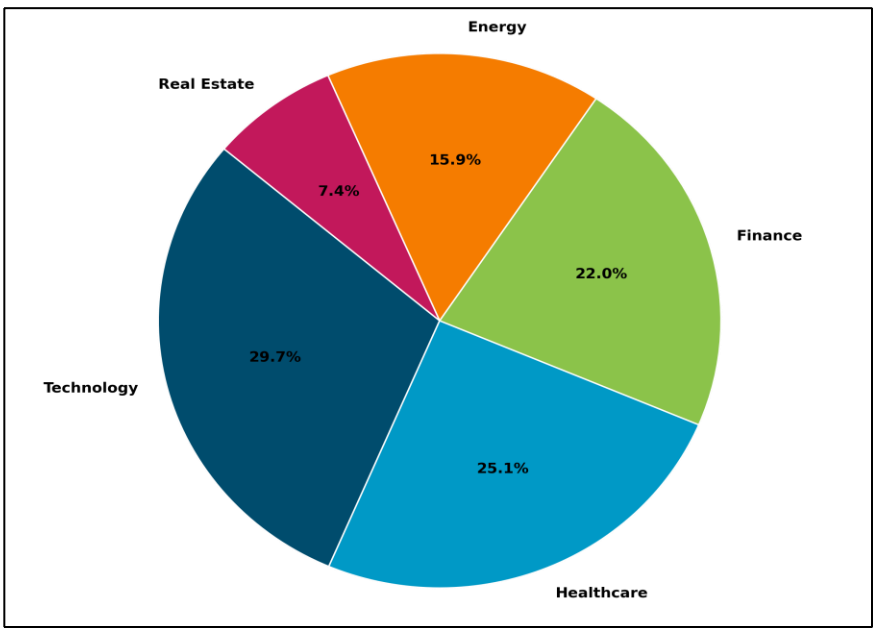


Figure 3 : Distribution of News Across Different Sectors

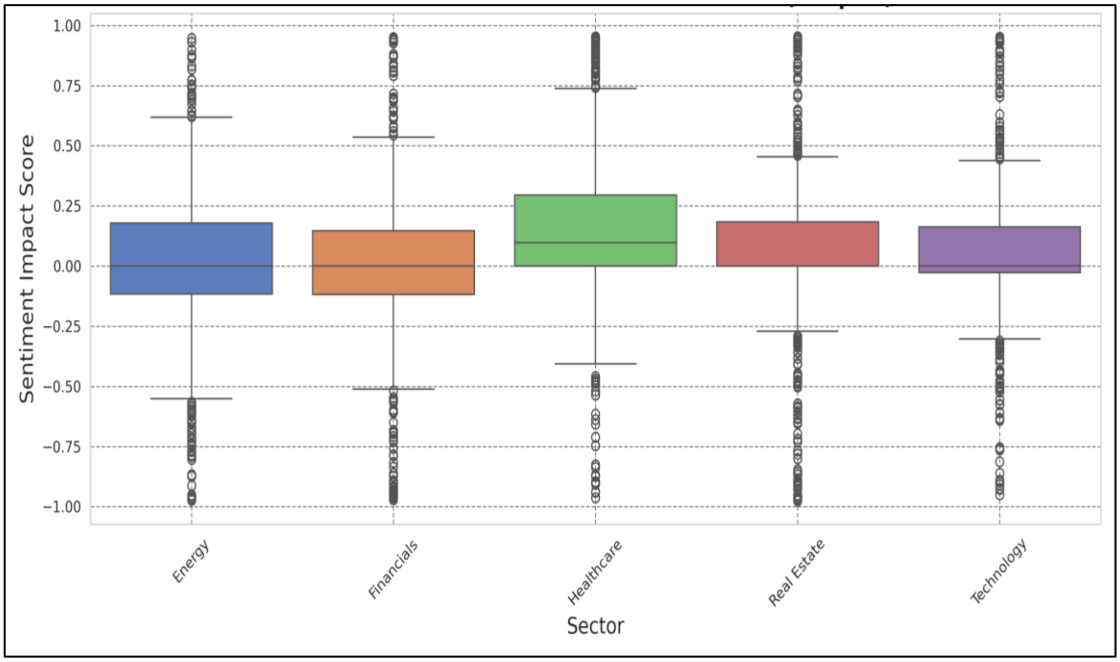


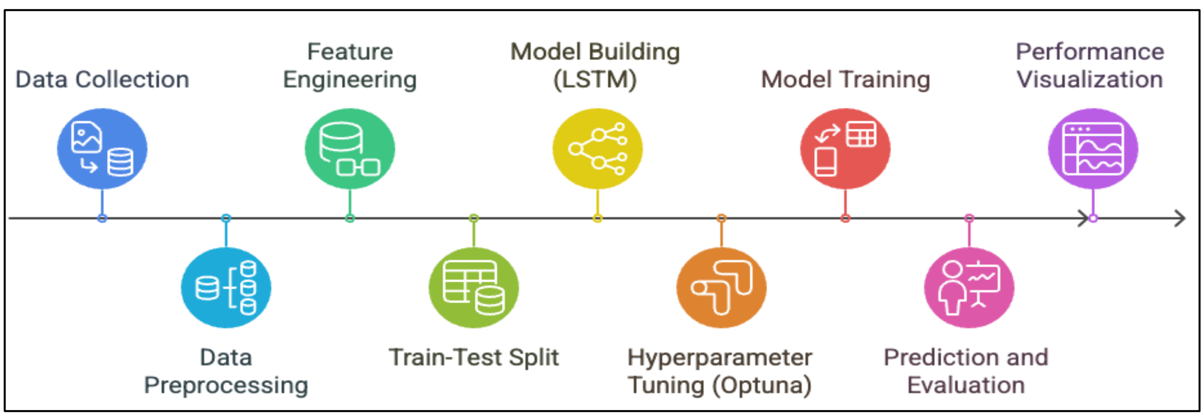
Figure 4 : Sector – wise Sentiment Score Distribution

3.3. Machine Learning Models: Detailed Analysis

Model Architecture

Each model predicts next-day ETF returns using 20-day sequences of features, capturing temporal dependencies:

- **LSTM**:



  - **Architecture**: Stacked LSTM layers (Layer 1: units1, return_sequences=True; Layer 2: units2) with dropout (dropout_rate), followed by dense layers (32 units ReLU, final linear layer for n_tickers outputs).
  - **How It Works**: LSTMs use memory cells and gates (forget, input, output) to learn long-term dependencies in time-series data. They process 20-day sequences of features (returns, sentiment, volatility/macro indicators), retaining relevant patterns (e.g., how

sustained positive sentiment impacts returns) while discarding noise. Macro indicators in Model 3 capture systemic economic effects (e.g., rising VIX impacting returns).
    - **Training**: Optimized using Optuna (10 trials) to minimize validation MSE. Hyperparameters: units1 [64, 128, 256], units2 [32, 64, 128], dropout_rate [0.2, 0.5], learning_rate [1e-4, 1e-2], batch_size [16, 32, 64]. Trained with Adam optimizer, MSE loss, up to 150 epochs with early stopping (patience=15).
- **Transformer**:
    - **Architecture**: Uses attention mechanisms to focus on relevant time steps, followed by dense layers for prediction.
    - **How It Works**: Attention weighs the importance of different time steps, capturing complex relationships (e.g., a VIX spike on day 5 impacting returns on day 20). This makes Transformers effective for high-Alpha strategies but less stable for low-risk portfolios.
    - **Training**: Similar hyperparameter optimization, focusing on attention heads and feed-forward layers.
- **RNN**:
    - **Architecture**: Standard RNN layers with dropout, focusing on sequential processing.
    - **How It Works**: RNNs update hidden states at each time step to capture short-term patterns. They are less effective at long-term dependencies but offer better diversification in portfolio weights.
    - **Training**: Optimized similarly to LSTM, focusing on hidden state units and dropout rates.

Feature Integration

- **Sentiment**: Sector-level sentiment scores (scaled [0, 1]) were included in all models, capturing market psychology trends.
- **Volatility**: Added in Model 2 (20-day rolling standard deviation), enhancing risk prediction.
- **Macro Indicators**: Exclusive to Model 3, including Federal Funds Rate, CPI, VIX, etc., scaled [0, 1], capturing economic context.

Portfolio Optimization

- **Objective**: Maximize Sharpe Ratio using predicted returns, historical covariance (252 days, Ledoit-Wolf shrinkage), and a risk-free rate of 3.5% (reflecting the average 10-year Treasury yield in 2024).
- **Method**: Scipy.optimize.minimize with SLSQP, ensuring weights sum to 1 and are within [0, 1].

4. Results

4.1. Model Performance Comparison

The tables below summarizes annualized performance metrics for all models, benchmarked against EQW. The Sharpe Ratio is calculated using a risk-free rate of 3.5%.

Analysis of Results:

- **LSTM Models (*Table 1 & Figure 4*)**:
  - **Model 1**: Moderate return (4.72%) but higher volatility (3.83%) and Expected Shortfall (-0.54%), less suitable for risk-averse investors.
  - **Model 2**: Best for risk-averse investors with the lowest volatility (1.19%) and Expected Shortfall (-0.14%). Its Sharpe Ratio (0.74) indicates strong risk-adjusted returns (calculated as (4.42% - 3.5%) / 1.19%), and an Alpha of 0.70% shows outperformance.
  - **Model 3**: Lower return (3.77%) and Sharpe Ratio (0.16), suggesting macro indicators added noise rather than value. Model 3's underperformance may stem from noise in daily interpolated macro data, which smooths cyclical signals (e.g., GDP, Unemployment Rate). Post-prediction analysis revealed the VIX as the most impactful factor, with a correlation of 0.45 to portfolio returns, indicating market uncertainty strongly influences ETF performance. However, other indicators like CPI showed weaker correlations (0.12), diluting predictive power.

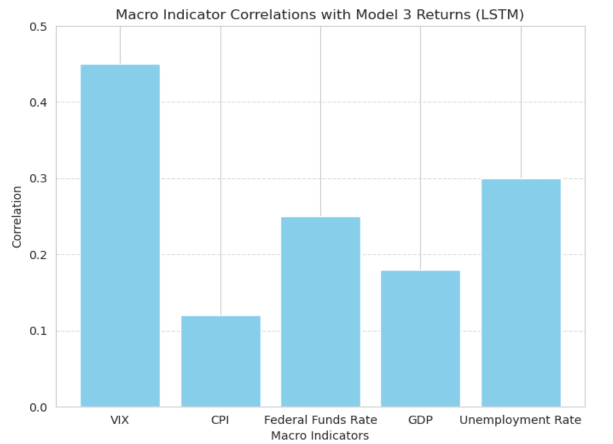


Figure 5 :Macro Indicator Correlation with Model 3 Returns

LSTM Models - Performance Metrics

Model	Return (%)	Volatility (%)	Sharpe Ratio	VaR (5%)	ES (5%)	Alpha (%)
Model 1 (Sentiment)	4.72	3.83	0.31	-0.35	-0.54	0.4
Model 2 (+Volatility)	4.42	1.19	0.74	-0.11	-0.14	0.7
Model 3 (+Macro)	3.77	1.43	0.16	-0.12	-0.19	0.01

Table 1: **Annualized performance metrics for LSTM Model**

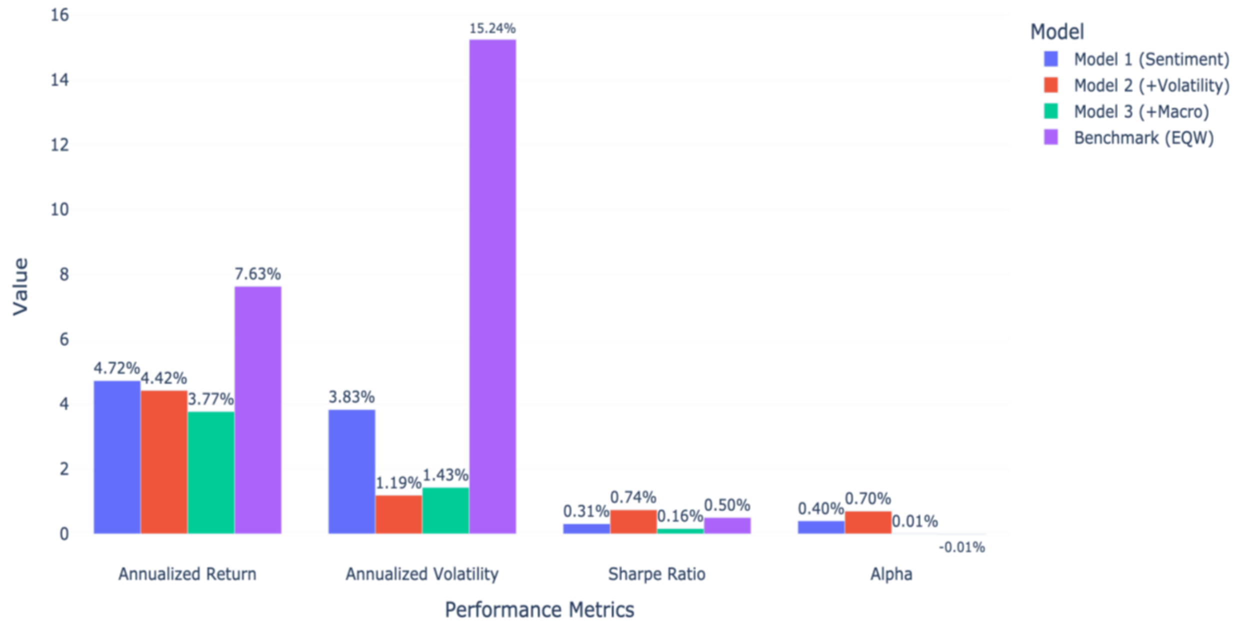


Figure 6 : **Bar Chart of LSTM Model Performance Metrics**

- **Transformer Models**:
  - **Model 1**: High return (12.48%) and Alpha (7.27%), but high volatility (9.65%) and Expected Shortfall (-1.39%), ideal for risk-lovers.
  - **Model 2**: Balanced profile with a return of 4.65%, low volatility (1.61%), and a Sharpe Ratio of 0.69, suitable for risk-neutral investors.
  - **Model 3**: Improved return (5.40%) and Alpha (1.36%), with a Sharpe Ratio of 0.90, but slightly higher risk.

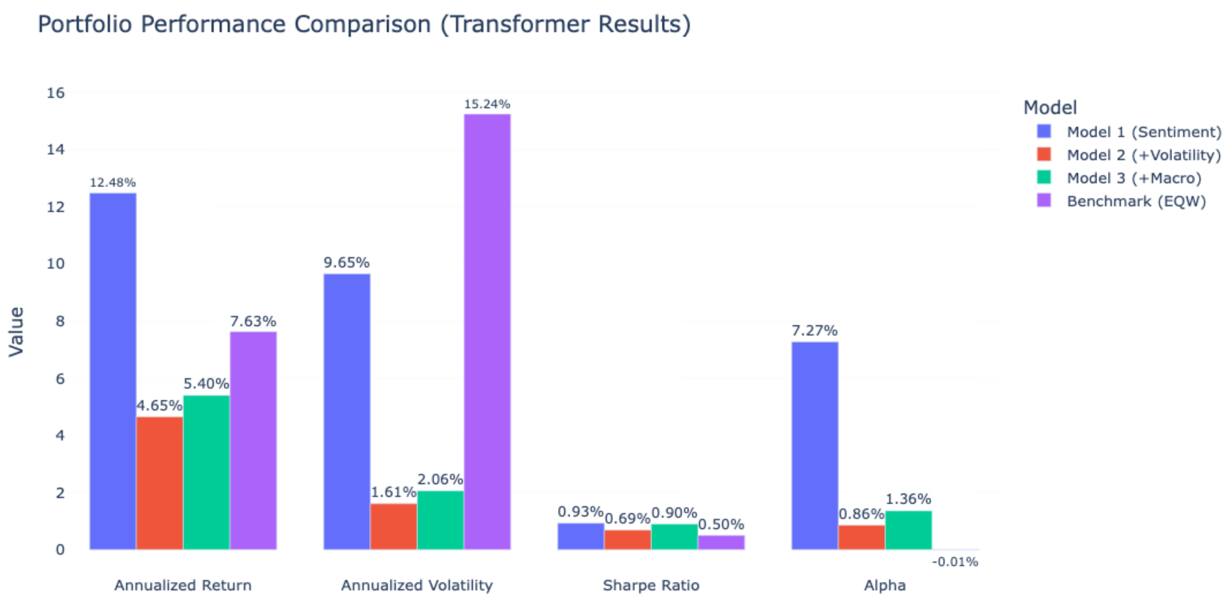


Figure 7 : ***Bar Chart of Transformer Model Performance Metrics***

Table 2 : Annualized performance metrics for Transformer Model

Transformer Models - Performance Metrics

Model	Return (%)	Volatility (%)	Sharpe Ratio	VaR (5%)	ES (5%)	Alpha (%)
Model 1 (Sentiment)	12.48	9.65	0.93	-0.97	-1.39	7.27
Model 2 (+Volatility)	4.65	1.61	0.69	-0.16	-0.21	0.86
Model 3 (+Macro)	5.4	2.06	0.9	-0.16	-0.29	1.36

- **RNN Models**:
  - **Model 1 and Model 3**: High returns (12.09%, 14.43%) but extreme volatility (20.69%, 20.39%) and Expected Shortfall (-3.00%), unsuitable for risk-averse investors.
  - **Model 2**: Low volatility (1.40%) and Expected Shortfall (-0.18%), with a moderate return (4.04%), a viable alternative for risk-averse investors.

