CFRM 421, Spring 2025

Financial Market Regime Classification

Group 1 members:

Eunice Seung-hyun Han, Xiaoquan Shang, Lanmin Lin, Ruoxuan Wang, Christine (Ziyu) Lin

Introduction

Problem Statement

Goal of Project

Financial markets are often volatile and unpredictable as they frequently switch between different states. These changes are affected by macroeconomic shocks, policy, and investor sentiment, and they can also be reflected in the dynamics of asset returns and volatility. Therefore, the ability to accurately identify and predict these market state transitions is helpful for investors and has practical implications in asset allocation, risk management, and tactical trading decisions.

In this project, we aim to identify and forecast market regimes in the S&P 500 Index by combining unsupervised and supervised learning methods. First, we apply K-Means clustering to rolling 21-day statistics—specifically, the mean (μ) and volatility (σ) of log returns—to uncover distinct market states without prior labeling. Then, we use supervised classification models to predict the regime label 21 trading days ahead, enabling forward-looking insights based on current market conditions.

We collected twenty years of daily closing price data from Yahoo Finance (https://finance.yahoo.com/), spanning from 2005 to 2025. Rolling statistics of mean (μ) and volatility (σ) were computed using non-overlapping 5-day

windows to capture short-term trends and risk. These features were then aggregated over 21-day periods to form input for K-Means clustering, enabling classification of each 3-week segment into one of several market regimes. While our current analysis uses five clusters (K=5), future work could explore a higher number of regimes to better capture rare or extreme events such as the 2008 financial crisis or the COVID-19 crash.

We then train a variety of supervised classification models to forecast future regimes based on recent market statistics. These include Softmax Regression with Neural Network, Random Forest, Support Vector Machine (RBF kernel), Gradient Boosting, and a regime-switching Markov chain model. For each model, we use standardized features and apply hyperparameter tuning via grid search or validation-based early stopping, depending on the algorithm. Model performance is evaluated using both classification accuracy and test loss. A summary comparison table is included to benchmark predictive effectiveness across models. Currently, we only use the rolling mean (μ) and volatility (σ) as input features for classification. However, including additional variables—such as trading volume, macroeconomic indicators, or technical signals—may further improve prediction accuracy and regime separation.

Data

We use daily historical price data for the S&P 500 Index (^GSPC), which is obtained from Yahoo Finance — a widely used and publicly accessible financial data platform. The dataset spans from May 2005 to May 2025, covering approximately 20 years of trading activity and capturing multiple market cycles, including bull markets, recessions, and crisis periods. For each trading day, the dataset provides open, high, low, close, adjusted close, and volume information. In our study, we focus specifically on the daily closing prices, as they best reflect end-of-day market consensus and are widely used in financial analysis.

To construct input features suitable for regime analysis, we calculate the mean (μ) and standard deviation (σ) of log returns over a non-overlapping 5-day rolling window, approximating weekly intervals. This transformation produces a sequence of (μ, σ) pairs that capture short-term trend and volatility characteristics of the market. These features form the basis for our modeling.

The resulting dataset supports a two-stage learning framework. First, the (μ, σ) features are used to identify market regimes through unsupervised clustering. Then, each observation is associated with the regime that occurs

21 days later, enabling the development of supervised classification models that forecast the regime in one month.

Target Variable

Our target variable is the market regime of 21 days in the future, derived via KMeans clustering on rolling (μ, σ). The future regime label is then aligned with current features to support supervised classification.

Main Features

The following features are used to capture return dynamics, volatility, momentum, macroeconomic conditions, and market sentiment:

• **log_return_5d**: 5-day log return of the s&p 500 index

• **log_return_21d**: 21-day log return capturing longer-term price trends

• **rolling_std_21d**: 21-day rolling standard deviation of percentage returns, used as a proxy for recent market volatility

• **RSI_14d**: 14-day relative strength index, measuring recent momentum and overbought/oversold conditions

• **MA_21d**: 21-day moving average of closing prices, representing short-term trend direction

• **sp500_volume**: daily trading volume of spy, used as a liquidity and participation proxy

• **vix**: daily close of the vix index, reflecting implied market volatility

• **10y_treasury**: yield of the 10-year u.s. treasury bond, representing long-term interest rate expectations

• **interest_rate**: 1-month treasury rate, used as a proxy for short-term interest rate levels

Import Libraries

In [ ]:
```
# Import necessary libraries for data collection, preprocessing, c

import yfinance as yf  # Download historical stock data
```

```
from sklearn.model_selection import train_test_split  # For splitt
import numpy as np  # Numerical operations
import pandas as pd # Data manipulation
import matplotlib.pyplot as plt  # Data visualization
import matplotlib.dates as mdates  # import date-handling tools
from matplotlib.dates import date2num, DateFormatter, YearLocator,
from matplotlib.collections import LineCollection
from matplotlib.lines import Line2D
import seaborn as sns
import itertools  # Useful tools for iteration
from sklearn.cluster import KMeans  # KMeans clustering for regime
from sklearn.metrics import silhouette_score  # For evaluating clu
from collections import OrderedDict  # Ordered dictionaries for cl
```

In [ ]:
```
# Import Scikit-learn tools for preprocessing, modeling, hyperpara

from sklearn.preprocessing import StandardScaler, LabelBinarizer
from sklearn.pipeline import Pipeline  # Build a machine learning
from sklearn.model_selection import GridSearchCV, TimeSeriesSplit
from sklearn.metrics import accuracy_score, classification_report,
from sklearn.feature_selection import mutual_info_classif
from sklearn.utils import class_weight
from sklearn.ensemble import GradientBoostingClassifier, RandomFor
from sklearn.svm import SVC
```

In [ ]:
```
# Import technical analysis tools and deep learning libraries

from ta.momentum import RSIIndicator  # Relative Strength Index (m
import tensorflow as tf  # TensorFlow for building deep learning m
from tensorflow import keras  # Keras high-level API
from tensorflow.keras import layers  # Layers for building neural
from tensorflow.keras.callbacks import EarlyStopping
```

In [ ]:
```
# Import IPython based environments that are used to show and pres
from IPython.display import Markdown, display
```

Data Processing

In [ ]:
```
# Download historical adjusted closing prices of the S&P 500 inde
# Time period: May 22, 2005 to May 22, 2025
sp500 = yf.download("^GSPC", start="2005-05-22", end="2025-05-22"
vix = yf.download("^VIX", start="2005-05-22", end="2025-05-22")
us10y = yf.download("^TNX", start="2005-05-22", end="2025-05-22")
ffr = yf.download("^IRX", start="2005-05-22", end="2025-05-22")
spy = yf.download("SPY", start="2005-05-22", end="2025-05-22")

# integrated data
df = sp500[["Close", "Volume"]].copy()
df.columns = [col if isinstance(col, str) else col[0] for col in
```

```
df.rename(columns={"Close": "sp500_close", "Volume": "sp500_volum
df["vix"] = vix["Close"]
df["10y_treasury"] = us10y["Close"]
df["interest_rate"] = ffr["Close"]
df["sp500_volume"] = spy["Volume"]

# feature construction:
# Returns
df["log_return_5d"] = np.log(df["sp500_close"] / df["sp500_close"
df["log_return_21d"] = np.log(df["sp500_close"] / df["sp500_close

# Momentum
df["RSI_14d"] = RSIIndicator(close=df["sp500_close"].astype(float

# Volatility
df["rolling_std_21d"] = df["sp500_close"].pct_change().rolling(wi

# Trend
df["MA_21d"] = df["sp500_close"].rolling(window=21).mean()

selected_features = [
    "log_return_5d", "log_return_21d",
    "RSI_14d", "rolling_std_21d", "sp500_volume",
    "MA_21d", "10y_treasury", "interest_rate", "vix"
]

df.dropna(subset=selected_features, inplace=True)
# extract feature data
X_raw = df[selected_features].copy()

# standardize
scaler = StandardScaler()
X_scaled = pd.DataFrame(
    scaler.fit_transform(X_raw),
    columns=X_raw.columns,
    index=X_raw.index
)

# X_scaled

# save data
# X_scaled.to_csv("X_features_scaled.csv")
```

```
[**********************100%**********************]  1 of 1 complet
ed
[**********************100%**********************]  1 of 1 complet
ed
[**********************100%**********************]  1 of 1 complet
ed
[**********************100%**********************]  1 of 1 complet
ed
[**********************100%**********************]  1 of 1 complet
ed
```

Feature Engineering

Step 1. Compute Rolling μ and σ from Log Returns

We begin by calculating the daily log returns of the S&P 500 index. To capture the average behavior and volatility of returns over time, we segment the return series into non-overlapping 5-day windows. For each window, we compute the mean (μ) and standard deviation (σ) of log returns, which respectively represent short-term return trends and volatility. These rolling statistics serve as simplified but informative features for identifying underlying market regimes in the following clustering step.

In [ ]:
```
# Define a function to compute rolling statistics (mean μ and st
# from the log returns of the closing price time series.
def rolling_mu_sigma_from_price(close_series, window=5):
    close_array = close_series.values
    log_returns = np.log(close_array[1:] / close_array[:-1]) # (

    mu_list, sigma_list, start_dates = [], [], []

    # Iterate through the log returns in non-overlapping windows
    for start in range(0, len(log_returns) - window + 1, window)
        window_returns = log_returns[start:start + window]
        mu = np.mean(window_returns) # Mean of returns (rolling
        sigma = np.std(window_returns) # Standard deviation (rol
        mu_list.append(mu)
        sigma_list.append(sigma)
        start_dates.append(close_series.index[start + 1])  # Ali

    # Create a DataFrame containing the rolling μ and σ values a
    df_simple = pd.DataFrame({
        "start_date": start_dates,
        "mu": mu_list,
        "sigma": sigma_list
    })

    return df_simple
```

In [ ]:
```
# Generate the rolling μ and σ features from S&P 500 closing pri
df_simple = rolling_mu_sigma_from_price(sp500["Close"], window=5
df_simple.head()
```

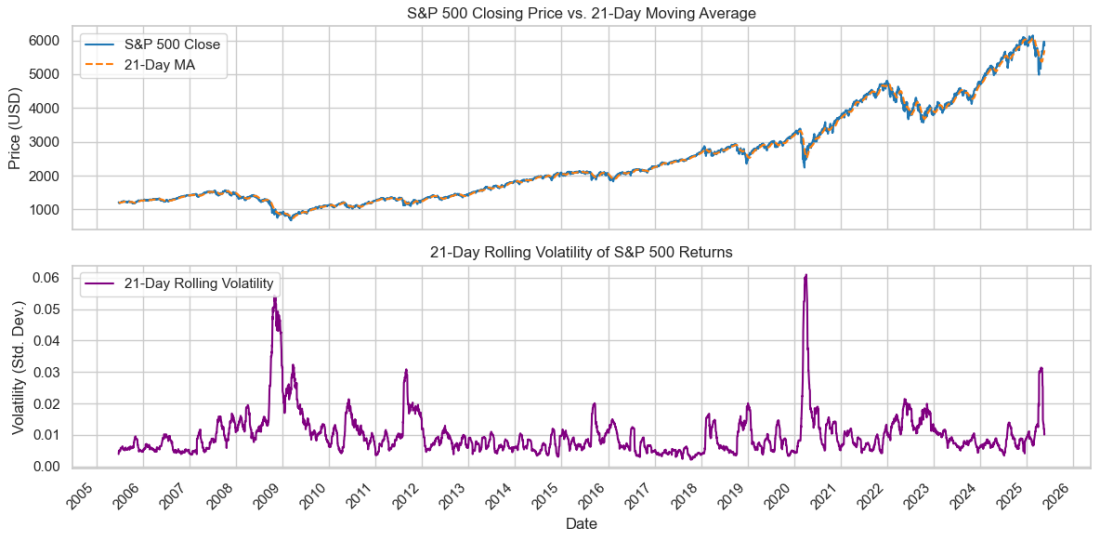
Out[ ]:

