IDEAT: Vix takas? Joku momentum indikaattori? Sentimentti?

Model settings

```
use_regime_split = False

#Default modelsD
RF = True # perus random forest
RF2 = False
GB = True # perus gradient boost
Hybrid = False

#Looping models
RF_feature_seek = False # random forest all combinations
seek_all = False
gb_loop = False

#DATA
FF5 = True
FF5_long = False
MSCI = False

RSI = True

local = False #ajetaanko colab vai oma kone
```

```
# Identify active model flags
active_modes = [name for name, flag in zip(
    ['RF', 'GB', 'RF_feature_seek', 'Hybrid', 'seek_all', 'gb_loop'
    [RF, GB, RF_feature_seek, Hybrid, seek_all, gb_loop]
) if flag]

if active_modes:
    print("✅ Active model modes:", ", ".join(active_modes))
else:
    print("⚠️ No active model mode selected.")

# Check dataset toggles: exactly one must be True
datasets = {
    'FF5': FF5,
    'FF5_long': FF5_long,
    'MSCI': MSCI
}
active_datasets = [name for name, flag in datasets.items() if flag]

if len(active_datasets) != 1:
    raise ValueError("Error: Exactly one of [FF5, FF5_long, MSCI] m
else:
    print(f"📊 Using dataset: {active_datasets[0]}")
```

```
✅ Active model modes: RF, GB
📊 Using dataset: FF5
```

```
# # Import Required Libraries
#
# Import all necessary libraries for data manipulation, visualizati
# machine learning, and regression analysis.

# %%
import os
import subprocess
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

from sklearn.ensemble import RandomForestClassifier, RandomForestRe
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler

import statsmodels.api as sm
from tabulate import tabulate

from IPython.display import display, HTML
```

```
if not local:

  %cd /content
  !rm -rf Gradu
  !git clone https://github.com/Elkkujou/Gradu.git
  %cd /content/Gradu
  !ls
  xls_file = pd.ExcelFile("/content/Gradu/THE_2ND_latest.xlsx")

else:



    repo_url = "https://github.com/Elkkujou/Gradu.git"
    repo_name = "Gradu"  # Name of the cloned folder

    # Check if the directory already exists
    if os.path.exists(repo_name):
        print(f"Folder '{repo_name}' already exists. Pulling latest
        # Change to the existing repo folder and pull the latest up
```

```
        subprocess.run(["git", "-C", repo_name, "pull"], check=True
    else:
        print(f"Cloning repository into '{repo_name}'...")
        subprocess.run(["git", "clone", repo_url], check=True)

    # List contents of the cloned repository
    subprocess.run(["ls", repo_name], check=True)
    xls_file = pd.ExcelFile("Gradu/THE_2ND_latest.xlsx")
```

```
/content
Cloning into 'Gradu'...
remote: Enumerating objects: 1139, done.
remote: Counting objects: 100% (189/189), done.
remote: Compressing objects: 100% (58/58), done.
remote: Total 1139 (delta 169), reused 131 (delta 131), pack-reused
Receiving objects: 100% (1139/1139), 314.71 MiB | 21.94 MiB/s, done.
Resolving deltas: 100% (567/567), done.
/content/Gradu
 chatti_RF.ipynb                       RF_Gradu.ipynb
 data+regimes.xlsx                     RF_REGIIMI_HYVÄ_TRAINING_2.ipy
 Fama_french_XGBOOST.ipynb             RF_REGIIMI_HYVÄ_TRAINING.ipynb
'Financial turbulence.ipynb'          'RF_REGIIMI_HYVÄ_TRAINING (MSCI
 FT_source.xlsx                       'RF_regime (3).ipynb'
 Gradient_boost_malli.ipynb            RF_Työ_1.ipynb
 MSCI_XGBOOST.ipynb                    RF_Työ.ipynb
 Regiimi_prediction.ipynb              THE_2ND_latest.xlsx
 regime_prediction_famafrench.ipynb    THE_2ND.xlsx
 regime_prediction_msci.ipynb          THE_ONE.xlsx
 regime_pred.txt
```

```
if FF5:
  SHEET_NAME = "ajodata_FF5"
  FEATURES = ['CPI%','T10Y3M', 'CFNAI', 'GARCH']
  FACTORS = [
    'SMB',
    'HML',
    'CMA',
    'RMW',
    #'RF'
]
  BENCHMARK = ['Mkt']
  show_benchmark = False
```

```
if FF5_long:
  SHEET_NAME = "ajodata_FF5_long"
  FEATURES = ['CPI%','T10Y3M', 'CFNAI', 'GARCH']
  FACTORS = [
    'SMB',
    'HML',
    'CMA',
    'RMW',
    #'RF'
]
  BENCHMARK = ['Mkt']
  show_benchmark = True
```

```
if MSCI:
  SHEET_NAME = "ajodata_MSCI"
  FEATURES = ['CPI%','T10Y3M', 'CFNAI', 'GARCH']
  FACTORS = [
    'Size',
    'value',
    'Quality',
    'min_vola']
  BENCHMARK = ['Us_standard']
  show_benchmark = True
```

Prepare data

```
df = xls_file.parse(SHEET_NAME)
df.columns = df.columns.get_level_values(0)

# Print headers dynamically
print(f"Headers in the '{SHEET_NAME}' sheet:")
print(df.columns)

REGIMES_COLUMN = 'Predicted_reg'

# Convert the leftmost column (assumed to be the date column) to da
date_column = df.columns[0]
df[date_column] = pd.to_datetime(df[date_column])

# Retrieve first and last observation dates and count observations
first_date = df[date_column].iloc[0]
last_date = df[date_column].iloc[-1]
n_observations = len(df)

# Create a DataFrame with the information
info_df = pd.DataFrame({
    "Description": ["First observation date", "Last observation dat
    "Value": [first_date, last_date, n_observations]
})

# Display the results as a neat HTML table
display(HTML(info_df.to_html(index=False, classes="table table-stri
```

```
Headers in the 'ajodata_FF5' sheet:
Index(['Date', 'SMB', 'HML', 'RMW', 'CMA', 'Mkt', 'RF', 'Mkt-RF', 'G
       'CPI%', 'T10YFF', 'Amihud', 'LEI%', 'Cape', 'Cape %', 'GDP',
       'T10Y3M', 'LEI', 'CFNAI', 'HV', 'VIX', 'EWMA', 'AvgShock', 'A
       'RealizedVol', 'RelVol_12m', 'VolBucket', 'GARCH_1M_REL', 'EW
       'EWMA_0.94', 'BAA10Y'],
      dtype='object')
```

Description	Value
First observation date	1963-07-30 00:00:00
Last observation date	2024-11-30 00:00:00
Total number of observations	737

```
if RSI:

  # --- Add lagged 12-month moving average for each factor return -
  for f in FACTORS:
      # shift by 1 so that MA at time t uses returns t-12…t-1
      df[f + '_MA12'] = (
          df[f]
          .shift(1)                          # drop “today”
          .rolling(window=12, min_periods=12)
          .mean()
      )

  # update your FEATURES list
  FEATURES += [f + '_MA12' for f in FACTORS]

  print("✨ Added lagged 12-month MA columns:", [f + '_MA12' for f
  print("🧩 New FEATURES list:", FEATURES)
```

```
✨ Added lagged 12-month MA columns: ['SMB_MA12', 'HML_MA12', 'CMA_M
🧩 New FEATURES list: ['CPI%', 'T10Y3M', 'CFNAI', 'GARCH', 'SMB_MA12
```

```
import numpy as np
import pandas as pd
from IPython.display import HTML, display

# ------------------------------------------------------------
# 1.  Inputs
# ------------------------------------------------------------
# df      : DataFrame of monthly total-return series (index = month
# FACTORS : list/tuple of column names in df you want analysed
#
# Example:
# df       = pd.read_csv("monthly_returns.csv", parse_dates=["Date"
# FACTORS  = ["Value", "Momentum", "Quality", "LowVol"]
# ------------------------------------------------------------

# ---------- Helper functions ----------
def annualized_return(returns: pd.Series) -> float:
    """Compounded annualised return from monthly series."""
    return (1 + returns).prod() ** (12 / len(returns)) - 1

def compute_metrics(returns: pd.Series) -> tuple[float, float, floa
    """Annualised return, annualised volatility, and cumulative ret
    ann_ret = annualized_return(returns)
    ann_vol = returns.std() * np.sqrt(12)          # monthly → annu
    cum_ret = (1 + returns).prod() - 1
```

```
    return ann_ret, ann_vol, cum_ret

# ---------- Win-rate (highest-return months, factors only) -------
monthly_winner = df[FACTORS].idxmax(axis=1)        # winner each mo
win_rates = (
    monthly_winner.value_counts()                  # how many wins
    .reindex(FACTORS, fill_value=0) / len(df)      # re-index ensur
)

# ---------- Number-format helper (space as thousands separator) --
NBSP = "\u202F"                                    # thin non-break

def pct_with_space(x: float) -> str:
    """Format a decimal as percentage with 1-dp and thin-space thou
    return f"{x * 100:,.1f}%".replace(",", NBSP)

# ---------- Build the summary table ----------
records = []
for col in FACTORS:
    r = df[col]
    ann_ret, ann_vol, cum_ret = compute_metrics(r)
    records.append({
        "Factor":                   col,
        "Annualised Return":        pct_with_space(ann_ret),
        "Annualised Volatility":    pct_with_space(ann_vol),
        "Total Cumulative Return":  pct_with_space(cum_ret),
        "Win rate":                 pct_with_space(win_rates[col]),
    })

metrics_df = pd.DataFrame(records)

# ---------- Display ----------
display(HTML(metrics_df.to_html(index=False)))
```

Factor	Annualised Return	Annualised Volatility	Total Cumulative Return	Win rate
SMB	1.9%	10.6%	216.7%	30.8%
HML	2.9%	10.4%	473.2%	23.7%
CMA	2.9%	7.2%	472.9%	16.0%
RMW	3.1%	7.7%	568.9%	29.4%

```
# Toggle for dropping rows with missing values in the FEATURES colu
drop_empty = True

# 1) Print the initial number of rows.
```

```
initial_rows = len(df)
print(f"Total number of rows before cleaning: {initial_rows}")

# 2) Show missing-value counts in FEATURES.
missing_counts = df[FEATURES].isna().sum()
print("\nMissing values in FEATURES before cleaning:")
print(missing_counts)

# 3) Drop any rows with NA in FEATURES, if requested.
if drop_empty:
    before = len(df)
    df.dropna(subset=FEATURES, inplace=True)
    df.reset_index(drop=True, inplace=True)
    dropped = before - len(df)
    print(f"\nDropped {dropped} rows due to missing FEATURES.")
else:
    print("\nKeeping all rows, including those with missing FEATURE

# 4) Re-check that FEATURES are now complete:
print("\nMissing values in FEATURES after cleaning:")
print(df[FEATURES].isna().sum())

# 5) Compute your target column in-place.
df['Winning Factor'] = df[FACTORS].idxmax(axis=1).astype('category'

# 6) (Optionally) create a numeric code column
df['Winning Factor Code'] = df['Winning Factor'].cat.codes

# 7) Quick summary:
print(f"\nFinal dataset now has {len(df)} rows.")
print("Target value counts:")
print(df['Winning Factor'].value_counts())
```

```
Total number of rows before cleaning: 737

Missing values in FEATURES before cleaning:
CPI%         0
T10Y3M       0
CFNAI       45
GARCH        0
SMB_MA12    12
HML_MA12    12
CMA_MA12    12
RMW_MA12    12
dtype: int64

Dropped 45 rows due to missing FEATURES.

Missing values in FEATURES after cleaning:
CPI%        0
```

```
T10Y3M      0
CFNAI       0
GARCH       0
SMB_MA12    0
HML_MA12    0
CMA_MA12    0
RMW_MA12    0
dtype: int64

Final dataset now has 692 rows.
Target value counts:
Winning Factor
SMB    210
RMW    209
HML    164
CMA    109
Name: count, dtype: int64
```

```
if use_regime_split:

    # --- Regime Mapping & Conversion to Numeric Codes (Dynamic) --

    # Dynamically extract the unique values in the REGIMES_COLUMN.
    unique_regimes = df[REGIMES_COLUMN].unique()

    # Convert the Regimes column to a categorical type with the uni
    df[REGIMES_COLUMN] = pd.Categorical(df[REGIMES_COLUMN], categor

    # Create a dictionary mapping numeric codes to the regime names
    regime_mapping = {i: cat for i, cat in enumerate(df[REGIMES_COL

    # Now encode the Regimes column as numeric codes.
    df[REGIMES_COLUMN] = df[REGIMES_COLUMN].cat.codes

    # Create a mapping from numeric codes to original regime names.
    regime_short_mapping = {code: name for code, name in regime_map

    # Calculate the number of observations for each regime using va
    obs_counts = df[REGIMES_COLUMN].value_counts(sort=False)

    # Create a DataFrame preview of the regime mapping, including c
    mapping_table_data = []
    for code in regime_mapping.keys():
        mapping_table_data.append({
            "Numeric Code": code,
            "Original Name": regime_mapping.get(code, "N/A"),
            "Observations": obs_counts.get(code, 0)
        })

    # Append a row with the total observations.
    total_obs = obs_counts.sum()
    mapping_table_data.append({
        "Numeric Code": "",
        "Original Name": "Total",
        "Observations": total_obs
    })

    # Create the DataFrame for regime mapping preview and print.
    regime_mapping_df = pd.DataFrame(mapping_table_data)

    from tabulate import tabulate
    print("Preview of Dynamic Regime Mapping:")
    print(tabulate(regime_mapping_df, headers="keys", tablefmt="psq
```

Models

Feature seek

```
    if RF_feature_seek:
        import itertools
        import os
        import time
        import pandas as pd
        import numpy as np
        from sklearn.ensemble import RandomForestClassifier

        # ------------------------
        # Parameters for Feature & Training Window Search
        # ------------------------
        min_features = 2                 # minimum number of features
        max_features = len(FEATURES)     # maximum number of features

        # Define fixed rolling window sizes (in years) to test (assumin
        training_window_years = [5, 10, 15, 20]

        # Also run an expanding window experiment
        run_expanding_window = True

        # Independent variable: minimum number of observations required
        # This is now decoupled from the training window calculation.
        min_obs_for_prediction = 60  # adjust this value as desired

        output_filename = "feature_subset_results.csv"
        if os.path.exists(output_filename):
            os.remove(output_filename)

        # Ensure the data is sorted by date.
        df_sorted = df.sort_values('Date').reset_index(drop=True)

        # ------------------------
        # Outer Loop: Fixed Rolling Window Modes
        # ------------------------
        for years in training_window_years:
            # Convert years to number of observations (assume 12 obs pe
            rolling_window_size = years * 12
            # Ensure predictions start only after both the rolling wind
            start_index = max(min_obs_for_prediction, rolling_window_si
```

```
            print(f"\n--- Testing fixed rolling window of {years} years
                  f"({rolling_window_size} observations, starting predi
            outer_start_time = time.time()

            # Inner loop over feature subset sizes
            for r in range(min_features, max_features + 1):
                # Loop over all combinations of size r
                for comb in itertools.combinations(FEATURES, r):
                    current_features = list(comb)
                    inner_start_time = time.time()
                    print(f"\nTesting feature combination: {current_fea
                    results = []

                    # Loop over test rows, starting when we have enough
                    for i in range(start_index, len(df_sorted)):
                        test_row = df_sorted.iloc[i]
                        Predicted_month = test_row['Date']

                        # Build fixed rolling training window (most rec
                        train_window = df_sorted.iloc[i - rolling_windo

                        # Ensure the last training observation is stric
                        last_train_date = train_window['Date'].iloc[-1]
                        if (last_train_date.year == Predicted_month.yea
                           (last_train_date.month >= Predicted_month.mo
                            continue

                        # (Optional) Regime check if use_regime_split i
                        if use_regime_split:
                            regime_counts = train_window[REGIMES_COLUMN
                            insufficient_regimes = regime_counts[regime
                            if insufficient_regimes:
                                continue
                            current_regime = test_row[REGIMES_COLUMN]
                            train_window = train_window[train_window[RE
                            if len(train_window) < min_obs_regime:
                                continue

                        # Prepare training data for the current feature
                        X_train = train_window[list(current_features)].
                        y_train = train_window['Winning Factor'].loc[X_
                        if len(X_train) < 1:
                            continue

                        # Train the RandomForest model.
                        rf_model = RandomForestClassifier(n_estimators=
                        rf_model.fit(X_train, y_train)
```

```
            # Use the last row of the training window as te
            X_test = train_window[list(current_features)].i
            if X_test.empty:
                continue

            predicted_probabilities = rf_model.predict_prob
            predicted_winner = rf_model.classes_[predicted_

            # Map the predicted probabilities to the full s
            full_probs = np.zeros(len(FACTORS))
            for cls, prob in zip(rf_model.classes_, predict
                try:
                    idx = FACTORS.index(cls)
                    full_probs[idx] = prob
                except ValueError:
                    continue

            allocated_return = (full_probs * test_row[FACTO

            # months_ahead: how many months ahead the predi
            months_ahead = (
                (Predicted_month.year - last_train_date.yea
                (Predicted_month.month - last_train_date.mo
            )

            # Collect feature levels for logging
            feature_levels = {
                f"Feature_Level_{f}": X_test[f].iloc[0] for
            }

            # Create a result row
            result = {
                "TrainingWindowYears": years,   # <--- Reco
                "Features_used": str(current_features),
                "Predicted_month": Predicted_month,
                "Allocated_Return": allocated_return,
                "Predicted_Winner": predicted_winner,
                "Actual_Winner": test_row['Winning Factor']
                "Prediction_Horizon_Months": months_ahead,
                **feature_levels
            }
            results.append(result)

        # End of inner test row loop for this feature combi
        if results:
            df_results_comb = pd.DataFrame(results)
            if not os.path.exists(output_filename):
                df_results_comb.to_csv(output_filename, mod
```

```
                    else:
                        df_results_comb.to_csv(output_filename, mod
                    elapsed_inner = time.time() - inner_start_time
                    minutes = int(elapsed_inner // 60)
                    seconds = int(elapsed_inner % 60)
                    print(f"Results for combination {current_featur
                          f"Time taken: {minutes:02d}:{seconds:02d}

        elapsed_outer = time.time() - outer_start_time
        minutes = int(elapsed_outer // 60)
        seconds = int(elapsed_outer % 60)
        print(f"Completed fixed rolling window of {years} years in

    # --------------------------
    # Expanding Window Mode
    # --------------------------
    if run_expanding_window:
        print("\n--- Testing Expanding Window Mode ---")
        outer_start_time = time.time()
        for r in range(min_features, max_features + 1):
            for comb in itertools.combinations(FEATURES, r):
                current_features = list(comb)
                inner_start_time = time.time()
                print(f"\nTesting feature combination (expanding):
                results = []

                # In expanding mode, the training window goes from
                # Start predictions only after the minimum observat
                for i in range(min_obs_for_prediction, len(df_sorte
                    test_row = df_sorted.iloc[i]
                    Predicted_month = test_row['Date']
                    train_window = df_sorted.iloc[:i].copy()
                    if train_window.empty:
                        continue

                    last_train_date = train_window['Date'].iloc[-1]
                    if (last_train_date.year == Predicted_month.yea
                       (last_train_date.month >= Predicted_month.mo
                        continue

                    X_train = train_window[list(current_features)].
                    y_train = train_window['Winning Factor'].loc[X_
                    if len(X_train) < 1:
                        continue

                    rf_model = RandomForestClassifier(n_estimators=
                    rf_model.fit(X_train, y_train)
```

```
                X_test = train_window[list(current_features)].i
                if X_test.empty:
                    continue

                predicted_probabilities = rf_model.predict_prob
                predicted_winner = rf_model.classes_[predicted_

                full_probs = np.zeros(len(FACTORS))
                for cls, prob in zip(rf_model.classes_, predict
                    try:
                        idx = FACTORS.index(cls)
                        full_probs[idx] = prob
                    except ValueError:
                        continue

                allocated_return = (full_probs * test_row[FACTO

                months_ahead = (
                    (Predicted_month.year - last_train_date.yea
                    (Predicted_month.month - last_train_date.mo
                )

                feature_levels = {
                    f"Feature_Level_{f}": X_test[f].iloc[0] for
                }

                result = {
                    "TrainingWindowYears": "expanding",  # <---
                    "Features_used": str(current_features),
                    "Predicted_month": Predicted_month,
                    "Allocated_Return": allocated_return,
                    "Predicted_Winner": predicted_winner,
                    "Actual_Winner": test_row['Winning Factor']
                    "Prediction_Horizon_Months": months_ahead,
                    **feature_levels
                }
                results.append(result)

            if results:
                df_results_comb = pd.DataFrame(results)
                if not os.path.exists(output_filename):
                    df_results_comb.to_csv(output_filename, mod
                else:
                    df_results_comb.to_csv(output_filename, mod
                elapsed_inner = time.time() - inner_start_time
                minutes = int(elapsed_inner // 60)
                seconds = int(elapsed_inner % 60)
                print(f"Expanding window: Results for combinati
```

```
                                f"appended to CSV. Time taken: {minutes:0

            elapsed_outer = time.time() - outer_start_time
            minutes = int(elapsed_outer // 60)
            seconds = int(elapsed_outer % 60)
            print(f"Completed Expanding Window Mode in {minutes:02d}:{s
```