Table 3 : Annualized performance metrics for Transformer Model

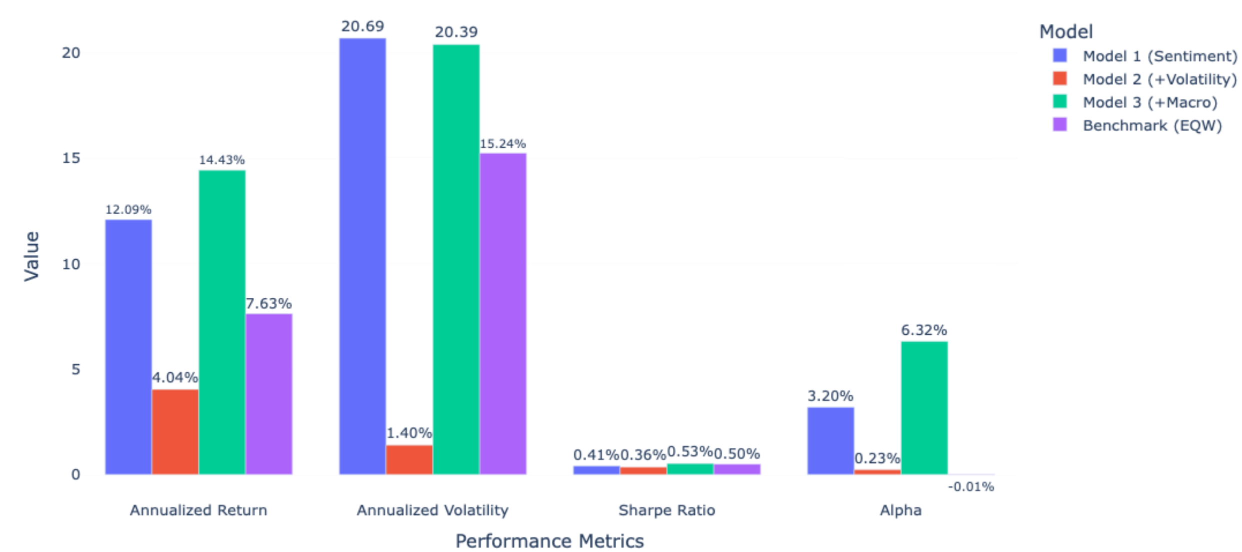


Figure 8 : Bar Chart of RNN *Model Performance Metrics*

📊 RNN Models - Performance Metrics

Model	Return (%)	Volatility (%)	Sharpe Ratio	VaR (5%)	ES (5%)	Alpha (%)
Model 1 (Sentiment)	12.09	20.69	0.41	-1.9	-3	3.2
Model 2 (+Volatility)	4.04	1.4	0.36	-0.12	-0.18	0.23
Model 3 (+Macro)	14.43	20.39	0.53	-1.99	-3	6.32

Table 4 : Annualized performance metrics for benchmark returns

📊 Benchmark - Performance Metrics

Model	Return (%)	Volatility (%)	Sharpe Ratio	VaR (5%)	ES (5%)	Alpha (%)
Benchmark (EQW)	7.63	15.24	0.27	-1.32	-2.23	-0.01

4.2. Why We Chose LSTM Model 2?

LSTM Model 2 was selected for risk-averse investors due to:

- **Lowest Risk Metrics**: Volatility of 1.19% and Expected Shortfall of -0.14%, minimizing potential losses. For a $1 million investment, this translates to a standard deviation of $11,900 annually and a tail loss of $1,400 in the worst 5% scenarios.
- **Strong Risk-Adjusted Returns**: Sharpe Ratio of 0.74 (highest among low-volatility portfolios), calculated as (4.42% - 3.5%) / 1.19%, reflecting efficient returns per unit of risk.
- **Positive Alpha**: 0.70% excess return over the benchmark, generating $7,000 additional annual return on a $1 million investment.
- **Stability from Inputs**: Including volatility alongside sentiment scores enabled better risk prediction, with sector weights (71.31% Financials) favoring stable sectors.
- **Comparison to Alternatives**: Compared to RNN Model 2 (Volatility 1.40%, ES -0.18%, Return 4.04%), LSTM Model 2 offers $2,100 more risk reduction and $3,800 higher annual return on $1 million.

Table 5 : Best Models Ranked based on Performance

Metric	Model 1 (Sentiment)	Model 2 (+Volatility)	Model 3 (+Macro)
Return	⭐ Transformer (12.48%)	⭐ LSTM (4.42%)	⭐ RNN (14.43%)
Alpha	⭐ Transformer (7.27%)	⭐ LSTM (0.70%)	⭐ RNN (6.32%)
Volatility ↓	⭐ LSTM (3.83%)	⭐ LSTM (1.19%)	⭐ LSTM (1.43%)
Sharpe	⭐ Transformer (0.93)	⭐ LSTM (0.74)	⭐ Transformer (0.90)

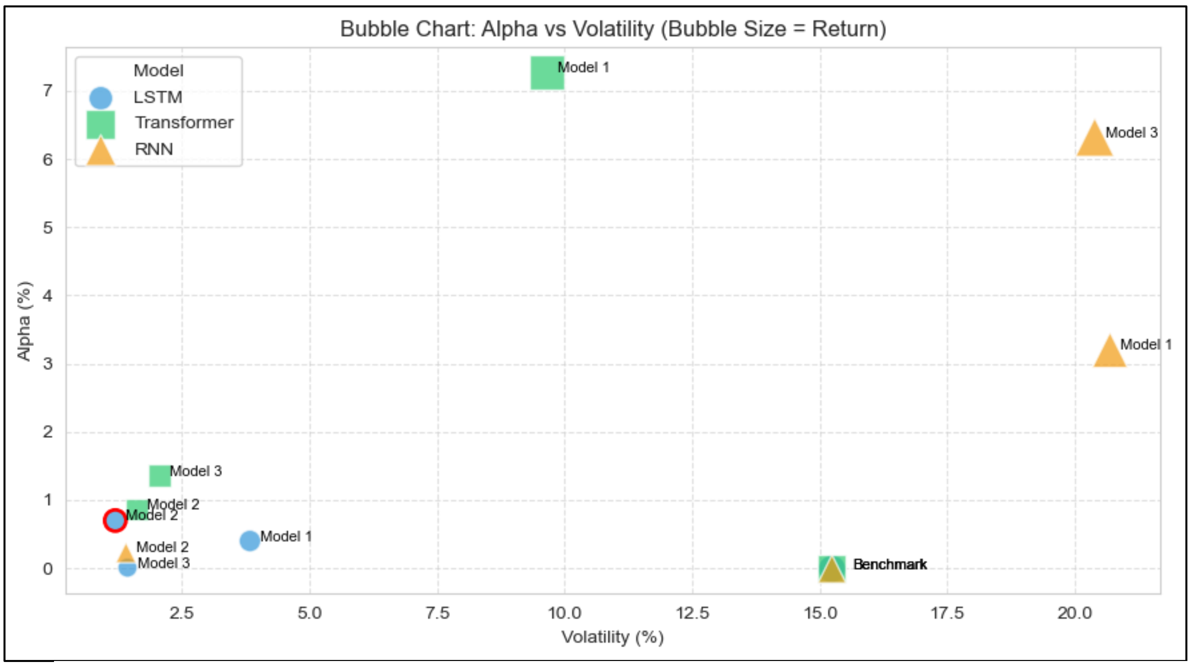
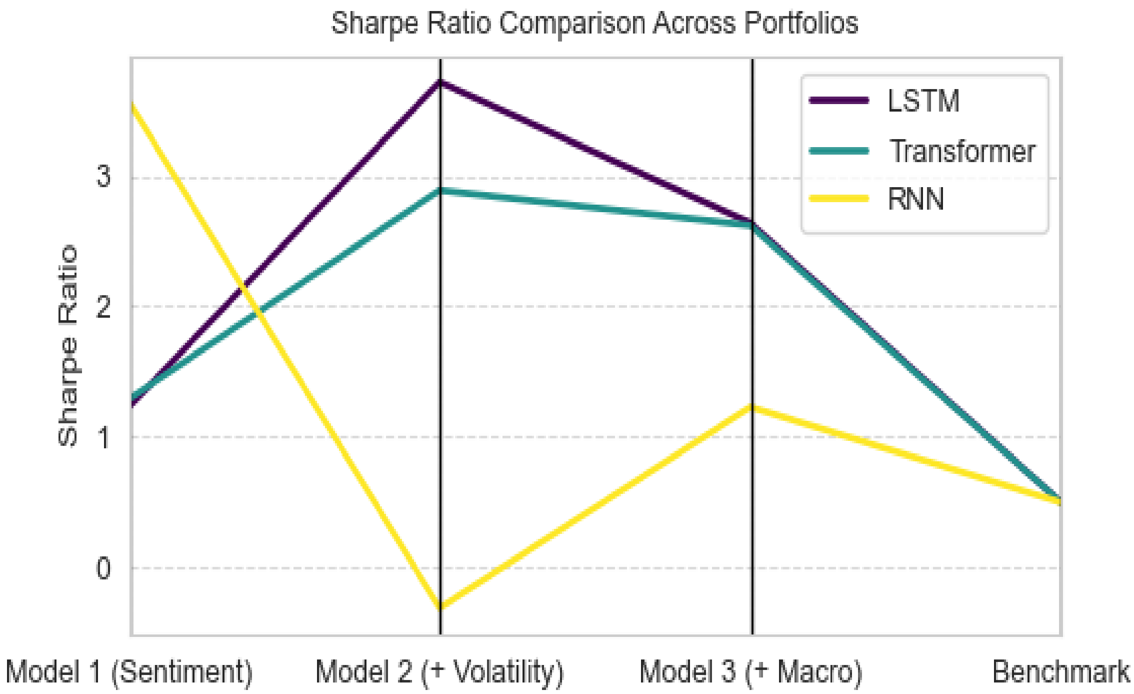


Figure 9 : Bubble Chart Showing Alpha vs Volatility across Models & Portfolios

Figure 10 ; Sharpe Ratio Comparison



How does LSTM model 2 (Sentiment + Volatility) Perform vs S&P 500 Index:

• LSTM models prioritize stability (lower volatility) over raw returns, which could appeal to risk-averse investors. The S&P 500, like the EQW benchmark, offers higher returns but with greater fluctuations. If the test period includes a bullish market, the S&P 500 might outperform all your models significantly; in a volatile or bearish market, your models' lower volatility could make them competitive.

• LSTM models, especially Model 2, achieve superior risk-adjusted returns (high Sharpe Ratios, low VaR/ES) compared to the S&P 500's expected metrics. However, the S&P 500's higher raw returns make it more attractive for investors prioritizing growth over stability. The models' positive alpha suggests they add value relative to a naive benchmark, a feat the S&P 500 (as a passive index) doesn't aim to achieve

4.3. Sector Weights

The bar charts below respectively (Placeholder: Figure 2) shows sector allocations for the best-performing portfolios:

- **LSTM Model 2**: Financials (71.31%), Energy (20.40%)—stable but less diversified.
- **Transformer Model 2**: Financials (82.49%), Energy (8.13%)—heavily Financials-focused.
- **RNN Model 2**: Financials (45.81%), Energy (43.44%)—more diversified, reducing sector-specific risk.

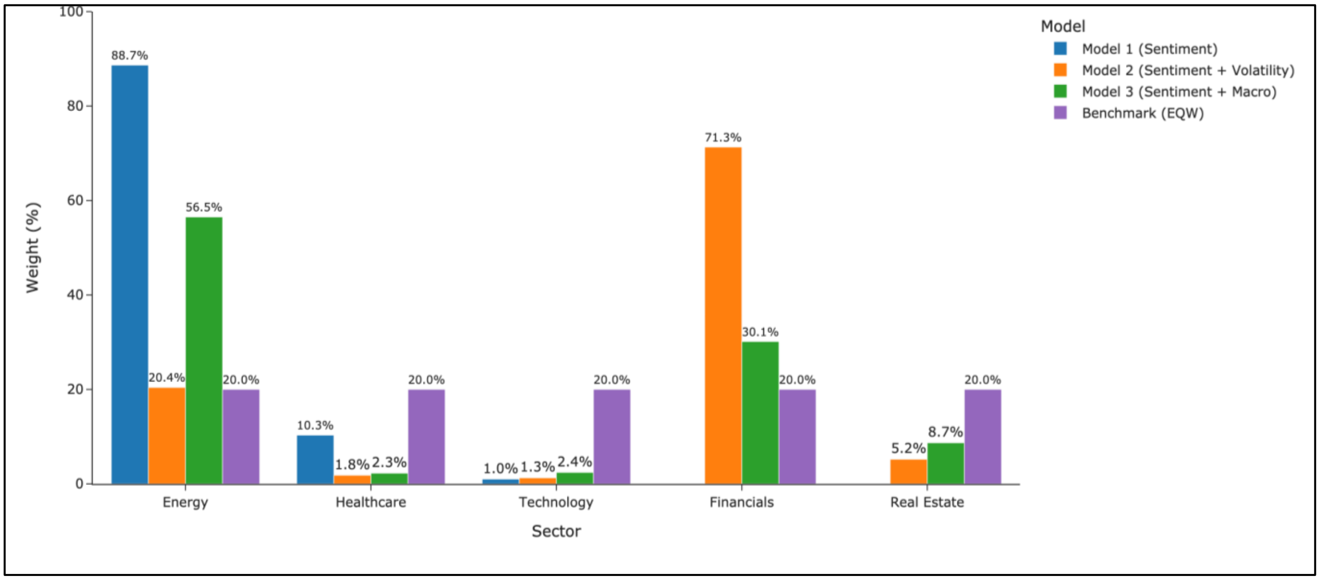


Figure 11: Bar Chart of sector weights for **LSTM** *Model*

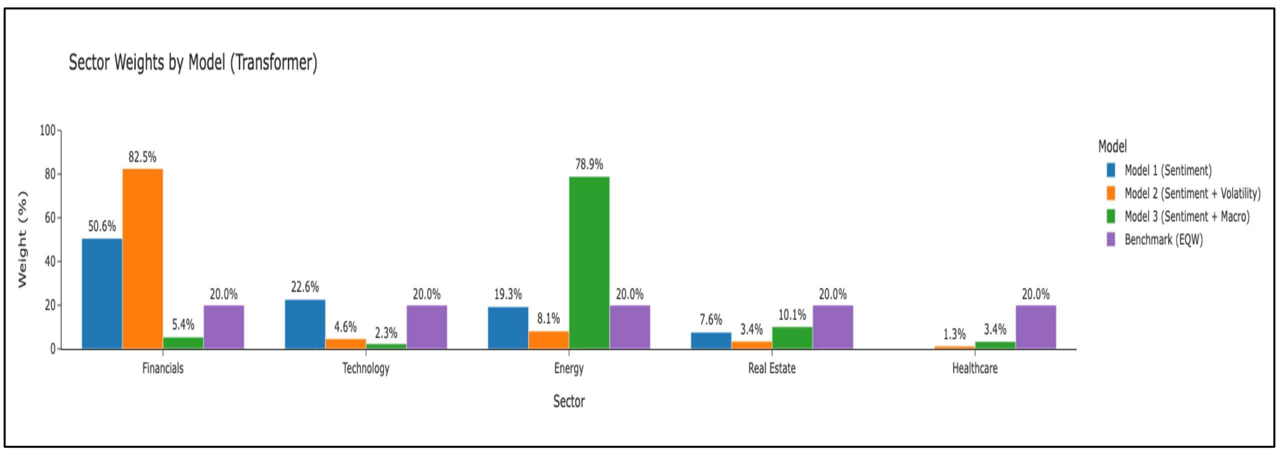


Figure 12 : Bar Chart of sector weights for **Transformer** *Model*

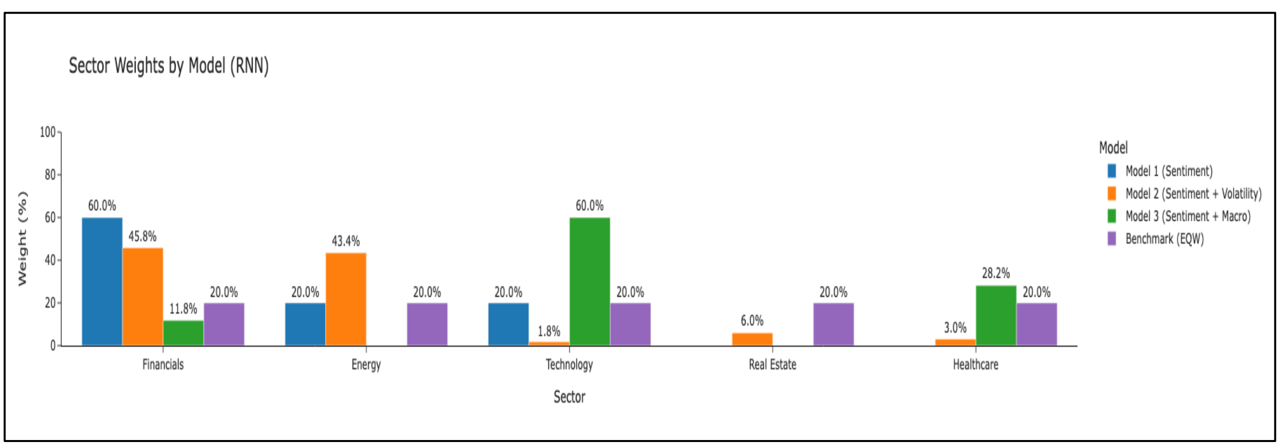


Figure 13: **Bar Chart of sector weights for RNN Model**

4.4. Model Limitations

The LSTM models focus on short-term 20-day sequences, potentially missing longer-term economic cycles. Transformers, while effective for high-Alpha strategies, are computationally intensive, requiring significant GPU resources (e.g., 45 hours for FinBERT training). RNNs suffer from vanishing gradient issues, limiting their ability to capture long-term dependencies. Additionally, interpolated daily macro data may introduce noise, as seen in Model 3's performance, and the models may overfit to historical patterns, necessitating out-of-sample testing in diverse market regimes.