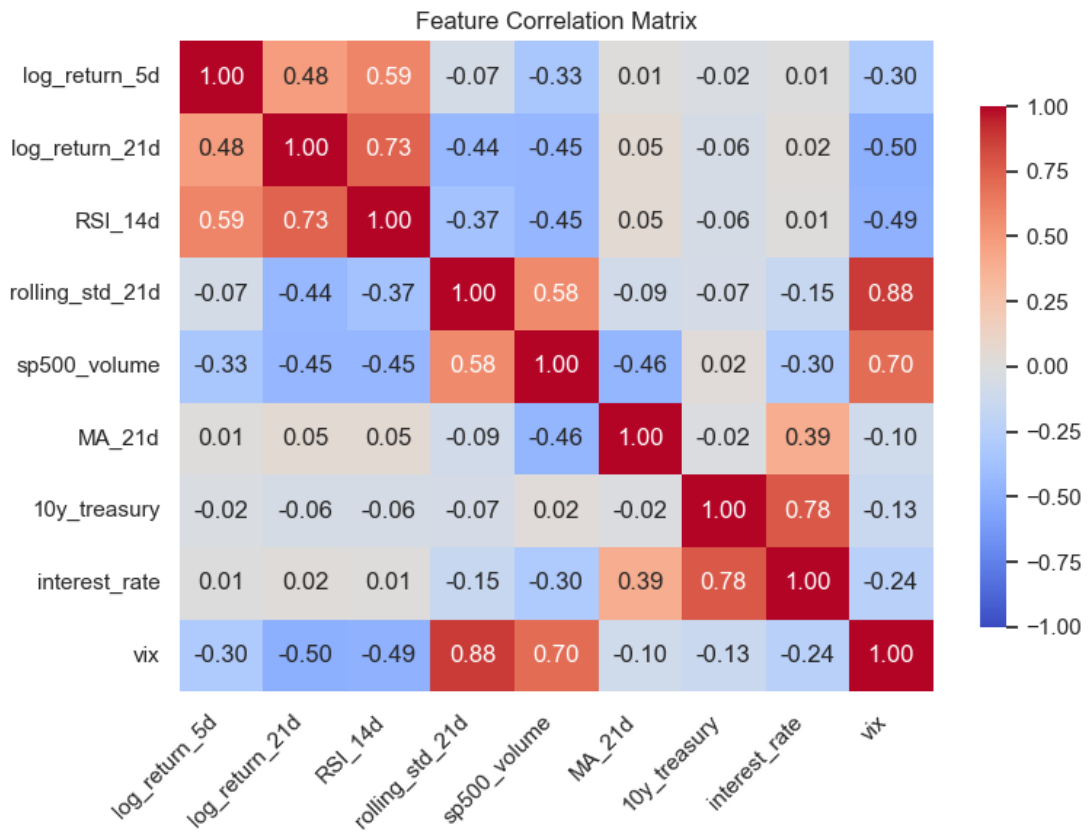
	start_date	mu	sigma
0	2005-05-24	-0.000396	0.004231
1	2005-06-01	0.000965	0.005052
2	2005-06-08	0.001108	0.002936
3	2005-06-15	0.001605	0.002619
4	2005-06-22	-0.001994	0.006939

In [ ]:

```
# === 1. Two-panel time-series plot: SP500 Close + MA21d and Rol
fig, axes = plt.subplots(2, 1, figsize=(12, 6), sharex=True)

# Top panel: SP500 close price and 21-day moving average
axes[0].plot(df.index, df["sp500_close"], label="S&P 500 Close",
axes[0].plot(df.index, df["MA_21d"], label="21-Day MA", linestyl
axes[0].set_ylabel("Price (USD)")
axes[0].set_title("S&P 500 Closing Price vs. 21-Day Moving Avera
axes[0].legend(loc="upper left")

# Bottom panel: 21-day rolling volatility
axes[1].plot(df.index, df["rolling_std_21d"], label="21-Day Roll
axes[1].set_ylabel("Volatility (Std. Dev.)")
axes[1].set_title("21-Day Rolling Volatility of S&P 500 Returns'
axes[1].legend(loc="upper left")


# === 2. Improve x-axis formatting for “finer” date ticks ===
# Choose a locator—e.g., one tick per year:
year_locator = mdates.YearLocator()       # major tick every Jar
year_formatter = mdates.DateFormatter('%Y')  # format ticks as '

# (If you wanted quarterly ticks, you could use QuarterLocator(t
for ax in axes:
    ax.xaxis.set_major_locator(year_locator)
    ax.xaxis.set_major_formatter(year_formatter)
    # Optionally add minor ticks every 6 months:
    ax.xaxis.set_minor_locator(mdates.MonthLocator(bymonth=(1, 7
    # Rotate labels on the bottom panel only (since sharex=True,
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right")

# Shared x-axis label
axes[-1].set_xlabel("Date")
plt.tight_layout()
plt.show()


# === 3. Feature Correlation Heatmap ===
# Compute correlation matrix from your standardized feature Data
corr_matrix = X_scaled.corr()
```

```
plt.figure(figsize=(8, 6))
sns.heatmap(
    corr_matrix,
    annot=True,
    fmt=".2f",
    cmap="coolwarm",
    vmin=-1,
    vmax=1,
    cbar_kws={"shrink": 0.8}
)
plt.title("Feature Correlation Matrix")
plt.xticks(rotation=45, ha="right")
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()
```

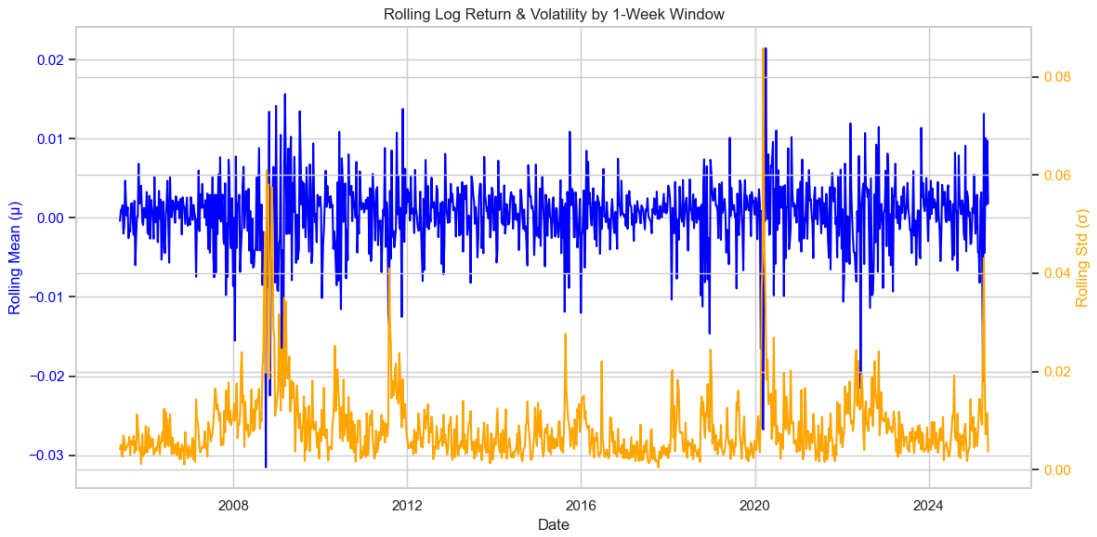




- Use **21-day moving averages** for trend tracking.
- Monitor **volatility spikes** to anticipate risk.
- Combine price and volatility insights for **informed decision-making**.

In [ ]:
```
# Plot the evolution of rolling log return (μ) and rolling volat
# Uses dual y-axes: μ on the left, σ on the right
fig, ax1 = plt.subplots(figsize=(12, 6))
x_dates = df_simple["start_date"]

# Left y-axis: rolling mean return (μ)
ax1.plot(x_dates, df_simple["mu"], color="blue", label="Rolling
ax1.set_ylabel("Rolling Mean (μ)", color="blue")
ax1.tick_params(axis='y', labelcolor="blue")

# Right y-axis: rolling standard deviation (σ)
ax2 = ax1.twinx()
ax2.plot(x_dates, df_simple["sigma"], color="orange", label="Rol
ax2.set_ylabel("Rolling Std (σ)", color="orange")
ax2.tick_params(axis='y', labelcolor="orange")

# Title and grid
ax1.set_title("Rolling Log Return & Volatility by 1-Week Window'
ax1.set_xlabel("Date")
ax1.grid(True)

plt.tight_layout()
```

```
plt.show()
```



This plot shows how the market's return (μ) and volatility (σ) evolved over time using a 1-week rolling window. Sharp drops in μ often coincide with spikes in σ, highlighting periods of high uncertainty such as the 2008 financial crisis or the 2020 COVID shock.Visualizing these together reveals how periods of stability and turbulence alternate over the dataset.

Step 2. Cluster market regimes using rolling μ and σ

Using the rolling mean (μ) and standard deviation (σ) computed in Step 1, we apply KMeans clustering to identify underlying market regimes. To determine the optimal number of clusters, we evaluate performance using both the Elbow Method (inertia) and the Silhouette Score across a range of cluster values (k = 5 to 21). These regimes are visualized both over time (plotted against the 3-week average return) and in the (μ, σ) space to assess the separation and stability of the clusters. Finally, we use the best-performing k to assign a single market regime label to each 5-day window, which will be used for later modeling and interpretation.

In [ ]:
```
# # Winsorization

# from scipy.stats.mstats import winsorize

# # Winsorization parameters
# lower_limit = 0.05   # bottom 5%
# upper_limit = 0.05   # top 5%

# df_simple_winsor = df_simple.copy()

# # Apply Winsorization to 'mu' and 'sigma'
# for col in ["mu", "sigma"]:
```

```
#     df_simple_winsor[col] = winsorize(df_simple[col], limits=(

# for col in ["mu", "sigma"]:
#     original_min, original_max = df_simple[col].min(), df_simp
#     winsor_min, winsor_max = df_simple_winsor[col].min(), df_s
#     print(f"{col}:\n  Original range: {original_min:.4f} to {o
#     print(f"  Winsorized range: {winsor_min:.4f} to {winsor_ma
```

In this project, we use rolling mean (`mu`) and standard deviation (`sigma`) of financial returns as inputs for KMeans clustering to identify market regimes. While Winsorization is a common technique for mitigating the influence of extreme outliers, we chose not to apply it. This decision is based on the fact that we are working with only two features, which are derived from rolling windows and tend to be statistically stable and smooth. Visual inspection showed no severe outliers, and both `mu` and `sigma` are robust against short-term fluctuations. Moreover, in financial time series, extreme values often represent meaningful market events—such as crashes or rallies—that are important to capture rather than suppress. Applying Winsorization could mask these signals and reduce the effectiveness of our clustering. However, if we were working in a higher-dimensional feature space—for example, including skewness, kurtosis, or momentum indicators—Winsorization would be more relevant, as outliers in one dimension can disproportionately affect clustering results. In summary, with our clean and interpretable low-dimensional data, Winsorization is unnecessary and could even be counterproductive, though it remains useful in more complex modeling contexts.

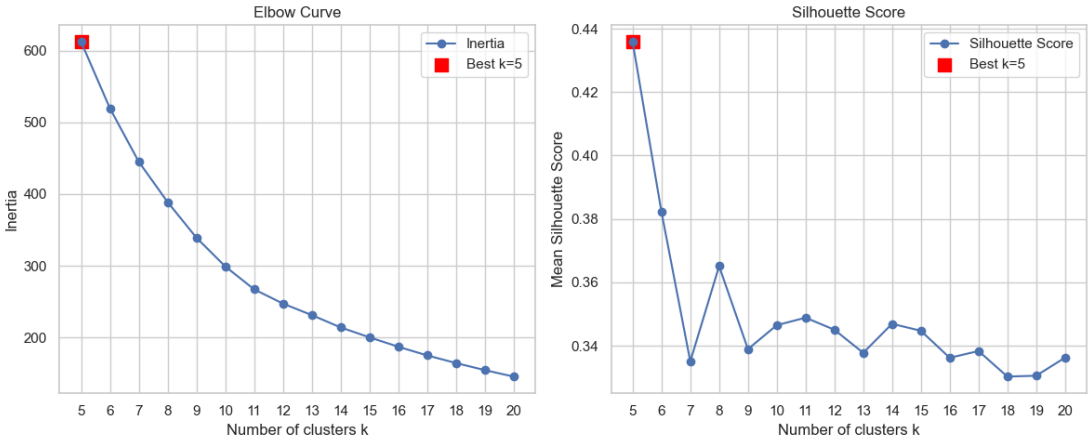
In [ ]:
```
#  Extract rolling mean (μ) and standard deviation (σ) values fo
X_mu_sigma = df_simple[["mu", "sigma"]].values
# Standardize features to ensure both μ and σ have equal weight
X_mu_sigma_scaled = StandardScaler().fit_transform(X_mu_sigma)

# Use KMeans to cluster the (μ, σ) pairs
# Try different numbers of clusters (k) from 7 to 20 to find the
Ks = range(5, 21)
inertias = [] # Inertia (within-cluster sum of squares) for elbo
sil_scores = [] # Silhouette score to evaluate cluster separatio

# Fit KMeans for each k and record performance metrics
for k in Ks:
    km = KMeans(n_clusters=k, n_init=10, random_state=42).fit(X_
    inertias.append(km.inertia_)
    sil_scores.append(silhouette_score(X_mu_sigma_scaled, km.lab

# Identify the optimal number of clusters (k)
best_idx = np.argmax(sil_scores)
k_best = Ks[best_idx]
```

```
print(f"Best k by silhouette score: {k_best}")
```

```
Best k by silhouette score: 5
```

The Elbow Curve above plots the inertia (within-cluster variance) against different values of k. The plot shows a clear elbow at k = 5, suggesting that using 5 clusters strikes a good balance between model simplicity and explanatory power. This value is selected as the optimal number of market regimes. The Silhouette Score plot evaluates how well-separated the clusters are for different values of k. The highest silhouette score occurs at k = 5, supporting it as the optimal choice for capturing distinct market regimes.