Seek all

```
if seek_all:
  import pandas as pd
  import numpy as np
  import csv
  import itertools
  import time
  import math
  from sklearn.ensemble import RandomForestClassifier
  from sklearn.model_selection import ParameterGrid

  # ——————————————————————————————————————
  # Assumes `df` (with 'Date' & 'Winning Factor'),
  # `FACTORS` (list of all factor names) and
  # `FEATURES` (base feature list) are defined above
  # ——————————————————————————————————————

  # ——————————————————————————————————————
  # Toggle feature-looping on/off
  # ——————————————————————————————————————
  loop_features = False   # False ⇒ single run on FEATURES; True ⇒

  # ——————————————————————————————————————
  # 1) Define always-on & optional features
  # ——————————————————————————————————————
  always_features = [
      "CMA_MA12", "SMB_MA12", "RMW_MA12", "HML_MA12",
      "CPI%"
  ]
  optional_features = [
      "LEI%", "Cape", "Cape %", "TED",
      "T10Y3", "LEI", "AR_Shock", "HV",
      "EWMA_0.94", "T10YFF", "CFNAI", "GARCH_1M", "VIX","BAA10Y"
  ]

  # ——————————————————————————————————————
```

```
# 1a) Control how many optional features per combo
# ──────────────────────────────────────────────
min_optional = 1   # minimum number of optional features in each
max_optional = 5   # maximum number of optional features in each

# ──────────────────────────────────────────────
# 2) Build feature_combinations (with new constraints)
# ──────────────────────────────────────────────
volatility_features = {"GARCH_1M", "VIX", "AR_Shock", "HV", "EWMA
lei_group          = {"LEI", "LEI%", "CFNAI"}
cape_group         = {"Cape", "Cape %"}

if loop_features:
    def valid_combo(combo):
        combo = set(combo)
        # 1) at most 2 volatility measures
        if len(combo & volatility_features) > 2:
            return False
        # 2) T10Y3 and T10YFF cannot co-exist
        if {"T10Y3", "T10YFF"} <= combo:
            return False
        # 3) only one of LEI, LEI%, CFNAI
        if len(combo & lei_group) > 1:
            return False
        # 4) only one of Cape, Cape %
        if len(combo & cape_group) > 1:
            return False
        return True

    feature_combinations = [
        always_features + list(combo)
        for r in range(min_optional, max_optional + 1)
        for combo in itertools.combinations(optional_features, r)
        if valid_combo(combo)
    ]
else:
    feature_combinations = [FEATURES]

# ──────────────────────────────────────────────
# 3) RF hyperparameter template (excluding max_features)
# ──────────────────────────────────────────────
param_grid_template = {
    'n_estimators':      [100, 300, 500],
    'max_depth':         [None, 7, 10, 15],
    'min_samples_split': [2, 4, 6],
    'min_samples_leaf':  [1, 3, 5, 7],
    'bootstrap':         [False, True],
    'n_jobs':            [-1]
```

```
}
base_param_list = list(ParameterGrid(param_grid_template))

# compute total iterations for progress display
total_iterations = sum(
    len(base_param_list) * len(range(2, len(feat_set) + 1, 2))
    for feat_set in feature_combinations
)

# ──────────────────────────────────────────────
# 4) CSV logging setup
# ──────────────────────────────────────────────
csv_file = 'rf_feature_search.csv'
fieldnames = [
    'Iteration','Training_Window','Features','Hyperparameters',
    'Num_Preds','First_Pred','Last_Pred',
    'CumAlloc_Post2000','CumEqual_Post2000',
    'CumAlloc_Total','CumEqual_Total',
    'Sharpe_Post2000','Win_Count_Post2000'
]
with open(csv_file, 'w', newline='') as f:
    writer = csv.DictWriter(f, fieldnames=fieldnames, delimiter='
    writer.writeheader()

# ──────────────────────────────────────────────
# 5) Rolling-window & data settings
# ──────────────────────────────────────────────
rolling_window_size = 60   # months in each fixed window
min_months_train    = 60   # minimum months of history required
min_obs_train       = 0    # minimum non-missing rows in X_train
use_fixed_window    = True # True: fixed-length rolling window; F

# ──────────────────────────────────────────────
# 6) Loop over feature sets & hyperparameters
# ──────────────────────────────────────────────
iteration = 0
summary_records = []

for feat_set in feature_combinations:
    n_feats = len(feat_set)
    # for n_feats = 8, this gives [2, 4, 6, 8]
    max_features_opts = list(range(2, n_feats + 1, 2))

    # inject max_features into each base parameter combination
    param_list = [
        {**p, 'max_features': mf}
        for p in base_param_list
        for mf in max_features_opts
```

```
        ]

        for rf_params in param_list:
            iteration += 1
            start_time = time.perf_counter()

            # --- print at start ---
            print(f"\n=== Iteration {iteration}/{total_iterations} ==
            if loop_features:
                print(f"Always-on features ({len(always_features)}):
                print(f"Optional count range: {min_optional}–{max_opt
            print(f"Using features ({n_feats}): {feat_set}")
            print(f"RF hyperparameters: {rf_params}")

            # drop rows with missing values in this feature set only
            df_iter = df.dropna(subset=feat_set).copy()
            df_sorted = df_iter.sort_values('Date').reset_index(drop=
            results = []

            # per-row prediction loop
            for i in range(1, len(df_sorted)):
                test_row = df_sorted.iloc[i]

                # build training window from prior rows
                if use_fixed_window:
                    start_idx    = max(0, i - rolling_window_size)
                    train_window = df_sorted.iloc[start_idx:i].copy()
                else:
                    train_window = df_sorted.iloc[:i].copy()

                if len(train_window) < min_months_train:
                    continue

                # prepare training data
                X_train = train_window[feat_set].dropna()
                y_train = train_window['Winning Factor'].loc[X_train.
                if len(X_train) < min_obs_train:
                    continue

                # fit RandomForest
                rf = RandomForestClassifier(**rf_params, random_state
                rf.fit(X_train, y_train)

                # predict on the last available row in train_window
                X_test = train_window[feat_set].iloc[[-1]].dropna()
                if X_test.empty:
                    continue
```

```
            probs      = rf.predict_proba(X_test)[0]
            full_probs = np.zeros(len(FACTORS))
            for cls, p in zip(rf.classes_, probs):
                full_probs[FACTORS.index(cls)] = p

            alloc_ret = (full_probs * test_row[FACTORS].values).s
            eq_ret    = test_row[FACTORS].mean()

            results.append({
                'Date': test_row['Date'],
                'Allocated_Return': alloc_ret,
                'Equal_Weight_Return': eq_ret
            })

        # compute summary metrics
        res_df = pd.DataFrame(results).sort_values('Date')
        if res_df.empty:
            summary = dict.fromkeys(fieldnames, np.nan)
            summary.update({
                'Iteration': iteration,
                'Training_Window': rolling_window_size,
                'Features': ",".join(feat_set),
                'Hyperparameters': ";".join(f"{k}={v}" for k,v in
                'Num_Preds': 0
            })
        else:
            post2k        = res_df[res_df['Date'] >= pd.Timestamp
            cum_alloc_2k  = (1 + post2k['Allocated_Return']).prod
            cum_eq_2k     = (1 + post2k['Equal_Weight_Return']).p
            cum_alloc_tot = (1 + res_df['Allocated_Return']).prod
            cum_eq_tot    = (1 + res_df['Equal_Weight_Return']).p

            sharpe_2k = (post2k['Allocated_Return'].mean() /
                         post2k['Allocated_Return'].std()) * np.sq
            win2k = (post2k['Allocated_Return'] > post2k['Equal_W

            summary = {
                'Iteration': iteration,
                'Training_Window': rolling_window_size,
                'Features': ",".join(feat_set),
                'Hyperparameters': ";".join(f"{k}={v}" for k,v in
                'Num_Preds': len(res_df),
                'First_Pred': res_df['Date'].iloc[0].strftime("%Y
                'Last_Pred':  res_df['Date'].iloc[-1].strftime("%
                'CumAlloc_Post2000': cum_alloc_2k,
                'CumEqual_Post2000':  cum_eq_2k,
                'CumAlloc_Total':      cum_alloc_tot,
                'CumEqual_Total':       cum_eq_tot,
```

```
                'Sharpe_Post2000':     sharpe_2k,
                'Win_Count_Post2000':  win2k
            }

        # write one line to CSV
        with open(csv_file, 'a', newline='') as f:
            writer = csv.DictWriter(f, fieldnames=fieldnames, del
            writer.writerow(summary)
            f.flush()

        # update running ranks & print end summary
        summary_records.append(summary)
        df_sum = pd.DataFrame(summary_records)
        df_sum['Rank_CumAlloc_Post2000']   = df_sum['CumAlloc_Pos
        df_sum['Rank_Sharpe_Post2000']     = df_sum['Sharpe_Post2
        df_sum['Rank_CumAlloc_Total']      = df_sum['CumAlloc_Tot
        df_sum['Rank_Win_Count_Post2000']  = df_sum['Win_Count_Pc

        cur = df_sum.iloc[-1]
        duration = time.perf_counter() - start_time

        # print end-of-iteration stats
        print(f"Completed iteration {iteration}/{total_iterations
        print(f"  CumAlloc_Post2000: {cur['CumAlloc_Post2000']:.4
        print(f"  Sharpe_Post2000:   {cur['Sharpe_Post2000']:.4f}
        print(f"  CumAlloc_Total:    {cur['CumAlloc_Total']:.4f}
        print(f"  Win_Count_Post2000:{int(cur['Win_Count_Post2000
        print(f"  EqualWeight_Post2000: {cur['CumEqual_Post2000']
        print(f"  EqualWeight_Total:     {cur['CumEqual_Total']:.
```