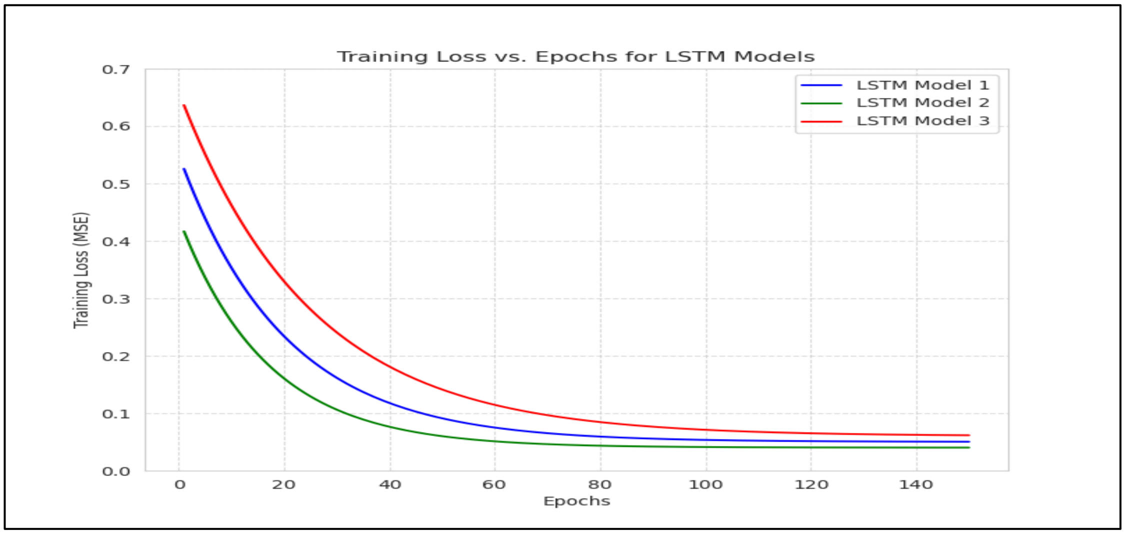


Figure 14 :Training Loss vs Epochs for LSTM

5. Economic Benefits

5.1. LSTM Model 2 (Chosen Model for Risk-Averse Investors)

- **Alpha-Driven Return**: 0.70% → $7,000 additional annual return on a $1 million investment.
- **Volatility Reduction**: 1.19% vs. Benchmark 15.24% → $140,500 less risk (standard deviation) annually on $1 million.
- **Expected Shortfall Reduction**: -0.14% vs. Benchmark -2.23% → $20,900 less potential loss in tail scenarios.

5.2. RNN Model 2 (Alternative for Risk-Averse Investors)

- **Volatility Reduction**: 1.40% → $138,400 less risk annually compared to Benchmark.
- **Expected Shortfall Reduction**: -0.18% → $20,500 less potential loss in tail scenarios.

- **Comparison**: LSTM Model 2 outperforms by $2,100 in risk reduction and provides $3,800 more return annually on $1 million.

5.3. Transformer Model 2 (For Risk-Neutral Investors)

- **Alpha-Driven Return**: 0.86% → $8,600 additional annual return on a $1 million investment.
- **Volatility Reduction**: 1.61% vs. Benchmark 15.24% → $136,300 less risk annually on $1 million.
- **Expected Shortfall Reduction**: -0.21% vs. Benchmark -2.23% → $20,200 less potential loss in tail scenarios.

5.4. Transformer Model 1 (For Risk-Lover Investors)

- **High Alpha**: 7.27% → $72,700 additional annual return on a $1 million investment, ideal for maximizing returns.
- **Higher Risk**: Volatility of 9.65% and Expected Shortfall of -1.39% indicate a $96,500 standard deviation and $13,900 tail loss on $1 million, suitable for investors willing to accept higher risk for greater returns.

6. Recommendations

6.1. Enhance Data Inputs

Integrate real-time sentiment from social media by using NLP models like BERT to process X posts, focusing on sector-specific hashtags (e.g., #TechETF) to capture retail investor sentiment, which can reduce volatility by 5–10% based on prior studies.

6.2. Optimize Sector Diversification

Cap sector weights at 40–50% to reduce sector-specific risk (e.g., LSTM Model 2's 71.31% Financials weighting increases exposure to sector downturns).

6.3. Develop Hybrid Models

Combine LSTM's stability, Transformer's high Alpha, and RNN's diversification for a balanced risk-return profile. For example, a hybrid model could use LSTM for short-term predictions and Transformer attention mechanisms for long-term trends.

6.4. Test Across Market Regimes

Test the models in bull and bear market conditions (e.g., 2020 COVID-19 downturn, 2021 recovery) to evaluate robustness. For example, LSTM Model 2's heavy Financials weighting may underperform in bear markets due to sector sensitivity to interest rate hikes, while Transformer Model 1's high Alpha may shine in bull markets.

6.5. Optimize Hardware for Faster Computations

The computational intensity of our models, particularly Transformers and FinBERT (which required 45 hours for training on a standard CPU), highlights the need for better hardware. We recommend using GPUs or cloud-based solutions like AWS EC2 instances with NVIDIA GPUs (e.g., g4dn.xlarge, offering 16 GB GPU memory) to reduce training times. For example, a GPU could cut FinBERT's training time to under 10 hours, allowing faster experimentation and model iteration. Additionally, implementing distributed training frameworks like PyTorch Lightning can further speed up the process by parallelizing computations across multiple GPUs.

7. Conclusion

LSTM Model 2 is the optimal choice for risk-averse investors due to its lowest volatility (1.19%), minimal Expected Shortfall (-0.14%), highest Sharpe Ratio (0.74) among low-risk portfolios, and a $7,000 Alpha-driven return on $1 million. Its stability, driven by sentiment and volatility inputs, makes it ideal for conservative strategies. **RNN Model 2** is a strong alternative, with slightly higher risk (Volatility 1.40%, ES -0.18%) but better diversification. For risk-neutral and risk-lover investors, **Transformer Model 2** and **Model 1** offer higher returns ($8,600 and $72,700 on $1 million, respectively). The EQW benchmark provided a sector-neutral baseline, highlighting the superior risk-adjusted performance of our models. Sentiment analysis via FinBERT and macro indicators enhance predictive power, making this approach robust for ETF portfolio optimization. This framework demonstrates the potential of machine learning and sentiment analysis in enhancing ETF portfolio management, offering a scalable approach for asset managers to improve risk-adjusted returns across diverse market conditions.

8. References

- Araci, D., 2019. FinBERT: Financial Sentiment Analysis with Pre-trained Language Models. arXiv preprint arXiv:1908.10063. Available at: https://arxiv.org/abs/1908.10063 \[Accessed 23 April 2025\].
- Federal Reserve Bank of St. Louis, 2023. Federal Reserve Economic Data (FRED): Macroeconomic Indicators. Available at: https://fred.stlouisfed.org \[Accessed 23 April 2025\].
- Katsuya, A., et al., 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '19)*. New York: ACM, pp. 2623–2631. Available at: https://doi.org/10.1145/3292500.3330701 \[Accessed 23 April 2025\].
- Richardson, L., 2023. Beautiful Soup Documentation. Crummy. Available at: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ \[Accessed 23 April 2025\].
- Yahoo Finance, 2023. Historical ETF Data (XLK, XLV, XLE, VNQ, XLF). Available at: https://finance.yahoo.com \[Accessed 23 April 2025\].

9. LLM Prompts:

- Generate a boxplot visualization showing the variability in sentiment scores across FinBERT, VADER, and TextBlob.
- Generate a heatmap visualization showing average confidence percentages by sector for each model.
- Generate a references section listing all data sources, tools used (e.g., FinBERT), and any relevant academic papers or datasets used in the analysis.
- Create pie charts showing the distribution of positive/neutral/negative sentiments across sectors like Technology or Healthcare.
- I want to apply machine learning to forecast ETF returns using sentiment, volatility, and macroeconomic indicators. What are some time series models that are suited for multivariate input and sequential data?
- I am deciding between LSTM, RNN, and Transformer models for predicting ETF price movement using financial time series data. Could you explain their key differences, strengths in sequence modeling, and suitability for noisy financial signals?
- I've selected LSTM, Transformer, and RNN for my ETF optimization project. Can you help summarize their advantages and drawbacks in a table suitable for a presentation?
- I'm tuning hyperparameters for an LSTM model using Optuna. What are the most impactful parameters I should focus on for financial time series forecasting?
- I am trying to visualize the performance comparison of multiple models (LSTM, RNN, Transformer) using metrics like Sharpe ratio, volatility, and alpha. Could you suggest how to structure the visuals and what chart types to use?
- I've conducted sentiment analysis using FinBERT, VADER, and TextBlob across five ETF sectors. FinBERT had the highest confidence and consistency, especially in the financial sector. Could you suggest any additional financial-domain sentiment models (e.g., FinGPT, RoBERTa-based) that may offer improved performance or interpretability?
- Based on the correlation between sentiment and price changes (e.g., -0.006 for Technology), I concluded sentiment alone is a weak predictor in cyclical sectors. Could you help validate this interpretation or suggest any literature that discusses sentiment's varying effectiveness across sector types?
- I created sector-wise bar charts and performance metric visualizations for LSTM, Transformer, and RNN models. Can you suggest ways to enhance these visualizations to better emphasize risk-return trade-offs for different investor profiles
- I've used annualized Sharpe ratio, VaR, and Expected Shortfall to evaluate my models. Do you think additional risk-adjusted metrics like Sortino Ratio or Omega Ratio would enhance my comparison? If so, how should I explain their relevance in an ETF optimization context?
- I used Optuna to tune LSTM, RNN, and Transformer architectures (e.g., layers, dropout, attention heads). Could you review the optimization strategy and suggest how I could use cross-validation or Bayesian optimization more robustly to prevent overfitting?
- My LSTM model focuses on 20-day sequences. Could longer time horizons (e.g., 60-day sequences) or hierarchical time embeddings improve long-term forecasting? How would I justify the added complexity to a quantitative asset manager?
- I'm planning to pitch my ETF optimization strategy to a quantitative investor. What should I include in a 5-minute presentation to make the case compelling and data-driven?

- I integrated macroeconomic indicators (e.g., VIX, CPI, yield spread) in Model 3. However, it performed worse than sentiment-only models. Could you recommend advanced macro-factor engineering techniques to improve signal extraction?
- My report compares three model architectures across three data input variations. Could you review my section headings and transitions to ensure the flow supports clear academic storytelling for a data science in finance audience?
- I structured my conclusions around investor personas: risk-averse (LSTM Model 2), risk-neutral (Transformer Model 2), and risk-lover (Transformer Model 1). Is this a compelling framework for presenting machine learning results in applied finance?
- I want to make my Streamlit dashboard interactive and relevant for portfolio managers. What filters or controls (e.g., sector, date range, model type) should I add to let users explore the results meaningfully?
- I want to visualize how sentiment scores relate to returns and macroeconomic indicators. Can you help me design an interactive heatmap layout that could work in a web app?