In [ ]:
```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Plot the Elbow Curve to visualize how inertia (within-cluster
# The 'elbow' point indicates diminishing returns and helps ider
# Elbow Curve (subplot 1)
axes[0].plot(Ks, inertias, marker='o', label='Inertia')
axes[0].scatter([k_best], [inertias[best_idx]], s=100, marker='s
axes[0].set_title('Elbow Curve')
axes[0].set_xlabel('Number of clusters k')
axes[0].set_ylabel('Inertia')
axes[0].set_xticks(Ks)
axes[0].legend()

# Plot the Silhouette Score curve to evaluate how well-separated
# Higher silhouette scores indicate better-defined and more cohe
# The red marker highlights the k value with the highest silhoue
# Silhouette Score vs. k (subplot 2)
axes[1].plot(Ks, sil_scores, marker='o', label='Silhouette Score
axes[1].scatter([k_best], [sil_scores[best_idx]], s=100, marker=
axes[1].set_title('Silhouette Score')
axes[1].set_xlabel('Number of clusters k')
axes[1].set_ylabel('Mean Silhouette Score')
axes[1].set_xticks(Ks)
axes[1].legend()

plt.tight_layout()
plt.show()
```



In [ ]:
```
# Perform final KMeans clustering using the optimal number of cl
# This produces the final regime labels for each 5-day window
final_kmeans = KMeans(n_clusters=k_best, n_init=10, random_state

# Assign the final regime labels to the dataframe
df_simple["regime_best"] = final_kmeans.fit_predict(X_mu_sigma_s
# Display final dataset with regime assignments and descriptive
df_simple
```

Out[ ]:

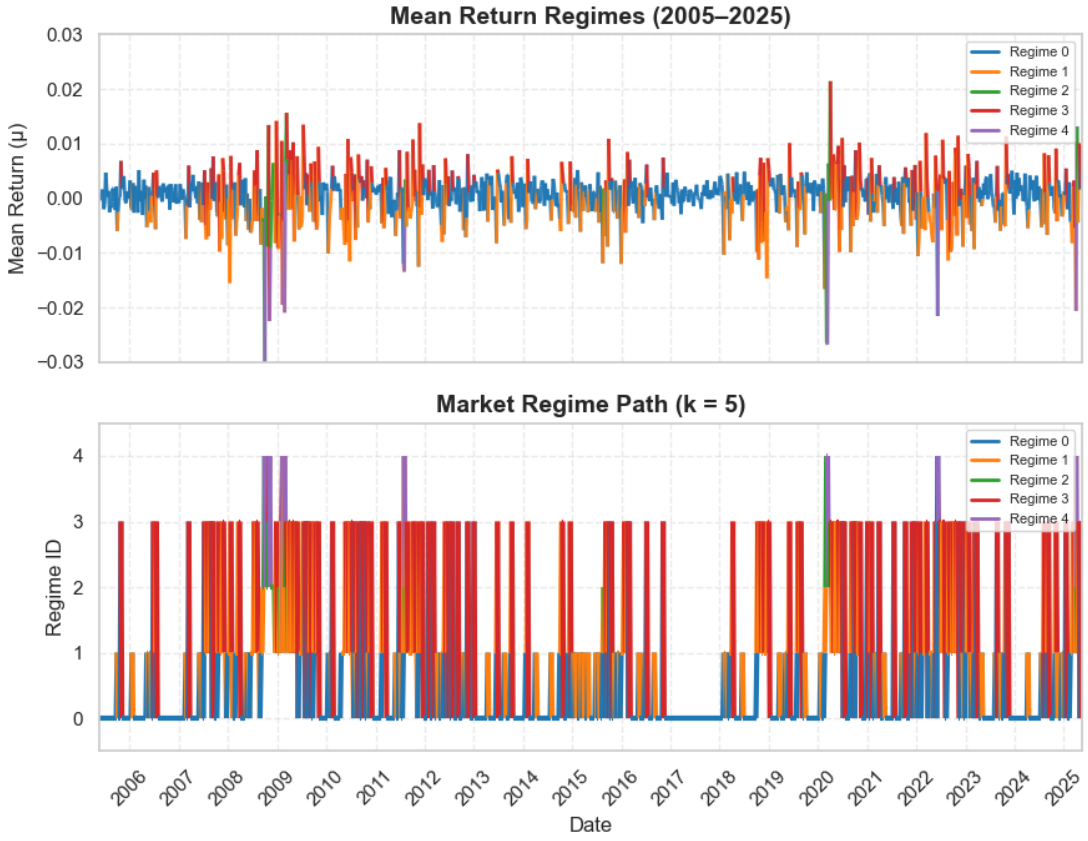
	start_date	mu	sigma	regime_best
0	2005-05-24	-0.000396	0.004231	0
1	2005-06-01	0.000965	0.005052	0
2	2005-06-08	0.001108	0.002936	0
3	2005-06-15	0.001605	0.002619	0
4	2005-06-22	-0.001994	0.006939	0
...	...	...	...	...
1001	2025-04-15	-0.004422	0.017915	1
1002	2025-04-23	0.010071	0.007156	3
1003	2025-04-30	0.001650	0.008275	0
1004	2025-05-07	0.009734	0.011469	3
1005	2025-05-14	0.001823	0.003634	0

1006 rows × 4 columns

In [ ]:
```
# Convert dates to numeric format
x_num = date2num(df_simple["start_date"])
y_values = df_simple["mu"].to_numpy()
regimes = df_simple["regime_best"].to_numpy()

# Prepare colors for the mean return plot segments
```

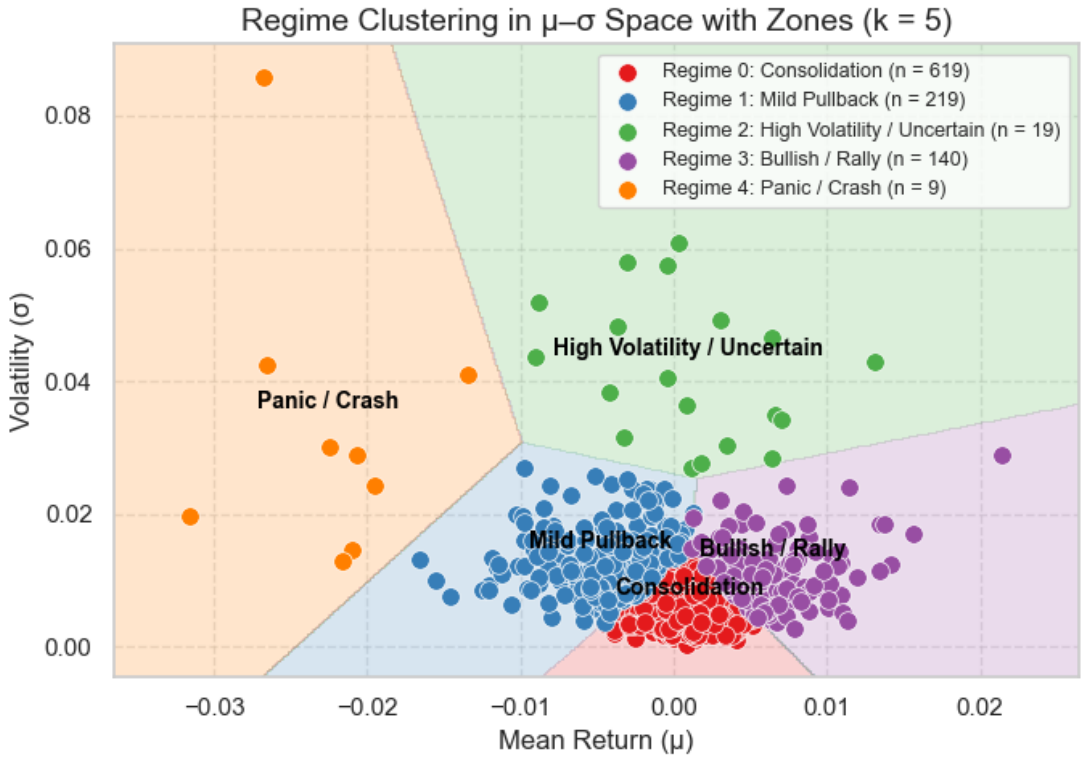
```
cmap = plt.get_cmap("tab10")
regime_colors_mean = [cmap(r % 10) for r in regimes[:-1]]

# Build line segments for the mean return plot
points_mean = np.array(list(zip(x_num, y_values))).reshape(-1, 1
segments_mean = np.concatenate([points_mean[:-1], points_mean[1:
lc_mean = LineCollection(segments_mean, colors=regime_colors_mea

# Prepare the regime path plot data
y_regime = regimes
regime_colors_path = [cmap(r % 10) for r in y_regime[:-1]]
points_path = np.array(list(zip(x_num, y_regime))).reshape(-1, 1
segments_path = np.concatenate([points_path[:-1], points_path[1:
lc_path = LineCollection(segments_path, colors=regime_colors_pat

# Create a 2x1 subplot
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(9, 7), sharex=True

# ---- Subplot 1: Mean Return Regimes ----
ax1.add_collection(lc_mean)
ax1.set_xlim(x_num.min(), x_num.max())
ax1.set_ylim(-0.03, 0.03)  # adjust as needed
ax1.set_title("Mean Return Regimes (2005–2025)", fontsize=14, fo
ax1.set_ylabel("Mean Return (μ)", fontsize=12)
ax1.grid(True, linestyle='--', alpha=0.4)
ax1.xaxis.set_major_locator(YearLocator())
ax1.xaxis.set_minor_locator(MonthLocator(bymonth=[6]))
ax1.xaxis.set_major_formatter(DateFormatter('%Y'))
ax1.tick_params(axis='x', rotation=45)

# Legend for regime colors in mean return plot
unique_regs = np.unique(regimes)
legend_lines = [
    Line2D([0], [0], color=cmap(r % 10), lw=2, label=f"Regime {r
    for r in unique_regs
]
ax1.legend(handles=legend_lines, loc='upper right', fontsize=8)

# ---- Subplot 2: Market Regime Path ----
ax2.add_collection(lc_path)
ax2.set_xlim(x_num.min(), x_num.max())
ax2.set_ylim(y_regime.min() - 0.5, y_regime.max() + 0.5)
ax2.set_title(f"Market Regime Path (k = {k_best})", fontsize=14,
ax2.set_ylabel("Regime ID", fontsize=12)
ax2.set_xlabel("Date", fontsize=12)
ax2.set_yticks(np.sort(unique_regs))
ax2.grid(True, linestyle='--', alpha=0.4)
ax2.xaxis.set_major_locator(YearLocator())
ax2.xaxis.set_minor_locator(MonthLocator(bymonth=[6]))
ax2.xaxis.set_major_formatter(DateFormatter('%Y'))
ax2.tick_params(axis='x', rotation=45)
```

```
# Add the same legend to the second plot
ax2.legend(handles=legend_lines, loc='upper right', fontsize=8)

plt.tight_layout()
plt.show()
```