Random forest

```
if RF:
    import pandas as pd
    import numpy as np
    from sklearn.ensemble import RandomForestClassifier
    from IPython.display import display, HTML

    # ------------------------------------------------------------
    # Assumes these are already defined:
    #   df, FEATURES, FACTORS, REGIMES_COLUMN, regime_short_mapping
    # ------------------------------------------------------------

    RF1_FEATURES = FEATURES
```

```
# ——————————————————
# 1) Parameters
# ——————————————————
min_months_train        = 60
min_obs_regime          = 50
min_obs_train           = 0
use_regime_split        = False
default_hyperparameters = False

use_fixed_window    = True
rolling_window_size = 60
n_jobs              = -1

# Toggle on/off heavy computations
compute_permutation_importance = False
compute_pdp                    = False
compute_shap                   = True  # ← collect SHAP data

if default_hyperparameters:
    rf_params = {
        'n_estimators': 100,
        'max_depth': None,
        'max_features': 'sqrt',
        'min_samples_split': 2,
        'min_samples_leaf': 1,
        'bootstrap': True,
        'n_jobs': n_jobs
    }
else:
    rf_params = {
        'n_estimators': 100,
        'max_depth': None,
        'max_features': None,
        'min_samples_split': 2,
        'min_samples_leaf': 5,
        'bootstrap': False,
        'n_jobs': n_jobs
    }

# ——————————————————
# prepare PDP & SHAP storage if asked
# ——————————————————
if compute_pdp:
    from sklearn.inspection import partial_dependence
    pdp_data = {feat: [] for feat in RF1_FEATURES}

if compute_shap:
    import shap
```

```
    shap_data = []

# ------------------
# 2) Sort data and init containers
# ------------------
df_sorted = df.sort_values('Date').reset_index(drop=True)
results   = []
rf_models = []

if compute_permutation_importance:
    from sklearn.inspection import permutation_importance
    perm_importances_list = []
    months_list           = []

# ------------------
# 3) Main Loop
# ------------------
for i in range(1, len(df_sorted)):
    test_row        = df_sorted.iloc[i]
    Predicted_month = test_row['Date']

    # build train window
    if use_fixed_window:
        start_idx    = max(0, i - rolling_window_size)
        train_window = df_sorted.iloc[start_idx:i].copy()
    else:
        train_window = df_sorted.iloc[:i].copy()

    if len(train_window) < min_months_train:
        print(f"{Predicted_month.date()}: only {len(train_window
        continue

    train_start_date = train_window['Date'].iloc[0]
    train_end_date   = train_window['Date'].iloc[-1]

    # optional regime-split
    if use_regime_split:
        counts = train_window[REGIMES_COLUMN].value_counts()
        bad    = counts[counts < min_obs_regime].index.tolist()
        if bad:
            print(f"{Predicted_month.date()}: insufficient regim
            continue
        current_regime = test_row[REGIMES_COLUMN]
        train_window   = train_window[train_window[REGIMES_COLUM
        if len(train_window) < min_obs_regime:
            print(f"{Predicted_month.date()}: regime {current_re
            continue
        regime_used = regime_short_mapping.get(current_regime, s
    else:
```

```
else:
    regime_used = 'NoRegime'

last_train_date = train_window['Date'].iloc[-1]
if (last_train_date.year == Predicted_month.year) and (last_
    print(f"{Predicted_month.date()}: last train {last_train
    continue

X_train = train_window[RF1_FEATURES].dropna()
y_train = train_window.loc[X_train.index, 'Winning Factor']
if len(X_train) < min_obs_train:
    print(f"{Predicted_month.date()}: {len(X_train)} < {min_
    continue

# fit RF
rf_model = RandomForestClassifier(**rf_params, random_state=
rf_model.fit(X_train, y_train)
rf_models.append(rf_model)

# —— collect PDP if toggled ——
if compute_pdp:
    for feat in RF1_FEATURES:
        pdp_res = partial_dependence(
            rf_model,
            X_train,
            features=[feat],
            grid_resolution=50,
            kind='average'
        )
        grid = pdp_res.get('values', pdp_res.get('grid_value
        avg  = pdp_res['average']  # shape = (n_factors, n_g
        pdp_data[feat].append({
            'month':   Predicted_month,
            'values':  grid,
            'average': avg
        })

# —— collect SHAP if toggled ——
if compute_shap:
    X_test = train_window[RF1_FEATURES].iloc[[-1]].dropna()
    if not X_test.empty:
        # try new shap.Explainer API
        try:
            explainer = shap.Explainer(rf_model, X_train, fe
            shap_exp  = explainer(X_test)
            vals      = shap_exp.values  # shape: (1, n_feat
        except Exception:
            # fallback to TreeExplainer
            explainer = shap.TreeExplainer(rf_model)
```

```
                    explainer = shap.TreeExplainer(rf_model)
                    sv_list   = explainer.shap_values(X_test)
                    if isinstance(sv_list, list):
                        vals = np.stack(sv_list, axis=-1)[0]  # (n_f
                        vals = vals.T                          # → (
                    else:
                        vals = sv_list  # e.g. (1, n_features)
                # normalize to (n_factors, n_features)
                if vals.ndim == 3:
                    shap_mat = vals[0].T
                elif vals.ndim == 2 and vals.shape[0] == len(FEATURE
                    shap_mat = vals.T
                elif vals.ndim == 2 and vals.shape[0] == 1:
                    shap_mat = np.tile(vals, (len(FACTORS), 1))
                else:
                    raise ValueError(f"Unexpected SHAP shape {vals.s
                shap_data.append({
                    'month':        Predicted_month,
                    'feature_vals': X_test.iloc[0].values,
                    'shap_values':  shap_mat
                })

            # optional: compute permutation importance
            if compute_permutation_importance:
                pi = permutation_importance(
                    rf_model, X_train, y_train,
                    n_repeats=10, random_state=42, n_jobs=n_jobs
                )
                perm_importances_list.append(pi.importances_mean)
                months_list.append(Predicted_month.strftime('%Y-%m-%d'))

            # predict (unchanged)
            X_test = train_window[RF1_FEATURES].iloc[[-1]].dropna()
            if X_test.empty:
                print(f"{Predicted_month.date()}: empty test features. S
                continue

            probs  = rf_model.predict_proba(X_test)[0]
            winner = rf_model.classes_[probs.argmax()]

            full_probs = np.zeros(len(FACTORS))
            for cls, p in zip(rf_model.classes_, probs):
                try:
                    idx = FACTORS.index(cls)
                    full_probs[idx] = p
                except ValueError:
                    pass

            allocated_return    = (full_probs * test_row[FACTORS].values
```

```
            equal_weight_return = test_row[FACTORS].mean()

            depths      = [e.tree_.max_depth for e in rf_model.estimator
            avg_depth   = np.mean(depths)
            max_depth   = np.max(depths)
            months_ahead = ((Predicted_month.year - last_train_date.year
                            (Predicted_month.month - last_train_date.mon

            feature_levels = {f"Feature_Level_{f}": X_test[f].iloc[0] fo

            print(f"{Predicted_month.date()}: trained {train_start_date.

            result = {
                'Regime': regime_used,
                'Predicted_month': Predicted_month,
                'Train_Start_Date': train_start_date,
                'Train_End_Date': train_end_date,
                'Train_Count': len(X_train),
                'Feature_Importances': rf_model.feature_importances_,
                'Predicted_Probabilities': full_probs,
                'Predicted_Winner': winner,
                'Allocated_Return': allocated_return,
                'Equal_Weight_Return': equal_weight_return,
                'Actual_Winner': test_row['Winning Factor'],
                'Num_Trees': rf_model.n_estimators,
                'Average_Tree_Depth': avg_depth,
                'Max_Tree_Depth': max_depth,
                'Prediction_Horizon_Months': months_ahead,
                **feature_levels
            }
            results.append(result)

        # --------------------
        # 4) Build results DataFrame
        # --------------------
        results_df_rf = pd.DataFrame(results)
        print("Columns:", results_df_rf.columns.tolist())
        display(results_df_rf.tail(10))

        # --------------------
        # 5) Cumulative Returns
        # --------------------
        filt = results_df_rf['Predicted_month'] >= pd.Timestamp("2000-01-
        if filt.any():
            cr_a = (1 + results_df_rf.loc[filt, 'Allocated_Return']).pro
            cr_e = (1 + results_df_rf.loc[filt, 'Equal_Weight_Return']).
            d0, d1 = results_df_rf.loc[filt, 'Predicted_month'].agg(['mi
            print(f"\nCumulative 2000-01-01–{d1.date()}: ML={cr_a:.4f},
```

```
    else:
        print("No preds ≥ 2000-01-01")

    if not results_df_rf.empty:
        cr_a = (1 + results_df_rf['Allocated_Return']).prod() - 1
        cr_e = (1 + results_df_rf['Equal_Weight_Return']).prod() - 1
        d0, d1 = results_df_rf['Predicted_month'].agg(['min','max'])
        print(f"\nTotal {d0.date()}–{d1.date()}: ML={cr_a:.4f}, EW={
    else:
        print("No predictions total.")
```

```
1967-05-30: only 1 rows. Skipping.
1967-06-30: only 2 rows. Skipping.
1967-07-30: only 3 rows. Skipping.
1967-08-30: only 4 rows. Skipping.
1967-09-30: only 5 rows. Skipping.
1967-10-30: only 6 rows. Skipping.
1967-11-30: only 7 rows. Skipping.
1967-12-30: only 8 rows. Skipping.
1968-01-30: only 9 rows. Skipping.
1968-02-29: only 10 rows. Skipping.
1968-03-30: only 11 rows. Skipping.
1968-04-30: only 12 rows. Skipping.
1968-05-30: only 13 rows. Skipping.
1968-06-30: only 14 rows. Skipping.
1968-07-30: only 15 rows. Skipping.
1968-08-30: only 16 rows. Skipping.
1968-09-30: only 17 rows. Skipping.
1968-10-30: only 18 rows. Skipping.
1968-11-30: only 19 rows. Skipping.
1968-12-30: only 20 rows. Skipping.
1969-01-30: only 21 rows. Skipping.
1969-02-28: only 22 rows. Skipping.
1969-03-30: only 23 rows. Skipping.
1969-04-30: only 24 rows. Skipping.
1969-05-30: only 25 rows. Skipping.
1969-06-30: only 26 rows. Skipping.
1969-07-30: only 27 rows. Skipping.
1969-08-30: only 28 rows. Skipping.
1969-09-30: only 29 rows. Skipping.
1969-10-30: only 30 rows. Skipping.
1969-11-30: only 31 rows. Skipping.
1969-12-30: only 32 rows. Skipping.
1970-01-30: only 33 rows. Skipping.
1970-02-28: only 34 rows. Skipping.
1970-03-30: only 35 rows. Skipping.
1970-04-30: only 36 rows. Skipping.
1970-05-30: only 37 rows. Skipping.
1970-06-30: only 38 rows. Skipping.
1970-07-30: only 39 rows. Skipping.
1970-08-30: only 40 rows. Skipping.
1970-09-30: only 41 rows. Skipping.
1970-10-30: only 42 rows. Skipping.
```

```
1970-11-30: only 43 rows. Skipping.
1970-12-30: only 44 rows. Skipping.
1971-01-30: only 45 rows. Skipping.
1971-02-28: only 46 rows. Skipping.
1971-03-30: only 47 rows. Skipping.
1971-04-30: only 48 rows. Skipping.
1971-05-30: only 49 rows. Skipping.
1971-06-30: only 50 rows. Skipping.
1971-07-30: only 51 rows. Skipping.
1971-08-30: only 52 rows. Skipping.
1971-09-30: only 53 rows. Skipping.
1971-10-30: only 54 rows. Skipping.
1971-11-30: only 55 rows. Skipping.
1971-12-30: only 56 rows. Skipping.
1972-01-30: only 57 rows. Skipping.
1972-02-29: only 58 rows. Skipping.
1972-03-30: only 59 rows. Skipping.
1972-04-30: trained 1967-04-30—1972-03-30 → predicted
1972-05-30: trained 1967-05-30—1972-04-30 → predicted
1972-06-30: trained 1967-06-30—1972-05-30 → predicted
1972-07-30: trained 1967-07-30—1972-06-30 → predicted
1972-08-30: trained 1967-08-30—1972-07-30 → predicted
1972-09-30: trained 1967-09-30—1972-08-30 → predicted
1972-10-30: trained 1967-10-30—1972-09-30 → predicted
1972-11-30: trained 1967-11-30—1972-10-30 → predicted
1972-12-30: trained 1967-12-30—1972-11-30 → predicted
1973-01-30: trained 1968-01-30—1972-12-30 → predicted
1973-02-28: trained 1968-02-29—1973-01-30 → predicted
1973-03-30: trained 1968-03-30—1973-02-28 → predicted
1973-04-30: trained 1968-04-30—1973-03-30 → predicted
1973-05-30: trained 1968-05-30—1973-04-30 → predicted
1973-06-30: trained 1968-06-30—1973-05-30 → predicted
1973-07-30: trained 1968-07-30—1973-06-30 → predicted
1973-08-30: trained 1968-08-30—1973-07-30 → predicted
1973-09-30: trained 1968-09-30—1973-08-30 → predicted
1973-10-30: trained 1968-10-30—1973-09-30 → predicted
1973-11-30: trained 1968-11-30—1973-10-30 → predicted
1973-12-30: trained 1968-12-30—1973-11-30 → predicted
1974-01-30: trained 1969-01-30—1973-12-30 → predicted
1974-02-28: trained 1969-02-28—1974-01-30 → predicted
1974-03-30: trained 1969-03-30—1974-02-28 → predicted
1974-04-30: trained 1969-04-30—1974-03-30 → predicted
1974-05-30: trained 1969-05-30—1974-04-30 → predicted
1974-06-30: trained 1969-06-30—1974-05-30 → predicted
1974-07-30: trained 1969-07-30—1974-06-30 → predicted
1974-08-30: trained 1969-08-30—1974-07-30 → predicted
1974-09-30: trained 1969-09-30—1974-08-30 → predicted
1974-10-30: trained 1969-10-30—1974-09-30 → predicted
1974-11-30: trained 1969-11-30—1974-10-30 → predicted
1974-12-30: trained 1969-12-30—1974-11-30 → predicted
1975-01-30: trained 1970-01-30—1974-12-30 → predicted
1975-02-28: trained 1970-02-28—1975-01-30 → predicted
1975-03-30: trained 1970-03-30—1975-02-28 → predicted
1975-04-30: trained 1970-04-30—1975-03-30 → predicted
```

```
1975-05-30: trained 1970-05-30—1975-04-30 → predicted
1975-06-30: trained 1970-06-30—1975-05-30 → predicted
1975-07-30: trained 1970-07-30—1975-06-30 → predicted
1975-08-30: trained 1970-08-30—1975-07-30 → predicted
1975-09-30: trained 1970-09-30—1975-08-30 → predicted
1975-10-30: trained 1970-10-30—1975-09-30 → predicted
1975-11-30: trained 1970-11-30—1975-10-30 → predicted
1975-12-30: trained 1970-12-30—1975-11-30 → predicted
1976-01-30: trained 1971-01-30—1975-12-30 → predicted
1976-02-29: trained 1971-02-28—1976-01-30 → predicted
1976-03-30: trained 1971-03-30—1976-02-29 → predicted
1976-04-30: trained 1971-04-30—1976-03-30 → predicted
1976-05-30: trained 1971-05-30—1976-04-30 → predicted
1976-06-30: trained 1971-06-30—1976-05-30 → predicted
1976-07-30: trained 1971-07-30—1976-06-30 → predicted
1976-08-30: trained 1971-08-30—1976-07-30 → predicted
1976-09-30: trained 1971-09-30—1976-08-30 → predicted
1976-10-30: trained 1971-10-30—1976-09-30 → predicted
1976-11-30: trained 1971-11-30—1976-10-30 → predicted
1976-12-30: trained 1971-12-30—1976-11-30 → predicted
1977-01-30: trained 1972-01-30—1976-12-30 → predicted
1977-02-28: trained 1972-02-29—1977-01-30 → predicted
1977-03-30: trained 1972-03-30—1977-02-28 → predicted
1977-04-30: trained 1972-04-30—1977-03-30 → predicted
1977-05-30: trained 1972-05-30—1977-04-30 → predicted
1977-06-30: trained 1972-06-30—1977-05-30 → predicted
1977-07-30: trained 1972-07-30—1977-06-30 → predicted
1977-08-30: trained 1972-08-30—1977-07-30 → predicted
1977-09-30: trained 1972-09-30—1977-08-30 → predicted
1977-10-30: trained 1972-10-30—1977-09-30 → predicted
1977-11-30: trained 1972-11-30—1977-10-30 → predicted
1977-12-30: trained 1972-12-30—1977-11-30 → predicted
1978-01-30: trained 1973-01-30—1977-12-30 → predicted
1978-02-28: trained 1973-02-28—1978-01-30 → predicted
1978-03-30: trained 1973-03-30—1978-02-28 → predicted
1978-04-30: trained 1973-04-30—1978-03-30 → predicted
1978-05-30: trained 1973-05-30—1978-04-30 → predicted
1978-06-30: trained 1973-06-30—1978-05-30 → predicted
1978-07-30: trained 1973-07-30—1978-06-30 → predicted
1978-08-30: trained 1973-08-30—1978-07-30 → predicted
1978-09-30: trained 1973-09-30—1978-08-30 → predicted
1978-10-30: trained 1973-10-30—1978-09-30 → predicted
1978-11-30: trained 1973-11-30—1978-10-30 → predicted
1978-12-30: trained 1973-12-30—1978-11-30 → predicted
1979-01-30: trained 1974-01-30—1978-12-30 → predicted
1979-02-28: trained 1974-02-28—1979-01-30 → predicted
1979-03-30: trained 1974-03-30—1979-02-28 → predicted
1979-04-30: trained 1974-04-30—1979-03-30 → predicted
1979-05-30: trained 1974-05-30—1979-04-30 → predicted
1979-06-30: trained 1974-06-30—1979-05-30 → predicted
1979-07-30: trained 1974-07-30—1979-06-30 → predicted
1979-08-30: trained 1974-08-30—1979-07-30 → predicted
1979-09-30: trained 1974-09-30—1979-08-30 → predicted
1979-10-30: trained 1974-10-30—1979-09-30 → predicted
```

```
1979-11-30: trained 1974-11-30—1979-10-30 → predicted
1979-12-30: trained 1974-12-30—1979-11-30 → predicted
1980-01-30: trained 1975-01-30—1979-12-30 → predicted
1980-02-29: trained 1975-02-28—1980-01-30 → predicted
1980-03-30: trained 1975-03-30—1980-02-29 → predicted
1980-04-30: trained 1975-04-30—1980-03-30 → predicted
1980-05-30: trained 1975-05-30—1980-04-30 → predicted
1980-06-30: trained 1975-06-30—1980-05-30 → predicted
1980-07-30: trained 1975-07-30—1980-06-30 → predicted
1980-08-30: trained 1975-08-30—1980-07-30 → predicted
1980-09-30: trained 1975-09-30—1980-08-30 → predicted
1980-10-30: trained 1975-10-30—1980-09-30 → predicted
1980-11-30: trained 1975-11-30—1980-10-30 → predicted
1980-12-30: trained 1975-12-30—1980-11-30 → predicted
1981-01-30: trained 1976-01-30—1980-12-30 → predicted
1981-02-28: trained 1976-02-29—1981-01-30 → predicted
1981-03-30: trained 1976-03-30—1981-02-28 → predicted
1981-04-30: trained 1976-04-30—1981-03-30 → predicted
1981-05-30: trained 1976-05-30—1981-04-30 → predicted
1981-06-30: trained 1976-06-30—1981-05-30 → predicted
1981-07-30: trained 1976-07-30—1981-06-30 → predicted
1981-08-30: trained 1976-08-30—1981-07-30 → predicted
1981-09-30: trained 1976-09-30—1981-08-30 → predicted
1981-10-30: trained 1976-10-30—1981-09-30 → predicted
1981-11-30: trained 1976-11-30—1981-10-30 → predicted
1981-12-30: trained 1976-12-30—1981-11-30 → predicted
1982-01-30: trained 1977-01-30—1981-12-30 → predicted
1982-02-28: trained 1977-02-28—1982-01-30 → predicted
1982-03-30: trained 1977-03-30—1982-02-28 → predicted
1982-04-30: trained 1977-04-30—1982-03-30 → predicted
1982-05-30: trained 1977-05-30—1982-04-30 → predicted
1982-06-30: trained 1977-06-30—1982-05-30 → predicted
1982-07-30: trained 1977-07-30—1982-06-30 → predicted
1982-08-30: trained 1977-08-30—1982-07-30 → predicted
1982-09-30: trained 1977-09-30—1982-08-30 → predicted
1982-10-30: trained 1977-10-30—1982-09-30 → predicted
1982-11-30: trained 1977-11-30—1982-10-30 → predicted
1982-12-30: trained 1977-12-30—1982-11-30 → predicted
1983-01-30: trained 1978-01-30—1982-12-30 → predicted
1983-02-28: trained 1978-02-28—1983-01-30 → predicted
1983-03-30: trained 1978-03-30—1983-02-28 → predicted
1983-04-30: trained 1978-04-30—1983-03-30 → predicted
1983-05-30: trained 1978-05-30—1983-04-30 → predicted
1983-06-30: trained 1978-06-30—1983-05-30 → predicted
1983-07-30: trained 1978-07-30—1983-06-30 → predicted
1983-08-30: trained 1978-08-30—1983-07-30 → predicted
1983-09-30: trained 1978-09-30—1983-08-30 → predicted
1983-10-30: trained 1978-10-30—1983-09-30 → predicted
1983-11-30: trained 1978-11-30—1983-10-30 → predicted
1983-12-30: trained 1978-12-30—1983-11-30 → predicted
1984-01-30: trained 1979-01-30—1983-12-30 → predicted
1984-02-29: trained 1979-02-28—1984-01-30 → predicted
1984-03-30: trained 1979-03-30—1984-02-29 → predicted
1984-04-30: trained 1979-04-30—1984-03-30 → predicted
```

```
1984-05-30: trained 1979-05-30—1984-04-30 → predicted
1984-06-30: trained 1979-06-30—1984-05-30 → predicted
1984-07-30: trained 1979-07-30—1984-06-30 → predicted
1984-08-30: trained 1979-08-30—1984-07-30 → predicted
1984-09-30: trained 1979-09-30—1984-08-30 → predicted
1984-10-30: trained 1979-10-30—1984-09-30 → predicted
1984-11-30: trained 1979-11-30—1984-10-30 → predicted
1984-12-30: trained 1979-12-30—1984-11-30 → predicted
1985-01-30: trained 1980-01-30—1984-12-30 → predicted
1985-02-28: trained 1980-02-29—1985-01-30 → predicted
1985-03-30: trained 1980-03-30—1985-02-28 → predicted
1985-04-30: trained 1980-04-30—1985-03-30 → predicted
1985-05-30: trained 1980-05-30—1985-04-30 → predicted
1985-06-30: trained 1980-06-30—1985-05-30 → predicted
1985-07-30: trained 1980-07-30—1985-06-30 → predicted
1985-08-30: trained 1980-08-30—1985-07-30 → predicted
1985-09-30: trained 1980-09-30—1985-08-30 → predicted
1985-10-30: trained 1980-10-30—1985-09-30 → predicted
1985-11-30: trained 1980-11-30—1985-10-30 → predicted
1985-12-30: trained 1980-12-30—1985-11-30 → predicted
1986-01-30: trained 1981-01-30—1985-12-30 → predicted
1986-02-28: trained 1981-02-28—1986-01-30 → predicted
1986-03-30: trained 1981-03-30—1986-02-28 → predicted
1986-04-30: trained 1981-04-30—1986-03-30 → predicted
1986-05-30: trained 1981-05-30—1986-04-30 → predicted
1986-06-30: trained 1981-06-30—1986-05-30 → predicted
1986-07-30: trained 1981-07-30—1986-06-30 → predicted
1986-08-30: trained 1981-08-30—1986-07-30 → predicted
1986-09-30: trained 1981-09-30—1986-08-30 → predicted
1986-10-30: trained 1981-10-30—1986-09-30 → predicted
1986-11-30: trained 1981-11-30—1986-10-30 → predicted
1986-12-30: trained 1981-12-30—1986-11-30 → predicted
1987-01-30: trained 1982-01-30—1986-12-30 → predicted
1987-02-28: trained 1982-02-28—1987-01-30 → predicted
1987-03-30: trained 1982-03-30—1987-02-28 → predicted
1987-04-30: trained 1982-04-30—1987-03-30 → predicted
1987-05-30: trained 1982-05-30—1987-04-30 → predicted
1987-06-30: trained 1982-06-30—1987-05-30 → predicted
1987-07-30: trained 1982-07-30—1987-06-30 → predicted
1987-08-30: trained 1982-08-30—1987-07-30 → predicted
1987-09-30: trained 1982-09-30—1987-08-30 → predicted
1987-10-30: trained 1982-10-30—1987-09-30 → predicted
1987-11-30: trained 1982-11-30—1987-10-30 → predicted
1987-12-30: trained 1982-12-30—1987-11-30 → predicted
1988-01-30: trained 1983-01-30—1987-12-30 → predicted
1988-02-29: trained 1983-02-28—1988-01-30 → predicted
1988-03-30: trained 1983-03-30—1988-02-29 → predicted
1988-04-30: trained 1983-04-30—1988-03-30 → predicted
1988-05-30: trained 1983-05-30—1988-04-30 → predicted
1988-06-30: trained 1983-06-30—1988-05-30 → predicted
1988-07-30: trained 1983-07-30—1988-06-30 → predicted
1988-08-30: trained 1983-08-30—1988-07-30 → predicted
1988-09-30: trained 1983-09-30—1988-08-30 → predicted
```

```
1988-11-30: trained 1983-11-30—1988-10-30 → predicted
1988-12-30: trained 1983-12-30—1988-11-30 → predicted
1989-01-30: trained 1984-01-30—1988-12-30 → predicted
1989-02-28: trained 1984-02-29—1989-01-30 → predicted
1989-03-30: trained 1984-03-30—1989-02-28 → predicted
1989-04-30: trained 1984-04-30—1989-03-30 → predicted
1989-05-30: trained 1984-05-30—1989-04-30 → predicted
1989-06-30: trained 1984-06-30—1989-05-30 → predicted
1989-07-30: trained 1984-07-30—1989-06-30 → predicted
1989-08-30: trained 1984-08-30—1989-07-30 → predicted
1989-09-30: trained 1984-09-30—1989-08-30 → predicted
1989-10-30: trained 1984-10-30—1989-09-30 → predicted