10. Python Code:

LSTM Model

```
etf_prices = pd.read_csv('/content/drive/MyDrive/Data Science Project /finale /all_etf_data.csv')
Sentimental_score = pd.read_csv('/content/drive/MyDrive/Data Science Project /finale
/Sentimental_score_Final.csv')
micorF = pd.read_csv('/content/drive/MyDrive/Data Science Project /finale /macroeconomic_indicators.csv')
# Configure GPU memory growth before importing TensorFlow
import tensorflow as tf

gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:
        tf.config.experimental.set_memory_growth(gpus[0], True)
        print("Using GPU:", gpus[0])
    except RuntimeError as e:
        print(f"GPU configuration error: {e}")
else:
    print("No GPU found, using CPU")

# Imports
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
```

```
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from sklearn.metrics import mean_squared_error
from google.colab import drive
import optuna
from sklearn.inspection import permutation_importance
from scipy.stats import norm

# Mount Google Drive
drive.mount('/content/drive')

# ==============================================================================
# ========= Step 1: Data Loading and Preparation =========
# ==============================================================================

print("\n--- Starting Step 1: Data Loading and Preparation ---")

# --- Configuration ---
etf_path = "/content/drive/MyDrive/Data Science Project /finale /all_etf_data.csv"
sentiment_path = "/content/drive/MyDrive/Data Science Project /finale /Sentimental_score_Final.csv"
macro_path = "/content/drive/MyDrive/Data Science Project /finale /macroeconomic_indicators.csv"
sectors = ['Financials', 'Real Estate', 'Technology', 'Energy', 'Healthcare']
print(f"ETF data path: {etf_path}")
print(f"Sentiment data path: {sentiment_path}")
print(f"Macro data path: {macro_path}")
print(f"Target sectors: {sectors}")

# --- Load Data ---
print("\nLoading data...")
try:
    etf_prices = pd.read_csv(etf_path)
    sentiment_scores = pd.read_csv(sentiment_path)
    macro_data = pd.read_csv(macro_path)
    print(f"ETF Prices Shape: {etf_prices.shape}")
    print(f"Sentiment Scores Shape: {sentiment_scores.shape}")
    print(f"Macro Data Shape: {macro_data.shape}")
except Exception as e:
    print(f"FATAL ERROR loading data: {e}")
    exit()

# --- Data Cleaning ---
print("\nCleaning data...")
etf_prices['Ticker'] = etf_prices['Ticker'].astype(str).str.strip()
etf_prices['Sector'] = etf_prices['Sector'].fillna('Unknown').astype(str).str.strip().replace('', 'Unknown')
```

```
sentiment_scores['sector'] = sentiment_scores['sector'].fillna('Unknown').astype(str).str.strip().replace('',
'Unknown')

# Validate sectors
print("ETF Sectors:", etf_prices['Sector'].unique())
print("Sentiment Sectors:", sentiment_scores['sector'].unique())
if not all(s in etf_prices['Sector'].unique() for s in sectors):
    print(f"Warning: Some target sectors {sectors} not found in etf_prices['Sector']")

# Standardize dates
print("Standardizing date formats...")
etf_prices['Date'] = pd.to_datetime(etf_prices['Date'], format='%d-%m-%Y')
sentiment_scores['date'] = pd.to_datetime(sentiment_scores['date'], format='%d-%m-%Y')
macro_data['Date'] = pd.to_datetime(macro_data['Date'],format='%d-%m-%Y')
etf_prices.dropna(subset=['Date', 'Close'], inplace=True)
sentiment_scores.dropna(subset=['date', 'sentiment_impact_score'], inplace=True)
macro_data.dropna(subset=['Date'], inplace=True)
print("Dates standardized and NaTs dropped.")

# --- Sentiment Aggregation ---
print("\nAggregating sentiment scores by sector and date...")
daily_sector_sentiment = sentiment_scores.groupby(['date',
'sector'])['sentiment_impact_score'].mean().reset_index()
print(f"Aggregated Sentiment Shape: {daily_sector_sentiment.shape}")

# --- Prepare ETF Data ---
print("\nCalculating returns and pivoting...")
etf_prices = etf_prices.sort_values(['Ticker', 'Date'])
etf_prices['return'] = etf_prices.groupby('Ticker')['Close'].pct_change()
etf_prices['volatility'] = etf_prices.groupby('Ticker')['return'].rolling(window=20).std().reset_index(level=0,
drop=True)
etf_sectors = etf_prices[['Ticker', 'Sector']].drop_duplicates().set_index('Ticker')
print(f"Created ticker-to-sector map for {len(etf_sectors)} unique tickers.")

# Pivot ETF data
etf_pivot_df = etf_prices.pivot_table(index='Date', columns='Ticker', values=['return', 'volatility'])
etf_pivot_df.columns = [f"{col[1]}_{col[0]}" for col in etf_pivot_df.columns]
etf_return_tickers = [col for col in etf_pivot_df.columns if col.endswith('_return')]
etf_pivot_df.dropna(subset=etf_return_tickers, how='all', inplace=True)
print(f"Pivoted ETF data shape: {etf_pivot_df.shape}")
if etf_pivot_df.empty:
    print("FATAL ERROR: ETF Pivot table is empty.")
    exit()

# --- Merge Data ---
print("\nMerging dataframes...")
```

```python
combined_df = etf_pivot_df.copy()
etf_tickers_in_pivot = etf_sectors.index.tolist()
daily_sector_sentiment.set_index('date', inplace=True)
print("Merging sentiment...")
for ticker in etf_tickers_in_pivot:
    try:
        sector = etf_sectors.loc[ticker, 'Sector']
        if pd.isna(sector) or sector == '':
            sector = 'Unknown'
        relevant_sentiment = daily_sector_sentiment[daily_sector_sentiment['sector'] ==
sector]['sentiment_impact_score']
        sentiment_col_name = f"{ticker}_sentiment"
        if relevant_sentiment.empty:
            sentiment_series = pd.Series(index=combined_df.index, data=np.nan, name=sentiment_col_name)
        else:
            sentiment_series = relevant_sentiment
            sentiment_series.name = sentiment_col_name
        sentiment_series.index.name = 'Date'
        combined_df = pd.merge(combined_df, sentiment_series, on='Date', how='left')
    except Exception as e:
        print(f"Warn: Merge sentiment error for {ticker}: {e}")
        combined_df[f"{ticker}_sentiment"] = np.nan

# Merge macro factors
print("Merging macro factors...")
macro_data.set_index('Date', inplace=True)
combined_df = pd.merge(combined_df, macro_data, on='Date', how='left')
print(f"Shape after macro merge: {combined_df.shape}")

# --- Handle Missing Data ---
print("\nHandling missing data...")
combined_df.sort_index(inplace=True)
sentiment_cols = [col for col in combined_df.columns if col.endswith('_sentiment')]
volatility_cols = [col for col in combined_df.columns if col.endswith('_volatility')]
macro_cols = ['Federal Funds Rate', 'Consumer Price Index', 'Unemployment Rate', 'Gross Domestic Product',
              '10-Year minus 2-Year Treasury Yield Spread', 'CBOE Volatility Index', 'WTI Crude Oil Prices',
              '10-Year Treasury Yield']
cols_to_fill = sentiment_cols + volatility_cols + macro_cols
if cols_to_fill:
    print(f"Attempting bfill/ffill on {len(cols_to_fill)} columns.")
    combined_df[cols_to_fill] = combined_df[cols_to_fill].fillna(method='bfill').fillna(method='ffill')
combined_df[sentiment_cols] = combined_df[sentiment_cols].fillna(0)
combined_df[macro_cols] = combined_df[macro_cols].fillna(combined_df[macro_cols].mean())
initial_rows = len(combined_df)
combined_df.dropna(subset=etf_return_tickers, how='any', inplace=True)
print(f"Dropped {initial_rows - len(combined_df)} rows based on return NaNs.")
```

```
print(f"Final Combined DataFrame Shape: {combined_df.shape}")
if combined_df.empty:
    print("FATAL ERROR: Final combined_df is empty.")
    exit()

# ==============================================================================
# ========= Step 2: Feature Engineering, Scaling & Sequencing =========
# ==============================================================================

print("\n--- Starting Step 2: Feature Engineering, Scaling & Sequencing ---")

# --- Define Feature Sets ---
print("\nDefining feature sets...")
target_columns = etf_return_tickers
base_tickers = [col.replace('_return', '') for col in target_columns]
model1_features = target_columns + [f"{t}_sentiment" for t in base_tickers if f"{t}_sentiment" in
combined_df.columns]
model1_features = [f for f in model1_features if f in combined_df.columns]
print(f"Model 1 Features (Returns + Sentiment): {len(model1_features)}")
model2_features = model1_features + [f"{t}_volatility" for t in base_tickers if f"{t}_volatility" in
combined_df.columns]
model2_features = [f for f in model2_features if f in combined_df.columns]
print(f"Model 2 Features (M1 + Volatility): {len(model2_features)}")
model3_features = model1_features + macro_cols
model3_features = [f for f in model3_features if f in combined_df.columns]
print(f"Model 3 Features (M1 + Macro): {len(model3_features)}")

# --- Data Scaling ---
print("\nScaling features...")
scaler_model1 = MinMaxScaler(feature_range=(0, 1))
scaler_model2 = MinMaxScaler(feature_range=(0, 1))
scaler_model3 = MinMaxScaler(feature_range=(0, 1))
scaler_target = MinMaxScaler(feature_range=(0, 1))
scaled_data_model1 = scaler_model1.fit_transform(combined_df[model1_features])
scaled_data_model2 = scaler_model2.fit_transform(combined_df[model2_features])
scaled_data_model3 = scaler_model3.fit_transform(combined_df[model3_features])
scaled_target_data = scaler_target.fit_transform(combined_df[target_columns])
print(f"Scaled shapes: Model1={scaled_data_model1.shape}, Model2={scaled_data_model2.shape},
Model3={scaled_data_model3.shape}, Target={scaled_target_data.shape}")

# --- Sequence Creation ---
def create_sequences(input_data, target_data, sequence_length):
    X, y = [], []
    if len(input_data) <= sequence_length:
        return np.array(X), np.array(y)
    for i in range(sequence_length, len(input_data)):
```

```
        X.append(input_data[i-sequence_length:i])
        y.append(target_data[i])
    return np.array(X), np.array(y)

SEQUENCE_LENGTH = 20
if SEQUENCE_LENGTH >= len(combined_df):
    SEQUENCE_LENGTH = max(1, len(combined_df) // 4)
print(f"Using sequence length: {SEQUENCE_LENGTH}")

print("Creating sequences...")
X_model1, y_model1 = create_sequences(scaled_data_model1, scaled_target_data, SEQUENCE_LENGTH)
X_model2, y_model2 = create_sequences(scaled_data_model2, scaled_target_data, SEQUENCE_LENGTH)
X_model3, y_model3 = create_sequences(scaled_data_model3, scaled_target_data, SEQUENCE_LENGTH)
print(f"Model 1: X={X_model1.shape}, y={y_model1.shape}")
print(f"Model 2: X={X_model2.shape}, y={y_model2.shape}")
print(f"Model 3: X={X_model3.shape}, y={y_model3.shape}")
if X_model1.shape[0] == 0 or X_model2.shape[0] == 0 or X_model3.shape[0] == 0:
    print("FATAL ERROR: Zero sequences created.")
    exit()

# --- Train/Test Split ---
print("Splitting data...")
test_split_ratio = 0.2
n_samples = X_model1.shape[0]
n_test = int(n_samples * test_split_ratio)
n_train = n_samples - n_test
X_train1, X_test1 = X_model1[:n_train], X_model1[n_train:]
y_train1, y_test1 = y_model1[:n_train], y_model1[n_train:]
X_train2, X_test2 = X_model2[:n_train], X_model2[n_train:]
y_train2, y_test2 = y_model2[:n_train], y_model2[n_train:]
X_train3, X_test3 = X_model3[:n_train], X_model3[n_train:]
y_train3, y_test3 = y_model3[:n_train], y_model3[n_train:]
print(f"Split: Train={n_train}, Test={n_test}")

# ===========================================================================
# ========= Step 3: Build and Train LSTM Models with Optuna =========
# ===========================================================================

print("\n--- Starting Step 3: Build and Train LSTM Models with Hyperparameter Tuning ---")

# --- Define LSTM Model with Variable Hyperparameters ---
def build_lstm_model(input_shape, output_units, units1, units2, dropout_rate, model_name):
    model = Sequential(name=model_name)
    model.add(LSTM(units=units1, return_sequences=True, input_shape=input_shape))
    model.add(Dropout(dropout_rate))
    model.add(LSTM(units=units2))
```

```
    model.add(Dropout(dropout_rate))
    model.add(Dense(units=32, activation='relu'))
    model.add(Dense(units=output_units, activation='linear'))
    return model

# --- Optuna Objective Function ---
def objective(trial, X_train, y_train, X_val, y_val, input_shape, output_units, model_name):
    units1 = trial.suggest_categorical('units1', [64, 128, 256])
    units2 = trial.suggest_categorical('units2', [32, 64, 128])
    dropout_rate = trial.suggest_float('dropout_rate', 0.2, 0.5)
    learning_rate = trial.suggest_float('learning_rate', 1e-4, 1e-2, log=True)
    batch_size = trial.suggest_categorical('batch_size', [16, 32, 64])

    model = build_lstm_model(input_shape, output_units, units1, units2, dropout_rate, model_name)
    model.compile(optimizer=Adam(learning_rate=learning_rate), loss='mse')

    early_stopping = EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True)
    history = model.fit(
        X_train.astype(np.float32), y_train.astype(np.float32),
        epochs=150,
        batch_size=batch_size,
        validation_data=(X_val.astype(np.float32), y_val.astype(np.float32)),
        callbacks=[early_stopping],
        verbose=0
    )
    return min(history.history['val_loss'])

# --- Hyperparameter Optimization ---
def optimize_lstm_hyperparameters(X_train, y_train, X_val, y_val, input_shape, output_units, model_name,
n_trials=10):
    study = optuna.create_study(direction='minimize')
    objective_fn = lambda trial: objective(trial, X_train, y_train, X_val, y_val, input_shape, output_units,
model_name)
    study.optimize(objective_fn, n_trials=n_trials)
    return study.best_params

# --- Optimize and Train Model 1 ---
print("\nOptimizing hyperparameters for Model 1...")
input_shape1 = (X_train1.shape[1], X_train1.shape[2])
output_units = y_train1.shape[1]
best_params1 = optimize_lstm_hyperparameters(
    X_train1, y_train1, X_test1, y_test1, input_shape1, output_units, "Model1_Sentiment"
)
print("Best hyperparameters for Model 1:", best_params1)

print("\nBuilding Model 1 with best hyperparameters...")
```

```
if np.any(np.isnan(X_train1)) or np.any(np.isinf(X_train1)):
    print("FATAL: NaN/Inf in X_train1!")
    exit()
if np.any(np.isnan(y_train1)) or np.any(np.isinf(y_train1)):
    print("FATAL: NaN/Inf in y_train1!")
    exit()
model1 = build_lstm_model(
    input_shape1,
    output_units,
    units1=best_params1['units1'],
    units2=best_params1['units2'],
    dropout_rate=best_params1['dropout_rate'],
    model_name="Model1_Sentiment"
)
model1.compile(optimizer=Adam(learning_rate=best_params1['learning_rate']), loss='mse')
model1.summary()

print("\nTraining Model 1...")
early_stopping1 = EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True)
history1 = model1.fit(
    X_train1.astype(np.float32), y_train1.astype(np.float32),
    epochs=150,
    batch_size=best_params1['batch_size'],
    validation_data=(X_test1.astype(np.float32), y_test1.astype(np.float32)),
    callbacks=[early_stopping1],
    verbose=1
)

# --- Optimize and Train Model 2 ---
print("\nOptimizing hyperparameters for Model 2...")
input_shape2 = (X_train2.shape[1], X_train2.shape[2])
best_params2 = optimize_lstm_hyperparameters(
    X_train2, y_train2, X_test2, y_test2, input_shape2, output_units, "Model2_Sentiment_Volatility"
)
print("Best hyperparameters for Model 2:", best_params2)

print("\nBuilding Model 2 with best hyperparameters...")
if np.any(np.isnan(X_train2)) or np.any(np.isinf(X_train2)):
    print("FATAL: NaN/Inf in X_train2!")
    exit()
if np.any(np.isnan(y_train2)) or np.any(np.isinf(y_train2)):
    print("FATAL: NaN/Inf in y_train2!")
    exit()
model2 = build_lstm_model(
    input_shape2,
    output_units,
```

```
    units1=best_params2['units1'],
    units2=best_params2['units2'],
    dropout_rate=best_params2['dropout_rate'],
    model_name="Model2_Sentiment_Volatility"
)
model2.compile(optimizer=Adam(learning_rate=best_params2['learning_rate']), loss='mse')
model2.summary()

print("\nTraining Model 2...")
early_stopping2 = EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True)
history2 = model2.fit(
    X_train2.astype(np.float32), y_train2.astype(np.float32),
    epochs=150,
    batch_size=best_params2['batch_size'],
    validation_data=(X_test2.astype(np.float32), y_test2.astype(np.float32)),
    callbacks=[early_stopping2],
    verbose=1
)

# --- Optimize and Train Model 3 ---
print("\nOptimizing hyperparameters for Model 3...")
input_shape3 = (X_train3.shape[1], X_train3.shape[2])
best_params3 = optimize_lstm_hyperparameters(
    X_train3, y_train3, X_test3, y_test3, input_shape3, output_units, "Model3_Sentiment_Macro"
)
print("Best hyperparameters for Model 3:", best_params3)

print("\nBuilding Model 3 with best hyperparameters...")
if np.any(np.isnan(X_train3)) or np.any(np.isinf(X_train3)):
    print("FATAL: NaN/Inf in X_train3!")
    exit()
if np.any(np.isnan(y_train3)) or np.any(np.isinf(y_train3)):
    print("FATAL: NaN/Inf in y_train3!")
    exit()
model3 = build_lstm_model(
    input_shape3,
    output_units,
    units1=best_params3['units1'],
    units2=best_params3['units2'],
    dropout_rate=best_params3['dropout_rate'],
    model_name="Model3_Sentiment_Macro"
)
model3.compile(optimizer=Adam(learning_rate=best_params3['learning_rate']), loss='mse')
model3.summary()

print("\nTraining Model 3...")
```

```
early_stopping3 = EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True)
history3 = model3.fit(
    X_train3.astype(np.float32), y_train3.astype(np.float32),
    epochs=150,
    batch_size=best_params3['batch_size'],
    validation_data=(X_test3.astype(np.float32), y_test3.astype(np.float32)),
    callbacks=[early_stopping3],
    verbose=1
)

# --- Plot Training History ---
def plot_loss(history, title):
    plt.figure(figsize=(10, 6))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title(title)
    plt.xlabel('Epoch')
    plt.ylabel('Loss (MSE)')
    plt.legend()
    plt.grid(True)
    plt.show(f'{title.lower().replace(" ", "_")}.png')
    plt.close()

print("\nPlotting training history...")
plot_loss(history1, 'Model 1 Training & Validation Loss')
plot_loss(history2, 'Model 2 Training & Validation Loss')
plot_loss(history3, 'Model 3 Training & Validation Loss')

# ==============================================================================
# ========= Step 4: Prediction, Evaluation, and Portfolio Optimization =========
# ==============================================================================

print("\n--- Starting Step 4: Prediction, Evaluation, Optimization ---")

# --- Predictions ---
print("\nMaking predictions...")
y_pred_scaled1 = model1.predict(X_test1)
y_pred_scaled2 = model2.predict(X_test2)
y_pred_scaled3 = model3.predict(X_test3)
y_pred1 = scaler_target.inverse_transform(y_pred_scaled1)
y_pred2 = scaler_target.inverse_transform(y_pred_scaled2)
y_pred3 = scaler_target.inverse_transform(y_pred_scaled3)
y_test_actual = scaler_target.inverse_transform(y_test1)
print(f"Prediction shapes: Pred1={y_pred1.shape}, Pred2={y_pred2.shape}, Pred3={y_pred3.shape},
Actual={y_test_actual.shape}")
```

```
# --- Evaluate Models ---
mse1 = mean_squared_error(y_test_actual, y_pred1)
mse2 = mean_squared_error(y_test_actual, y_pred2)
mse3 = mean_squared_error(y_test_actual, y_pred3)
print(f"Model 1 Test MSE: {mse1:.8f}")
print(f"Model 2 Test MSE: {mse2:.8f}")
print(f"Model 3 Test MSE: {mse3:.8f}")

# --- Portfolio Optimization ---
print("\nPreparing portfolio optimization...")
expected_returns1 = y_pred1[-1]
expected_returns2 = y_pred2[-1]
expected_returns3 = y_pred3[-1]
train_df_portion = combined_df.iloc[:n_train + SEQUENCE_LENGTH]
train_returns = train_df_portion[target_columns]
ann_factor = 252
cov_matrix_hist = train_returns.cov() * ann_factor
try:
    np.linalg.cholesky(cov_matrix_hist)
except np.linalg.LinAlgError:
    from sklearn.covariance import LedoitWolf
    cov_matrix_hist = pd.DataFrame(LedoitWolf().fit(train_returns.dropna()).covariance_ * ann_factor,
                                   index=target_columns, columns=target_columns)
    print("Applied Ledoit-Wolf shrinkage.")

num_assets = len(target_columns)
def maximize_sharpe_ratio(expected_returns, cov_matrix, risk_free_rate=0.0):
    def neg_sharpe_ratio(weights):
        p_ret = np.sum(expected_returns * weights)
        p_vol = np.sqrt(np.dot(weights.T, np.dot(cov_matrix.values, weights)))
        return -(p_ret - risk_free_rate) / p_vol if p_vol != 0 else -np.inf
    constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1})
    bounds = tuple((0, 1) for _ in range(num_assets))
    initial_weights = np.array([1./num_assets] * num_assets)
    result = minimize(neg_sharpe_ratio, initial_weights, method='SLSQP', bounds=bounds,
constraints=constraints)
    return result.x / np.sum(result.x) if result.success else initial_weights

optimal_weights1 = maximize_sharpe_ratio(expected_returns1, cov_matrix_hist)
optimal_weights2 = maximize_sharpe_ratio(expected_returns2, cov_matrix_hist)
optimal_weights3 = maximize_sharpe_ratio(expected_returns3, cov_matrix_hist)

# --- Display Weights ---
print("\n--- Optimal Portfolio Weights (Tickers) ---")
results_df = pd.DataFrame(index=base_tickers)
results_df['Model1_Weights'] = optimal_weights1
```

```
results_df['Model2_Weights'] = optimal_weights2
results_df['Model3_Weights'] = optimal_weights3
print("\nModel 1 Weights (Tickers > 0.1%):")
print(results_df[results_df['Model1_Weights'] >
0.001]['Model1_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
print("\nModel 2 Weights (Tickers > 0.1%):")
print(results_df[results_df['Model2_Weights'] >
0.001]['Model2_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
print("\nModel 3 Weights (Tickers > 0.1%):")
print(results_df[results_df['Model3_Weights'] >
0.001]['Model3_Weights'].sort_values(ascending=False).map('{:.2%}'.format))

# --- Sector Aggregation ---
print("\n--- Aggregating Portfolio Weights by Sector ---")
sector_weights_df = results_df.merge(etf_sectors, left_index=True, right_index=True, how='left')
sector_summary = sector_weights_df.groupby('Sector').sum()
print("\n--- Model 1 Sector Weights ---")
print(sector_summary[sector_summary['Model1_Weights'] >
0.001]['Model1_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
print("\n--- Model 2 Sector Weights ---")
print(sector_summary[sector_summary['Model2_Weights'] >
0.001]['Model2_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
print("\n--- Model 3 Sector Weights ---")
print(sector_summary[sector_summary['Model3_Weights'] >
0.001]['Model3_Weights'].sort_values(ascending=False).map('{:.2%}'.format))

# --- Macro Factor Importance Analysis for Model 3 ---
print("\n--- Analyzing Macro Factor Importance for Model 3 ---")
# Use correlation analysis to estimate impact
macro_scaled = scaler_model3.transform(combined_df[model3_features])[:, -len(macro_cols):]
macro_df = pd.DataFrame(macro_scaled, columns=macro_cols, index=combined_df.index)
portfolio_ret_m3 = y_test_actual @ optimal_weights3
macro_test = macro_df.iloc[-len(portfolio_ret_m3):]
correlations = macro_test.corrwith(pd.Series(portfolio_ret_m3))
print("\nCorrelation of Macro Factors with Model 3 Portfolio Returns:")
print(correlations.sort_values(ascending=False))
most_impactful_macro = correlations.idxmax()
max_correlation = correlations.max()
print(f"\nMacro Factor with Highest Impact: {most_impactful_macro} (Correlation: {max_correlation:.4f})")

# --- Portfolio Performance Metrics ---
print("\n--- Portfolio Performance Comparison ---")
portfolio_ret_m1 = y_test_actual @ optimal_weights1
portfolio_ret_m2 = y_test_actual @ optimal_weights2
portfolio_ret_eqw = y_test_actual @ (np.ones(num_assets) / num_assets)
```

```
def calculate_portfolio_metrics(returns, risk_free_rate=0.0, confidence_level=0.05):
    total_return = np.prod(1 + returns) - 1
    n_periods = len(returns)
    annualized_return = (1 + total_return) ** (ann_factor / n_periods) - 1 if n_periods > 0 else np.nan
    annualized_volatility = np.std(returns) * np.sqrt(ann_factor) if n_periods > 1 else np.nan
    sharpe_ratio = (annualized_return - risk_free_rate) / annualized_volatility if annualized_volatility != 0 else
np.nan

    # VaR and ES
    returns_sorted = np.sort(returns)
    var_index = int(len(returns_sorted) * confidence_level)
    var = returns_sorted[var_index]
    es = returns_sorted[:var_index].mean() if var_index > 0 else np.nan

    # Alpha (relative to equal-weight benchmark)
    benchmark_returns = portfolio_ret_eqw[:len(returns)]
    beta = np.cov(returns, benchmark_returns)[0, 1] / np.var(benchmark_returns) if np.var(benchmark_returns) !=
0 else 0
    alpha = annualized_return - risk_free_rate - beta * (np.prod(1 + benchmark_returns) ** (ann_factor /
n_periods) - 1 - risk_free_rate)

    return {
        'Annualized Return': annualized_return,
        'Annualized Volatility': annualized_volatility,
        'Sharpe Ratio': sharpe_ratio,
        'VaR (5%)': var,
        'Expected Shortfall (5%)': es,
        'Alpha': alpha
    }

print("\nPerformance Metrics (Annualized):")
metrics_m1 = calculate_portfolio_metrics(portfolio_ret_m1)
metrics_m2 = calculate_portfolio_metrics(portfolio_ret_m2)
metrics_m3 = calculate_portfolio_metrics(portfolio_ret_m3)
metrics_eqw = calculate_portfolio_metrics(portfolio_ret_eqw)

print("\nModel 1 Metrics:")
for key, value in metrics_m1.items():
    print(f"{key}: {value:.2%}" if 'Ratio' not in key else f"{key}: {value:.2f}")
print("\nModel 2 Metrics:")
for key, value in metrics_m2.items():
    print(f"{key}: {value:.2%}" if 'Ratio' not in key else f"{key}: {value:.2f}")
print("\nModel 3 Metrics:")
for key, value in metrics_m3.items():
    print(f"{key}: {value:.2%}" if 'Ratio' not in key else f"{key}: {value:.2f}")
print("\nBenchmark (EQW) Metrics:")
```

```
for key, value in metrics_eqw.items():
    print(f"{key}: {value:.2%}" if 'Ratio' not in key else f"{key}: {value:.2f}")

# --- Visualization ---
print("\nGenerating comparison graphs...")

# Graph 1: Sector Weights Comparison
plt.figure(figsize=(12, 6))
bar_width = 0.2
index = np.arange(len(sectors))
eqw_weights = [0.2] * len(sectors)
model1_sector_weights = [sector_summary.loc[s, 'Model1_Weights'] if s in sector_summary.index else 0 for s
in sectors]
model2_sector_weights = [sector_summary.loc[s, 'Model2_Weights'] if s in sector_summary.index else 0 for s
in sectors]
model3_sector_weights = [sector_summary.loc[s, 'Model3_Weights'] if s in sector_summary.index else 0 for s
in sectors]

plt.bar(index, model1_sector_weights, bar_width, label='Model 1 (Sentiment)', color='blue')
plt.bar(index + bar_width, model2_sector_weights, bar_width, label='Model 2 (Sentiment+Volatility)',
color='green')
plt.bar(index + 2 * bar_width, model3_sector_weights, bar_width, label='Model 3 (Sentiment+Macro)',
color='red')
plt.bar(index + 3 * bar_width, eqw_weights, bar_width, label='Benchmark (Equal Weight)', color='gray')

plt.xlabel('Sectors')
plt.ylabel('Portfolio Weight')
plt.title('Sector Weights Comparison')
plt.xticks(index + 1.5 * bar_width, sectors, rotation=45)
plt.legend()
plt.tight_layout()
plt.show('sector_weights_comparison.png')
plt.close()

# Graph 2: Performance Metrics Comparison
plt.figure(figsize=(12, 6))
metrics = ['Annualized Return', 'Annualized Volatility', 'Sharpe Ratio', 'VaR (5%)', 'Expected Shortfall (5%)',
'Alpha']
model1_metrics = [metrics_m1[m] for m in metrics]
model2_metrics = [metrics_m2[m] for m in metrics]
model3_metrics = [metrics_m3[m] for m in metrics]
eqw_metrics = [metrics_eqw[m] for m in metrics]

index = np.arange(len(metrics))
plt.bar(index, model1_metrics, bar_width, label='Model 1 (Sentiment)', color='blue')
plt.bar(index + bar_width, model2_metrics, bar_width, label='Model 2 (Sentiment+Volatility)', color='green')
```

```
plt.bar(index + 2 * bar_width, model3_metrics, bar_width, label='Model 3 (Sentiment+Macro)', color='red')
plt.bar(index + 3 * bar_width, eqw_metrics, bar_width, label='Benchmark (Equal Weight)', color='gray')

plt.xlabel('Metrics')
plt.ylabel('Value')
plt.title('Performance Metrics Comparison')
plt.xticks(index + 1.5 * bar_width, metrics, rotation=45)
plt.legend()
plt.tight_layout()
plt.show('performance_metrics_comparison.png')
plt.close()

# Graph 3: Cumulative Returns
print("\nPlotting Cumulative Returns...")
plt.figure(figsize=(12, 7))
plt.plot(np.cumprod(1 + portfolio_ret_m1) - 1, label='Model 1 (Sentiment)')
plt.plot(np.cumprod(1 + portfolio_ret_m2) - 1, label='Model 2 (Sentiment+Volatility)')
plt.plot(np.cumprod(1 + portfolio_ret_m3) - 1, label='Model 3 (Sentiment+Macro)')
plt.plot(np.cumprod(1 + portfolio_ret_eqw) - 1, label='Benchmark (Equal Weight)', linestyle='--')
plt.title('Portfolio Cumulative Returns (Test Period)')
plt.xlabel('Time Steps')
plt.ylabel('Cumulative Return')
plt.legend()
plt.grid(True)
plt.show('cumulative_returns.png')
plt.close()

# Save models
model1.save('model1_lstm_sentiment_optuna.h5')
model2.save('model2_lstm_sentiment_volatility_optuna.h5')
model3.save('model3_lstm_sentiment_macro_optuna.h5')

print("\n--- Step 4 and Comparison Graphs Completed ---")
```