In [ ]:
```
# Regime clustering (detection) plot in μ–σ space

sns.set_theme(style="whitegrid", context="notebook")

X_mu_sigma = df_simple[["mu", "sigma"]].values
scaler = StandardScaler()
X_scaled_new = scaler.fit_transform(X_mu_sigma)

kmeans = KMeans(n_clusters=k_best, n_init=10, random_state=42).f
df_simple["regime_best"] = kmeans.labels_
centers = scaler.inverse_transform(kmeans.cluster_centers_)

regime_labels_en = [
    "Consolidation",
    "Mild Pullback",
    "High Volatility / Uncertain",
    "Bullish / Rally",
    "Panic / Crash"]
color_list = sns.color_palette("Set1", k_best)
counts = df_simple["regime_best"].value_counts().sort_index()

x_min, x_max = X_mu_sigma[:, 0].min() - 0.005, X_mu_sigma[:, 0].
```

```
y_min, y_max = X_mu_sigma[:, 1].min() - 0.005, X_mu_sigma[:, 1].
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500),
                     np.linspace(y_min, y_max, 500))
grid = np.c_[xx.ravel(), yy.ravel()]
grid_scaled = scaler.transform(grid)
Z = kmeans.predict(grid_scaled)
Z = Z.reshape(xx.shape)

plt.figure(figsize=(7,5))
plt.contourf(xx, yy, Z, alpha=0.2, levels=np.arange(k_best+1)-0.
for i in range(k_best):
    cluster_data = df_simple[df_simple["regime_best"] == i]
    plt.scatter(cluster_data["mu"], cluster_data["sigma"],
                color=color_list[i],
                label=f"Regime {i}: {regime_labels_en[i]} (n = {
                s=70, edgecolor='white', linewidth=0.5)
    x, y = centers[i]
    plt.text(x, y + 0.0025, regime_labels_en[i], fontsize=10, ha

plt.title(f"Regime Clustering in μ–σ Space with Zones (k = {k_be
plt.xlabel("Mean Return (μ)", fontsize=12)
plt.ylabel("Volatility (σ)", fontsize=12)
plt.grid(True, linestyle='--', alpha=0.5)
plt.legend(loc="best", fontsize=9)
plt.tight_layout()
plt.show()
```



The μ–σ regime clustering plot reveals a clear and intuitive structure that aligns with common financial market behavior. The overall trend moves diagonally from the upper-left to the lower-right, reflecting a natural

progression from high-risk, negative-return environments to low-risk, positive-return ones. In the top-left corner, we observe the Panic/Crash regime, characterized by deeply negative returns and extremely high volatility—typical of sudden market crashes or systemic shocks. Just below that, the High Volatility/Uncertain regime emerges, where returns are near zero but risk remains elevated, indicating a turbulent yet directionless market. Moving left to center, the Mild Pullback regime represents short-term corrections with slightly negative returns and moderate volatility. At the center-bottom, the Consolidation regime appears, marked by both low volatility and near-zero returns, reflecting indecisive, range-bound markets. Finally, the bottom-right quadrant captures the Bullish/Rally regime, where positive returns and relatively low volatility indicate strong investor confidence and upward trends. This layout makes intuitive sense: volatility tends to rise during extreme movements (both up and down), and stable trends are often accompanied by lower risk. The regime zones produced by clustering in μ–σ space thus provide an interpretable and data-driven segmentation of market behavior that mirrors theoretical expectations and empirical patterns.

In [ ]:
```
# Summarize the characteristics of each market regime
# Calculate the average return (μ) and volatility (σ) within eac

df_regime_stats = df_simple.groupby("regime_best")[["mu", "sigma
print(df_regime_stats)
```

```
                  mu     sigma
regime_best
3           0.006513  0.011082
0           0.001037  0.005304
2           0.000916  0.041532
1          -0.004781  0.012457
4          -0.022587  0.033351
```

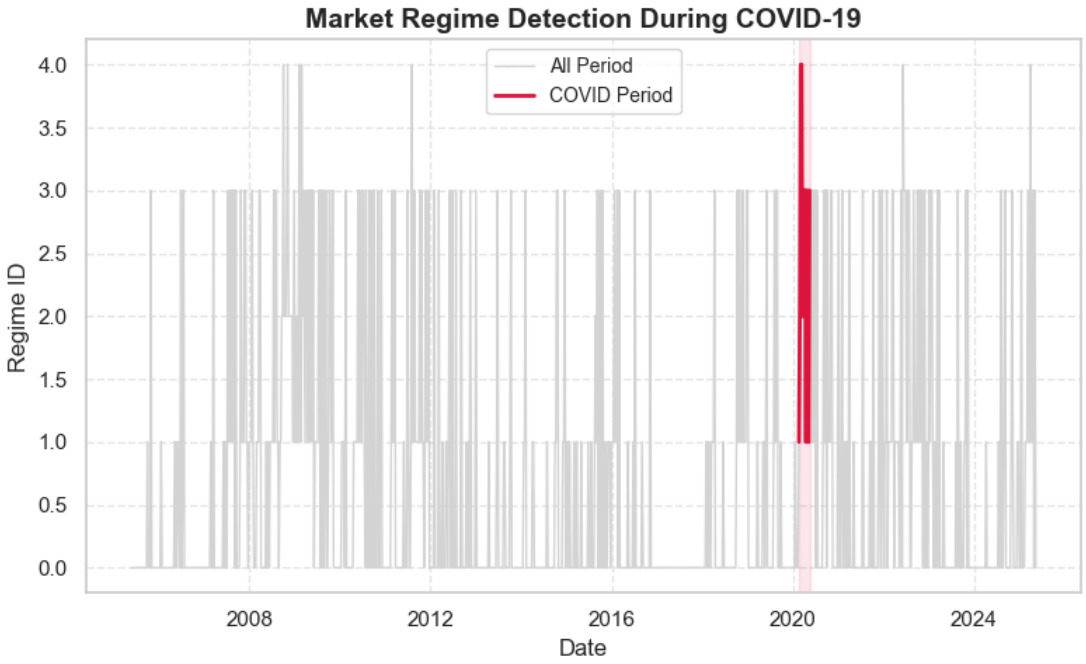
Based on the table and plot shown above, we can interpret the regimes as follows:

- **Regime 0**: Consolidation: μ = 0.001037, σ = 0.005304, n = 619. This regime reflects stable markets with low volatility and near-zero returns, indicating sideways price movement and investor indecision.
- **Regime 1**: Mild Pullback: μ = -0.004781, σ = 0.012457, n = 219. Slightly negative returns with moderate volatility suggest short-term corrections or minor downward movements within a larger trend.
- **Regime 2**: High Volatility / Uncertain: μ = 0.000916, σ = 0.041532, n =

19. High volatility with neutral returns implies a highly uncertain market environment, often driven by macroeconomic or geopolitical shocks.

- **Regime 3**: Bullish / Rally: μ = 0.006513, σ = 0.011082, n = 140. Positive returns with low volatility characterize a strong upward trend and market optimism.
- **Regime 4**: Panic / Crash: μ = -0.022587, σ = 0.033351, n = 9. Sharp negative returns and high volatility indicate market distress, panic selling, or systemic risk events.

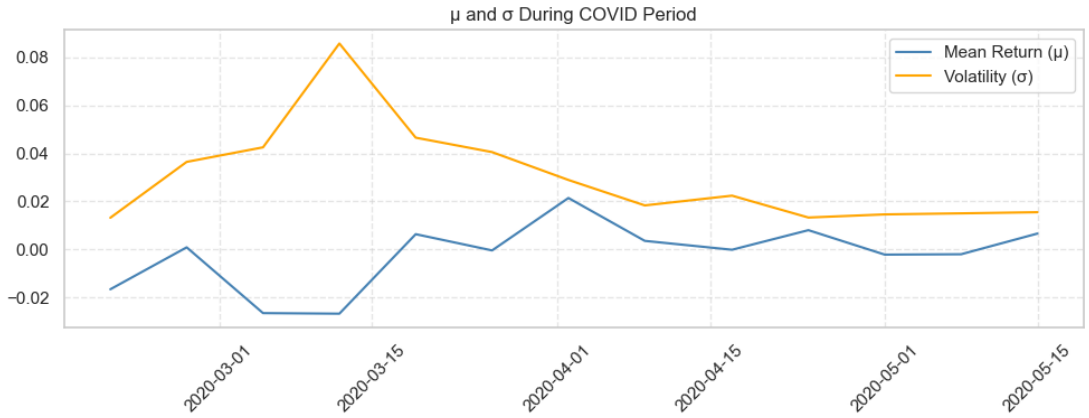
In [ ]:
```
# Step: Define COVID window
covid_start = pd.to_datetime("2020-02-15")
covid_end = pd.to_datetime("2020-05-15")
df_covid = df_simple[(df_simple["start_date"] >= covid_start) &

# Plot regime path with COVID highlighting
plt.figure(figsize=(8, 5))

# Full regime path (light gray)
plt.plot(df_simple["start_date"], df_simple["regime_best"], colo

# COVID regime path (highlighted)
plt.plot(df_covid["start_date"], df_covid["regime_best"], color=

# Add shaded COVID window
plt.axvspan(covid_start, covid_end, color='crimson', alpha=0.1)
plt.xlabel("Date", fontsize=12)
plt.ylabel("Regime ID", fontsize=12)
plt.title("Market Regime Detection During COVID-19", fontsize=14
plt.grid(True, linestyle='--', alpha=0.5)
plt.legend(fontsize=10)
plt.tight_layout()
plt.show()

# regime distribution during COVID
print("Regime Distribution During COVID Period:")
covid_counts = df_covid["regime_best"].value_counts().sort_index
for i, count in covid_counts.items():
    label = regime_labels_en[i]
    print(f"Regime {i}: {label:<30} — {count} windows")
```




```
Regime Distribution During COVID Period:
Regime 1: Mild Pullback                  — 4 windows
Regime 2: High Volatility / Uncertain    — 3 windows
Regime 3: Bullish / Rally                — 4 windows
Regime 4: Panic / Crash                  — 2 windows
```

In [ ]:
```
print("")
print(df_covid[["start_date", "mu", "sigma", "regime_best"]])
plt.figure(figsize=(10,4))
plt.plot(df_covid["start_date"], df_covid["mu"], label="Mean Ret
plt.plot(df_covid["start_date"], df_covid["sigma"], label="Volat
plt.legend()
plt.title("μ and σ During COVID Period")
plt.xticks(rotation=45)
plt.grid(True, linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
```

```
    start_date        mu     sigma  regime_best
742 2020-02-20 -0.016604  0.013189            1
743 2020-02-27  0.000879  0.036475            2
744 2020-03-05 -0.026522  0.042538            4
745 2020-03-12 -0.026757  0.085807            4
746 2020-03-19  0.006358  0.046569            2
747 2020-03-26 -0.000409  0.040558            2
748 2020-04-02  0.021435  0.028944            3
749 2020-04-09  0.003573  0.018347            3
750 2020-04-17 -0.000125  0.022403            1
751 2020-04-24  0.008031  0.013294            3
752 2020-05-01 -0.002157  0.014599            1
753 2020-05-08 -0.002002  0.015037            1
754 2020-05-15  0.006621  0.015510            3
```



Justifying the Choice of k=5 Through Economic Reasoning: Choosing the number of clusters (k) in unsupervised learning is challenging—statistical tools like the silhouette score offer helpful guidance, but they don't ensure economic interpretability. To address this, we validate our choice of k=5 using a real-world stress event: the COVID-19 market crash (2020-02-15 to 2020-05-15). During this period, our model correctly identifies the emergence of the "Panic / Crash" regime (Regime 4), aligning closely with the known market turmoil. This provides strong economic justification for selecting k = 5, demonstrating that the clustering structure not only fits the data statistically but also captures both normal and extreme market behaviors in a manner consistent with financial intuition.

Step 3. Define Supervised Learning Targets

To build predictive models, we define the target variable at time t+21 (approximately one month ahead) for each observation at time t.

The target can be:

- **μ_{t+21} (mean return)**: the average log return over the next window
- **σ_{t+21} (volatility)**: the standard deviation of returns over that same future window
- **Regime$_{t+21}$**: the cluster label (market regime) that corresponds to conditions at t+21

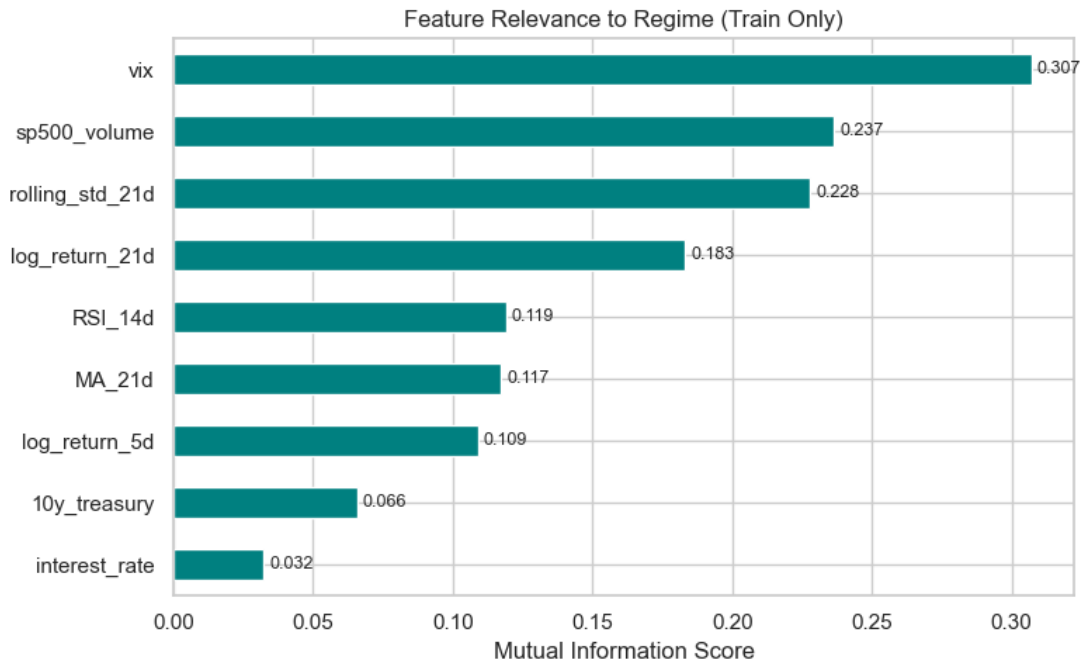
We use the **future regime label** as the prediction target, allowing us to train a supervised model that forecasts the market regime based on current market conditions.