1989-11-30: trained 1984-11-30—1989-10-30 → predicted
1989-12-30: trained 1984-12-30—1989-11-30 → predicted
1990-01-30: trained 1985-01-30—1989-12-30 → predicted
1990-02-28: trained 1985-02-28—1990-01-30 → predicted
1990-03-30: trained 1985-03-30—1990-02-28 → predicted
1990-04-30: trained 1985-04-30—1990-03-30 → predicted
1990-05-30: trained 1985-05-30—1990-04-30 → predicted
1990-06-30: trained 1985-06-30—1990-05-30 → predicted
1990-07-30: trained 1985-07-30—1990-06-30 → predicted
1990-08-30: trained 1985-08-30—1990-07-30 → predicted
1990-09-30: trained 1985-09-30—1990-08-30 → predicted
1990-10-30: trained 1985-10-30—1990-09-30 → predicted
1990-11-30: trained 1985-11-30—1990-10-30 → predicted
1990-12-30: trained 1985-12-30—1990-11-30 → predicted
1991-01-30: trained 1986-01-30—1990-12-30 → predicted
1991-02-28: trained 1986-02-28—1991-01-30 → predicted
1991-03-30: trained 1986-03-30—1991-02-28 → predicted
1991-04-30: trained 1986-04-30—1991-03-30 → predicted
1991-05-30: trained 1986-05-30—1991-04-30 → predicted
1991-06-30: trained 1986-06-30—1991-05-30 → predicted
1991-07-30: trained 1986-07-30—1991-06-30 → predicted
1991-08-30: trained 1986-08-30—1991-07-30 → predicted
1991-09-30: trained 1986-09-30—1991-08-30 → predicted
1991-10-30: trained 1986-10-30—1991-09-30 → predicted
1991-11-30: trained 1986-11-30—1991-10-30 → predicted
1991-12-30: trained 1986-12-30—1991-11-30 → predicted
1992-01-30: trained 1987-01-30—1991-12-30 → predicted
1992-02-29: trained 1987-02-28—1992-01-30 → predicted
1992-03-30: trained 1987-03-30—1992-02-29 → predicted
1992-04-30: trained 1987-04-30—1992-03-30 → predicted
1992-05-30: trained 1987-05-30—1992-04-30 → predicted
1992-06-30: trained 1987-06-30—1992-05-30 → predicted
1992-07-30: trained 1987-07-30—1992-06-30 → predicted
1992-08-30: trained 1987-08-30—1992-07-30 → predicted
1992-09-30: trained 1987-09-30—1992-08-30 → predicted
1992-10-30: trained 1987-10-30—1992-09-30 → predicted
1992-11-30: trained 1987-11-30—1992-10-30 → predicted
1992-12-30: trained 1987-12-30—1992-11-30 → predicted
1993-01-30: trained 1988-01-30—1992-12-30 → predicted
1993-02-28: trained 1988-02-29—1993-01-30 → predicted
1993-03-30: trained 1988-03-30—1993-02-28 → predicted
```

```
1993-04-30: trained 1988-04-30—1993-03-30 → predicted
1993-05-30: trained 1988-05-30—1993-04-30 → predicted
1993-06-30: trained 1988-06-30—1993-05-30 → predicted
1993-07-30: trained 1988-07-30—1993-06-30 → predicted
1993-08-30: trained 1988-08-30—1993-07-30 → predicted
1993-09-30: trained 1988-09-30—1993-08-30 → predicted
1993-10-30: trained 1988-10-30—1993-09-30 → predicted
1993-11-30: trained 1988-11-30—1993-10-30 → predicted
1993-12-30: trained 1988-12-30—1993-11-30 → predicted
1994-01-30: trained 1989-01-30—1993-12-30 → predicted
1994-02-28: trained 1989-02-28—1994-01-30 → predicted
1994-03-30: trained 1989-03-30—1994-02-28 → predicted
1994-04-30: trained 1989-04-30—1994-03-30 → predicted
1994-05-30: trained 1989-05-30—1994-04-30 → predicted
1994-06-30: trained 1989-06-30—1994-05-30 → predicted
1994-07-30: trained 1989-07-30—1994-06-30 → predicted
1994-08-30: trained 1989-08-30—1994-07-30 → predicted
1994-09-30: trained 1989-09-30—1994-08-30 → predicted
1994-10-30: trained 1989-10-30—1994-09-30 → predicted
1994-11-30: trained 1989-11-30—1994-10-30 → predicted
1994-12-30: trained 1989-12-30—1994-11-30 → predicted
1995-01-30: trained 1990-01-30—1994-12-30 → predicted
1995-02-28: trained 1990-02-28—1995-01-30 → predicted
1995-03-30: trained 1990-03-30—1995-02-28 → predicted
1995-04-30: trained 1990-04-30—1995-03-30 → predicted
1995-05-30: trained 1990-05-30—1995-04-30 → predicted
1995-06-30: trained 1990-06-30—1995-05-30 → predicted
1995-07-30: trained 1990-07-30—1995-06-30 → predicted
1995-08-30: trained 1990-08-30—1995-07-30 → predicted
1995-09-30: trained 1990-09-30—1995-08-30 → predicted
1995-10-30: trained 1990-10-30—1995-09-30 → predicted
1995-11-30: trained 1990-11-30—1995-10-30 → predicted
1995-12-30: trained 1990-12-30—1995-11-30 → predicted
1996-01-30: trained 1991-01-30—1995-12-30 → predicted
1996-02-29: trained 1991-02-28—1996-01-30 → predicted
1996-03-30: trained 1991-03-30—1996-02-29 → predicted
1996-04-30: trained 1991-04-30—1996-03-30 → predicted
1996-05-30: trained 1991-05-30—1996-04-30 → predicted
1996-06-30: trained 1991-06-30—1996-05-30 → predicted
1996-07-30: trained 1991-07-30—1996-06-30 → predicted
1996-08-30: trained 1991-08-30—1996-07-30 → predicted
1996-09-30: trained 1991-09-30—1996-08-30 → predicted
1996-10-30: trained 1991-10-30—1996-09-30 → predicted
1996-11-30: trained 1991-11-30—1996-10-30 → predicted
1996-12-30: trained 1991-12-30—1996-11-30 → predicted
1997-01-30: trained 1992-01-30—1996-12-30 → predicted
1997-02-28: trained 1992-02-29—1997-01-30 → predicted
1997-03-30: trained 1992-03-30—1997-02-28 → predicted
1997-04-30: trained 1992-04-30—1997-03-30 → predicted
1997-05-30: trained 1992-05-30—1997-04-30 → predicted
1997-06-30: trained 1992-06-30—1997-05-30 → predicted
1997-07-30: trained 1992-07-30—1997-06-30 → predicted
1997-08-30: trained 1992-08-30—1997-07-30 → predicted
1997-09-30: trained 1992-09-30—1997-08-30 → predicted
1997-10-30: trained 1992-10-30—1997-09-30 → predicted
```

```
1997-10-30: trained 1992-10-30—1997-09-30 → predicted
1997-11-30: trained 1992-11-30—1997-10-30 → predicted
1997-12-30: trained 1992-12-30—1997-11-30 → predicted
1998-01-30: trained 1993-01-30—1997-12-30 → predicted
1998-02-28: trained 1993-02-28—1998-01-30 → predicted
1998-03-30: trained 1993-03-30—1998-02-28 → predicted
1998-04-30: trained 1993-04-30—1998-03-30 → predicted
1998-05-30: trained 1993-05-30—1998-04-30 → predicted
1998-06-30: trained 1993-06-30—1998-05-30 → predicted
1998-07-30: trained 1993-07-30—1998-06-30 → predicted
1998-08-30: trained 1993-08-30—1998-07-30 → predicted
1998-09-30: trained 1993-09-30—1998-08-30 → predicted
1998-10-30: trained 1993-10-30—1998-09-30 → predicted
1998-11-30: trained 1993-11-30—1998-10-30 → predicted
1998-12-30: trained 1993-12-30—1998-11-30 → predicted
1999-01-30: trained 1994-01-30—1998-12-30 → predicted
1999-02-28: trained 1994-02-28—1999-01-30 → predicted
1999-03-30: trained 1994-03-30—1999-02-28 → predicted
1999-04-30: trained 1994-04-30—1999-03-30 → predicted
1999-05-30: trained 1994-05-30—1999-04-30 → predicted
1999-06-30: trained 1994-06-30—1999-05-30 → predicted
1999-07-30: trained 1994-07-30—1999-06-30 → predicted
1999-08-30: trained 1994-08-30—1999-07-30 → predicted
1999-09-30: trained 1994-09-30—1999-08-30 → predicted
1999-10-30: trained 1994-10-30—1999-09-30 → predicted
1999-11-30: trained 1994-11-30—1999-10-30 → predicted
1999-12-30: trained 1994-12-30—1999-11-30 → predicted
2000-01-30: trained 1995-01-30—1999-12-30 → predicted
2000-02-29: trained 1995-02-28—2000-01-30 → predicted
2000-03-30: trained 1995-03-30—2000-02-29 → predicted
2000-04-30: trained 1995-04-30—2000-03-30 → predicted
2000-05-30: trained 1995-05-30—2000-04-30 → predicted
2000-06-30: trained 1995-06-30—2000-05-30 → predicted
2000-07-30: trained 1995-07-30—2000-06-30 → predicted
2000-08-30: trained 1995-08-30—2000-07-30 → predicted
2000-09-30: trained 1995-09-30—2000-08-30 → predicted
2000-10-30: trained 1995-10-30—2000-09-30 → predicted
2000-11-30: trained 1995-11-30—2000-10-30 → predicted
2000-12-30: trained 1995-12-30—2000-11-30 → predicted
2001-01-30: trained 1996-01-30—2000-12-30 → predicted
2001-02-28: trained 1996-02-29—2001-01-30 → predicted
2001-03-30: trained 1996-03-30—2001-02-28 → predicted
2001-04-30: trained 1996-04-30—2001-03-30 → predicted
2001-05-30: trained 1996-05-30—2001-04-30 → predicted
2001-06-30: trained 1996-06-30—2001-05-30 → predicted
2001-07-30: trained 1996-07-30—2001-06-30 → predicted
2001-08-30: trained 1996-08-30—2001-07-30 → predicted
2001-09-30: trained 1996-09-30—2001-08-30 → predicted
2001-10-30: trained 1996-10-30—2001-09-30 → predicted
2001-11-30: trained 1996-11-30—2001-10-30 → predicted
2001-12-30: trained 1996-12-30—2001-11-30 → predicted
2002-01-30: trained 1997-01-30—2001-12-30 → predicted
2002-02-28: trained 1997-02-28—2002-01-30 → predicted
2002-03-30: trained 1997-03-30—2002-02-28 → predicted
2002-04-30: trained 1997-04-30—2002-03-30 → predicted
```

```
2002-04-30: trained 1997-04-30–2002-03-30 → predicted
2002-05-30: trained 1997-05-30–2002-04-30 → predicted
2002-06-30: trained 1997-06-30–2002-05-30 → predicted
2002-07-30: trained 1997-07-30–2002-06-30 → predicted
2002-08-30: trained 1997-08-30–2002-07-30 → predicted
2002-09-30: trained 1997-09-30–2002-08-30 → predicted
2002-10-30: trained 1997-10-30–2002-09-30 → predicted
2002-11-30: trained 1997-11-30–2002-10-30 → predicted
2002-12-30: trained 1997-12-30–2002-11-30 → predicted
2003-01-30: trained 1998-01-30–2002-12-30 → predicted
2003-02-28: trained 1998-02-28–2003-01-30 → predicted
2003-03-30: trained 1998-03-30–2003-02-28 → predicted
2003-04-30: trained 1998-04-30–2003-03-30 → predicted
2003-05-30: trained 1998-05-30–2003-04-30 → predicted
2003-06-30: trained 1998-06-30–2003-05-30 → predicted
2003-07-30: trained 1998-07-30–2003-06-30 → predicted
2003-08-30: trained 1998-08-30–2003-07-30 → predicted
2003-09-30: trained 1998-09-30–2003-08-30 → predicted
2003-10-30: trained 1998-10-30–2003-09-30 → predicted
2003-11-30: trained 1998-11-30–2003-10-30 → predicted
2003-12-30: trained 1998-12-30–2003-11-30 → predicted
2004-01-30: trained 1999-01-30–2003-12-30 → predicted
2004-02-29: trained 1999-02-28–2004-01-30 → predicted
2004-03-30: trained 1999-03-30–2004-02-29 → predicted
2004-04-30: trained 1999-04-30–2004-03-30 → predicted
2004-05-30: trained 1999-05-30–2004-04-30 → predicted
2004-06-30: trained 1999-06-30–2004-05-30 → predicted
2004-07-30: trained 1999-07-30–2004-06-30 → predicted
2004-08-30: trained 1999-08-30–2004-07-30 → predicted
2004-09-30: trained 1999-09-30–2004-08-30 → predicted
2004-10-30: trained 1999-10-30–2004-09-30 → predicted
2004-11-30: trained 1999-11-30–2004-10-30 → predicted
2004-12-30: trained 1999-12-30–2004-11-30 → predicted
2005-01-30: trained 2000-01-30–2004-12-30 → predicted
2005-02-28: trained 2000-02-29–2005-01-30 → predicted
2005-03-30: trained 2000-03-30–2005-02-28 → predicted
2005-04-30: trained 2000-04-30–2005-03-30 → predicted
2005-05-30: trained 2000-05-30–2005-04-30 → predicted
2005-06-30: trained 2000-06-30–2005-05-30 → predicted
2005-07-30: trained 2000-07-30–2005-06-30 → predicted
2005-08-30: trained 2000-08-30–2005-07-30 → predicted
2005-09-30: trained 2000-09-30–2005-08-30 → predicted
2005-10-30: trained 2000-10-30–2005-09-30 → predicted
2005-11-30: trained 2000-11-30–2005-10-30 → predicted
2005-12-30: trained 2000-12-30–2005-11-30 → predicted
2006-01-30: trained 2001-01-30–2005-12-30 → predicted
2006-02-28: trained 2001-02-28–2006-01-30 → predicted
2006-03-30: trained 2001-03-30–2006-02-28 → predicted
2006-04-30: trained 2001-04-30–2006-03-30 → predicted
2006-05-30: trained 2001-05-30–2006-04-30 → predicted
2006-06-30: trained 2001-06-30–2006-05-30 → predicted
2006-07-30: trained 2001-07-30–2006-06-30 → predicted
2006-08-30: trained 2001-08-30–2006-07-30 → predicted
2006-09-30: trained 2001-09-30–2006-08-30 → predicted
2006-10-30: trained 2001-10-30–2006-09-30 → predicted
```

```
2006-10-30: trained 2001-10-30–2006-09-30 → predicted
2006-11-30: trained 2001-11-30–2006-10-30 → predicted
2006-12-30: trained 2001-12-30–2006-11-30 → predicted
2007-01-30: trained 2002-01-30–2006-12-30 → predicted
2007-02-28: trained 2002-02-28–2007-01-30 → predicted
2007-03-30: trained 2002-03-30–2007-02-28 → predicted
2007-04-30: trained 2002-04-30–2007-03-30 → predicted
2007-05-30: trained 2002-05-30–2007-04-30 → predicted
2007-06-30: trained 2002-06-30–2007-05-30 → predicted
2007-07-30: trained 2002-07-30–2007-06-30 → predicted
2007-08-30: trained 2002-08-30–2007-07-30 → predicted
2007-09-30: trained 2002-09-30–2007-08-30 → predicted
2007-10-30: trained 2002-10-30–2007-09-30 → predicted
2007-11-30: trained 2002-11-30–2007-10-30 → predicted
2007-12-30: trained 2002-12-30–2007-11-30 → predicted
2008-01-30: trained 2003-01-30–2007-12-30 → predicted
2008-02-29: trained 2003-02-28–2008-01-30 → predicted
2008-03-30: trained 2003-03-30–2008-02-29 → predicted
2008-04-30: trained 2003-04-30–2008-03-30 → predicted
2008-05-30: trained 2003-05-30–2008-04-30 → predicted
2008-06-30: trained 2003-06-30–2008-05-30 → predicted
2008-07-30: trained 2003-07-30–2008-06-30 → predicted
2008-08-30: trained 2003-08-30–2008-07-30 → predicted
2008-09-30: trained 2003-09-30–2008-08-30 → predicted
2008-10-30: trained 2003-10-30–2008-09-30 → predicted
2008-11-30: trained 2003-11-30–2008-10-30 → predicted
2008-12-30: trained 2003-12-30–2008-11-30 → predicted
2009-01-30: trained 2004-01-30–2008-12-30 → predicted
2009-02-28: trained 2004-02-29–2009-01-30 → predicted
2009-03-30: trained 2004-03-30–2009-02-28 → predicted
2009-04-30: trained 2004-04-30–2009-03-30 → predicted
2009-05-30: trained 2004-05-30–2009-04-30 → predicted
2009-06-30: trained 2004-06-30–2009-05-30 → predicted
2009-07-30: trained 2004-07-30–2009-06-30 → predicted
2009-08-30: trained 2004-08-30–2009-07-30 → predicted
2009-09-30: trained 2004-09-30–2009-08-30 → predicted
2009-10-30: trained 2004-10-30–2009-09-30 → predicted
2009-11-30: trained 2004-11-30–2009-10-30 → predicted
2009-12-30: trained 2004-12-30–2009-11-30 → predicted
2010-01-30: trained 2005-01-30–2009-12-30 → predicted
2010-02-28: trained 2005-02-28–2010-01-30 → predicted
2010-03-30: trained 2005-03-30–2010-02-28 → predicted
2010-04-30: trained 2005-04-30–2010-03-30 → predicted
2010-05-30: trained 2005-05-30–2010-04-30 → predicted
2010-06-30: trained 2005-06-30–2010-05-30 → predicted
2010-07-30: trained 2005-07-30–2010-06-30 → predicted
2010-08-30: trained 2005-08-30–2010-07-30 → predicted
2010-09-30: trained 2005-09-30–2010-08-30 → predicted
2010-10-30: trained 2005-10-30–2010-09-30 → predicted
2010-11-30: trained 2005-11-30–2010-10-30 → predicted
2010-12-30: trained 2005-12-30–2010-11-30 → predicted
2011-01-30: trained 2006-01-30–2010-12-30 → predicted
2011-02-28: trained 2006-02-28–2011-01-30 → predicted
2011-03-30: trained 2006-03-30–2011-02-28 → predicted
```

```
2011-04-30: trained 2006-04-30—2011-03-30 → predicted
2011-05-30: trained 2006-05-30—2011-04-30 → predicted
2011-06-30: trained 2006-06-30—2011-05-30 → predicted
2011-07-30: trained 2006-07-30—2011-06-30 → predicted
2011-08-30: trained 2006-08-30—2011-07-30 → predicted
2011-09-30: trained 2006-09-30—2011-08-30 → predicted
2011-10-30: trained 2006-10-30—2011-09-30 → predicted
2011-11-30: trained 2006-11-30—2011-10-30 → predicted
2011-12-30: trained 2006-12-30—2011-11-30 → predicted
2012-01-30: trained 2007-01-30—2011-12-30 → predicted
2012-02-29: trained 2007-02-28—2012-01-30 → predicted
2012-03-30: trained 2007-03-30—2012-02-29 → predicted
2012-04-30: trained 2007-04-30—2012-03-30 → predicted
2012-05-30: trained 2007-05-30—2012-04-30 → predicted
2012-06-30: trained 2007-06-30—2012-05-30 → predicted
2012-07-30: trained 2007-07-30—2012-06-30 → predicted
2012-08-30: trained 2007-08-30—2012-07-30 → predicted
2012-09-30: trained 2007-09-30—2012-08-30 → predicted
2012-10-30: trained 2007-10-30—2012-09-30 → predicted
2012-11-30: trained 2007-11-30—2012-10-30 → predicted
2012-12-30: trained 2007-12-30—2012-11-30 → predicted
2013-01-30: trained 2008-01-30—2012-12-30 → predicted
2013-02-28: trained 2008-02-29—2013-01-30 → predicted
2013-03-30: trained 2008-03-30—2013-02-28 → predicted
2013-04-30: trained 2008-04-30—2013-03-30 → predicted
2013-05-30: trained 2008-05-30—2013-04-30 → predicted
2013-06-30: trained 2008-06-30—2013-05-30 → predicted
2013-07-30: trained 2008-07-30—2013-06-30 → predicted
2013-08-30: trained 2008-08-30—2013-07-30 → predicted
2013-09-30: trained 2008-09-30—2013-08-30 → predicted
2013-10-30: trained 2008-10-30—2013-09-30 → predicted
2013-11-30: trained 2008-11-30—2013-10-30 → predicted
2013-12-30: trained 2008-12-30—2013-11-30 → predicted
2014-01-30: trained 2009-01-30—2013-12-30 → predicted
2014-02-28: trained 2009-02-28—2014-01-30 → predicted
2014-03-30: trained 2009-03-30—2014-02-28 → predicted
2014-04-30: trained 2009-04-30—2014-03-30 → predicted
2014-05-30: trained 2009-05-30—2014-04-30 → predicted
2014-06-30: trained 2009-06-30—2014-05-30 → predicted
2014-07-30: trained 2009-07-30—2014-06-30 → predicted
2014-08-30: trained 2009-08-30—2014-07-30 → predicted
2014-09-30: trained 2009-09-30—2014-08-30 → predicted
2014-10-30: trained 2009-10-30—2014-09-30 → predicted
2014-11-30: trained 2009-11-30—2014-10-30 → predicted
2014-12-30: trained 2009-12-30—2014-11-30 → predicted
2015-01-30: trained 2010-01-30—2014-12-30 → predicted
2015-02-28: trained 2010-02-28—2015-01-30 → predicted
2015-03-30: trained 2010-03-30—2015-02-28 → predicted
2015-04-30: trained 2010-04-30—2015-03-30 → predicted
2015-05-30: trained 2010-05-30—2015-04-30 → predicted
2015-06-30: trained 2010-06-30—2015-05-30 → predicted
2015-07-30: trained 2010-07-30—2015-06-30 → predicted
2015-08-30: trained 2010-08-30—2015-07-30 → predicted
2015-09-30: trained 2010-09-30—2015-08-30 → predicted
```

```
2015-10-30: trained 2010-10-30—2015-09-30 → predicted
2015-11-30: trained 2010-11-30—2015-10-30 → predicted
2015-12-30: trained 2010-12-30—2015-11-30 → predicted
2016-01-30: trained 2011-01-30—2015-12-30 → predicted
2016-02-29: trained 2011-02-28—2016-01-30 → predicted
2016-03-30: trained 2011-03-30—2016-02-29 → predicted
2016-04-30: trained 2011-04-30—2016-03-30 → predicted
2016-05-30: trained 2011-05-30—2016-04-30 → predicted
2016-06-30: trained 2011-06-30—2016-05-30 → predicted
2016-07-30: trained 2011-07-30—2016-06-30 → predicted
2016-08-30: trained 2011-08-30—2016-07-30 → predicted
2016-09-30: trained 2011-09-30—2016-08-30 → predicted
2016-10-30: trained 2011-10-30—2016-09-30 → predicted
2016-11-30: trained 2011-11-30—2016-10-30 → predicted
2016-12-30: trained 2011-12-30—2016-11-30 → predicted
2017-01-30: trained 2012-01-30—2016-12-30 → predicted
2017-02-28: trained 2012-02-29—2017-01-30 → predicted
2017-03-30: trained 2012-03-30—2017-02-28 → predicted
2017-04-30: trained 2012-04-30—2017-03-30 → predicted
2017-05-30: trained 2012-05-30—2017-04-30 → predicted
2017-06-30: trained 2012-06-30—2017-05-30 → predicted
2017-07-30: trained 2012-07-30—2017-06-30 → predicted
2017-08-30: trained 2012-08-30—2017-07-30 → predicted
2017-09-30: trained 2012-09-30—2017-08-30 → predicted
2017-10-30: trained 2012-10-30—2017-09-30 → predicted
2017-11-30: trained 2012-11-30—2017-10-30 → predicted
2017-12-30: trained 2012-12-30—2017-11-30 → predicted
2018-01-30: trained 2013-01-30—2017-12-30 → predicted
2018-02-28: trained 2013-02-28—2018-01-30 → predicted
2018-03-30: trained 2013-03-30—2018-02-28 → predicted
2018-04-30: trained 2013-04-30—2018-03-30 → predicted
2018-05-30: trained 2013-05-30—2018-04-30 → predicted
2018-06-30: trained 2013-06-30—2018-05-30 → predicted
2018-07-30: trained 2013-07-30—2018-06-30 → predicted
2018-08-30: trained 2013-08-30—2018-07-30 → predicted
2018-09-30: trained 2013-09-30—2018-08-30 → predicted
2018-10-30: trained 2013-10-30—2018-09-30 → predicted
2018-11-30: trained 2013-11-30—2018-10-30 → predicted
2018-12-30: trained 2013-12-30—2018-11-30 → predicted
2019-01-30: trained 2014-01-30—2018-12-30 → predicted
2019-02-28: trained 2014-02-28—2019-01-30 → predicted
2019-03-30: trained 2014-03-30—2019-02-28 → predicted
2019-04-30: trained 2014-04-30—2019-03-30 → predicted
2019-05-30: trained 2014-05-30—2019-04-30 → predicted
2019-06-30: trained 2014-06-30—2019-05-30 → predicted
2019-07-30: trained 2014-07-30—2019-06-30 → predicted
2019-08-30: trained 2014-08-30—2019-07-30 → predicted
2019-09-30: trained 2014-09-30—2019-08-30 → predicted
2019-10-30: trained 2014-10-30—2019-09-30 → predicted
2019-11-30: trained 2014-11-30—2019-10-30 → predicted
2019-12-30: trained 2014-12-30—2019-11-30 → predicted
2020-01-30: trained 2015-01-30—2019-12-30 → predicted
2020-02-29: trained 2015-02-28—2020-01-30 → predicted
2020-03-30: trained 2015-03-30—2020-02-29 → predicted
```

```
2020-04-30: trained 2015-04-30—2020-03-30 → predicted
2020-05-30: trained 2015-05-30—2020-04-30 → predicted
2020-06-30: trained 2015-06-30—2020-05-30 → predicted
2020-07-30: trained 2015-07-30—2020-06-30 → predicted
2020-08-30: trained 2015-08-30—2020-07-30 → predicted
2020-09-30: trained 2015-09-30—2020-08-30 → predicted
2020-10-30: trained 2015-10-30—2020-09-30 → predicted
2020-11-30: trained 2015-11-30—2020-10-30 → predicted
2020-12-30: trained 2015-12-30—2020-11-30 → predicted
2021-01-30: trained 2016-01-30—2020-12-30 → predicted
2021-02-28: trained 2016-02-29—2021-01-30 → predicted
2021-03-30: trained 2016-03-30—2021-02-28 → predicted
2021-04-30: trained 2016-04-30—2021-03-30 → predicted
2021-05-30: trained 2016-05-30—2021-04-30 → predicted
2021-06-30: trained 2016-06-30—2021-05-30 → predicted
2021-07-30: trained 2016-07-30—2021-06-30 → predicted
2021-08-30: trained 2016-08-30—2021-07-30 → predicted
2021-09-30: trained 2016-09-30—2021-08-30 → predicted
2021-10-30: trained 2016-10-30—2021-09-30 → predicted
2021-11-30: trained 2016-11-30—2021-10-30 → predicted
2021-12-30: trained 2016-12-30—2021-11-30 → predicted
2022-01-30: trained 2017-01-30—2021-12-30 → predicted
2022-02-28: trained 2017-02-28—2022-01-30 → predicted
2022-03-30: trained 2017-03-30—2022-02-28 → predicted
2022-04-30: trained 2017-04-30—2022-03-30 → predicted
2022-05-30: trained 2017-05-30—2022-04-30 → predicted
2022-06-30: trained 2017-06-30—2022-05-30 → predicted
2022-07-30: trained 2017-07-30—2022-06-30 → predicted
2022-08-30: trained 2017-08-30—2022-07-30 → predicted
2022-09-30: trained 2017-09-30—2022-08-30 → predicted
2022-10-30: trained 2017-10-30—2022-09-30 → predicted
2022-11-30: trained 2017-11-30—2022-10-30 → predicted
2022-12-30: trained 2017-12-30—2022-11-30 → predicted
2023-01-30: trained 2018-01-30—2022-12-30 → predicted
2023-02-28: trained 2018-02-28—2023-01-30 → predicted
2023-03-30: trained 2018-03-30—2023-02-28 → predicted
2023-04-30: trained 2018-04-30—2023-03-30 → predicted
2023-05-30: trained 2018-05-30—2023-04-30 → predicted
2023-06-30: trained 2018-06-30—2023-05-30 → predicted
2023-07-30: trained 2018-07-30—2023-06-30 → predicted
2023-08-30: trained 2018-08-30—2023-07-30 → predicted
2023-09-30: trained 2018-09-30—2023-08-30 → predicted
2023-10-30: trained 2018-10-30—2023-09-30 → predicted
2023-11-30: trained 2018-11-30—2023-10-30 → predicted
2023-12-30: trained 2018-12-30—2023-11-30 → predicted
2024-01-30: trained 2019-01-30—2023-12-30 → predicted
2024-02-29: trained 2019-02-28—2024-01-30 → predicted
2024-03-30: trained 2019-03-30—2024-02-29 → predicted
2024-04-30: trained 2019-04-30—2024-03-30 → predicted
2024-05-30: trained 2019-05-30—2024-04-30 → predicted
2024-06-30: trained 2019-06-30—2024-05-30 → predicted
2024-07-30: trained 2019-07-30—2024-06-30 → predicted
2024-08-30: trained 2019-08-30—2024-07-30 → predicted
2024-09-30: trained 2019-09-30—2024-08-30 → predicted
```

```
2024-10-30: trained 2019-10-30—2024-09-30 → predicted
2024-11-30: trained 2019-11-30—2024-10-30 → predicted
Columns: ['Regime', 'Predicted_month', 'Train_Start_Date', 'Train_En
```