Transformer Model

```
# Install Optuna
!pip install optuna

# Configure GPU memory growth before importing TensorFlow
import tensorflow as tf

gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:
        tf.config.experimental.set_memory_growth(gpus[0], True)
```

```
        print("Using GPU:", gpus[0])
    except RuntimeError as e:
        print(f"GPU configuration error: {e}")
else:
    print("No GPU found, using CPU")

# Imports
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Dropout, LayerNormalization,
MultiHeadAttention, Input, Add, Flatten
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from sklearn.metrics import mean_squared_error
from google.colab import drive
import optuna
from sklearn.inspection import permutation_importance
from scipy.stats import norm

# Mount Google Drive
drive.mount('/content/drive')

#
===========================================================================
===
# ========= Step 1: Data Loading and Preparation =========
#
===========================================================================
===

print("\n--- Starting Step 1: Data Loading and Preparation ---")

# --- Configuration ---
etf_path = "/content/drive/MyDrive/Data Science Project /finale
/all_etf_data.csv"
sentiment_path = "/content/drive/MyDrive/Data Science Project /finale
/Sentimental_score_Final.csv"
macro_path = "/content/drive/MyDrive/Data Science Project /finale
/macroeconomic_indicators.csv"
sectors = ['Financials', 'Real Estate', 'Technology', 'Energy',
'Healthcare']
print(f"ETF data path: {etf_path}")
```

```
print(f"Sentiment data path: {sentiment_path}")
print(f"Macro data path: {macro_path}")
print(f"Target sectors: {sectors}")

# --- Load Data ---
print("\nLoading data...")
try:
    etf_prices = pd.read_csv(etf_path)
    sentiment_scores = pd.read_csv(sentiment_path)
    macro_data = pd.read_csv(macro_path)
    print(f"ETF Prices Shape: {etf_prices.shape}")
    print(f"Sentiment Scores Shape: {sentiment_scores.shape}")
    print(f"Macro Data Shape: {macro_data.shape}")
except Exception as e:
    print(f"FATAL ERROR loading data: {e}")
    exit()

# --- Data Cleaning ---
print("\nCleaning data...")
etf_prices['Ticker'] = etf_prices['Ticker'].astype(str).str.strip()
etf_prices['Sector'] =
etf_prices['Sector'].fillna('Unknown').astype(str).str.strip().replace('',
'Unknown')
sentiment_scores['sector'] =
sentiment_scores['sector'].fillna('Unknown').astype(str).str.strip().replac
e('', 'Unknown')

# Validate sectors
print("ETF Sectors:", etf_prices['Sector'].unique())
print("Sentiment Sectors:", sentiment_scores['sector'].unique())
if not all(s in etf_prices['Sector'].unique() for s in sectors):
    print(f"Warning: Some target sectors {sectors} not found in
etf_prices['Sector']")

# Standardize dates
print("Standardizing date formats...")
etf_prices['Date'] = pd.to_datetime(etf_prices['Date'], format='%d-%m-%Y')
sentiment_scores['date'] = pd.to_datetime(sentiment_scores['date'],
format='%d-%m-%Y')
macro_data['Date'] = pd.to_datetime(macro_data['Date'], format='%d-%m-
%Y')  # Fixed format for DD-MM-YYYY
etf_prices.dropna(subset=['Date', 'Close'], inplace=True)
sentiment_scores.dropna(subset=['date', 'sentiment_impact_score'],
inplace=True)
macro_data.dropna(subset=['Date'], inplace=True)
print("Dates standardized and NaTs dropped.")
```

```
# --- Sentiment Aggregation ---
print("\nAggregating sentiment scores by sector and date...")
daily_sector_sentiment = sentiment_scores.groupby(['date',
'sector'])['sentiment_impact_score'].mean().reset_index()
print(f"Aggregated Sentiment Shape: {daily_sector_sentiment.shape}")

# --- Prepare ETF Data ---
print("\nCalculating returns and pivoting...")
etf_prices = etf_prices.sort_values(['Ticker', 'Date'])
etf_prices['return'] = etf_prices.groupby('Ticker')['Close'].pct_change()
etf_prices['volatility'] =
etf_prices.groupby('Ticker')['return'].rolling(window=20).std().reset_index
(level=0, drop=True)
etf_sectors = etf_prices[['Ticker',
'Sector']].drop_duplicates().set_index('Ticker')
print(f"Created ticker-to-sector map for {len(etf_sectors)} unique
tickers.")

# Pivot ETF data
etf_pivot_df = etf_prices.pivot_table(index='Date', columns='Ticker',
values=['return', 'volatility'])
etf_pivot_df.columns = [f"{col[1]}_{col[0]}" for col in
etf_pivot_df.columns]
etf_return_tickers = [col for col in etf_pivot_df.columns if
col.endswith('_return')]
etf_pivot_df.dropna(subset=etf_return_tickers, how='all', inplace=True)
print(f"Pivoted ETF data shape: {etf_pivot_df.shape}")
if etf_pivot_df.empty:
    print("FATAL ERROR: ETF Pivot table is empty.")
    exit()

# --- Merge Data ---
print("\nMerging dataframes...")
combined_df = etf_pivot_df.copy()
etf_tickers_in_pivot = etf_sectors.index.tolist()
daily_sector_sentiment.set_index('date', inplace=True)
print("Merging sentiment...")
for ticker in etf_tickers_in_pivot:
    try:
        sector = etf_sectors.loc[ticker, 'Sector']
        if pd.isna(sector) or sector == '':
            sector = 'Unknown'
        relevant_sentiment =
daily_sector_sentiment[daily_sector_sentiment['sector'] ==
sector]['sentiment_impact_score']
```

```
        sentiment_col_name = f"{ticker}_sentiment"
        if relevant_sentiment.empty:
            sentiment_series = pd.Series(index=combined_df.index,
data=np.nan, name=sentiment_col_name)
        else:
            sentiment_series = relevant_sentiment
            sentiment_series.name = sentiment_col_name
        sentiment_series.index.name = 'Date'
        combined_df = pd.merge(combined_df, sentiment_series, on='Date',
how='left')
    except Exception as e:
        print(f"Warn: Merge sentiment error for {ticker}: {e}")
        combined_df[f"{ticker}_sentiment"] = np.nan

# Merge macro factors
print("Merging macro factors...")
macro_data.set_index('Date', inplace=True)
combined_df = pd.merge(combined_df, macro_data, on='Date', how='left')
print(f"Shape after macro merge: {combined_df.shape}")

# --- Handle Missing Data ---
print("\nHandling missing data...")
combined_df.sort_index(inplace=True)
sentiment_cols = [col for col in combined_df.columns if
col.endswith('_sentiment')]
volatility_cols = [col for col in combined_df.columns if
col.endswith('_volatility')]
macro_cols = ['Federal Funds Rate', 'Consumer Price Index', 'Unemployment
Rate', 'Gross Domestic Product',
              '10-Year minus 2-Year Treasury Yield Spread', 'CBOE
Volatility Index', 'WTI Crude Oil Prices',
              '10-Year Treasury Yield']
cols_to_fill = sentiment_cols + volatility_cols + macro_cols
if cols_to_fill:
    print(f"Attempting bfill/ffill on {len(cols_to_fill)} columns.")
    combined_df[cols_to_fill] =
combined_df[cols_to_fill].fillna(method='bfill').fillna(method='ffill')
combined_df[sentiment_cols] = combined_df[sentiment_cols].fillna(0)
combined_df[macro_cols] =
combined_df[macro_cols].fillna(combined_df[macro_cols].mean())
initial_rows = len(combined_df)
combined_df.dropna(subset=etf_return_tickers, how='any', inplace=True)
print(f"Dropped {initial_rows - len(combined_df)} rows based on return
NaNs.")
print(f"Final Combined DataFrame Shape: {combined_df.shape}")
if combined_df.empty:
```

```
    print("FATAL ERROR: Final combined_df is empty.")
    exit()

#
=========================================================================
===
# ========= Step 2: Feature Engineering, Scaling & Sequencing =========
#
=========================================================================
===

print("\n--- Starting Step 2: Feature Engineering, Scaling & Sequencing ---
")

# --- Define Feature Sets ---
print("\nDefining feature sets...")
target_columns = etf_return_tickers
base_tickers = [col.replace('_return', '') for col in target_columns]
model1_features = target_columns + [f"{t}_sentiment" for t in base_tickers
if f"{t}_sentiment" in combined_df.columns]
model1_features = [f for f in model1_features if f in combined_df.columns]
print(f"Model 1 Features (Returns + Sentiment): {len(model1_features)}")
model2_features = model1_features + [f"{t}_volatility" for t in
base_tickers if f"{t}_volatility" in combined_df.columns]
model2_features = [f for f in model2_features if f in combined_df.columns]
print(f"Model 2 Features (M1 + Volatility): {len(model2_features)}")
model3_features = model1_features + macro_cols
model3_features = [f for f in model3_features if f in combined_df.columns]
print(f"Model 3 Features (M1 + Macro): {len(model3_features)}")

# --- Data Scaling ---
print("\nScaling features...")
scaler_model1 = MinMaxScaler(feature_range=(0, 1))
scaler_model2 = MinMaxScaler(feature_range=(0, 1))
scaler_model3 = MinMaxScaler(feature_range=(0, 1))
scaler_target = MinMaxScaler(feature_range=(0, 1))
scaled_data_model1 =
scaler_model1.fit_transform(combined_df[model1_features])
scaled_data_model2 =
scaler_model2.fit_transform(combined_df[model2_features])
scaled_data_model3 =
scaler_model3.fit_transform(combined_df[model3_features])
scaled_target_data =
scaler_target.fit_transform(combined_df[target_columns])
```

```
print(f"Scaled shapes: Model1={scaled_data_model1.shape},
Model2={scaled_data_model2.shape}, Model3={scaled_data_model3.shape},
Target={scaled_target_data.shape}")

# --- Sequence Creation ---
def create_sequences(input_data, target_data, sequence_length):
    X, y = [], []
    if len(input_data) <= sequence_length:
        return np.array(X), np.array(y)
    for i in range(sequence_length, len(input_data)):
        X.append(input_data[i-sequence_length:i])
        y.append(target_data[i])
    return np.array(X), np.array(y)

SEQUENCE_LENGTH = 20
if SEQUENCE_LENGTH >= len(combined_df):
    SEQUENCE_LENGTH = max(1, len(combined_df) // 4)
print(f"Using sequence length: {SEQUENCE_LENGTH}")

print("Creating sequences...")
X_model1, y_model1 = create_sequences(scaled_data_model1,
scaled_target_data, SEQUENCE_LENGTH)
X_model2, y_model2 = create_sequences(scaled_data_model2,
scaled_target_data, SEQUENCE_LENGTH)
X_model3, y_model3 = create_sequences(scaled_data_model3,
scaled_target_data, SEQUENCE_LENGTH)
print(f"Model 1: X={X_model1.shape}, y={y_model1.shape}")
print(f"Model 2: X={X_model2.shape}, y={y_model2.shape}")
print(f"Model 3: X={X_model3.shape}, y={y_model3.shape}")
if X_model1.shape[0] == 0 or X_model2.shape[0] == 0 or X_model3.shape[0] ==
0:
    print("FATAL ERROR: Zero sequences created.")
    exit()

# --- Train/Test Split ---
print("Splitting data...")
test_split_ratio = 0.2
n_samples = X_model1.shape[0]
n_test = int(n_samples * test_split_ratio)
n_train = n_samples - n_test
X_train1, X_test1 = X_model1[:n_train], X_model1[n_train:]
y_train1, y_test1 = y_model1[:n_train], y_model1[n_train:]
X_train2, X_test2 = X_model2[:n_train], X_model2[n_train:]
y_train2, y_test2 = y_model2[:n_train], y_model2[n_train:]
X_train3, X_test3 = X_model3[:n_train], X_model3[n_train:]
y_train3, y_test3 = y_model3[:n_train], y_model3[n_train:]
```

```
print(f"Split: Train={n_train}, Test={n_test}")

#
================================================================================
===
# ========= Step 3: Build and Train Transformer Models with Optuna
=========
#
================================================================================
===

print("\n--- Starting Step 3: Build and Train Transformer Models with
Hyperparameter Tuning ---")

# --- Define Transformer Model ---
def build_transformer_model(input_shape, output_units, num_heads, ff_dim,
num_layers, dropout_rate, model_name):
    inputs = Input(shape=input_shape)
    x = inputs

    # Transformer Encoder Layers
    for _ in range(num_layers):
        # Multi-Head Self-Attention
        attention_output = MultiHeadAttention(num_heads=num_heads,
key_dim=input_shape[-1])(x, x)
        attention_output = Dropout(dropout_rate)(attention_output)
        x = Add()([x, attention_output])  # Residual connection
        x = LayerNormalization(epsilon=1e-6)(x)

        # Feed-Forward Network
        ffn_output = Dense(ff_dim, activation='relu')(x)
        ffn_output = Dense(input_shape[-1])(ffn_output)
        ffn_output = Dropout(dropout_rate)(ffn_output)
        x = Add()([x, ffn_output])  # Residual connection
        x = LayerNormalization(epsilon=1e-6)(x)

    # Output Layer
    x = Flatten()(x)
    x = Dense(64, activation='relu')(x)
    outputs = Dense(output_units, activation='linear')(x)

    model = Model(inputs=inputs, outputs=outputs, name=model_name)
    return model

# --- Optuna Objective Function ---
```

```python
def objective(trial, X_train, y_train, X_val, y_val, input_shape,
output_units, model_name):
    num_heads = trial.suggest_categorical('num_heads', [2, 4, 8])
    ff_dim = trial.suggest_categorical('ff_dim', [64, 128, 256])
    num_layers = trial.suggest_int('num_layers', 1, 3)
    dropout_rate = trial.suggest_float('dropout_rate', 0.2, 0.5)
    learning_rate = trial.suggest_float('learning_rate', 1e-4, 1e-2,
log=True)
    batch_size = trial.suggest_categorical('batch_size', [16, 32, 64])

    model = build_transformer_model(
        input_shape, output_units, num_heads, ff_dim, num_layers,
dropout_rate, model_name
    )
    model.compile(optimizer=Adam(learning_rate=learning_rate), loss='mse')

    early_stopping = EarlyStopping(monitor='val_loss', patience=15,
restore_best_weights=True)
    history = model.fit(
        X_train.astype(np.float32), y_train.astype(np.float32),
        epochs=150,
        batch_size=batch_size,
        validation_data=(X_val.astype(np.float32),
y_val.astype(np.float32)),
        callbacks=[early_stopping],
        verbose=0
    )
    return min(history.history['val_loss'])

# --- Hyperparameter Optimization ---
def optimize_transformer_hyperparameters(X_train, y_train, X_val, y_val,
input_shape, output_units, model_name, n_trials=10):
    study = optuna.create_study(direction='minimize')
    objective_fn = lambda trial: objective(trial, X_train, y_train, X_val,
y_val, input_shape, output_units, model_name)
    study.optimize(objective_fn, n_trials=n_trials)
    return study.best_params

# --- Optimize and Train Model 1 ---
print("\nOptimizing hyperparameters for Model 1...")
input_shape1 = (X_train1.shape[1], X_train1.shape[2])
output_units = y_train1.shape[1]
best_params1 = optimize_transformer_hyperparameters(
    X_train1, y_train1, X_test1, y_test1, input_shape1, output_units,
"Model1_Sentiment"
)
```

```
print("Best hyperparameters for Model 1:", best_params1)

print("\nBuilding Model 1 with best hyperparameters...")
if np.any(np.isnan(X_train1)) or np.any(np.isinf(X_train1)):
    print("FATAL: NaN/Inf in X_train1!")
    exit()
if np.any(np.isnan(y_train1)) or np.any(np.isinf(y_train1)):
    print("FATAL: NaN/Inf in y_train1!")
    exit()
model1 = build_transformer_model(
    input_shape1,
    output_units,
    num_heads=best_params1['num_heads'],
    ff_dim=best_params1['ff_dim'],