In [ ]:
```
# Ensure datetime alignment across features and labels
df_simple["start_date"] = pd.to_datetime(df_simple["start_date"]
X_scaled.index = pd.to_datetime(X_scaled.index)
```

```
# Find common dates shared between the rolling stats and the sta
valid_dates = df_simple["start_date"][df_simple["start_date"].is

# Extract rows from both datasets corresponding to common dates
X_supervised = X_scaled.loc[valid_dates].copy()
y_supervised = df_simple.set_index("start_date").loc[valid_dates

# Combine features and labels into a unified training table
df_train = X_supervised.copy()
df_train["date"] = valid_dates.values
df_train["regime_label"] = y_supervised.values


# Split into training and testing sets (70% train, 30% test)
X = df_train.drop(columns=["date", "regime_label"])
y = df_train["regime_label"].astype(int)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_s
X_train

# Set global random seed for reproducibility
def reset_session(seed=42):
    tf.random.set_seed(seed)
    np.random.seed(seed)
    tf.keras.backend.clear_session()
```

In [ ]:

```
from sklearn.feature_selection import mutual_info_classif

# Compute mutual information between each feature and the target
mi_train = mutual_info_classif(X_train, y_train, random_state=42
mi_series_train = pd.Series(mi_train, index=X_train.columns).sor

# Plot the mutual information scores for each feature
plt.figure(figsize=(8, 5))
ax = mi_series_train.plot(kind='barh', color='teal')

# Add score labels next to each bar
for i, (value, name) in enumerate(zip(mi_series_train, mi_series
    ax.text(value + 0.002, i, f"{value:.3f}", va='center', fonts
plt.title("Feature Relevance to Regime (Train Only)")
plt.xlabel("Mutual Information Score")
plt.tight_layout()
plt.show()

# Select features with MI > 0.1 as relevant predictors
selected_features = mi_series_train[mi_series_train > 0.1].index
```



The mutual information analysis reveals that VIX, trading volume, and 21-day log returns are the most predictive features for market regime classification at a 1-month horizon. These indicators likely reflect market sentiment, volatility expectations, and price momentum — all critical for capturing regime dynamics. In contrast, macroeconomic indicators like interest rates show limited predictive power for short-term regime shifts.

In [ ]:
```
# Select only the most relevant features (MI > 0.1) for training
X_train_sel = X_train[selected_features]
X_test_sel = X_test[selected_features]
```

Based on mutual information analysis, we selected the top three features with the highest relevance to regime prediction: 21-day log return, S&P 500 trading volume, and VIX. These inputs are now used as the final features for supervised learning to forecast market regime labels.

Train Supervised Models to Predict Market Regimes

We use the selected price-derived features (e.g., log returns, volatility, trading volume, VIX) at day *t* to predict the market regime (cluster label) at *t + 21* (approximately one month ahead). This is formulated as a multi-class classification task, where the target is the regime cluster ID.

We train and evaluate several classification models:

- **Regime-switching model (Markov chain)**
- **Softmax Regression with Neural Network**
- **Random Forest**
- **Gradient Boosting**
- **Support Vector Machine (SVM)**

To further enhance model performance and robustness, we also experiment with ensemble approaches, combining the outputs of multiple models.

In [ ]:
```
# Get data set prepared
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_sel)
X_test_scaled = scaler.transform(X_test_sel)

# Same train and test set for everyone just to clarify we use
X_train_eh, X_test_eh, y_train_eh, y_test_eh = X_train_scaled,
X_train_lm, X_test_lm, y_train_lm, y_test_lm = X_train_scaled,
X_train_ss, X_test_ss, y_train_ss, y_test_ss = X_train_scaled,
X_train_rx, X_test_rx, y_train_rx, y_test_rx= X_train_scaled,
X_train_cl, X_test_cl, y_train_cl, y_test_cl = X_train_scaled,
```

Regime-switching model (Markov chain)

Description:

This **Regime-switching model using a Markov chain** is a straightforward way to capture the idea that markets move between a few identifiable states (for example, low-volatility/high-return or high-volatility/low-return regimes) rather than behaving homogeneously. In this approach, it labels each day with a regime and then estimates the probability of moving from today's regime i to tomorrow's regime j by simply counting **how often those transitions occurred** in the historical data.

Those probabilities **form a matrix** (one for each pair of regimes), which can be raised to a power (21 days ahead) to forecast the most likely regime in the future. This is easily computes just counts and divisions, allows to interpret as it shows exactly how likely each jump is, and serves as a low-parameter baseline against the other complex models.

In [ ]:
```
reset_session()
```

```
n_total = len(df_simple)
n_train = len(X_train_eh)
n_test  = len(X_test_eh)

# Extract the sequence of “current” regimes from distribution_
all_regimes = df_simple['regime_best'].to_numpy()

# regimes corresponding to the training window (first n_train
regimes_train = all_regimes[:n_train]

# regimes corresponding to the test window (next n_test rows)
regimes_test  = all_regimes[n_train : n_train + n_test]

# Ground-truth “future” regime_target at t+21 for test
y_true = y_test_eh

# Number of distinct regimes (k_best from k-means)
k_best = df_simple['regime_best'].nunique()
states = np.arange(k_best)

def estimate_transitions(labels, k):
    counts = np.zeros((k, k), dtype=int)
    for t in range(len(labels) - 1):
        i = labels[t]
        j = labels[t + 1]
        counts[i, j] += 1

    row_sums = counts.sum(axis=1)
    probs = np.zeros_like(counts, dtype=float)
    for i in range(k):
        if row_sums[i] > 0:
            probs[i, :] = counts[i, :] / row_sums[i]
        else:
            probs[i, :] = np.nan

    return counts, probs

counts_train, P_train = estimate_transitions(regimes_train, k_

idx = [f"from {i}" for i in range(k_best)]
cols = [f"to {j}"   for j in range(k_best)]

df_counts_train = pd.DataFrame(counts_train, index=idx, columns
df_P_train      = pd.DataFrame(P_train,      index=idx, columns

counts_test, P_test = estimate_transitions(regimes_test, k_bes

df_counts_test = pd.DataFrame(counts_test, index=idx, columns=
df_P_test      = pd.DataFrame(P_test,      index=idx, columns=

display(Markdown("#### Transition Probabilities for TRAINING S
```

```
display(df_P_train)

display(Markdown("#### Counts of Transitions for TRAINING Set"
display(df_counts_train)

display(Markdown("#### Transition Probabilities for TEST Set")
display(df_P_test)

display(Markdown("#### Counts of Transitions for TEST Set"))
display(df_counts_test)

print(f"Total transitions out of regime 0 (train): ", df_count
print(f"Total transitions out of regime 1 (train): ", df_count
print(f"Total transitions out of regime 0 (test): ", df_counts_
print(f"Total transitions out of regime 1 (test): ", df_counts_

regime_test_next    = regimes_test[1:]       # actual regime a
regime_test_current = regimes_test[:-1]      # regime at t

y1_true = regime_test_next
y1_pred = np.array([np.nan if np.isnan(P_test[s]).all() else n

valid_mask = ~np.isnan(y1_pred)
y1_true = y1_true[valid_mask].astype(int)
y1_pred = y1_pred[valid_mask].astype(int)
#print("Prediction of the test data:", y1_pred)

acc1 = accuracy_score(y1_true, y1_pred)
print(f"\nOne-step Markov 1-day accuracy on TEST: {acc1:.6f}")

cm1 = confusion_matrix(y1_true, y1_pred, labels=states)

# Calculate difference matrix for the bottom-right subplot
diff = P_test - P_train

# Create 2x2 subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

# Subplot (0,0): force a square-cell heatmap
ax_cm = axes[0, 0]
im_cm = ax_cm.imshow(cm1, aspect='equal', cmap='Blues')
ax_cm.set_title('1-Step Markov Confusion Matrix (Test)')
ax_cm.set_xlabel('Predicted Next Regime')
ax_cm.set_ylabel('Actual Next Regime')
ax_cm.set_xticks(states)
ax_cm.set_yticks(states)
for i in states:
    for j in states:
        ax_cm.text(j, i, cm1[i, j], ha='center', va='center',
                   color=('white' if cm1[i, j] > cm1.max()/2 e
cbar_cm = fig.colorbar(im_cm, ax=ax_cm, fraction=0.046, pad=0.
cbar_cm.set_label('Probability')
```

```
# Subplot (0,1): also force square-cell
ax_train = axes[0, 1]
im_train = ax_train.imshow(P_train, aspect='equal', cmap='Blue
ax_train.set_title('Train Transition Probability')
ax_train.set_xlabel('To Regime (j)')
ax_train.set_ylabel('From Regime (i)')
ax_train.set_xticks(states)
ax_train.set_yticks(states)
for i in states:
    for j in states:
        val = P_train[i, j]
        if not np.isnan(val):
            ax_train.text(j, i, f"{val:.2f}", ha='center', va=
                          color=('white' if val > 0.5 else 'bl
cbar_train = fig.colorbar(im_train, ax=ax_train, fraction=0.04
cbar_train.set_label('Probability')

# ---- Subplot (1,0): Test Transition Probability Heatmap ----
ax_test = axes[1, 0]
im_test = ax_test.imshow(P_test, cmap='Blues', vmin=0, vmax=1)
ax_test.set_title('Test Transition Probability')
ax_test.set_xlabel('To Regime (j)')
ax_test.set_ylabel('From Regime (i)')
ax_test.set_xticks(states)
ax_test.set_yticks(states)
for i in states:
    for j in states:
        val = P_test[i, j]
        if not np.isnan(val):
            color = 'white' if val > 0.5 else 'black'
            ax_test.text(j, i, f"{val:.2f}", ha='center', va='
cbar_test = fig.colorbar(im_test, ax=ax_test, fraction=0.046, 
cbar_test.set_label('Probability')

# ---- Subplot (1,1): Difference Heatmap (Test - Train) ----
ax_diff = axes[1, 1]
im_diff = ax_diff.imshow(diff, cmap='RdBu', vmin=-1, vmax=1)
ax_diff.set_title('Difference: Test - Train Probability')
ax_diff.set_xlabel('To Regime (j)')
ax_diff.set_ylabel('From Regime (i)')
ax_diff.set_xticks(states)
ax_diff.set_yticks(states)
for i in states:
    for j in states:
        val = diff[i, j]
        if not np.isnan(val):
            color = 'white' if abs(val) > 0.5 else 'black'
            ax_diff.text(j, i, f"{val:.2f}", ha='center', va='
cbar_diff = fig.colorbar(im_diff, ax=ax_diff, fraction=0.046, 
cbar_diff.set_label('Difference')
```

```
plt.tight_layout()
plt.show()


dates_test = df_simple['start_date'].iloc[n_train+1 : n_train+
regimes_actual_ts = y1_true
regimes_pred_ts = y1_pred

fig3, ax3 = plt.subplots(figsize=(12, 4))
ax3.plot(dates_test, regimes_actual_ts, label='Actual Regime (
ax3.plot(dates_test, regimes_pred_ts, label='Predicted Regime
ax3.set_title('Test Set: Actual vs Predicted 1-Step Regime')
ax3.set_xlabel('Date')
ax3.set_ylabel('Regime ID')
ax3.xaxis.set_major_locator(YearLocator())
ax3.xaxis.set_major_formatter(DateFormatter('%Y'))
ax3.legend()
ax3.grid(True, linestyle='--', alpha=0.5)
plt.xticks(rotation=45)

plt.tight_layout()

# Display all figures
plt.show()
```