	Regime	Predicted_month	Train_Start_Date	Train_End_Date	Tra
622	NoRegime	2024-02-29	2019-02-28	2024-01-30	
623	NoRegime	2024-03-30	2019-03-30	2024-02-29	
624	NoRegime	2024-04-30	2019-04-30	2024-03-30	
625	NoRegime	2024-05-30	2019-05-30	2024-04-30	
626	NoRegime	2024-06-30	2019-06-30	2024-05-30	
627	NoRegime	2024-07-30	2019-07-30	2024-06-30	
628	NoRegime	2024-08-30	2019-08-30	2024-07-30	
629	NoRegime	2024-09-30	2019-09-30	2024-08-30	
630	NoRegime	2024-10-30	2019-10-30	2024-09-30	
631	NoRegime	2024-11-30	2019-11-30	2024-10-30	

10 rows × 23 columns

```
Cumulative 2000-01-01—2024-11-30: ML=2.4346, EW=1.1529

Total 1972-04-30—2024-11-30: ML=11.1029, EW=3.5990
```

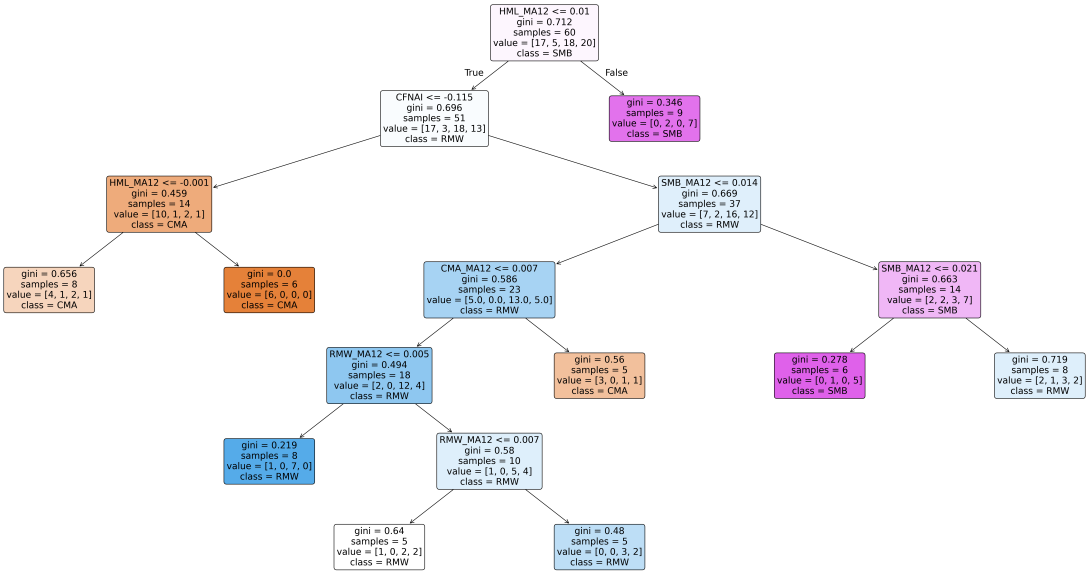
RF tree visualization

```
if RF:
    # ──────────────────────────────────────────────────────────────────────
    # CELL: Interactive selector for prediction date & tree number
    # ──────────────────────────────────────────────────────────────────────
    import ipywidgets as widgets
    from IPython.display import display
    from sklearn import tree
    import matplotlib.pyplot as plt

    # build a list of the dates you predicted on
    prediction_dates = [
        res['Predicted_month'].date()
        for res in results  # or results_df_rf['Predicted_month']
    ]

    # map date → model
    model_by_date = dict(zip(prediction_dates, rf_models))

    # widget for selecting date
    date_widget = widgets.Dropdown(
        options=prediction_dates,
        description='Date:',
    )

    # widget for selecting tree index
    tree_widget = widgets.IntSlider(
        value=0,
        min=0,
        max=rf_params['n_estimators'] - 1,
        step=1,
        description='Tree #:',
    )

    def plot_tree_for_selection(chosen_date, tree_idx):
        model     = model_by_date[chosen_date]
        estimator = model.estimators_[tree_idx]

        # extra-large figure & text for bigger boxes and labels
        plt.figure(figsize=(30, 15), dpi=300)
        tree.plot_tree(
            estimator,
            feature_names=RF1_FEATURES,
            class_names=[str(c) for c in model.classes_],
            filled=True,
            rounded=True,
            fontsize=16,         # bump this up for larger node lab
        )
```

```
        plt.axis('off')         # hide axes
        plt.tight_layout()      # remove extra padding
        plt.show()

    # wire up interactive display
    interactive_ui = widgets.interactive(
        plot_tree_for_selection,
        chosen_date=date_widget,
        tree_idx=tree_widget
    )
    display(interactive_ui)
```