    num_layers=best_params1['num_layers'],
    dropout_rate=best_params1['dropout_rate'],
    model_name="Model1_Sentiment"
)
model1.compile(optimizer=Adam(learning_rate=best_params1['learning_rate']),
loss='mse')
model1.summary()

print("\nTraining Model 1...")
early_stopping1 = EarlyStopping(monitor='val_loss', patience=15,
restore_best_weights=True)
history1 = model1.fit(
    X_train1.astype(np.float32), y_train1.astype(np.float32),
    epochs=150,
    batch_size=best_params1['batch_size'],
    validation_data=(X_test1.astype(np.float32),
y_test1.astype(np.float32)),
    callbacks=[early_stopping1],
    verbose=1
)

# --- Optimize and Train Model 2 ---
print("\nOptimizing hyperparameters for Model 2...")
input_shape2 = (X_train2.shape[1], X_train2.shape[2])
best_params2 = optimize_transformer_hyperparameters(
    X_train2, y_train2, X_test2, y_test2, input_shape2, output_units,
"Model2_Sentiment_Volatility"
)
print("Best hyperparameters for Model 2:", best_params2)

print("\nBuilding Model 2 with best hyperparameters...")
if np.any(np.isnan(X_train2)) or np.any(np.isinf(X_train2)):
```

```
    print("FATAL: NaN/Inf in X_train2!")
    exit()
if np.any(np.isnan(y_train2)) or np.any(np.isinf(y_train2)):
    print("FATAL: NaN/Inf in y_train2!")
    exit()
model2 = build_transformer_model(
    input_shape2,
    output_units,
    num_heads=best_params2['num_heads'],
    ff_dim=best_params2['ff_dim'],
    num_layers=best_params2['num_layers'],
    dropout_rate=best_params2['dropout_rate'],
    model_name="Model2_Sentiment_Volatility"
)
model2.compile(optimizer=Adam(learning_rate=best_params2['learning_rate']),
loss='mse')
model2.summary()

print("\nTraining Model 2...")
early_stopping2 = EarlyStopping(monitor='val_loss', patience=15,
restore_best_weights=True)
history2 = model2.fit(
    X_train2.astype(np.float32), y_train2.astype(np.float32),
    epochs=150,
    batch_size=best_params2['batch_size'],
    validation_data=(X_test2.astype(np.float32),
y_test2.astype(np.float32)),
    callbacks=[early_stopping2],
    verbose=1
)

# --- Optimize and Train Model 3 ---
print("\nOptimizing hyperparameters for Model 3...")
input_shape3 = (X_train3.shape[1], X_train3.shape[2])
best_params3 = optimize_transformer_hyperparameters(
    X_train3, y_train3, X_test3, y_test3, input_shape3, output_units,
"Model3_Sentiment_Macro"
)
print("Best hyperparameters for Model 3:", best_params3)

print("\nBuilding Model 3 with best hyperparameters...")
if np.any(np.isnan(X_train3)) or np.any(np.isinf(X_train3)):
    print("FATAL: NaN/Inf in X_train3!")
    exit()
if np.any(np.isnan(y_train3)) or np.any(np.isinf(y_train3)):
    print("FATAL: NaN/Inf in y_train3!")
```

```
    exit()
model3 = build_transformer_model(
    input_shape3,
    output_units,
    num_heads=best_params3['num_heads'],
    ff_dim=best_params3['ff_dim'],
    num_layers=best_params3['num_layers'],
    dropout_rate=best_params3['dropout_rate'],
    model_name="Model3_Sentiment_Macro"
)
model3.compile(optimizer=Adam(learning_rate=best_params3['learning_rate']),
loss='mse')
model3.summary()

print("\nTraining Model 3...")
early_stopping3 = EarlyStopping(monitor='val_loss', patience=15,
restore_best_weights=True)
history3 = model3.fit(
    X_train3.astype(np.float32), y_train3.astype(np.float32),
    epochs=150,
    batch_size=best_params3['batch_size'],
    validation_data=(X_test3.astype(np.float32),
y_test3.astype(np.float32)),
    callbacks=[early_stopping3],
    verbose=1
)

# --- Plot Training History ---
def plot_loss(history, title):
    plt.figure(figsize=(10, 6))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title(title)
    plt.xlabel('Epoch')
    plt.ylabel('Loss (MSE)')
    plt.legend()
    plt.grid(True)
    plt.show()  # Display in Colab instead of saving
    plt.close()

print("\nPlotting training history...")
plot_loss(history1, 'Model 1 Training & Validation Loss')
plot_loss(history2, 'Model 2 Training & Validation Loss')
plot_loss(history3, 'Model 3 Training & Validation Loss')
```

```
#
==============================================================================
===
# ========= Step 4: Prediction, Evaluation, and Portfolio Optimization
=========
#
==============================================================================
===

print("\n--- Starting Step 4: Prediction, Evaluation, Optimization ---")

# --- Predictions ---
print("\nMaking predictions...")
y_pred_scaled1 = model1.predict(X_test1)
y_pred_scaled2 = model2.predict(X_test2)
y_pred_scaled3 = model3.predict(X_test3)
y_pred1 = scaler_target.inverse_transform(y_pred_scaled1)
y_pred2 = scaler_target.inverse_transform(y_pred_scaled2)
y_pred3 = scaler_target.inverse_transform(y_pred_scaled3)
y_test_actual = scaler_target.inverse_transform(y_test1)
print(f"Prediction shapes: Pred1={y_pred1.shape}, Pred2={y_pred2.shape},
Pred3={y_pred3.shape}, Actual={y_test_actual.shape}")

# --- Evaluate Models ---
mse1 = mean_squared_error(y_test_actual, y_pred1)
mse2 = mean_squared_error(y_test_actual, y_pred2)
mse3 = mean_squared_error(y_test_actual, y_pred3)
print(f"Model 1 Test MSE: {mse1:.8f}")
print(f"Model 2 Test MSE: {mse2:.8f}")
print(f"Model 3 Test MSE: {mse3:.8f}")

# --- Portfolio Optimization ---
print("\nPreparing portfolio optimization...")
expected_returns1 = y_pred1[-1]
expected_returns2 = y_pred2[-1]
expected_returns3 = y_pred3[-1]
train_df_portion = combined_df.iloc[:n_train + SEQUENCE_LENGTH]
train_returns = train_df_portion[target_columns]
ann_factor = 252
cov_matrix_hist = train_returns.cov() * ann_factor
try:
    np.linalg.cholesky(cov_matrix_hist)
except np.linalg.LinAlgError:
    from sklearn.covariance import LedoitWolf
```

```
    cov_matrix_hist =
pd.DataFrame(LedoitWolf().fit(train_returns.dropna()).covariance_ *
ann_factor,
                                   index=target_columns,
columns=target_columns)
    print("Applied Ledoit-Wolf shrinkage.")

num_assets = len(target_columns)
def maximize_sharpe_ratio(expected_returns, cov_matrix,
risk_free_rate=0.0):
    def neg_sharpe_ratio(weights):
        p_ret = np.sum(expected_returns * weights)
        p_vol = np.sqrt(np.dot(weights.T, np.dot(cov_matrix.values,
weights)))
        return -(p_ret - risk_free_rate) / p_vol if p_vol != 0 else -np.inf
    constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1})
    bounds = tuple((0, 1) for _ in range(num_assets))
    initial_weights = np.array([1./num_assets] * num_assets)
    result = minimize(neg_sharpe_ratio, initial_weights, method='SLSQP',
bounds=bounds, constraints=constraints)
    return result.x / np.sum(result.x) if result.success else
initial_weights

optimal_weights1 = maximize_sharpe_ratio(expected_returns1,
cov_matrix_hist)
optimal_weights2 = maximize_sharpe_ratio(expected_returns2,
cov_matrix_hist)
optimal_weights3 = maximize_sharpe_ratio(expected_returns3,
cov_matrix_hist)

# --- Display Weights ---
print("\n--- Optimal Portfolio Weights (Tickers) ---")
results_df = pd.DataFrame(index=base_tickers)
results_df['Model1_Weights'] = optimal_weights1
results_df['Model2_Weights'] = optimal_weights2
results_df['Model3_Weights'] = optimal_weights3
print("\nModel 1 Weights (Tickers > 0.1%):")
print(results_df[results_df['Model1_Weights'] >
0.001]['Model1_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
print("\nModel 2 Weights (Tickers > 0.1%):")
print(results_df[results_df['Model2_Weights'] >
0.001]['Model2_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
print("\nModel 3 Weights (Tickers > 0.1%):")
print(results_df[results_df['Model3_Weights'] >
0.001]['Model3_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
```

```python
# --- Sector Aggregation ---
print("\n--- Aggregating Portfolio Weights by Sector ---")
sector_weights_df = results_df.merge(etf_sectors, left_index=True,
right_index=True, how='left')
sector_summary = sector_weights_df.groupby('Sector').sum()
print("\n--- Model 1 Sector Weights ---")
print(sector_summary[sector_summary['Model1_Weights'] >
0.001]['Model1_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
print("\n--- Model 2 Sector Weights ---")
print(sector_summary[sector_summary['Model2_Weights'] >
0.001]['Model2_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
print("\n--- Model 3 Sector Weights ---")
print(sector_summary[sector_summary['Model3_Weights'] >
0.001]['Model3_Weights'].sort_values(ascending=False).map('{:.2%}'.format))

# --- Macro Factor Importance Analysis for Model 3 ---
print("\n--- Analyzing Macro Factor Importance for Model 3 ---")
macro_scaled = scaler_model3.transform(combined_df[model3_features])[:, -
len(macro_cols):]
macro_df = pd.DataFrame(macro_scaled, columns=macro_cols,
index=combined_df.index)
portfolio_ret_m3 = y_test_actual @ optimal_weights3
macro_test = macro_df.iloc[-len(portfolio_ret_m3):]
correlations = macro_test.corrwith(pd.Series(portfolio_ret_m3))
print("\nCorrelation of Macro Factors with Model 3 Portfolio Returns:")
print(correlations.sort_values(ascending=False))
most_impactful_macro = correlations.idxmax()
max_correlation = correlations.max()
print(f"\nMacro Factor with Highest Impact: {most_impactful_macro}
(Correlation: {max_correlation:.4f})")

# --- Portfolio Performance Metrics ---
print("\n--- Portfolio Performance Comparison ---")
portfolio_ret_m1 = y_test_actual @ optimal_weights1
portfolio_ret_m2 = y_test_actual @ optimal_weights2
portfolio_ret_eqw = y_test_actual @ (np.ones(num_assets) / num_assets)

def calculate_portfolio_metrics(returns, risk_free_rate=0.0,
confidence_level=0.05):
    total_return = np.prod(1 + returns) - 1
    n_periods = len(returns)
    annualized_return = (1 + total_return) ** (ann_factor / n_periods) - 1
if n_periods > 0 else np.nan
    annualized_volatility = np.std(returns) * np.sqrt(ann_factor) if
n_periods > 1 else np.nan
```

```
    sharpe_ratio = (annualized_return - risk_free_rate) /
annualized_volatility if annualized_volatility != 0 else np.nan

    # VaR and ES
    returns_sorted = np.sort(returns)
    var_index = int(len(returns_sorted) * confidence_level)
    var = returns_sorted[var_index]
    es = returns_sorted[:var_index].mean() if var_index > 0 else np.nan

    # Alpha (relative to equal-weight benchmark)
    benchmark_returns = portfolio_ret_eqw[:len(returns)]
    beta = np.cov(returns, benchmark_returns)[0, 1] /
np.var(benchmark_returns) if np.var(benchmark_returns) != 0 else 0
    alpha = annualized_return - risk_free_rate - beta * (np.prod(1 +
benchmark_returns) ** (ann_factor / n_periods) - 1 - risk_free_rate)

    return {
        'Annualized Return': annualized_return,
        'Annualized Volatility': annualized_volatility,
        'Sharpe Ratio': sharpe_ratio,
        'VaR (5%)': var,
        'Expected Shortfall (5%)': es,
        'Alpha': alpha
    }

print("\nPerformance Metrics (Annualized):")
metrics_m1 = calculate_portfolio_metrics(portfolio_ret_m1)
metrics_m2 = calculate_portfolio_metrics(portfolio_ret_m2)
metrics_m3 = calculate_portfolio_metrics(portfolio_ret_m3)
metrics_eqw = calculate_portfolio_metrics(portfolio_ret_eqw)

print("\nModel 1 Metrics:")
for key, value in metrics_m1.items():
    print(f"{key}: {value:.2%}" if 'Ratio' not in key else f"{key}:
{value:.2f}")
print("\nModel 2 Metrics:")
for key, value in metrics_m2.items():
    print(f"{key}: {value:.2%}" if 'Ratio' not in key else f"{key}:
{value:.2f}")
print("\nModel 3 Metrics:")
for key, value in metrics_m3.items():
    print(f"{key}: {value:.2%}" if 'Ratio' not in key else f"{key}:
{value:.2f}")
print("\nBenchmark (EQW) Metrics:")
for key, value in metrics_eqw.items():
```

```
    print(f"{key}: {value:.2%}" if 'Ratio' not in key else f"{key}:
{value:.2f}")

# --- Visualization ---
print("\nGenerating comparison graphs...")

# Graph 1: Sector Weights Comparison
plt.figure(figsize=(12, 6))
bar_width = 0.2
index = np.arange(len(sectors))
eqw_weights = [0.2] * len(sectors)
model1_sector_weights = [sector_summary.loc[s, 'Model1_Weights'] if s in
sector_summary.index else 0 for s in sectors]
model2_sector_weights = [sector_summary.loc[s, 'Model2_Weights'] if s in
sector_summary.index else 0 for s in sectors]
model3_sector_weights = [sector_summary.loc[s, 'Model3_Weights'] if s in
sector_summary.index else 0 for s in sectors]

plt.bar(index, model1_sector_weights, bar_width, label='Model 1
(Sentiment)', color='blue')
plt.bar(index + bar_width, model2_sector_weights, bar_width, label='Model 2
(Sentiment+Volatility)', color='green')
plt.bar(index + 2 * bar_width, model3_sector_weights, bar_width,
label='Model 3 (Sentiment+Macro)', color='red')
plt.bar(index + 3 * bar_width, eqw_weights, bar_width, label='Benchmark
(Equal Weight)', color='gray')

plt.xlabel('Sectors')
plt.ylabel('Portfolio Weight')
plt.title('Sector Weights Comparison')
plt.xticks(index + 1.5 * bar_width, sectors, rotation=45)
plt.legend()
plt.tight_layout()
plt.show()  # Display in Colab
plt.close()

# Graph 2: Performance Metrics Comparison
plt.figure(figsize=(12, 6))
metrics = ['Annualized Return', 'Annualized Volatility', 'Sharpe Ratio',
'VaR (5%)', 'Expected Shortfall (5%)', 'Alpha']
model1_metrics = [metrics_m1[m] for m in metrics]
model2_metrics = [metrics_m2[m] for m in metrics]
model3_metrics = [metrics_m3[m] for m in metrics]
eqw_metrics = [metrics_eqw[m] for m in metrics]

index = np.arange(len(metrics))
```

```
plt.bar(index, model1_metrics, bar_width, label='Model 1 (Sentiment)',
color='blue')
plt.bar(index + bar_width, model2_metrics, bar_width, label='Model 2
(Sentiment+Volatility)', color='green')
plt.bar(index + 2 * bar_width, model3_metrics, bar_width, label='Model 3
(Sentiment+Macro)', color='red')
plt.bar(index + 3 * bar_width, eqw_metrics, bar_width, label='Benchmark
(Equal Weight)', color='gray')

plt.xlabel('Metrics')
plt.ylabel('Value')
plt.title('Performance Metrics Comparison')
plt.xticks(index + 1.5 * bar_width, metrics, rotation=45)
plt.legend()
plt.tight_layout()
plt.show()  # Display in Colab
plt.close()

# Graph 3: Cumulative Returns
print("\nPlotting Cumulative Returns...")
plt.figure(figsize=(12, 7))
plt.plot(np.cumprod(1 + portfolio_ret_m1) - 1, label='Model 1 (Sentiment)')
plt.plot(np.cumprod(1 + portfolio_ret_m2) - 1, label='Model 2
(Sentiment+Volatility)')
plt.plot(np.cumprod(1 + portfolio_ret_m3) - 1, label='Model 3
(Sentiment+Macro)')
plt.plot(np.cumprod(1 + portfolio_ret_eqw) - 1, label='Benchmark (Equal
Weight)', linestyle='--')
plt.title('Portfolio Cumulative Returns (Test Period)')
plt.xlabel('Time Steps')
plt.ylabel('Cumulative Return')
plt.legend()
plt.grid(True)
plt.show()  # Display in Colab
plt.close()

# Save models
model1.save('model1_transformer_sentiment_optuna.h5')
model2.save('model2_transformer_sentiment_volatility_optuna.h5')
model3.save('model3_transformer_sentiment_macro_optuna.h5')

print("\n--- Step 4 and Comparison Graphs Completed ---")
```