Transition Probabilities for TRAINING Set

	to 0	to 1	to 2	to 3	to 4
from 0	0.809843	0.138702	0.000000	0.051454	0.000000
from 1	0.290541	0.351351	0.020270	0.324324	0.013514
from 2	0.000000	0.333333	0.466667	0.133333	0.066667
from 3	0.500000	0.333333	0.011905	0.130952	0.023810
from 4	0.000000	0.200000	0.800000	0.000000	0.000000

Counts of Transitions for TRAINING Set

	to 0	to 1	to 2	to 3	to 4
from 0	362	62	0	23	0
from 1	43	52	3	48	2
from 2	0	5	7	2	1
from 3	42	28	1	11	2
from 4	0	1	4	0	0

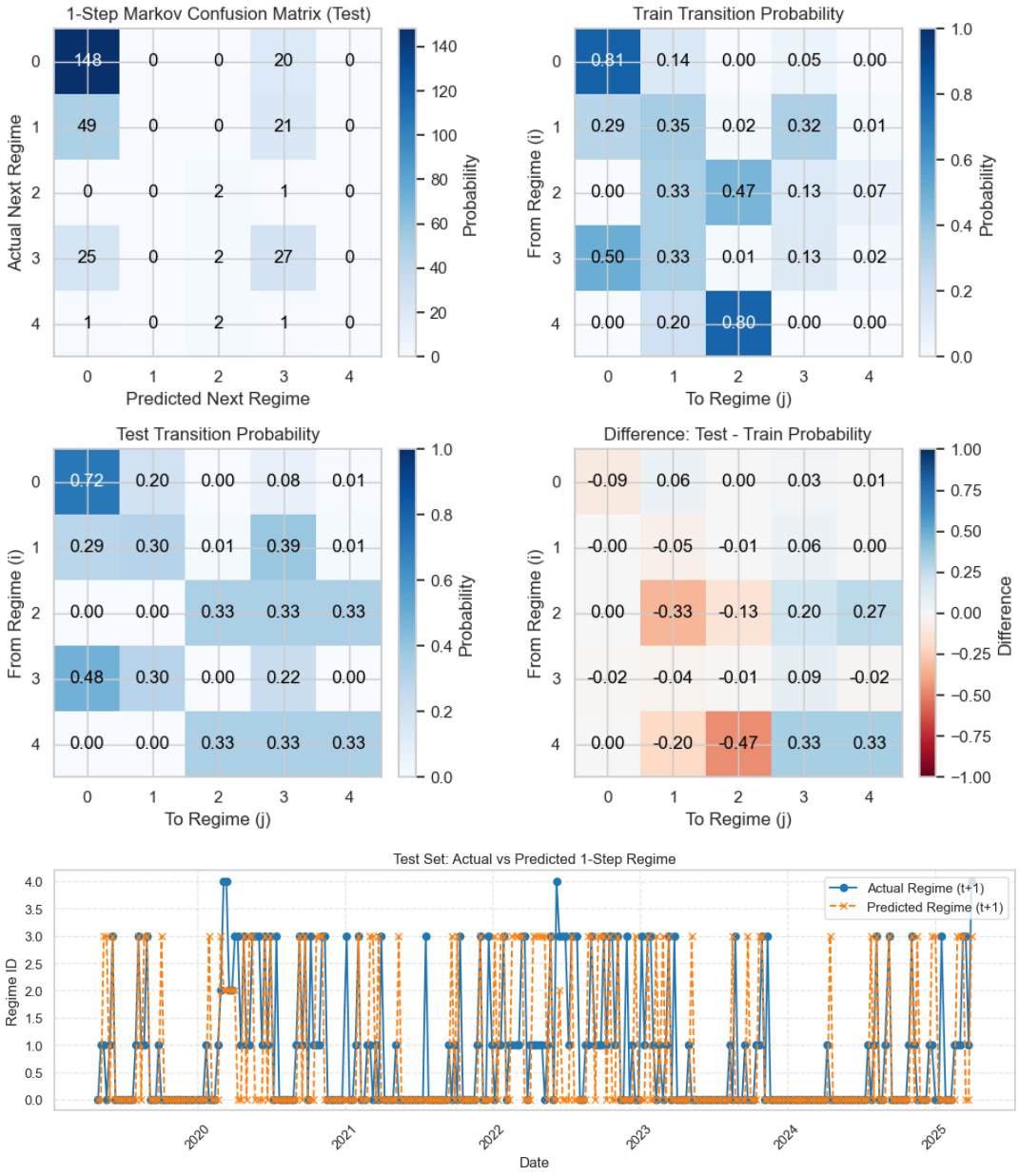
Transition Probabilities for TEST Set

	to 0	to 1	to 2	to 3	to 4
from 0	0.721893	0.195266	0.000000	0.076923	0.005917
from 1	0.285714	0.300000	0.014286	0.385714	0.014286
from 2	0.000000	0.000000	0.333333	0.333333	0.333333
from 3	0.481481	0.296296	0.000000	0.222222	0.000000
from 4	0.000000	0.000000	0.333333	0.333333	0.333333

Counts of Transitions for TEST Set

	to 0	to 1	to 2	to 3	to 4
from 0	122	33	0	13	1
from 1	20	21	1	27	1
from 2	0	0	1	1	1
from 3	26	16	0	12	0
from 4	0	0	1	1	1

```
Total transitions out of regime 0 (train):  447
Total transitions out of regime 1 (train):  148
Total transitions out of regime 0 (test):  169
Total transitions out of regime 1 (test):  70

One-step Markov 1-day accuracy on TEST: 0.591973
```



Softmax Regression with Neural Network

Description:

The model is a **softmax regression neural network implemented using Keras**, designed for multi-class classification of market regimes. It consists of an input layer matching the number of standardized financial features, followed by a dense hidden layer with 64 neurons and ReLU activation to capture non-linear patterns.

We used a dropout layer with a rate of 0.3 for regularization to prevent overfitting. The output layer uses a softmax activation function to produce class probabilities across multiple regime labels. The model is compiled with sparse categorical cross-entropy loss and optimized

using the Adam optimizer with a learning rate of 0.001. **To account for class imbalance,** class weights are computed and applied during training. An early stopping mechanism is used to monitor validation loss and restore the best model weights, ensuring stable and efficient convergence.

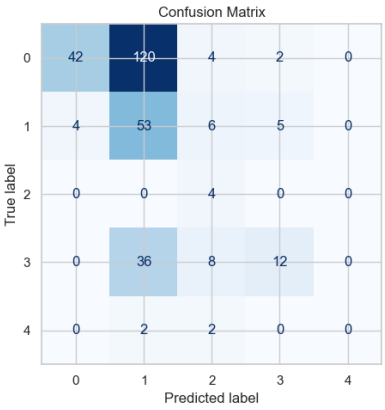
In [ ]:
```
reset_session()

num_classes = len(np.unique(y_train))
class_weights = class_weight.compute_class_weight(
    class_weight='balanced',
    classes=np.unique(y_train),
    y=y_train)
class_weight_dict = dict(enumerate(class_weights))
print("Class Weights:", class_weight_dict)

keras.backend.clear_session()
model = keras.Sequential([
    layers.Input(shape=(X_train_scaled.shape[1],)),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='softmax')])

model.compile(loss='sparse_categorical_crossentropy', optimize

early_stop = EarlyStopping(monitor='val_loss', patience=5, res
history = model.fit(X_train_scaled, y_train, validation_split=
test_loss, test_acc = model.evaluate(X_test_scaled, y_test, ve
print(f"Test Accuracy: {test_acc:.3f}")

# Generate predictions for the confusion matrix
y_pred_probs = model.predict(X_test_scaled)
y_pred_classes = np.argmax(y_pred_probs, axis=1)

# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred_classes)
# Create 1x2 subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# ---- Subplot 1: Training vs Validation Accuracy ----
ax1.plot(history.history['accuracy'], label='Train Accuracy')
ax1.plot(history.history['val_accuracy'], label='Validation Ac
ax1.set_title("Training vs Validation Accuracy")
ax1.set_xlabel("Epoch")
ax1.set_ylabel("Accuracy")
ax1.legend()
ax1.grid(True)

# ---- Subplot 2: Confusion Matrix ----
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(cmap="Blues", ax=ax2, colorbar=False)
```

```
ax2.set_title("Confusion Matrix")

plt.tight_layout()
plt.show()
print("Classification Report:")
print(classification_report(y_test, y_pred_classes, digits=3,
```

```
Class Weights: {0: np.float64(0.3146067415730337), 1: np.float6
4(0.9271523178807947), 2: np.float64(9.333333333333334), 3: np.
float64(1.6666666666666667), 4: np.float64(28.0)}
Epoch 1/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.1357 - los
s: 1.9509 - val_accuracy: 0.1571 - val_loss: 1.7232
Epoch 2/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.1774 - los
s: 1.7905 - val_accuracy: 0.3357 - val_loss: 1.5709
Epoch 3/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3010 - los
s: 1.7151 - val_accuracy: 0.5929 - val_loss: 1.4366
Epoch 4/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3907 - los
s: 1.7214 - val_accuracy: 0.7143 - val_loss: 1.3133
Epoch 5/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4401 - los
s: 1.7448 - val_accuracy: 0.7500 - val_loss: 1.2054
Epoch 6/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5046 - los
s: 1.3786 - val_accuracy: 0.7571 - val_loss: 1.1069
Epoch 7/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5244 - los
s: 1.5142 - val_accuracy: 0.7714 - val_loss: 1.0230
Epoch 8/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5494 - los
s: 1.2598 - val_accuracy: 0.7857 - val_loss: 0.9509
Epoch 9/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5663 - los
s: 1.4466 - val_accuracy: 0.7857 - val_loss: 0.8953
Epoch 10/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5629 - los
s: 1.2609 - val_accuracy: 0.7929 - val_loss: 0.8574
Epoch 11/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5431 - los
s: 1.2792 - val_accuracy: 0.7714 - val_loss: 0.8160
Epoch 12/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5781 - los
s: 1.2600 - val_accuracy: 0.7857 - val_loss: 0.7859
Epoch 13/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5613 - los
s: 1.2033 - val_accuracy: 0.7857 - val_loss: 0.7672
Epoch 14/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5554 - los
s: 1.3204 - val_accuracy: 0.7643 - val_loss: 0.7536
Epoch 15/50
```

```
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5806 - los
s: 1.2866 - val_accuracy: 0.7643 - val_loss: 0.7376
Epoch 16/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5791 - los
s: 1.0667 - val_accuracy: 0.7714 - val_loss: 0.7221
Epoch 17/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5718 - los
s: 1.3224 - val_accuracy: 0.7571 - val_loss: 0.7148
Epoch 18/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5916 - los
s: 1.1879 - val_accuracy: 0.7643 - val_loss: 0.7003
Epoch 19/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5658 - los
s: 1.1175 - val_accuracy: 0.7643 - val_loss: 0.6898
Epoch 20/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5525 - los
s: 1.2579 - val_accuracy: 0.7643 - val_loss: 0.6863
Epoch 21/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5809 - los
s: 1.0780 - val_accuracy: 0.7643 - val_loss: 0.6804
Epoch 22/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5519 - los
s: 1.0814 - val_accuracy: 0.7643 - val_loss: 0.6789
Epoch 23/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5644 - los
s: 1.0537 - val_accuracy: 0.7571 - val_loss: 0.6733
Epoch 24/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5454 - los
s: 1.1552 - val_accuracy: 0.7571 - val_loss: 0.6679
Epoch 25/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5600 - los
s: 1.0698 - val_accuracy: 0.7571 - val_loss: 0.6635
Epoch 26/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5603 - los
s: 1.1249 - val_accuracy: 0.7643 - val_loss: 0.6617
Epoch 27/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5907 - los
s: 1.0947 - val_accuracy: 0.7571 - val_loss: 0.6581
Epoch 28/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5823 - los
s: 1.0219 - val_accuracy: 0.7571 - val_loss: 0.6503
Epoch 29/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5430 - los
s: 1.0018 - val_accuracy: 0.7571 - val_loss: 0.6473
Epoch 30/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5392 - los
s: 1.1434 - val_accuracy: 0.7571 - val_loss: 0.6495
Epoch 31/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5598 - los
s: 1.1896 - val_accuracy: 0.7643 - val_loss: 0.6462
Epoch 32/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5399 - los
```

```
s: 1.0380 - val_accuracy: 0.7571 - val_loss: 0.6481
Epoch 33/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5413 - los
s: 1.0163 - val_accuracy: 0.7571 - val_loss: 0.6508
Epoch 34/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5403 - los
s: 1.2330 - val_accuracy: 0.7571 - val_loss: 0.6448
Epoch 35/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5631 - los
s: 1.2665 - val_accuracy: 0.7786 - val_loss: 0.6434
Epoch 36/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5854 - los
s: 0.9924 - val_accuracy: 0.7643 - val_loss: 0.6416
Epoch 37/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5606 - los
s: 0.9667 - val_accuracy: 0.7571 - val_loss: 0.6416
Epoch 38/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5722 - los
s: 0.9472 - val_accuracy: 0.7643 - val_loss: 0.6363
Epoch 39/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5532 - los
s: 1.0658 - val_accuracy: 0.7571 - val_loss: 0.6338
Epoch 40/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5881 - los
s: 1.0116 - val_accuracy: 0.7571 - val_loss: 0.6310
Epoch 41/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5695 - los
s: 0.9475 - val_accuracy: 0.7643 - val_loss: 0.6295
Epoch 42/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5501 - los
s: 1.0403 - val_accuracy: 0.7643 - val_loss: 0.6279
Epoch 43/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5731 - los
s: 0.9286 - val_accuracy: 0.7643 - val_loss: 0.6246
Epoch 44/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5864 - los
s: 0.9644 - val_accuracy: 0.7643 - val_loss: 0.6273
Epoch 45/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5750 - los
s: 0.9717 - val_accuracy: 0.7643 - val_loss: 0.6269
Epoch 46/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5665 - los
s: 0.9481 - val_accuracy: 0.7643 - val_loss: 0.6260
Epoch 47/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5819 - los
s: 1.0247 - val_accuracy: 0.7571 - val_loss: 0.6261
Epoch 48/50
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5920 - los
s: 0.9530 - val_accuracy: 0.7571 - val_loss: 0.6246
Test Accuracy: 0.370
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
```




```
Classification Report:
              precision    recall  f1-score   support

           0      0.913     0.250     0.393       168
           1      0.251     0.779     0.380        68
           2      0.167     1.000     0.286         4
           3      0.632     0.214     0.320        56
           4      0.000     0.000     0.000         4

    accuracy                          0.370       300
   macro avg      0.392     0.449     0.276       300
weighted avg      0.688     0.370     0.369       300
```