Date: 1972-04-30

Tree #: 0



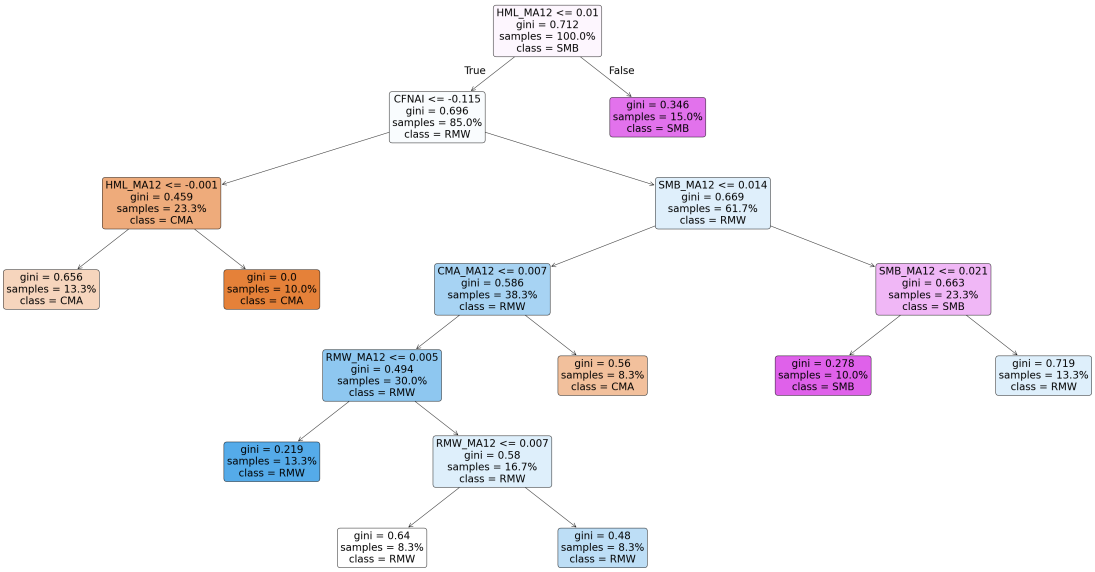

```
import ipywidgets as widgets
from IPython.display import display
from sklearn import tree
import matplotlib.pyplot as plt
import matplotlib as mpl
```

```
# ──────────────────────────────────────────────────────────────────────
# Assumptions: you have these already in your namespace:
#   • results       – list or iterable of dicts/DataFrame rows with
#   • rf_models     – list of fitted RandomForestClassifier/Regress
#   • rf_params     – dict with at least 'n_estimators'
#   • RF1_FEATURES  – list of feature-name strings
# ──────────────────────────────────────────────────────────────────────

prediction_dates = [
    res['Predicted_month'].date()
    for res in results
]
model_by_date = dict(zip(prediction_dates, rf_models))

date_widget = widgets.Dropdown(options=prediction_dates, descriptio
tree_widget = widgets.IntSlider(
    value=0,
    min=0,
    max=rf_params['n_estimators'] - 1,
    step=1,
    description='Tree #:'
)

def plot_tree_for_selection(chosen_date, tree_idx):
    model     = model_by_date[chosen_date]
    estimator = model.estimators_[tree_idx]

    # huge canvas
    plt.figure(figsize=(40, 20), dpi=100)
    ax = plt.gca()

    # plot with big text & proportional boxes
    tree.plot_tree(
        estimator,
        feature_names=RF1_FEATURES,
        class_names=[str(c) for c in model.classes_],
        filled=True,
        rounded=True,
        proportion=True,
        fontsize=24,
        ax=ax
    )

    # pad-out each box
    for patch in ax.patches:
        if isinstance(patch, mpl.patches.FancyBboxPatch):
            patch.set_boxstyle("round,pad=1.2")
```

```
        # REMOVE any "value =" line from the node text
        for txt in ax.texts:
            lines = txt.get_text().split('\n')
            # filter out the value line
            filtered = [ln for ln in lines if not ln.strip().startswith
            txt.set_text('\n'.join(filtered))

        plt.axis('off')
        plt.tight_layout()
        plt.show()

    interactive_ui = widgets.interactive(
        plot_tree_for_selection,
        chosen_date=date_widget,
        tree_idx=tree_widget
    )
    display(interactive_ui)
```

Date: 1972-04-30

Tree #: 0



RF2

```
# Cell 2 — second RF run under RF2
if RF2:
    import pandas as pd
    import numpy as np
    from sklearn.ensemble import RandomForestClassifier
    from IPython.display import display, HTML

    # ——————————————————
    # RF2: Drop _MA12 and GARCH_1M
    # ——————————————————
    RF2_FEATURES = [f for f in FEATURES if not f.endswith('_MA12')
    #RF2_FEATURES = [f for f in FEATURES if f not in ['Cape']]


    # ——————————————————
    # 1) Parameters
    # ——————————————————
    min_months_train    = 60
    min_obs_regime      = 50
    min_obs_train       = 0
    use_regime_split    = False
    default_hyperparams = False

    use_fixed_window    = True
    rolling_window_size = 60

    n_jobs = -1

    # ——————————————————
    # Hyperparameter Settings
    # ——————————————————
    if default_hyperparams:
        rf_params = {
            'n_estimators': 100,
            'max_depth': None,
            'max_features': 'sqrt',
            'min_samples_split': 2,
            'min_samples_leaf': 1,
            'bootstrap': True,
            'n_jobs': n_jobs
        }
    else:
        rf_params = {
            'n_estimators': 100,
```

```
            'max_depth': None,
            'max_features': None,
            'min_samples_split': 2,
            'min_samples_leaf': 5,
            'bootstrap': False,
            'n_jobs': n_jobs
        }

    df_sorted = df.sort_values('Date').reset_index(drop=True)
    results_rf2 = []

    # --------------------
    # 2) Main Loop
    # --------------------
    for i in range(1, len(df_sorted)):
        test_row = df_sorted.iloc[i]
        Predicted_month = test_row['Date']

        # Build window
        if use_fixed_window:
            start_idx = max(0, i - rolling_window_size)
            train_window = df_sorted.iloc[start_idx:i].copy()
        else:
            train_window = df_sorted.iloc[:i].copy()

        # Skip if insufficient data or overlapping dates
        if len(train_window) < min_months_train:
            continue
        last_train_date = train_window['Date'].iloc[-1]
        if (last_train_date.year == Predicted_month.year) and (last
            continue

        # Prepare X_train / y_train using RF2_FEATURES
        X_train = train_window[RF2_FEATURES].dropna()
        y_train = train_window['Winning Factor'].loc[X_train.index]
        if len(X_train) < min_obs_train:
            continue

        # Train RF
        rf_model = RandomForestClassifier(**rf_params, random_state
        rf_model.fit(X_train, y_train)

        # Prepare X_test
        X_test = train_window[RF2_FEATURES].iloc[[-1]].dropna()
        if X_test.empty:
            continue

        # Predict and map probabilities
```

```
            probs = rf_model.predict_proba(X_test)[0]
            full_probs = np.zeros(len(FACTORS))
            for cls, p in zip(rf_model.classes_, probs):
                if cls in FACTORS:
                    full_probs[FACTORS.index(cls)] = p

            # Calculate returns
            allocated_return    = (full_probs * test_row[FACTORS].value
            equal_weight_return = np.mean(test_row[FACTORS].values)

            # Feature levels from RF2_FEATURES
            feature_levels = {f"Feature_Level_{f}": X_test[f].iloc[0] f

            result = {
                'Regime': 'NoRegime',
                'Predicted_month': Predicted_month,
                'Train_Start_Date': train_window['Date'].iloc[0],
                'Train_End_Date': last_train_date,
                'Train_Count': len(X_train),
                'Feature_Importances': rf_model.feature_importances_,
                'Predicted_Probabilities': full_probs,
                'Predicted_Winner': rf_model.classes_[probs.argmax()],
                'Allocated_Return': allocated_return,
                'Equal_Weight_Return': equal_weight_return,
                'Actual_Winner': test_row['Winning Factor'],
                'Num_Trees': rf_model.n_estimators,
                'Average_Tree_Depth': np.mean([t.tree_.max_depth for t
                'Max_Tree_Depth': np.max([t.tree_.max_depth for t in rf
                'Prediction_Horizon_Months': ((Predicted_month.year - l
                                              (Predicted_month.month - l
                **feature_levels
            }
            results_rf2.append(result)

        # ------------------
        # 3) Build RF2 results DataFrame
        # ------------------
        results_df_rf2 = pd.DataFrame(results_rf2)
        print("Final results_df_rf2 columns:", results_df_rf2.columns.t
        display(results_df_rf2.tail(10))

        # ------------------
        # Cumulative returns (2000 onward & total)
        # ------------------
        filtered2 = results_df_rf2[results_df_rf2['Predicted_month'] >=
        if not filtered2.empty:
            cum_alloc2 = (1 + filtered2['Allocated_Return']).prod() - 1
            cum_eq2    = (1 + filtered2['Equal_Weight_Return']).prod()
```

```
        print(f"Cumulative 2000–present — RF2: {cum_alloc2:.4f}  /

    if not results_df_rf2.empty:
        cum_alloc_all2 = (1 + results_df_rf2['Allocated_Return']).p
        cum_eq_all2    = (1 + results_df_rf2['Equal_Weight_Return']
        print(f"Total cumulative — RF2: {cum_alloc_all2:.4f}  /  Eq
```