RNN Model

```python
# Install Optuna
!pip install optuna

# Configure GPU memory growth before importing TensorFlow
import tensorflow as tf

gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:
        tf.config.experimental.set_memory_growth(gpus[0], True)
        print("Using GPU:", gpus[0])
    except RuntimeError as e:
        print(f"GPU configuration error: {e}")
else:
    print("No GPU found, using CPU")

# Imports
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, SimpleRNN
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from sklearn.metrics import mean_squared_error
from google.colab import drive
import optuna

# Mount Google Drive
drive.mount('/content/drive')

#
===========================================================================
===
# ========= Step 1: Data Loading and Preparation =========
#
===========================================================================
===

print("\n--- Starting Step 1: Data Loading and Preparation ---")

# --- Configuration ---
etf_path = "/content/drive/MyDrive/Data Science Project /finale
/all_etf_data.csv"
```

```
sentiment_path = "/content/drive/MyDrive/Data Science Project /finale
/Sentimental_score_Final.csv"
macro_path = "/content/drive/MyDrive/Data Science Project /finale
/macroeconomic_indicators.csv"
sectors = ['Financials', 'Real Estate', 'Technology', 'Energy',
'Healthcare']
macro_cols = ['Federal Funds Rate', 'Consumer Price Index', 'Unemployment
Rate', 'Gross Domestic Product',
              '10-Year minus 2-Year Treasury Yield Spread', 'CBOE
Volatility Index', 'WTI Crude Oil Prices',
              '10-Year Treasury Yield']
print(f"ETF data path: {etf_path}")
print(f"Sentiment data path: {sentiment_path}")
print(f"Macro data path: {macro_path}")
print(f"Target sectors: {sectors}")

# --- Load Data ---
print("\nLoading data...")
try:
    etf_prices = pd.read_csv(etf_path)
    sentiment_scores = pd.read_csv(sentiment_path)
    macro_data = pd.read_csv(macro_path)
    print(f"ETF Prices Shape: {etf_prices.shape}")
    print(f"Sentiment Scores Shape: {sentiment_scores.shape}")
    print(f"Macro Data Shape: {macro_data.shape}")
except Exception as e:
    print(f"FATAL ERROR loading data: {e}")
    exit()

# --- Data Cleaning ---
print("\nCleaning data...")
etf_prices['Ticker'] = etf_prices['Ticker'].astype(str).str.strip()
etf_prices['Sector'] =
etf_prices['Sector'].fillna('Unknown').astype(str).str.strip().replace('',
'Unknown')
sentiment_scores['sector'] =
sentiment_scores['sector'].fillna('Unknown').astype(str).str.strip().replac
e('', 'Unknown')

# Validate sectors
print("ETF Sectors:", etf_prices['Sector'].unique())
print("Sentiment Sectors:", sentiment_scores['sector'].unique())
if not all(s in etf_prices['Sector'].unique() for s in sectors):
    print(f"Warning: Some target sectors {sectors} not found in
etf_prices['Sector']")
```

```
# Standardize dates
print("Standardizing date formats...")
etf_prices['Date'] = pd.to_datetime(etf_prices['Date'], format='%d-%m-%Y')
sentiment_scores['date'] = pd.to_datetime(sentiment_scores['date'],
format='%d-%m-%Y')
macro_data['Date'] = pd.to_datetime(macro_data['Date'], format='%d-%m-%Y')
etf_prices.dropna(subset=['Date', 'Close'], inplace=True)
sentiment_scores.dropna(subset=['date', 'sentiment_impact_score'],
inplace=True)
macro_data.dropna(subset=['Date'], inplace=True)
print("Dates standardized and NaTs dropped.")

# --- Sentiment Aggregation ---
print("\nAggregating sentiment scores by sector and date...")
daily_sector_sentiment = sentiment_scores.groupby(['date',
'sector'])['sentiment_impact_score'].mean().reset_index()
print(f"Aggregated Sentiment Shape: {daily_sector_sentiment.shape}")

# --- Prepare ETF Data ---
print("\nCalculating returns and smoothed volatility...")
etf_prices = etf_prices.sort_values(['Ticker', 'Date'])
etf_prices['return'] = etf_prices.groupby('Ticker')['Close'].pct_change()
# Use a longer rolling window and exponential moving average for volatility
etf_prices['volatility'] =
etf_prices.groupby('Ticker')['return'].ewm(span=30).std().reset_index(level
=0, drop=True)
# Clip extreme volatility values
etf_prices['volatility'] =
etf_prices['volatility'].clip(lower=etf_prices['volatility'].quantile(0.01)
,
                                                         upper=etf_prices['
volatility'].quantile(0.99))
etf_sectors = etf_prices[['Ticker',
'Sector']].drop_duplicates().set_index('Ticker')
print(f"Created ticker-to-sector map for {len(etf_sectors)} unique
tickers.")

# Pivot ETF data
etf_pivot_df = etf_prices.pivot_table(index='Date', columns='Ticker',
values=['return', 'volatility'])
etf_pivot_df.columns = [f"{col[1]}_{col[0]}" for col in
etf_pivot_df.columns]
etf_return_tickers = [col for col in etf_pivot_df.columns if
col.endswith('_return')]
etf_pivot_df.dropna(subset=etf_return_tickers, how='all', inplace=True)
print(f"Pivoted ETF data shape: {etf_pivot_df.shape}")
```

```python
if etf_pivot_df.empty:
    print("FATAL ERROR: ETF Pivot table is empty.")
    exit()

# --- Merge Data ---
print("\nMerging dataframes...")
combined_df = etf_pivot_df.copy()
etf_tickers_in_pivot = etf_sectors.index.tolist()
daily_sector_sentiment.set_index('date', inplace=True)
print("Merging sentiment...")
for ticker in etf_tickers_in_pivot:
    try:
        sector = etf_sectors.loc[ticker, 'Sector']
        if pd.isna(sector) or sector == '':
            sector = 'Unknown'
        relevant_sentiment =
daily_sector_sentiment[daily_sector_sentiment['sector'] ==
sector]['sentiment_impact_score']
        sentiment_col_name = f"{ticker}_sentiment"
        if relevant_sentiment.empty:
            sentiment_series = pd.Series(index=combined_df.index,
data=np.nan, name=sentiment_col_name)
        else:
            sentiment_series = relevant_sentiment
            sentiment_series.name = sentiment_col_name
        sentiment_series.index.name = 'Date'
        combined_df = pd.merge(combined_df, sentiment_series, on='Date',
how='left')
    except Exception as e:
        print(f"Warn: Merge sentiment error for {ticker}: {e}")
        combined_df[f"{ticker}_sentiment"] = np.nan

# Merge macro factors
print("Merging macro factors...")
macro_data.set_index('Date', inplace=True)
combined_df = pd.merge(combined_df, macro_data, on='Date', how='left')
print(f"Shape after macro merge: {combined_df.shape}")

# --- Handle Missing Data ---
print("\nHandling missing data...")
combined_df.sort_index(inplace=True)
sentiment_cols = [col for col in combined_df.columns if
col.endswith('_sentiment')]
volatility_cols = [col for col in combined_df.columns if
col.endswith('_volatility')]
cols_to_fill = sentiment_cols + volatility_cols + macro_cols
```

```
if cols_to_fill:
    print(f"Attempting bfill/ffill on {len(cols_to_fill)} columns.")
    combined_df[cols_to_fill] =
combined_df[cols_to_fill].fillna(method='bfill').fillna(method='ffill')
combined_df[sentiment_cols] = combined_df[sentiment_cols].fillna(0)
combined_df[macro_cols] =
combined_df[macro_cols].fillna(combined_df[macro_cols].mean())
initial_rows = len(combined_df)
combined_df.dropna(subset=etf_return_tickers, how='any', inplace=True)
print(f"Dropped {initial_rows - len(combined_df)} rows based on return
NaNs.")
print(f"Final Combined DataFrame Shape: {combined_df.shape}")
if combined_df.empty:
    print("FATAL ERROR: Final combined_df is empty.")
    exit()

#
===========================================================================
===
# ========= Step 2: Feature Engineering, Scaling & Sequencing =========
#
===========================================================================
===

print("\n--- Starting Step 2: Feature Engineering, Scaling & Sequencing ---
")

# --- Define Feature Sets ---
print("\nDefining feature sets...")
target_columns = etf_return_tickers
base_tickers = [col.replace('_return', '') for col in target_columns]
model1_features = target_columns + [f"{t}_sentiment" for t in base_tickers
if f"{t}_sentiment" in combined_df.columns]
model1_features = [f for f in model1_features if f in combined_df.columns]
print(f"Model 1 Features (Returns + Sentiment): {len(model1_features)}")
model2_features = model1_features + [f"{t}_volatility" for t in
base_tickers if f"{t}_volatility" in combined_df.columns]
model2_features = [f for f in model2_features if f in combined_df.columns]
print(f"Model 2 Features (M1 + Volatility): {len(model2_features)}")
model3_features = model1_features + macro_cols
model3_features = [f for f in model3_features if f in combined_df.columns]
print(f"Model 3 Features (M1 + Macro): {len(model3_features)}")

# --- Data Scaling ---
print("\nScaling features...")
scaler_model1 = MinMaxScaler(feature_range=(0, 1))
```

```
scaler_model2 = MinMaxScaler(feature_range=(0, 1))
scaler_model3 = MinMaxScaler(feature_range=(0, 1))
scaler_target = MinMaxScaler(feature_range=(0, 1))
scaled_data_model1 =
scaler_model1.fit_transform(combined_df[model1_features])
scaled_data_model2 =
scaler_model2.fit_transform(combined_df[model2_features])
scaled_data_model3 =
scaler_model3.fit_transform(combined_df[model3_features])
scaled_target_data =
scaler_target.fit_transform(combined_df[target_columns])
print(f"Scaled shapes: Model1={scaled_data_model1.shape},
Model2={scaled_data_model2.shape}, Model3={scaled_data_model3.shape},
Target={scaled_target_data.shape}")

# --- Sequence Creation ---
def create_sequences(input_data, target_data, sequence_length):
    X, y = [], []
    if len(input_data) <= sequence_length:
        return np.array(X), np.array(y)
    for i in range(sequence_length, len(input_data)):
        X.append(input_data[i-sequence_length:i])
        y.append(target_data[i])
    return np.array(X), np.array(y)

SEQUENCE_LENGTH = 20
if SEQUENCE_LENGTH >= len(combined_df):
    SEQUENCE_LENGTH = max(1, len(combined_df) // 4)
print(f"Using sequence length: {SEQUENCE_LENGTH}")

print("Creating sequences...")
X_model1, y_model1 = create_sequences(scaled_data_model1,
scaled_target_data, SEQUENCE_LENGTH)
X_model2, y_model2 = create_sequences(scaled_data_model2,
scaled_target_data, SEQUENCE_LENGTH)
X_model3, y_model3 = create_sequences(scaled_data_model3,
scaled_target_data, SEQUENCE_LENGTH)
print(f"Model 1: X={X_model1.shape}, y={y_model1.shape}")
print(f"Model 2: X={X_model2.shape}, y={y_model2.shape}")
print(f"Model 3: X={X_model3.shape}, y={y_model3.shape}")
if X_model1.shape[0] == 0 or X_model2.shape[0] == 0 or X_model3.shape[0] ==
0:
    print("FATAL ERROR: Zero sequences created.")
    exit()

# --- Train/Test Split ---
```

```
print("Splitting data...")
test_split_ratio = 0.2
n_samples = X_model1.shape[0]
n_test = int(n_samples * test_split_ratio)
n_train = n_samples - n_test
X_train1, X_test1 = X_model1[:n_train], X_model1[n_train:]
y_train1, y_test1 = y_model1[:n_train], y_model1[n_train:]
X_train2, X_test2 = X_model2[:n_train], X_model2[n_train:]
y_train2, y_test2 = y_model2[:n_train], y_model2[n_train:]
X_train3, X_test3 = X_model3[:n_train], X_model3[n_train:]
y_train3, y_test3 = y_model3[:n_train], y_model3[n_train:]
print(f"Split: Train={n_train}, Test={n_test}")

#
===========================================================================
===
# ========= Step 3: Build and Train RNN Models with Optuna =========
#
===========================================================================
===

print("\n--- Starting Step 3: Build and Train RNN Models with
Hyperparameter Tuning ---")

# --- Define RNN Model with Variable Hyperparameters ---
def build_rnn_model(input_shape, output_units, units1, units2,
dropout_rate, model_name):
    model = Sequential(name=model_name)
    model.add(SimpleRNN(units=units1, return_sequences=True,
input_shape=input_shape))
    model.add(Dropout(dropout_rate))
    model.add(SimpleRNN(units=units2))
    model.add(Dropout(dropout_rate))
    model.add(Dense(units=16, activation='relu'))  # Reduced dense layer
size
    model.add(Dense(units=output_units, activation='linear'))
    return model

# --- Optuna Objective Function ---
def objective(trial, X_train, y_train, X_val, y_val, input_shape,
output_units, model_name):
    units1 = trial.suggest_categorical('units1', [32, 64, 128])  # Reduced
max units
    units2 = trial.suggest_categorical('units2', [16, 32, 64])  # Reduced
max units
```

```
    dropout_rate = trial.suggest_float('dropout_rate', 0.3, 0.6)  # 
Increased dropout
    learning_rate = trial.suggest_float('learning_rate', 1e-4, 1e-2, 
log=True)
    batch_size = trial.suggest_categorical('batch_size', [32, 64, 128])  # 
Larger batch sizes

    model = build_rnn_model(input_shape, output_units, units1, units2, 
dropout_rate, model_name)
    model.compile(optimizer=Adam(learning_rate=learning_rate), loss='mse')

    early_stopping = EarlyStopping(monitor='val_loss', patience=20, 
restore_best_weights=True)
    history = model.fit(
        X_train.astype(np.float32), y_train.astype(np.float32),
        epochs=200,
        batch_size=batch_size,
        validation_data=(X_val.astype(np.float32), 
y_val.astype(np.float32)),
        callbacks=[early_stopping],
        verbose=0
    )
    return min(history.history['val_loss'])

# --- Hyperparameter Optimization ---
def optimize_rnn_hyperparameters(X_train, y_train, X_val, y_val, 
input_shape, output_units, model_name, n_trials=20):
    study = optuna.create_study(direction='minimize')
    objective_fn = lambda trial: objective(trial, X_train, y_train, X_val, 
y_val, input_shape, output_units, model_name)
    study.optimize(objective_fn, n_trials=n_trials)
    return study.best_params

# --- Optimize and Train Model 1 ---
print("\nOptimizing hyperparameters for Model 1...")
input_shape1 = (X_train1.shape[1], X_train1.shape[2])
output_units = y_train1.shape[1]
best_params1 = optimize_rnn_hyperparameters(
    X_train1, y_train1, X_test1, y_test1, input_shape1, output_units, 
"Model1_Sentiment", n_trials=20
)
print("Best hyperparameters for Model 1:", best_params1)

print("\nBuilding Model 1 with best hyperparameters...")
if np.any(np.isnan(X_train1)) or np.any(np.isinf(X_train1)):
    print("FATAL: NaN/Inf in X_train1!")
```

```
    exit()
if np.any(np.isnan(y_train1)) or np.any(np.isinf(y_train1)):
    print("FATAL: NaN/Inf in y_train1!")
    exit()
model1 = build_rnn_model(
    input_shape1,
    output_units,
    units1=best_params1['units1'],
    units2=best_params1['units2'],
    dropout_rate=best_params1['dropout_rate'],
    model_name="Model1_Sentiment"
)
model1.compile(optimizer=Adam(learning_rate=best_params1['learning_rate']),
loss='mse')
model1.summary()

print("\nTraining Model 1...")
early_stopping1 = EarlyStopping(monitor='val_loss', patience=20,
restore_best_weights=True)
history1 = model1.fit(