Gradient Boosting

Desciption:

Gradient Boosting is a powerful ensemble method that **builds a strong classifier by combining many shallow decision trees,** where each tree corrects the errors of the previous one. It is well-suited for financial data because it can capture complex, nonlinear relationships while controlling for overfitting through parameters like learning rate, tree depth, and subsampling.

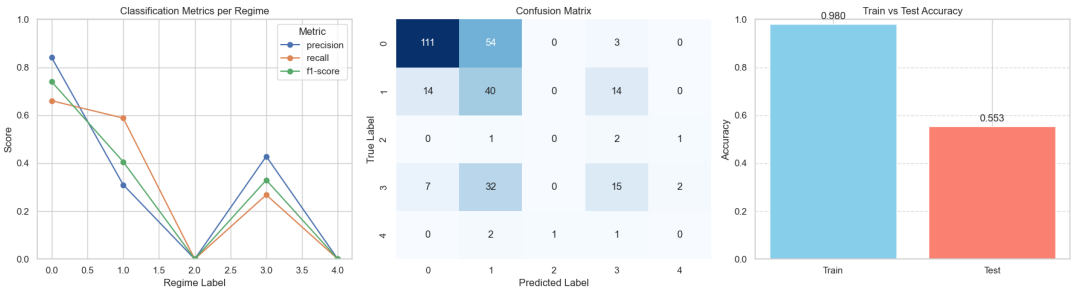
In our project, we used GradientBoostingClassifier within a pipeline that includes standard scaling and time-series cross-validation (TimeSeriesSplit) to respect the **chronological nature of financial data**. A grid search was used to tune hyperparameters and select the best-performing model.

We chose Gradient Boosting because it handles limited features well, provides good predictive performance, and offers some interpretability through feature importance. It serves as both a strong predictive model and a benchmark against which more complex models like neural networks or XGBoost can be compared.

In [ ]:
```
pipe_gb = Pipeline([
    ("scaler", StandardScaler()),
    ("gb", GradientBoostingClassifier(random_state=42))
])

param_grid_gb = {
    'gb__n_estimators': [100, 200],
    'gb__learning_rate': [0.1, 0.05],
    'gb__max_depth': [3, 5],
    'gb__subsample': [0.8, 1.0]
}

tscv = TimeSeriesSplit(n_splits=5)

grid_gb = GridSearchCV(
    pipe_gb,
    param_grid=param_grid_gb,
    cv=tscv,
    scoring='accuracy',
    n_jobs=-1,
    verbose=1,
    refit=True
)

grid_gb.fit(X_train_lm, y_train_lm)

print("Best GB parameters: ", grid_gb.best_params_)
print("Best CV accuracy: ", grid_gb.best_score_)

best_gb = grid_gb.best_estimator_
y_pred_lm = best_gb.predict(X_test_lm)

test_acc_lm = accuracy_score(y_test_lm, y_pred_lm)
test_f1_lm = f1_score(y_test_lm, y_pred_lm, average="macro")
train_acc_lm = accuracy_score(y_train_lm, best_gb.predict(X_tra

print("Train Accuracy:", train_acc_lm)
print("Test Accuracy:", test_acc_lm)
print("Test F1 (Macro):", test_f1_lm)
print("\nClassification report on test set:\n")
print(classification_report(y_test_lm, y_pred_lm, zero_divisio

report_dict_gb = classification_report(y_test_lm, y_pred_lm, ou
df_report_gb = pd.DataFrame(report_dict_gb).T
regime_metrics_gb = df_report_gb[df_report_gb.index.str.isnume
regime_metrics_gb.index = regime_metrics_gb.index.astype(int)
regime_metrics_gb.sort_index(inplace=True)

cm_gb = confusion_matrix(y_test_lm, y_pred_lm)
fig, axs = plt.subplots(1, 3, figsize=(18, 5))

regime_metrics_gb[['precision', 'recall', 'f1-score']].plot(ax=
```

```
axs[0].set_title("Classification Metrics per Regime")
axs[0].set_xlabel("Regime Label")
axs[0].set_ylabel("Score")
axs[0].set_ylim(0, 1)
axs[0].legend(title="Metric")
axs[0].grid(True)

sns.heatmap(cm_gb, annot=True, fmt='d', cmap='Blues', cbar=Fal
axs[1].set_title("Confusion Matrix")
axs[1].set_xlabel("Predicted Label")
axs[1].set_ylabel("True Label")

axs[2].bar(['Train', 'Test'], [train_acc_lm, test_acc_lm], col
axs[2].set_ylim(0, 1)
axs[2].set_title("Train vs Test Accuracy")
axs[2].set_ylabel("Accuracy")
for i, v in enumerate([train_acc_lm, test_acc_lm]):
    axs[2].text(i, v + 0.02, f"{v:.3f}", ha='center')
axs[2].grid(axis='y', linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits
Best GB parameters:  {'gb__learning_rate': 0.05, 'gb__max_dept
h': 3, 'gb__n_estimators': 200, 'gb__subsample': 0.8}
Best CV accuracy:  0.6137931034482759
Train Accuracy: 0.98
Test Accuracy: 0.5533333333333333
Test F1 (Macro): 0.2951523400457411

Classification report on test set:

              precision    recall  f1-score   support

           0       0.84      0.66      0.74       168
           1       0.31      0.59      0.41        68
           2       0.00      0.00      0.00         4
           3       0.43      0.27      0.33        56
           4       0.00      0.00      0.00         4

    accuracy                           0.55       300
   macro avg       0.32      0.30      0.30       300
weighted avg       0.62      0.55      0.57       300
```



Random Forest

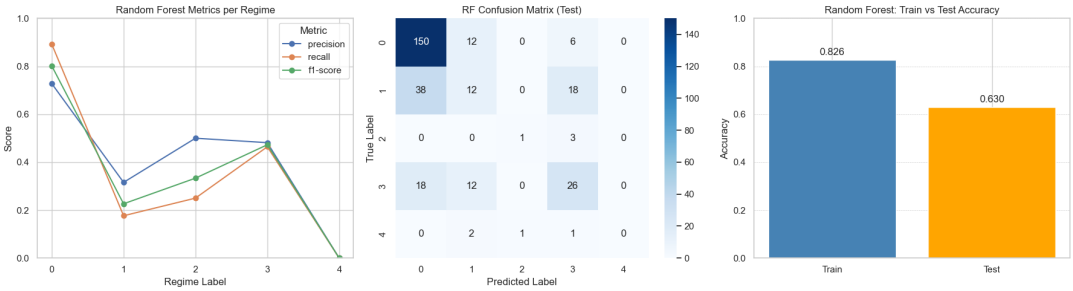
Desciption:

We include Random Forest as one of the classification models due to its robustness to overfitting, ability to handle nonlinear relationships, and effectiveness in high-dimensional settings. Its ensemble nature allows it to **capture complex interactions between features** like rolling mean and standard deviation, making it well-suited for predicting market regime transitions based on historical volatility and return patterns.

In [ ]:
```
reset_session()

pipe_rx = Pipeline([
    ('scaler', StandardScaler()),
    ('rf', RandomForestClassifier(random_state=42))])

param_grid_rx = {
    'rf__n_estimators': [100, 200],
    'rf__max_depth': [None, 5, 10],
    'rf__min_samples_split': [2, 5],
    'rf__criterion': ['gini', 'entropy']
}

grid_rx = GridSearchCV(
    estimator=pipe_rx,
    param_grid=param_grid_rx,
    cv=3,
    scoring='accuracy',
    verbose=1,
    n_jobs=-1,
    refit=True
)

# Fit grid search
grid_rx.fit(X_train_rx, y_train_rx)

# Best model
print("Best Random Forest params (rx):", grid_rx.best_params_)
print("Best CV accuracy (rx):", grid_rx.best_score_)

# Test set evaluation
best_rf_rx = grid_rx.best_estimator_
y_pred_rx = best_rf_rx.predict(X_test_rx)

print("Test accuracy (rx):", accuracy_score(y_test_rx, y_pred_
print("\nClassification report (rx):\n", classification_report
report_dict_rf = classification_report(y_test_rx, y_pred_rx, o
```

```
# Convert to DataFrame
df_report_rf = pd.DataFrame(report_dict_rf).T

# Keep only numeric regime labels (e.g., "0", "1", "2", ...)
regime_metrics_rf = df_report_rf[df_report_rf.index.str.isnume
regime_metrics_rf.index = regime_metrics_rf.index.astype(int)
regime_metrics_rf.sort_index(inplace=True)
cm = confusion_matrix(y_test_rx, y_pred_rx)
train_acc = accuracy_score(y_train_rx, best_rf_rx.predict(X_tr
test_acc  = accuracy_score(y_test_rx,  best_rf_rx.predict(X_te

fig, axs = plt.subplots(1, 3, figsize=(18, 5))

regime_metrics_rf[['precision', 'recall', 'f1-score']].plot(ax
axs[0].set_title("Random Forest Metrics per Regime")
axs[0].set_xlabel("Regime Label")
axs[0].set_ylabel("Score")
axs[0].set_ylim(0, 1)
axs[0].legend(title="Metric")
axs[0].grid(True)
axs[0].set_xticks(regime_metrics_rf.index)

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=best_rf_rx.named_steps['rf'].classes_,
            yticklabels=best_rf_rx.named_steps['rf'].classes_,
            ax=axs[1])
axs[1].set_xlabel('Predicted Label')
axs[1].set_ylabel('True Label')
axs[1].set_title('RF Confusion Matrix (Test)')

axs[2].bar(['Train', 'Test'], [train_acc, test_acc], color=['s
axs[2].set_ylim(0, 1)
axs[2].set_ylabel('Accuracy')
axs[2].set_title('Random Forest: Train vs Test Accuracy')
for i, v in enumerate([train_acc, test_acc]):
    axs[2].text(i, v + 0.02, f"{v:.3f}", ha='center')
axs[2].grid(axis='y', linestyle='--', linewidth=0.5)

plt.tight_layout()
plt.show()
```

```
Fitting 3 folds for each of 24 candidates, totalling 72 fits
Best Random Forest params (rx): {'rf__criterion': 'gini', 'rf__
max_depth': 5, 'rf__min_samples_split': 5, 'rf__n_estimators':
100}
Best CV accuracy (rx): 0.6843537165425578
Test accuracy (rx): 0.63

Classification report (rx):
              precision    recall  f1-score   support

           0       0.73      0.89      0.80       168
           1       0.32      0.18      0.23        68
           2       0.50      0.25      0.33         4
           3       0.48      0.46      0.47        56
           4       0.00      0.00      0.00         4

    accuracy                           0.63       300
   macro avg       0.41      0.36      0.37       300
weighted avg       0.58      0.63      0.59       300
```



Support Vector Machine

Description:

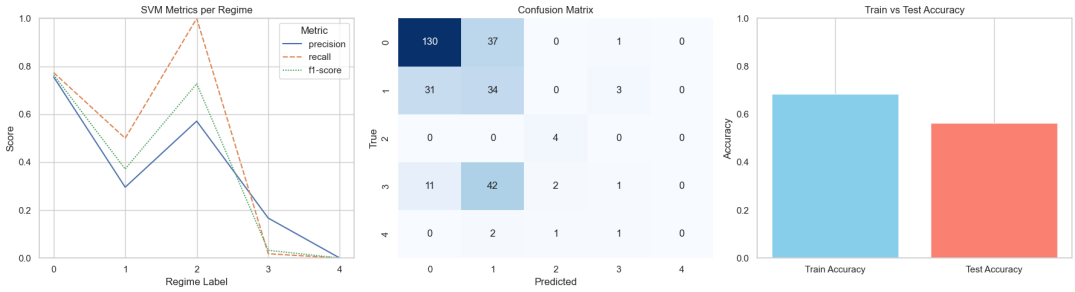
This part applies a **Support Vector Machine (SVM)** model to classify future market regimes based on selected features derived from price, volume, and volatility indicators (e.g., log returns, VIX, SP500 volume). The SVM algorithm is well-suited for handling high-dimensional and non-linear classification tasks. An **RBF kernel** is used to model complex boundaries between regime clusters, and **class weighting** is applied to address label imbalance across regime categories.