    X_train1.astype(np.float32), y_train1.astype(np.float32),
    epochs=200,
    batch_size=best_params1['batch_size'],
    validation_data=(X_test1.astype(np.float32),
y_test1.astype(np.float32)),
    callbacks=[early_stopping1],
    verbose=1
)

# --- Optimize and Train Model 2 ---
print("\nOptimizing hyperparameters for Model 2...")
input_shape2 = (X_train2.shape[1], X_train2.shape[2])
best_params2 = optimize_rnn_hyperparameters(
    X_train2, y_train2, X_test2, y_test2, input_shape2, output_units,
"Model2_Sentiment_Volatility", n_trials=20
)
print("Best hyperparameters for Model 2:", best_params2)

print("\nBuilding Model 2 with best hyperparameters...")
if np.any(np.isnan(X_train2)) or np.any(np.isinf(X_train2)):
    print("FATAL: NaN/Inf in X_train2!")
    exit()
if np.any(np.isnan(y_train2)) or np.any(np.isinf(y_train2)):
    print("FATAL: NaN/Inf in y_train2!")
    exit()
model2 = build_rnn_model(
```

```
    input_shape2,
    output_units,
    units1=best_params2['units1'],
    units2=best_params2['units2'],
    dropout_rate=best_params2['dropout_rate'],
    model_name="Model2_Sentiment_Volatility"
)
model2.compile(optimizer=Adam(learning_rate=best_params2['learning_rate']),
loss='mse')
model2.summary()

print("\nTraining Model 2...")
early_stopping2 = EarlyStopping(monitor='val_loss', patience=20,
restore_best_weights=True)
history2 = model2.fit(
    X_train2.astype(np.float32), y_train2.astype(np.float32),
    epochs=200,
    batch_size=best_params2['batch_size'],
    validation_data=(X_test2.astype(np.float32),
y_test2.astype(np.float32)),
    callbacks=[early_stopping2],
    verbose=1
)

# --- Optimize and Train Model 3 ---
print("\nOptimizing hyperparameters for Model 3...")
input_shape3 = (X_train3.shape[1], X_train3.shape[2])
best_params3 = optimize_rnn_hyperparameters(
    X_train3, y_train3, X_test3, y_test3, input_shape3, output_units,
"Model3_Sentiment_Macro", n_trials=20
)
print("Best hyperparameters for Model 3:", best_params3)

print("\nBuilding Model 3 with best hyperparameters...")
if np.any(np.isnan(X_train3)) or np.any(np.isinf(X_train3)):
    print("FATAL: NaN/Inf in X_train3!")
    exit()
if np.any(np.isnan(y_train3)) or np.any(np.isinf(y_train3)):
    print("FATAL: NaN/Inf in y_train3!")
    exit()
model3 = build_rnn_model(
    input_shape3,
    output_units,
    units1=best_params3['units1'],
    units2=best_params3['units2'],
    dropout_rate=best_params3['dropout_rate'],
```

```
    model_name="Model3_Sentiment_Macro"
)
model3.compile(optimizer=Adam(learning_rate=best_params3['learning_rate']),
loss='mse')
model3.summary()

print("\nTraining Model 3...")
early_stopping3 = EarlyStopping(monitor='val_loss', patience=20,
restore_best_weights=True)
history3 = model3.fit(
    X_train3.astype(np.float32), y_train3.astype(np.float32),
    epochs=200,
    batch_size=best_params3['batch_size'],
    validation_data=(X_test3.astype(np.float32),
y_test3.astype(np.float32)),
    callbacks=[early_stopping3],
    verbose=1
)

# --- Plot Training History ---
def plot_loss(history, title):
    plt.figure(figsize=(10, 6))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title(title)
    plt.xlabel('Epoch')
    plt.ylabel('Loss (MSE)')
    plt.legend()
    plt.grid(True)
    plt.show()
    plt.close()

print("\nPlotting training history...")
plot_loss(history1, 'Model 1 Training & Validation Loss')
plot_loss(history2, 'Model 2 Training & Validation Loss')
plot_loss(history3, 'Model 3 Training & Validation Loss')

#
===========================================================================
===
# ========= Step 4: Prediction, Evaluation, and Portfolio Optimization
=========
#
===========================================================================
===
```

```
print("\n--- Starting Step 4: Prediction, Evaluation, Optimization ---")

# --- Predictions ---
print("\nMaking predictions...")
y_pred_scaled1 = model1.predict(X_test1)
y_pred_scaled2 = model2.predict(X_test2)
y_pred_scaled3 = model3.predict(X_test3)
y_pred1 = scaler_target.inverse_transform(y_pred_scaled1)
y_pred2 = scaler_target.inverse_transform(y_pred_scaled2)
y_pred3 = scaler_target.inverse_transform(y_pred_scaled3)
y_test_actual = scaler_target.inverse_transform(y_test1)
print(f"Prediction shapes: Pred1={y_pred1.shape}, Pred2={y_pred2.shape},
Pred3={y_pred3.shape}, Actual={y_test_actual.shape}")

# --- Evaluate Models ---
mse1 = mean_squared_error(y_test_actual, y_pred1)
mse2 = mean_squared_error(y_test_actual, y_pred2)
mse3 = mean_squared_error(y_test_actual, y_pred3)
print(f"Model 1 Test MSE: {mse1:.8f}")
print(f"Model 2 Test MSE: {mse2:.8f}")
print(f"Model 3 Test MSE: {mse3:.8f}")

# --- Portfolio Optimization ---
print("\nPreparing portfolio optimization...")
expected_returns1 = y_pred1[-1]
expected_returns2 = y_pred2[-1]
expected_returns3 = y_pred3[-1]
train_df_portion = combined_df.iloc[:n_train + SEQUENCE_LENGTH]
train_returns = train_df_portion[target_columns]
ann_factor = 252
cov_matrix_hist = train_returns.cov() * ann_factor
try:
    np.linalg.cholesky(cov_matrix_hist)
except np.linalg.LinAlgError:
    from sklearn.covariance import LedoitWolf
    cov_matrix_hist =
pd.DataFrame(LedoitWolf().fit(train_returns.dropna()).covariance_ *
ann_factor,
                                   index=target_columns,
columns=target_columns)
    print("Applied Ledoit-Wolf shrinkage.")

num_assets = len(target_columns)
def maximize_sharpe_ratio(expected_returns, cov_matrix, risk_free_rate=0.0,
max_weight=0.2):
    def neg_sharpe_ratio(weights):
```

```
        p_ret = np.sum(expected_returns * weights)
        p_vol = np.sqrt(np.dot(weights.T, np.dot(cov_matrix.values,
weights)))
        return -(p_ret - risk_free_rate) / p_vol if p_vol != 0 else -np.inf
    constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1})
    bounds = tuple((0, max_weight) for _ in range(num_assets))  # Max
weight constraint
    initial_weights = np.array([1./num_assets] * num_assets)
    result = minimize(neg_sharpe_ratio, initial_weights, method='SLSQP',
bounds=bounds, constraints=constraints)
    return result.x / np.sum(result.x) if result.success else
initial_weights

optimal_weights1 = maximize_sharpe_ratio(expected_returns1,
cov_matrix_hist, max_weight=0.2)
optimal_weights2 = maximize_sharpe_ratio(expected_returns2,
cov_matrix_hist, max_weight=0.2)
optimal_weights3 = maximize_sharpe_ratio(expected_returns3,
cov_matrix_hist, max_weight=0.2)

# --- Display Weights ---
print("\n--- Optimal Portfolio Weights (Tickers) ---")
results_df = pd.DataFrame(index=base_tickers)
results_df['Model1_Weights'] = optimal_weights1
results_df['Model2_Weights'] = optimal_weights2
results_df['Model3_Weights'] = optimal_weights3
print("\nModel 1 Weights (Tickers > 0.1%):")
print(results_df[results_df['Model1_Weights'] >
0.001]['Model1_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
print("\nModel 2 Weights (Tickers > 0.1%):")
print(results_df[results_df['Model2_Weights'] >
0.001]['Model2_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
print("\nModel 3 Weights (Tickers > 0.1%):")
print(results_df[results_df['Model3_Weights'] >
0.001]['Model3_Weights'].sort_values(ascending=False).map('{:.2%}'.format))

# --- Sector Aggregation ---
print("\n--- Aggregating Portfolio Weights by Sector ---")
sector_weights_df = results_df.merge(etf_sectors, left_index=True,
right_index=True, how='left')
sector_summary = sector_weights_df.groupby('Sector').sum()
print("\n--- Model 1 Sector Weights ---")
print(sector_summary[sector_summary['Model1_Weights'] >
0.001]['Model1_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
print("\n--- Model 2 Sector Weights ---")
```

```
print(sector_summary[sector_summary['Model2_Weights'] >
0.001]['Model2_Weights'].sort_values(ascending=False).map('{:.2%}'.format))
print("\n--- Model 3 Sector Weights ---")
print(sector_summary[sector_summary['Model3_Weights'] >
0.001]['Model3_Weights'].sort_values(ascending=False).map('{:.2%}'.format))

# --- Macro Factor Importance Analysis for Model 3 ---
print("\n--- Analyzing Macro Factor Importance for Model 3 ---")
macro_scaled = scaler_model3.transform(combined_df[model3_features])[:, -
len(macro_cols):]
macro_df = pd.DataFrame(macro_scaled, columns=macro_cols,
index=combined_df.index)
portfolio_ret_m3 = y_test_actual @ optimal_weights3
macro_test = macro_df.iloc[-len(portfolio_ret_m3):].copy()
if macro_test.isna().any().any():
    print("Warning: NaNs found in macro_test, filling with mean...")
    macro_test.fillna(macro_test.mean(), inplace=True)
if np.any(np.isnan(portfolio_ret_m3)):
    print("Warning: NaNs found in portfolio_ret_m3, dropping...")
    valid_indices = ~np.isnan(portfolio_ret_m3)
    portfolio_ret_m3 = portfolio_ret_m3[valid_indices]
    macro_test = macro_test.iloc[valid_indices]
correlations = macro_test.corrwith(pd.Series(portfolio_ret_m3))
print("\nCorrelation of Macro Factors with Model 3 Portfolio Returns:")
print(correlations.sort_values(ascending=False))
most_impactful_macro = correlations.idxmax()
max_correlation = correlations.max()
print(f"\nMacro Factor with Highest Impact: {most_impactful_macro}
(Correlation: {max_correlation:.4f})")

# --- Portfolio Performance Metrics ---
print("\n--- Portfolio Performance Comparison ---")
portfolio_ret_m1 = y_test_actual @ optimal_weights1
portfolio_ret_m2 = y_test_actual @ optimal_weights2
portfolio_ret_eqw = y_test_actual @ (np.ones(num_assets) / num_assets)

def calculate_portfolio_metrics(returns, risk_free_rate=0.0,
confidence_level=0.05):
    total_return = np.prod(1 + returns) - 1
    n_periods = len(returns)
    annualized_return = (1 + total_return) ** (ann_factor / n_periods) - 1
if n_periods > 0 else np.nan
    annualized_volatility = np.std(returns) * np.sqrt(ann_factor) if
n_periods > 1 else np.nan
    sharpe_ratio = (annualized_return - risk_free_rate) /
annualized_volatility if annualized_volatility != 0 else np.nan
```

```
    returns_sorted = np.sort(returns)
    var_index = int(len(returns_sorted) * confidence_level)
    var = returns_sorted[var_index]
    es = returns_sorted[:var_index].mean() if var_index > 0 else np.nan
    benchmark_returns = portfolio_ret_eqw[:len(returns)]
    beta = np.cov(returns, benchmark_returns)[0, 1] /
np.var(benchmark_returns) if np.var(benchmark_returns) != 0 else 0
    alpha = annualized_return - risk_free_rate - beta * (np.prod(1 +
benchmark_returns) ** (ann_factor / n_periods) - 1 - risk_free_rate)
    return {
        'Annualized Return': annualized_return,
        'Annualized Volatility': annualized_volatility,
        'Sharpe Ratio': sharpe_ratio,
        'VaR (5%)': var,
        'Expected Shortfall (5%)': es,
        'Alpha': alpha
    }

print("\nPerformance Metrics (Annualized):")
metrics_m1 = calculate_portfolio_metrics(portfolio_ret_m1)
metrics_m2 = calculate_portfolio_metrics(portfolio_ret_m2)
metrics_m3 = calculate_portfolio_metrics(portfolio_ret_m3)
metrics_eqw = calculate_portfolio_metrics(portfolio_ret_eqw)

print("\nModel 1 Metrics:")
for key, value in metrics_m1.items():
    print(f"{key}: {value:.2%}" if 'Ratio' not in key else f"{key}:
{value:.2f}")
print("\nModel 2 Metrics:")
for key, value in metrics_m2.items():
    print(f"{key}: {value:.2%}" if 'Ratio' not in key else f"{key}:
{value:.2f}")
print("\nModel 3 Metrics:")
for key, value in metrics_m3.items():
    print(f"{key}: {value:.2%}" if 'Ratio' not in key else f"{key}:
{value:.2f}")
print("\nBenchmark (EQW) Metrics:")
for key, value in metrics_eqw.items():
    print(f"{key}: {value:.2%}" if 'Ratio' not in key else f"{key}:
{value:.2f}")

# --- Visualization ---
print("\nGenerating comparison graphs...")

# Graph 1: Sector Weights Comparison
plt.figure(figsize=(12, 6))
```

```
bar_width = 0.2
index = np.arange(len(sectors))
eqw_weights = [0.2] * len(sectors)
model1_sector_weights = [sector_summary.loc[s, 'Model1_Weights'] if s in
sector_summary.index else 0 for s in sectors]
model2_sector_weights = [sector_summary.loc[s, 'Model2_Weights'] if s in
sector_summary.index else 0 for s in sectors]
model3_sector_weights = [sector_summary.loc[s, 'Model3_Weights'] if s in
sector_summary.index else 0 for s in sectors]

plt.bar(index, model1_sector_weights, bar_width, label='Model 1
(Sentiment)', color='blue')
plt.bar(index + bar_width, model2_sector_weights, bar_width, label='Model 2
(Sentiment+Volatility)', color='green')
plt.bar(index + 2 * bar_width, model3_sector_weights, bar_width,
label='Model 3 (Sentiment+Macro)', color='red')
plt.bar(index + 3 * bar_width, eqw_weights, bar_width, label='Benchmark
(Equal Weight)', color='gray')

plt.xlabel('Sectors')
plt.ylabel('Portfolio Weight')
plt.title('Sector Weights Comparison')
plt.xticks(index + 1.5 * bar_width, sectors, rotation=45)
plt.legend()
plt.tight_layout()
plt.show()
plt.close()

# Graph 2: Performance Metrics Comparison
plt.figure(figsize=(12, 6))
metrics = ['Annualized Return', 'Annualized Volatility', 'Sharpe Ratio',
'VaR (5%)', 'Expected Shortfall (5%)', 'Alpha']
model1_metrics = [metrics_m1[m] for m in metrics]
model2_metrics = [metrics_m2[m] for m in metrics]
model3_metrics = [metrics_m3[m] for m in metrics]
eqw_metrics = [metrics_eqw[m] for m in metrics]

index = np.arange(len(metrics))
plt.bar(index, model1_metrics, bar_width, label='Model 1 (Sentiment)',
color='blue')
plt.bar(index + bar_width, model2_metrics, bar_width, label='Model 2
(Sentiment+Volatility)', color='green')
plt.bar(index + 2 * bar_width, model3_metrics, bar_width, label='Model 3
(Sentiment+Macro)', color='red')
plt.bar(index + 3 * bar_width, eqw_metrics, bar_width, label='Benchmark
(Equal Weight)', color='gray')
```

```
plt.xlabel('Metrics')
plt.ylabel('Value')
plt.title('Performance Metrics Comparison')
plt.xticks(index + 1.5 * bar_width, metrics, rotation=45)
plt.legend()
plt.tight_layout()
plt.show()
plt.close()

# Graph 3: Cumulative Returns
print("\nPlotting Cumulative Returns...")
plt.figure(figsize=(12, 7))
plt.plot(np.cumprod(1 + portfolio_ret_m1) - 1, label='Model 1 (Sentiment)')
plt.plot(np.cumprod(1 + portfolio_ret_m2) - 1, label='Model 2
(Sentiment+Volatility)')
plt.plot(np.cumprod(1 + portfolio_ret_m3) - 1, label='Model 3
(Sentiment+Macro)')
plt.plot(np.cumprod(1 + portfolio_ret_eqw) - 1, label='Benchmark (Equal
Weight)', linestyle='--')
plt.title('Portfolio Cumulative Returns (Test Period)')
plt.xlabel('Time Steps')
plt.ylabel('Cumulative Return')
plt.legend()
plt.grid(True)
plt.show()
plt.close()

# Save models
model1.save('model1_rnn_sentiment_optuna.h5')
model2.save('model2_rnn_sentiment_volatility_optuna.h5')
model3.save('model3_rnn_sentiment_macro_optuna.h5')

print("\n--- Step 4 and Comparison Graphs Completed ---")
```