To preserve the chronological structure of financial data, **TimeSeriesSplit** cross-validation is employed during model tuning. A **GridSearchCV** procedure is used to identify optimal hyperparameters (`C` and `gamma`) by maximizing validation accuracy. The final model is evaluated on a temporally separated test set, with performance reported through accuracy, macro-averaged F1-score, and a detailed classification report.

In [ ]:
```
reset_session()

# 1. Define pipeline (scaler + SVC)
pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("svc", SVC(kernel='rbf', class_weight='balanced', probabi
])

# 2. Hyperparameter grid
param_grid = {
    'svc__C': [0.1, 1, 10],
    'svc__gamma': ['scale', 'auto', 0.01, 0.1, 1]
}

# 3. TimeSeriesSplit
tscv = TimeSeriesSplit(n_splits=5)

# 4. GridSearchCV
grid = GridSearchCV(pipe, param_grid, cv=tscv, scoring='accura
grid.fit(X_train_cl, y_train_cl)

print("Best SVM parameters: ", grid.best_params_)
print("Best CV accuracy: ", grid.best_score_)

# 5. Evaluate best model on test set
best_svm = grid.best_estimator_
y_pred_cl = best_svm.predict(X_test_cl)

test_acc = accuracy_score(y_test_cl, y_pred_cl)
test_f1 = f1_score(y_test_cl, y_pred_cl, average="macro")

# Predict test labels using the best model
y_pred_cl = best_svm.predict(X_test_cl)

# Compute prediction accuracy
prediction_accuracy = accuracy_score(y_test_cl, y_pred_cl)

print("Test Accuracy:", test_acc)
print("Test F1 (Macro):", test_f1)
print("\nClassification report on test set:\n")
print(classification_report(y_test_cl, y_pred_cl, zero_divisio
print(f"Prediction Accuracy on Test Set: {prediction_accuracy:
# Generate classification report as dictionary
report_dict = classification_report(y_test_cl, y_pred_cl, outp

# Generate classification report as dictionary
report_dict = classification_report(y_test_cl, y_pred_cl, outp
df_report = pd.DataFrame(report_dict).T

# Keep only numeric regime labels (e.g., "0", "1", "2", ...)
regime_metrics = df_report[df_report.index.str.isnumeric()].co
regime_metrics.index = regime_metrics.index.astype(int)
```

```
regime_metrics.sort_index(inplace=True)

# Compute confusion matrix
cm = confusion_matrix(y_test_cl, y_pred_cl)

# Calculate train accuracy for comparison (if not already calcu
train_acc = accuracy_score(y_train_cl, best_svm.predict(X_trai

# Create subplots
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# 1. Classification metrics per regime
sns.lineplot(ax=axes[0], data=regime_metrics[['precision', 're
axes[0].set_title("SVM Metrics per Regime")
axes[0].set_xlabel("Regime Label")
axes[0].set_ylabel("Score")
axes[0].set_ylim(0, 1)
axes[0].grid(True)
axes[0].legend(title="Metric")
axes[0].set_xticks(regime_metrics.index)

# 2. Confusion matrix
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
axes[1].set_title("Confusion Matrix")
axes[1].set_xlabel("Predicted")
axes[1].set_ylabel("True")

# 3. Train vs Test accuracy
axes[2].bar(['Train Accuracy', 'Test Accuracy'], [train_acc, te
axes[2].set_ylim(0, 1)
axes[2].set_ylabel("Accuracy")
axes[2].set_title("Train vs Test Accuracy")
axes[2].grid(axis='y')

plt.tight_layout()
plt.show()
```

```
Fitting 5 folds for each of 15 candidates, totalling 75 fits
Best SVM parameters:  {'svc__C': 1, 'svc__gamma': 0.01}
Best CV accuracy:  0.6517241379310346
Test Accuracy: 0.5633333333333334
Test F1 (Macro): 0.3791642747190699

Classification report on test set:

              precision    recall  f1-score   support

           0       0.76      0.77      0.76       168
           1       0.30      0.50      0.37        68
           2       0.57      1.00      0.73         4
           3       0.17      0.02      0.03        56
           4       0.00      0.00      0.00         4

    accuracy                           0.56       300
   macro avg       0.36      0.46      0.38       300
weighted avg       0.53      0.56      0.53       300

Prediction Accuracy on Test Set: 0.5633
```



Test Loss Compare

To evaluate and compare the generalization performance of each model, the following table summarizes the test accuracy across all classification models. These metrics provide insight into both overall correctness (accuracy). Lower-performing models may indicate sensitivity to class imbalance or overfitting, while higher scores suggest better regime prediction capabilities.

In [ ]:
```
# ———————————— 1. Softmax Regression————————————

nn_test_loss = test_loss
nn_test_acc  = test_acc

# ———————————— 2. Gradient Boosting ————————————
y_proba_gb = best_gb.predict_proba(X_test_lm)
gb_test_loss = log_loss(y_test_lm, y_proba_gb)

y_pred_gb   = best_gb.predict(X_test_lm)
gb_test_acc = accuracy_score(y_test_lm, y_pred_gb)
```

```
# ———————————— 3. Random Forest ————————————
y_proba_rf = best_rf_rx.predict_proba(X_test_rx)
rf_test_loss = log_loss(y_test_rx, y_proba_rf)

y_pred_rf   = best_rf_rx.predict(X_test_rx)
rf_test_acc = accuracy_score(y_test_rx, y_pred_rf)

# ———————————— 4. SVM (RBF) ————————————
y_proba_svm = best_svm.predict_proba(X_test_cl)
svm_test_loss = log_loss(y_test_cl, y_proba_svm)

y_pred_svm   = best_svm.predict(X_test_cl)
svm_test_acc = accuracy_score(y_test_cl, y_pred_svm)

# ———————————— 5. Markov  ————————————
probs_pred = np.array([P_test[s] for s in regime_test_current[

lb = LabelBinarizer()
states = np.arange(k_best)
lb.fit(states)

y1_true_onehot = lb.transform(y1_true)
if y1_true_onehot.shape[1] != k_best:
    y1_true_onehot = np.eye(k_best)[y1_true]

markov_test_loss = log_loss(y1_true_onehot, probs_pred)
markov_test_acc  = acc1


losses = {
    "1-Step Markov": markov_test_loss,
    "Softmax Regression": nn_test_loss,
    "Gradient Boosting": gb_test_loss,
    "Random Forest": rf_test_loss,
    "SVM (RBF)": svm_test_loss
}

accuracies = {
    "1-Step Markov": markov_test_acc,
    "Softmax Regression": nn_test_acc,
    "Gradient Boosting": gb_test_acc,
    "Random Forest": rf_test_acc,
    "SVM (RBF)": svm_test_acc
}

df = pd.DataFrame.from_dict(losses, orient="index", columns=["
df.index.name = "Model"
df.reset_index(inplace=True)

df["Test Accuracy"] = df["Model"].map(accuracies)
```

```
print(df.to_markdown(index=False))
```

Model	Test Loss	Test Accuracy
1-Step Markov	0.935775	0.591973
Softmax Regression	1.23905	0.563333
Gradient Boosting	1.22999	0.553333
Random Forest	0.835526	0.63
SVM (RBF)	0.935575	0.563333

Result Analysis & Conclusion

Project Summary

In this project, we developed a two-stage machine learning framework to identify and forecast market regimes in the S&P 500 Index. We utilized 20 years of daily closing price data from Yahoo Finance (May 2005 to May 2025) and extracted rolling features—specifically, the mean (μ) and standard deviation (σ) of log returns—calculated over non-overlapping 5-day windows. These features effectively capture short-term market trends and volatility. Using KMeans clustering with K = 5, we segmented the data into five distinct market regimes. These unsupervised labels served as the prediction target for supervised models trained to forecast the regime 21 trading days (approximately one month) into the future.

Model Evaluation

We evaluated five models for regime classification: the 1-Step Markov Model, Softmax Regression with Neural Network, Gradient Boosting (GBDT), Random Forest, and Support Vector Machine (SVM with RBF kernel). To assess performance, we considered both test accuracy and test loss. While accuracy measures how often the model predicts the correct regime label, test loss reflects the confidence of the model's predicted probability distribution—lower loss indicates more reliable and calibrated predictions, even when accuracy remains the same. Among all models, Random Forest achieved the best overall performance, with the highest test accuracy and lowest test loss. This suggests that Random Forest is best at capturing the underlying patterns in our μ–σ-based clustering structure. The 1-Step Markov Model also performed competitively, indicating that temporal dependencies in market regime transitions are meaningful and helpful for prediction. Other models such as SVM and Softmax Regression

showed moderate performance, while Gradient boosting underperformed in this setup.

It's important to note that overall accuracy levels are relatively low across all models. This is largely due to the limited feature space used in clustering and classification—specifically, we only used two features: the mean return (μ) and volatility (σ). These features represent basic distributional properties (location and scale) but do not capture richer information like skewness, kurtosis, or inter-feature correlations. As a result, even the best models face difficulty in fully separating complex real-world market behavior. Nonetheless, the fact that reasonable classification performance is achieved using only two features highlights the economic relevance of μ and σ in regime identification.

Key Insights

Our framework combines unsupervised clustering with supervised classification to predict market regimes in a data-efficient yet interpretable way. We use two engineered features (μ and σ) to identify regime structure through clustering, ensuring interpretability and economic consistency. Once regimes are defined, we leverage richer technical and macro features—selected using mutual information (MI > 0.1)—to train predictive models. These features, led by VIX, trading volume, and 21-day volatility, offer the strongest predictive signals. Among the models tested, Random Forest achieves the best overall performance, confirming the advantage of tree-based methods in capturing nonlinear patterns. The Markov model's competitive accuracy highlights the role of temporal transitions in regime dynamics. Overall, our pipeline balances simplicity and predictive power and provides a foundation for interpretable regime forecasting..

Limitations

Despite promising outcomes, our approach has several limitations. Using a fixed number of clusters (K = 5) may not capture rare or extreme regimes like financial crises. Our model relies exclusively on technical features, omitting important macroeconomic and sentiment-based indicators. Moreover, we evaluated model performance solely on classification metrics like accuracy and loss, without assessing real-world financial outcomes such as Sharpe ratios or drawdowns.

Future Directions

While our classification models relied solely on rolling mean (μ) and standard deviation (σ) of log returns as input features, incorporating a broader set of financial indicators—such as momentum signals, macroeconomic variables, or sentiment data—could potentially improve test accuracy and model robustness. Expanding the feature space may help the classifiers capture more nuanced market behavior and reduce misclassification across similar regime states. To build on this work, future efforts could explore dynamic clustering methods such as Gaussian Mixture Models or silhouette scoring, and test time-sequence models like LSTMs or Hidden Markov Models to better capture regime dynamics. Incorporating macroeconomic indicators, news sentiment, and alternative data sources could significantly improve the model's forecasting power. Applying the framework to other indices and asset classes would also help assess its generalizability. Finally, linking regime forecasts to backtested trading strategies would offer insights into the financial viability of the approach.

Final Thoughts

Overall, this project presents a practical and data-driven method for identifying and forecasting regime shifts in financial markets. By capturing transitions between distinct market states, our approach provides valuable insights for investors, portfolio managers, and researchers seeking to improve risk-adjusted decision-making, market timing, and tactical asset allocation.