Gradient boosting

```
if GB:
    import pandas as pd
    import numpy as np
    from xgboost import XGBClassifier
    from IPython.display import display, HTML

    # ——————————————————
    # 1) Parameters
    # ——————————————————
    min_months_train = 60     # Minimum months of data needed (5 yea
    min_obs_regime = 50       # Minimum observations per regime if s
    min_obs_train = 0         # Minimum total observations after dro
    use_regime_split = False  # Toggle regime-based training or not
    default_hyperparameters = False  # If True, override manually se

    # Toggle for training window type:
    use_fixed_window = True   # True for fixed (rolling) window, Fal
    rolling_window_size = 60  # When using a fixed window, use this

    df_sorted = df.sort_values('Date').reset_index(drop=True)
    results = []

    # ——————————————————
    # 2) Main Loop: Predict for each row in df_sorted
    # ——————————————————
    for i in range(1, len(df_sorted)):
        test_row = df_sorted.iloc[i]
        Predicted_month = test_row['Date']

        # Build training window: either fixed-size (rolling) or expa
        if use_fixed_window:
            start_idx = max(0, i - rolling_window_size)
            train_window = df_sorted.iloc[start_idx:i].copy()
        else:
            train_window = df_sorted.iloc[:i].copy()
```

```
            # Check that we have enough training rows (i.e., months)
            if len(train_window) < min_months_train:
                print(f"Test row date: {Predicted_month.date()} - Insuff
                continue

            # Get first and last training dates
            train_start_date = train_window['Date'].iloc[0]
            train_end_date = train_window['Date'].iloc[-1]

            # Regime-based checks (if enabled)
            if use_regime_split:
                regime_counts = train_window[REGIMES_COLUMN].value_count
                insufficient_regimes = regime_counts[regime_counts < min
                if insufficient_regimes:
                    regime_str_list = [regime_short_mapping.get(r, str(r
                    regime_str = ", ".join(regime_str_list)
                    print(f"Test row date: {Predicted_month.date()}")
                    print(f"  Regime split active. Insufficient data in:
                    continue

                # Use only training data for the current regime
                current_regime = test_row[REGIMES_COLUMN]
                train_window = train_window[train_window[REGIMES_COLUMN]
                if len(train_window) < min_obs_regime:
                    regime_str = regime_short_mapping.get(current_regime
                    print(f"Test row date: {Predicted_month.date()}")
                    print(f"  Regime split active ({regime_str}). Only {
                    continue
                regime_used = regime_short_mapping.get(current_regime, s
            else:
                regime_used = 'NoRegime'

            # Ensure the last training date is strictly before the test
            last_train_date = train_window['Date'].iloc[-1]
            if (last_train_date.year == Predicted_month.year) and (last_
                print(f"Test row {Predicted_month.date()}: last training
                continue

            # Prepare training data
            X_train = train_window[FEATURES].dropna()
            y_train = train_window['Winning Factor'].loc[X_train.index]
            if len(X_train) < min_obs_train:
                print(f"   -> After dropping NAs: {len(X_train)} < {min_
                continue

            # Convert y_train from strings to numeric codes and save map
            y_train_cat = y_train.astype('category')
            mapping = dict(enumerate(y_train_cat.cat.categories))
```

```
            y_train_numeric = y_train_cat.cat.codes

            # --------------------
            # Set hyperparameters based on default_hyperparameters flag
            # --------------------
            if default_hyperparameters:
                xgb_params = {
                    'n_estimators': 100,
                    'max_depth': 3,
                    'learning_rate': 0.1,
                    'subsample': 1.0,
                    'colsample_bytree': 1.0,
                    'random_state': 42,
                    'eval_metric': 'mlogloss'
                }
            else:
                # Use manually defined hyperparameters (from Optuna or o
              xgb_params = {
                  'n_estimators':      200,
                  'max_depth':         15,
                  'learning_rate':     0.17,
                  'subsample':         0.75,
                  'colsample_bytree':  0.5,
                  'min_child_weight':  7,
                  'random_state':      42,
                  'n_jobs':            -1,
                  'eval_metric':       'mlogloss'
              }


            # Fit XGBoost gradient boosting classifier on numeric labels
            xgb_model = XGBClassifier(**xgb_params)
            xgb_model.fit(X_train, y_train_numeric)

            # Prepare test data (using the last row in the training wind
            X_test = train_window[FEATURES].iloc[[-1]].dropna()
            if X_test.empty:
                print("   -> Test features empty, skipping iteration.")
                continue

            predicted_probabilities = xgb_model.predict_proba(X_test)[0]
            # Get predicted numeric class and convert back to original f
            predicted_numeric = xgb_model.classes_[predicted_probabiliti
            predicted_winner = mapping[predicted_numeric]

            # Map predicted probabilities onto the full set of FACTORS
            full_probs = np.zeros(len(FACTORS))
            for code, prob in zip(xgb_model.classes_, predicted_probabil
                factor_name = mapping[code]
```

```
                factor_name = mapping[code]
                try:
                    idx = FACTORS.index(factor_name)
                    full_probs[idx] = prob
                except ValueError:
                    pass  # Skip if factor not found in FACTORS

            # Compute allocated return and equal weight return using the
            allocated_return = (full_probs * test_row[FACTORS].values).s
            equal_weight_return = np.mean(test_row[FACTORS].values)

            # Tree depth statistics are not required for XGB; set to Non
            avg_depth = None
            max_depth = None

            # Calculate prediction horizon (months ahead)
            months_ahead = (Predicted_month.year - last_train_date.year)

            # Store the actual feature levels used in X_test
            feature_levels = {f"Feature_Level_{f}": X_test[f].iloc[0] fo

            print(f"Test row date: {Predicted_month.date()} -> Model tra

            # Build the result dictionary for this iteration
            result = {
                'Regime': regime_used,
                'Predicted_month': Predicted_month,
                'Train_Start_Date': train_start_date,
                'Train_End_Date': train_end_date,
                'Train_Count': len(X_train),
                'Feature_Importances': xgb_model.feature_importances_,
                'Predicted_Probabilities': full_probs,
                'Predicted_Winner': predicted_winner,
                'Allocated_Return': allocated_return,
                'Equal_Weight_Return': equal_weight_return,
                'Actual_Winner': test_row['Winning Factor'],
                'Num_Trees': xgb_model.n_estimators,
                'Average_Tree_Depth': avg_depth,
                'Max_Tree_Depth': max_depth,
                'Prediction_Horizon_Months': months_ahead,
                **feature_levels
            }
            results.append(result)

        # ------------------
        # 3) Build the final results DataFrame for GB
        # ------------------
        results_df_gb = pd.DataFrame(results)
        print("Final results_df_gb columns:", results_df_gb.columns.toli
```

```
        print("Final results_df_gb columns:", results_df_gb.columns.toli
        display(results_df_gb.tail(10))

        # --------------------
        # 4) Calculate and Print Cumulative Returns (Filtered: from 1 Ja
        # --------------------
        filtered_results = results_df_gb[results_df_gb['Predicted_month'
        if not filtered_results.empty:
            cum_return_allocated = (1 + filtered_results['Allocated_Retu
            cum_return_equal = (1 + filtered_results['Equal_Weight_Retur

            first_pred_month = filtered_results.iloc[0]['Predicted_month
            last_pred_month = filtered_results.iloc[-1]['Predicted_month

            print("\nCumulative returns {} - {} - ML strategy: {:.4f} /
                first_pred_month.date(), last_pred_month.date(),
                cum_return_allocated, cum_return_equal))
        else:
            print("No predictions from 1 Jan 2000 onwards.")

        # --------------------
        # 5) Calculate and Print Cumulative Returns for Total Time
        # --------------------
        if not results_df_gb.empty:
            cum_return_allocated_total = (1 + results_df_gb['Allocated_R
            cum_return_equal_total = (1 + results_df_gb['Equal_Weight_Re

            first_total_month = results_df_gb.iloc[0]['Predicted_month']
            last_total_month = results_df_gb.iloc[-1]['Predicted_month']

            print("\nCumulative returns {} - {} - ML strategy: {:.4f} /
                first_total_month.date(), last_total_month.date(),
                cum_return_allocated_total, cum_return_equal_total))
        else:
            print("No predictions available for total time.")
```

```
Test row date: 1967-05-30 - Insufficient training rows (1 rows). Ski
Test row date: 1967-06-30 - Insufficient training rows (2 rows). Ski
Test row date: 1967-07-30 - Insufficient training rows (3 rows). Ski
Test row date: 1967-08-30 - Insufficient training rows (4 rows). Ski
Test row date: 1967-09-30 - Insufficient training rows (5 rows). Ski
Test row date: 1967-10-30 - Insufficient training rows (6 rows). Ski
Test row date: 1967-11-30 - Insufficient training rows (7 rows). Ski
Test row date: 1967-12-30 - Insufficient training rows (8 rows). Ski
Test row date: 1968-01-30 - Insufficient training rows (9 rows). Ski
Test row date: 1968-02-29 - Insufficient training rows (10 rows). Sk
Test row date: 1968-03-30 - Insufficient training rows (11 rows). Sk
Test row date: 1968-04-30 - Insufficient training rows (12 rows). Sk
Test row date: 1968-05-30 - Insufficient training rows (13 rows). Sk
```

```
Test row date: 1968-06-30 - Insufficient training rows (14 rows). Sk
Test row date: 1968-07-30 - Insufficient training rows (15 rows). Sk
Test row date: 1968-08-30 - Insufficient training rows (16 rows). Sk
Test row date: 1968-09-30 - Insufficient training rows (17 rows). Sk
Test row date: 1968-10-30 - Insufficient training rows (18 rows). Sk
Test row date: 1968-11-30 - Insufficient training rows (19 rows). Sk
Test row date: 1968-12-30 - Insufficient training rows (20 rows). Sk
Test row date: 1969-01-30 - Insufficient training rows (21 rows). Sk
Test row date: 1969-02-28 - Insufficient training rows (22 rows). Sk
Test row date: 1969-03-30 - Insufficient training rows (23 rows). Sk
Test row date: 1969-04-30 - Insufficient training rows (24 rows). Sk
Test row date: 1969-05-30 - Insufficient training rows (25 rows). Sk
Test row date: 1969-06-30 - Insufficient training rows (26 rows). Sk
Test row date: 1969-07-30 - Insufficient training rows (27 rows). Sk
Test row date: 1969-08-30 - Insufficient training rows (28 rows). Sk
Test row date: 1969-09-30 - Insufficient training rows (29 rows). Sk
Test row date: 1969-10-30 - Insufficient training rows (30 rows). Sk
Test row date: 1969-11-30 - Insufficient training rows (31 rows). Sk
Test row date: 1969-12-30 - Insufficient training rows (32 rows). Sk
Test row date: 1970-01-30 - Insufficient training rows (33 rows). Sk
Test row date: 1970-02-28 - Insufficient training rows (34 rows). Sk
Test row date: 1970-03-30 - Insufficient training rows (35 rows). Sk
Test row date: 1970-04-30 - Insufficient training rows (36 rows). Sk
Test row date: 1970-05-30 - Insufficient training rows (37 rows). Sk
Test row date: 1970-06-30 - Insufficient training rows (38 rows). Sk
Test row date: 1970-07-30 - Insufficient training rows (39 rows). Sk
Test row date: 1970-08-30 - Insufficient training rows (40 rows). Sk
Test row date: 1970-09-30 - Insufficient training rows (41 rows). Sk
Test row date: 1970-10-30 - Insufficient training rows (42 rows). Sk
Test row date: 1970-11-30 - Insufficient training rows (43 rows). Sk
Test row date: 1970-12-30 - Insufficient training rows (44 rows). Sk
Test row date: 1971-01-30 - Insufficient training rows (45 rows). Sk
Test row date: 1971-02-28 - Insufficient training rows (46 rows). Sk
Test row date: 1971-03-30 - Insufficient training rows (47 rows). Sk
Test row date: 1971-04-30 - Insufficient training rows (48 rows). Sk
Test row date: 1971-05-30 - Insufficient training rows (49 rows). Sk
Test row date: 1971-06-30 - Insufficient training rows (50 rows). Sk
Test row date: 1971-07-30 - Insufficient training rows (51 rows). Sk
Test row date: 1971-08-30 - Insufficient training rows (52 rows). Sk
Test row date: 1971-09-30 - Insufficient training rows (53 rows). Sk
Test row date: 1971-10-30 - Insufficient training rows (54 rows). Sk
Test row date: 1971-11-30 - Insufficient training rows (55 rows). Sk
Test row date: 1971-12-30 - Insufficient training rows (56 rows). Sk
Test row date: 1972-01-30 - Insufficient training rows (57 rows). Sk
Test row date: 1972-02-29 - Insufficient training rows (58 rows). Sk
Test row date: 1972-03-30 - Insufficient training rows (59 rows). Sk
Test row date: 1972-04-30 -> Model trained, prediction made (using:
Test row date: 1972-05-30 -> Model trained, prediction made (using:
Test row date: 1972-06-30 -> Model trained, prediction made (using:
Test row date: 1972-07-30 -> Model trained, prediction made (using:
Test row date: 1972-08-30 -> Model trained, prediction made (using:
Test row date: 1972-09-30 -> Model trained, prediction made (using:
Test row date: 1972-10-30 -> Model trained, prediction made (using:
Test row date: 1972-11-30 -> Model trained, prediction made (using:
```

```
Test row date: 1972-12-30 -> Model trained, prediction made (using:
Test row date: 1973-01-30 -> Model trained, prediction made (using:
Test row date: 1973-02-28 -> Model trained, prediction made (using:
Test row date: 1973-03-30 -> Model trained, prediction made (using:
Test row date: 1973-04-30 -> Model trained, prediction made (using:
Test row date: 1973-05-30 -> Model trained, prediction made (using:
Test row date: 1973-06-30 -> Model trained, prediction made (using:
Test row date: 1973-07-30 -> Model trained, prediction made (using:
Test row date: 1973-08-30 -> Model trained, prediction made (using:
Test row date: 1973-09-30 -> Model trained, prediction made (using:
Test row date: 1973-10-30 -> Model trained, prediction made (using:
Test row date: 1973-11-30 -> Model trained, prediction made (using:
Test row date: 1973-12-30 -> Model trained, prediction made (using:
Test row date: 1974-01-30 -> Model trained, prediction made (using:
Test row date: 1974-02-28 -> Model trained, prediction made (using:
Test row date: 1974-03-30 -> Model trained, prediction made (using:
Test row date: 1974-04-30 -> Model trained, prediction made (using:
Test row date: 1974-05-30 -> Model trained, prediction made (using:
Test row date: 1974-06-30 -> Model trained, prediction made (using:
Test row date: 1974-07-30 -> Model trained, prediction made (using:
Test row date: 1974-08-30 -> Model trained, prediction made (using:
Test row date: 1974-09-30 -> Model trained, prediction made (using:
Test row date: 1974-10-30 -> Model trained, prediction made (using:
Test row date: 1974-11-30 -> Model trained, prediction made (using:
Test row date: 1974-12-30 -> Model trained, prediction made (using:
Test row date: 1975-01-30 -> Model trained, prediction made (using:
Test row date: 1975-02-28 -> Model trained, prediction made (using:
Test row date: 1975-03-30 -> Model trained, prediction made (using:
Test row date: 1975-04-30 -> Model trained, prediction made (using:
Test row date: 1975-05-30 -> Model trained, prediction made (using:
Test row date: 1975-06-30 -> Model trained, prediction made (using:
Test row date: 1975-07-30 -> Model trained, prediction made (using:
Test row date: 1975-08-30 -> Model trained, prediction made (using:
Test row date: 1975-09-30 -> Model trained, prediction made (using:
Test row date: 1975-10-30 -> Model trained, prediction made (using:
Test row date: 1975-11-30 -> Model trained, prediction made (using:
Test row date: 1975-12-30 -> Model trained, prediction made (using:
Test row date: 1976-01-30 -> Model trained, prediction made (using:
Test row date: 1976-02-29 -> Model trained, prediction made (using:
Test row date: 1976-03-30 -> Model trained, prediction made (using:
Test row date: 1976-04-30 -> Model trained, prediction made (using:
Test row date: 1976-05-30 -> Model trained, prediction made (using:
Test row date: 1976-06-30 -> Model trained, prediction made (using:
Test row date: 1976-07-30 -> Model trained, prediction made (using:
Test row date: 1976-08-30 -> Model trained, prediction made (using:
Test row date: 1976-09-30 -> Model trained, prediction made (using:
Test row date: 1976-10-30 -> Model trained, prediction made (using:
Test row date: 1976-11-30 -> Model trained, prediction made (using:
Test row date: 1976-12-30 -> Model trained, prediction made (using:
Test row date: 1977-01-30 -> Model trained, prediction made (using:
Test row date: 1977-02-28 -> Model trained, prediction made (using:
Test row date: 1977-03-30 -> Model trained, prediction made (using:
Test row date: 1977-04-30 -> Model trained, prediction made (using:
Test row date: 1977-05-30 -> Model trained, prediction made (using:
```

```
Test row date: 1977-06-30 -> Model trained, prediction made (using:
Test row date: 1977-07-30 -> Model trained, prediction made (using:
Test row date: 1977-08-30 -> Model trained, prediction made (using:
Test row date: 1977-09-30 -> Model trained, prediction made (using:
Test row date: 1977-10-30 -> Model trained, prediction made (using:
Test row date: 1977-11-30 -> Model trained, prediction made (using:
Test row date: 1977-12-30 -> Model trained, prediction made (using:
Test row date: 1978-01-30 -> Model trained, prediction made (using:
Test row date: 1978-02-28 -> Model trained, prediction made (using:
Test row date: 1978-03-30 -> Model trained, prediction made (using:
Test row date: 1978-04-30 -> Model trained, prediction made (using:
Test row date: 1978-05-30 -> Model trained, prediction made (using:
Test row date: 1978-06-30 -> Model trained, prediction made (using:
Test row date: 1978-07-30 -> Model trained, prediction made (using:
Test row date: 1978-08-30 -> Model trained, prediction made (using:
Test row date: 1978-09-30 -> Model trained, prediction made (using:
Test row date: 1978-10-30 -> Model trained, prediction made (using:
Test row date: 1978-11-30 -> Model trained, prediction made (using:
Test row date: 1978-12-30 -> Model trained, prediction made (using:
Test row date: 1979-01-30 -> Model trained, prediction made (using:
Test row date: 1979-02-28 -> Model trained, prediction made (using:
Test row date: 1979-03-30 -> Model trained, prediction made (using:
Test row date: 1979-04-30 -> Model trained, prediction made (using:
Test row date: 1979-05-30 -> Model trained, prediction made (using:
Test row date: 1979-06-30 -> Model trained, prediction made (using:
Test row date: 1979-07-30 -> Model trained, prediction made (using:
Test row date: 1979-08-30 -> Model trained, prediction made (using:
Test row date: 1979-09-30 -> Model trained, prediction made (using:
Test row date: 1979-10-30 -> Model trained, prediction made (using:
Test row date: 1979-11-30 -> Model trained, prediction made (using:
Test row date: 1979-12-30 -> Model trained, prediction made (using:
Test row date: 1980-01-30 -> Model trained, prediction made (using:
Test row date: 1980-02-29 -> Model trained, prediction made (using:
Test row date: 1980-03-30 -> Model trained, prediction made (using:
Test row date: 1980-04-30 -> Model trained, prediction made (using:
Test row date: 1980-05-30 -> Model trained, prediction made (using:
Test row date: 1980-06-30 -> Model trained, prediction made (using:
Test row date: 1980-07-30 -> Model trained, prediction made (using:
Test row date: 1980-08-30 -> Model trained, prediction made (using:
Test row date: 1980-09-30 -> Model trained, prediction made (using:
Test row date: 1980-10-30 -> Model trained, prediction made (using:
Test row date: 1980-11-30 -> Model trained, prediction made (using:
Test row date: 1980-12-30 -> Model trained, prediction made (using:
Test row date: 1981-01-30 -> Model trained, prediction made (using:
Test row date: 1981-02-28 -> Model trained, prediction made (using:
Test row date: 1981-03-30 -> Model trained, prediction made (using:
Test row date: 1981-04-30 -> Model trained, prediction made (using:
Test row date: 1981-05-30 -> Model trained, prediction made (using:
Test row date: 1981-06-30 -> Model trained, prediction made (using:
Test row date: 1981-07-30 -> Model trained, prediction made (using:
Test row date: 1981-08-30 -> Model trained, prediction made (using:
Test row date: 1981-09-30 -> Model trained, prediction made (using:
Test row date: 1981-10-30 -> Model trained, prediction made (using:
Test row date: 1981-11-30 -> Model trained, prediction made (using:
```

```
Test row date: 1981-12-30 -> Model trained, prediction made (using:
Test row date: 1982-01-30 -> Model trained, prediction made (using:
Test row date: 1982-02-28 -> Model trained, prediction made (using:
Test row date: 1982-03-30 -> Model trained, prediction made (using:
Test row date: 1982-04-30 -> Model trained, prediction made (using:
Test row date: 1982-05-30 -> Model trained, prediction made (using:
Test row date: 1982-06-30 -> Model trained, prediction made (using:
Test row date: 1982-07-30 -> Model trained, prediction made (using:
Test row date: 1982-08-30 -> Model trained, prediction made (using:
Test row date: 1982-09-30 -> Model trained, prediction made (using:
Test row date: 1982-10-30 -> Model trained, prediction made (using:
Test row date: 1982-11-30 -> Model trained, prediction made (using:
Test row date: 1982-12-30 -> Model trained, prediction made (using:
Test row date: 1983-01-30 -> Model trained, prediction made (using:
Test row date: 1983-02-28 -> Model trained, prediction made (using:
Test row date: 1983-03-30 -> Model trained, prediction made (using:
Test row date: 1983-04-30 -> Model trained, prediction made (using:
Test row date: 1983-05-30 -> Model trained, prediction made (using:
Test row date: 1983-06-30 -> Model trained, prediction made (using:
Test row date: 1983-07-30 -> Model trained, prediction made (using:
Test row date: 1983-08-30 -> Model trained, prediction made (using:
Test row date: 1983-09-30 -> Model trained, prediction made (using:
Test row date: 1983-10-30 -> Model trained, prediction made (using:
Test row date: 1983-11-30 -> Model trained, prediction made (using:
Test row date: 1983-12-30 -> Model trained, prediction made (using:
Test row date: 1984-01-30 -> Model trained, prediction made (using:
Test row date: 1984-02-29 -> Model trained, prediction made (using:
Test row date: 1984-03-30 -> Model trained, prediction made (using:
Test row date: 1984-04-30 -> Model trained, prediction made (using:
Test row date: 1984-05-30 -> Model trained, prediction made (using:
Test row date: 1984-06-30 -> Model trained, prediction made (using:
Test row date: 1984-07-30 -> Model trained, prediction made (using:
Test row date: 1984-08-30 -> Model trained, prediction made (using:
Test row date: 1984-09-30 -> Model trained, prediction made (using:
Test row date: 1984-10-30 -> Model trained, prediction made (using:
Test row date: 1984-11-30 -> Model trained, prediction made (using:
Test row date: 1984-12-30 -> Model trained, prediction made (using:
Test row date: 1985-01-30 -> Model trained, prediction made (using:
Test row date: 1985-02-28 -> Model trained, prediction made (using:
Test row date: 1985-03-30 -> Model trained, prediction made (using:
Test row date: 1985-04-30 -> Model trained, prediction made (using:
Test row date: 1985-05-30 -> Model trained, prediction made (using:
Test row date: 1985-06-30 -> Model trained, prediction made (using:
Test row date: 1985-07-30 -> Model trained, prediction made (using:
Test row date: 1985-08-30 -> Model trained, prediction made (using:
Test row date: 1985-09-30 -> Model trained, prediction made (using:
Test row date: 1985-10-30 -> Model trained, prediction made (using:
Test row date: 1985-11-30 -> Model trained, prediction made (using:
Test row date: 1985-12-30 -> Model trained, prediction made (using:
Test row date: 1986-01-30 -> Model trained, prediction made (using:
Test row date: 1986-02-28 -> Model trained, prediction made (using:
Test row date: 1986-03-30 -> Model trained, prediction made (using:
Test row date: 1986-04-30 -> Model trained, prediction made (using:
Test row date: 1986-05-30 -> Model trained, prediction made (using:
```

```
Test row date: 1986-06-30 -> Model trained, prediction made (using:
Test row date: 1986-07-30 -> Model trained, prediction made (using:
Test row date: 1986-08-30 -> Model trained, prediction made (using:
Test row date: 1986-09-30 -> Model trained, prediction made (using:
Test row date: 1986-10-30 -> Model trained, prediction made (using:
Test row date: 1986-11-30 -> Model trained, prediction made (using:
Test row date: 1986-12-30 -> Model trained, prediction made (using:
Test row date: 1987-01-30 -> Model trained, prediction made (using:
Test row date: 1987-02-28 -> Model trained, prediction made (using:
Test row date: 1987-03-30 -> Model trained, prediction made (using:
Test row date: 1987-04-30 -> Model trained, prediction made (using:
Test row date: 1987-05-30 -> Model trained, prediction made (using:
Test row date: 1987-06-30 -> Model trained, prediction made (using:
Test row date: 1987-07-30 -> Model trained, prediction made (using:
Test row date: 1987-08-30 -> Model trained, prediction made (using:
Test row date: 1987-09-30 -> Model trained, prediction made (using:
Test row date: 1987-10-30 -> Model trained, prediction made (using:
Test row date: 1987-11-30 -> Model trained, prediction made (using:
Test row date: 1987-12-30 -> Model trained, prediction made (using:
Test row date: 1988-01-30 -> Model trained, prediction made (using:
Test row date: 1988-02-29 -> Model trained, prediction made (using:
Test row date: 1988-03-30 -> Model trained, prediction made (using:
Test row date: 1988-04-30 -> Model trained, prediction made (using:
Test row date: 1988-05-30 -> Model trained, prediction made (using:
Test row date: 1988-06-30 -> Model trained, prediction made (using:
Test row date: 1988-07-30 -> Model trained, prediction made (using:
Test row date: 1988-08-30 -> Model trained, prediction made (using:
Test row date: 1988-09-30 -> Model trained, prediction made (using:
Test row date: 1988-10-30 -> Model trained, prediction made (using:
Test row date: 1988-11-30 -> Model trained, prediction made (using:
Test row date: 1988-12-30 -> Model trained, prediction made (using:
Test row date: 1989-01-30 -> Model trained, prediction made (using:
Test row date: 1989-02-28 -> Model trained, prediction made (using:
Test row date: 1989-03-30 -> Model trained, prediction made (using:
Test row date: 1989-04-30 -> Model trained, prediction made (using:
Test row date: 1989-05-30 -> Model trained, prediction made (using:
Test row date: 1989-06-30 -> Model trained, prediction made (using:
Test row date: 1989-07-30 -> Model trained, prediction made (using:
Test row date: 1989-08-30 -> Model trained, prediction made (using:
Test row date: 1989-09-30 -> Model trained, prediction made (using:
Test row date: 1989-10-30 -> Model trained, prediction made (using:
Test row date: 1989-11-30 -> Model trained, prediction made (using:
Test row date: 1989-12-30 -> Model trained, prediction made (using:
Test row date: 1990-01-30 -> Model trained, prediction made (using:
Test row date: 1990-02-28 -> Model trained, prediction made (using:
Test row date: 1990-03-30 -> Model trained, prediction made (using:
Test row date: 1990-04-30 -> Model trained, prediction made (using:
Test row date: 1990-05-30 -> Model trained, prediction made (using:
Test row date: 1990-06-30 -> Model trained, prediction made (using:
Test row date: 1990-07-30 -> Model trained, prediction made (using:
Test row date: 1990-08-30 -> Model trained, prediction made (using:
Test row date: 1990-09-30 -> Model trained, prediction made (using:
Test row date: 1990-10-30 -> Model trained, prediction made (using:
Test row date: 1990-11-30 -> Model trained, prediction made (using:
```

```
Test row date: 1990-12-30 -> Model trained, prediction made (using:
Test row date: 1991-01-30 -> Model trained, prediction made (using:
Test row date: 1991-02-28 -> Model trained, prediction made (using:
Test row date: 1991-03-30 -> Model trained, prediction made (using:
Test row date: 1991-04-30 -> Model trained, prediction made (using:
Test row date: 1991-05-30 -> Model trained, prediction made (using:
Test row date: 1991-06-30 -> Model trained, prediction made (using:
Test row date: 1991-07-30 -> Model trained, prediction made (using:
Test row date: 1991-08-30 -> Model trained, prediction made (using:
Test row date: 1991-09-30 -> Model trained, prediction made (using:
Test row date: 1991-10-30 -> Model trained, prediction made (using:
Test row date: 1991-11-30 -> Model trained, prediction made (using:
Test row date: 1991-12-30 -> Model trained, prediction made (using:
Test row date: 1992-01-30 -> Model trained, prediction made (using:
Test row date: 1992-02-29 -> Model trained, prediction made (using:
Test row date: 1992-03-30 -> Model trained, prediction made (using:
Test row date: 1992-04-30 -> Model trained, prediction made (using:
Test row date: 1992-05-30 -> Model trained, prediction made (using:
Test row date: 1992-06-30 -> Model trained, prediction made (using:
Test row date: 1992-07-30 -> Model trained, prediction made (using:
Test row date: 1992-08-30 -> Model trained, prediction made (using:
Test row date: 1992-09-30 -> Model trained, prediction made (using:
Test row date: 1992-10-30 -> Model trained, prediction made (using:
Test row date: 1992-11-30 -> Model trained, prediction made (using:
Test row date: 1992-12-30 -> Model trained, prediction made (using:
Test row date: 1993-01-30 -> Model trained, prediction made (using:
Test row date: 1993-02-28 -> Model trained, prediction made (using:
Test row date: 1993-03-30 -> Model trained, prediction made (using:
Test row date: 1993-04-30 -> Model trained, prediction made (using:
Test row date: 1993-05-30 -> Model trained, prediction made (using:
Test row date: 1993-06-30 -> Model trained, prediction made (using:
Test row date: 1993-07-30 -> Model trained, prediction made (using:
Test row date: 1993-08-30 -> Model trained, prediction made (using:
Test row date: 1993-09-30 -> Model trained, prediction made (using:
Test row date: 1993-10-30 -> Model trained, prediction made (using:
Test row date: 1993-11-30 -> Model trained, prediction made (using:
Test row date: 1993-12-30 -> Model trained, prediction made (using:
Test row date: 1994-01-30 -> Model trained, prediction made (using:
Test row date: 1994-02-28 -> Model trained, prediction made (using:
Test row date: 1994-03-30 -> Model trained, prediction made (using:
Test row date: 1994-04-30 -> Model trained, prediction made (using:
Test row date: 1994-05-30 -> Model trained, prediction made (using:
Test row date: 1994-06-30 -> Model trained, prediction made (using:
Test row date: 1994-07-30 -> Model trained, prediction made (using:
Test row date: 1994-08-30 -> Model trained, prediction made (using:
Test row date: 1994-09-30 -> Model trained, prediction made (using:
Test row date: 1994-10-30 -> Model trained, prediction made (using:
Test row date: 1994-11-30 -> Model trained, prediction made (using:
Test row date: 1994-12-30 -> Model trained, prediction made (using:
Test row date: 1995-01-30 -> Model trained, prediction made (using:
Test row date: 1995-02-28 -> Model trained, prediction made (using:
Test row date: 1995-03-30 -> Model trained, prediction made (using:
Test row date: 1995-04-30 -> Model trained, prediction made (using:
Test row date: 1995-05-30 -> Model trained, prediction made (using:
```

```
Test row date: 1995-06-30 -> Model trained, prediction made (using:
Test row date: 1995-07-30 -> Model trained, prediction made (using:
Test row date: 1995-08-30 -> Model trained, prediction made (using:
Test row date: 1995-09-30 -> Model trained, prediction made (using:
Test row date: 1995-10-30 -> Model trained, prediction made (using:
Test row date: 1995-11-30 -> Model trained, prediction made (using:
Test row date: 1995-12-30 -> Model trained, prediction made (using:
Test row date: 1996-01-30 -> Model trained, prediction made (using:
Test row date: 1996-02-29 -> Model trained, prediction made (using:
Test row date: 1996-03-30 -> Model trained, prediction made (using:
Test row date: 1996-04-30 -> Model trained, prediction made (using:
Test row date: 1996-05-30 -> Model trained, prediction made (using:
Test row date: 1996-06-30 -> Model trained, prediction made (using:
Test row date: 1996-07-30 -> Model trained, prediction made (using:
Test row date: 1996-08-30 -> Model trained, prediction made (using:
Test row date: 1996-09-30 -> Model trained, prediction made (using:
Test row date: 1996-10-30 -> Model trained, prediction made (using:
Test row date: 1996-11-30 -> Model trained, prediction made (using:
Test row date: 1996-12-30 -> Model trained, prediction made (using:
Test row date: 1997-01-30 -> Model trained, prediction made (using:
Test row date: 1997-02-28 -> Model trained, prediction made (using:
Test row date: 1997-03-30 -> Model trained, prediction made (using:
Test row date: 1997-04-30 -> Model trained, prediction made (using:
Test row date: 1997-05-30 -> Model trained, prediction made (using:
Test row date: 1997-06-30 -> Model trained, prediction made (using:
Test row date: 1997-07-30 -> Model trained, prediction made (using:
Test row date: 1997-08-30 -> Model trained, prediction made (using:
Test row date: 1997-09-30 -> Model trained, prediction made (using:
Test row date: 1997-10-30 -> Model trained, prediction made (using:
Test row date: 1997-11-30 -> Model trained, prediction made (using:
Test row date: 1997-12-30 -> Model trained, prediction made (using:
Test row date: 1998-01-30 -> Model trained, prediction made (using:
Test row date: 1998-02-28 -> Model trained, prediction made (using:
Test row date: 1998-03-30 -> Model trained, prediction made (using:
Test row date: 1998-04-30 -> Model trained, prediction made (using:
Test row date: 1998-05-30 -> Model trained, prediction made (using:
Test row date: 1998-06-30 -> Model trained, prediction made (using:
Test row date: 1998-07-30 -> Model trained, prediction made (using:
Test row date: 1998-08-30 -> Model trained, prediction made (using:
Test row date: 1998-09-30 -> Model trained, prediction made (using:
Test row date: 1998-10-30 -> Model trained, prediction made (using:
Test row date: 1998-11-30 -> Model trained, prediction made (using:
Test row date: 1998-12-30 -> Model trained, prediction made (using:
Test row date: 1999-01-30 -> Model trained, prediction made (using:
Test row date: 1999-02-28 -> Model trained, prediction made (using:
Test row date: 1999-03-30 -> Model trained, prediction made (using:
Test row date: 1999-04-30 -> Model trained, prediction made (using:
Test row date: 1999-05-30 -> Model trained, prediction made (using:
Test row date: 1999-06-30 -> Model trained, prediction made (using:
Test row date: 1999-07-30 -> Model trained, prediction made (using:
Test row date: 1999-08-30 -> Model trained, prediction made (using:
Test row date: 1999-09-30 -> Model trained, prediction made (using:
Test row date: 1999-10-30 -> Model trained, prediction made (using:
Test row date: 1999-11-30 -> Model trained, prediction made (using:
```

```
Test row date: 1999-12-30 -> Model trained, prediction made (using:
Test row date: 2000-01-30 -> Model trained, prediction made (using:
Test row date: 2000-02-29 -> Model trained, prediction made (using:
Test row date: 2000-03-30 -> Model trained, prediction made (using:
Test row date: 2000-04-30 -> Model trained, prediction made (using:
Test row date: 2000-05-30 -> Model trained, prediction made (using:
Test row date: 2000-06-30 -> Model trained, prediction made (using:
Test row date: 2000-07-30 -> Model trained, prediction made (using:
Test row date: 2000-08-30 -> Model trained, prediction made (using:
Test row date: 2000-09-30 -> Model trained, prediction made (using:
Test row date: 2000-10-30 -> Model trained, prediction made (using:
Test row date: 2000-11-30 -> Model trained, prediction made (using:
Test row date: 2000-12-30 -> Model trained, prediction made (using:
Test row date: 2001-01-30 -> Model trained, prediction made (using:
Test row date: 2001-02-28 -> Model trained, prediction made (using:
Test row date: 2001-03-30 -> Model trained, prediction made (using:
Test row date: 2001-04-30 -> Model trained, prediction made (using:
Test row date: 2001-05-30 -> Model trained, prediction made (using:
Test row date: 2001-06-30 -> Model trained, prediction made (using:
Test row date: 2001-07-30 -> Model trained, prediction made (using:
Test row date: 2001-08-30 -> Model trained, prediction made (using:
Test row date: 2001-09-30 -> Model trained, prediction made (using:
Test row date: 2001-10-30 -> Model trained, prediction made (using:
Test row date: 2001-11-30 -> Model trained, prediction made (using:
Test row date: 2001-12-30 -> Model trained, prediction made (using:
Test row date: 2002-01-30 -> Model trained, prediction made (using:
Test row date: 2002-02-28 -> Model trained, prediction made (using:
Test row date: 2002-03-30 -> Model trained, prediction made (using:
Test row date: 2002-04-30 -> Model trained, prediction made (using:
Test row date: 2002-05-30 -> Model trained, prediction made (using:
Test row date: 2002-06-30 -> Model trained, prediction made (using:
Test row date: 2002-07-30 -> Model trained, prediction made (using:
Test row date: 2002-08-30 -> Model trained, prediction made (using:
Test row date: 2002-09-30 -> Model trained, prediction made (using:
Test row date: 2002-10-30 -> Model trained, prediction made (using:
Test row date: 2002-11-30 -> Model trained, prediction made (using:
Test row date: 2002-12-30 -> Model trained, prediction made (using:
Test row date: 2003-01-30 -> Model trained, prediction made (using:
Test row date: 2003-02-28 -> Model trained, prediction made (using:
Test row date: 2003-03-30 -> Model trained, prediction made (using:
Test row date: 2003-04-30 -> Model trained, prediction made (using:
Test row date: 2003-05-30 -> Model trained, prediction made (using:
Test row date: 2003-06-30 -> Model trained, prediction made (using:
Test row date: 2003-07-30 -> Model trained, prediction made (using:
Test row date: 2003-08-30 -> Model trained, prediction made (using:
Test row date: 2003-09-30 -> Model trained, prediction made (using:
Test row date: 2003-10-30 -> Model trained, prediction made (using:
Test row date: 2003-11-30 -> Model trained, prediction made (using:
Test row date: 2003-12-30 -> Model trained, prediction made (using:
Test row date: 2004-01-30 -> Model trained, prediction made (using:
Test row date: 2004-02-29 -> Model trained, prediction made (using:
Test row date: 2004-03-30 -> Model trained, prediction made (using:
Test row date: 2004-04-30 -> Model trained, prediction made (using:
Test row date: 2004-05-30 -> Model trained, prediction made (using:
```

```
Test row date: 2004-06-30 -> Model trained, prediction made (using:
Test row date: 2004-07-30 -> Model trained, prediction made (using:
Test row date: 2004-08-30 -> Model trained, prediction made (using:
Test row date: 2004-09-30 -> Model trained, prediction made (using:
Test row date: 2004-10-30 -> Model trained, prediction made (using:
Test row date: 2004-11-30 -> Model trained, prediction made (using:
Test row date: 2004-12-30 -> Model trained, prediction made (using:
Test row date: 2005-01-30 -> Model trained, prediction made (using:
Test row date: 2005-02-28 -> Model trained, prediction made (using:
Test row date: 2005-03-30 -> Model trained, prediction made (using:
Test row date: 2005-04-30 -> Model trained, prediction made (using:
Test row date: 2005-05-30 -> Model trained, prediction made (using:
Test row date: 2005-06-30 -> Model trained, prediction made (using:
Test row date: 2005-07-30 -> Model trained, prediction made (using:
Test row date: 2005-08-30 -> Model trained, prediction made (using:
Test row date: 2005-09-30 -> Model trained, prediction made (using:
Test row date: 2005-10-30 -> Model trained, prediction made (using:
Test row date: 2005-11-30 -> Model trained, prediction made (using:
Test row date: 2005-12-30 -> Model trained, prediction made (using:
Test row date: 2006-01-30 -> Model trained, prediction made (using:
Test row date: 2006-02-28 -> Model trained, prediction made (using:
Test row date: 2006-03-30 -> Model trained, prediction made (using:
Test row date: 2006-04-30 -> Model trained, prediction made (using:
Test row date: 2006-05-30 -> Model trained, prediction made (using:
Test row date: 2006-06-30 -> Model trained, prediction made (using:
Test row date: 2006-07-30 -> Model trained, prediction made (using:
Test row date: 2006-08-30 -> Model trained, prediction made (using:
Test row date: 2006-09-30 -> Model trained, prediction made (using:
Test row date: 2006-10-30 -> Model trained, prediction made (using:
Test row date: 2006-11-30 -> Model trained, prediction made (using:
Test row date: 2006-12-30 -> Model trained, prediction made (using:
Test row date: 2007-01-30 -> Model trained, prediction made (using:
Test row date: 2007-02-28 -> Model trained, prediction made (using:
Test row date: 2007-03-30 -> Model trained, prediction made (using:
Test row date: 2007-04-30 -> Model trained, prediction made (using:
Test row date: 2007-05-30 -> Model trained, prediction made (using:
Test row date: 2007-06-30 -> Model trained, prediction made (using:
Test row date: 2007-07-30 -> Model trained, prediction made (using:
Test row date: 2007-08-30 -> Model trained, prediction made (using:
Test row date: 2007-09-30 -> Model trained, prediction made (using:
Test row date: 2007-10-30 -> Model trained, prediction made (using:
Test row date: 2007-11-30 -> Model trained, prediction made (using:
Test row date: 2007-12-30 -> Model trained, prediction made (using:
Test row date: 2008-01-30 -> Model trained, prediction made (using:
Test row date: 2008-02-29 -> Model trained, prediction made (using:
Test row date: 2008-03-30 -> Model trained, prediction made (using:
Test row date: 2008-04-30 -> Model trained, prediction made (using:
Test row date: 2008-05-30 -> Model trained, prediction made (using:
Test row date: 2008-06-30 -> Model trained, prediction made (using:
Test row date: 2008-07-30 -> Model trained, prediction made (using:
Test row date: 2008-08-30 -> Model trained, prediction made (using:
Test row date: 2008-09-30 -> Model trained, prediction made (using:
Test row date: 2008-10-30 -> Model trained, prediction made (using:
Test row date: 2008-11-30 -> Model trained, prediction made (using:
```

```
Test row date: 2008-12-30 -> Model trained, prediction made (using:
Test row date: 2009-01-30 -> Model trained, prediction made (using:
Test row date: 2009-02-28 -> Model trained, prediction made (using:
Test row date: 2009-03-30 -> Model trained, prediction made (using:
Test row date: 2009-04-30 -> Model trained, prediction made (using:
Test row date: 2009-05-30 -> Model trained, prediction made (using:
Test row date: 2009-06-30 -> Model trained, prediction made (using:
Test row date: 2009-07-30 -> Model trained, prediction made (using:
Test row date: 2009-08-30 -> Model trained, prediction made (using:
Test row date: 2009-09-30 -> Model trained, prediction made (using:
Test row date: 2009-10-30 -> Model trained, prediction made (using:
Test row date: 2009-11-30 -> Model trained, prediction made (using:
Test row date: 2009-12-30 -> Model trained, prediction made (using:
Test row date: 2010-01-30 -> Model trained, prediction made (using:
Test row date: 2010-02-28 -> Model trained, prediction made (using:
Test row date: 2010-03-30 -> Model trained, prediction made (using:
Test row date: 2010-04-30 -> Model trained, prediction made (using:
Test row date: 2010-05-30 -> Model trained, prediction made (using:
Test row date: 2010-06-30 -> Model trained, prediction made (using:
Test row date: 2010-07-30 -> Model trained, prediction made (using:
Test row date: 2010-08-30 -> Model trained, prediction made (using:
Test row date: 2010-09-30 -> Model trained, prediction made (using:
Test row date: 2010-10-30 -> Model trained, prediction made (using:
Test row date: 2010-11-30 -> Model trained, prediction made (using:
Test row date: 2010-12-30 -> Model trained, prediction made (using:
Test row date: 2011-01-30 -> Model trained, prediction made (using:
Test row date: 2011-02-28 -> Model trained, prediction made (using:
Test row date: 2011-03-30 -> Model trained, prediction made (using:
Test row date: 2011-04-30 -> Model trained, prediction made (using:
Test row date: 2011-05-30 -> Model trained, prediction made (using:
Test row date: 2011-06-30 -> Model trained, prediction made (using:
Test row date: 2011-07-30 -> Model trained, prediction made (using:
Test row date: 2011-08-30 -> Model trained, prediction made (using:
Test row date: 2011-09-30 -> Model trained, prediction made (using:
Test row date: 2011-10-30 -> Model trained, prediction made (using:
Test row date: 2011-11-30 -> Model trained, prediction made (using:
Test row date: 2011-12-30 -> Model trained, prediction made (using:
Test row date: 2012-01-30 -> Model trained, prediction made (using:
Test row date: 2012-02-29 -> Model trained, prediction made (using:
Test row date: 2012-03-30 -> Model trained, prediction made (using:
Test row date: 2012-04-30 -> Model trained, prediction made (using:
Test row date: 2012-05-30 -> Model trained, prediction made (using:
Test row date: 2012-06-30 -> Model trained, prediction made (using:
Test row date: 2012-07-30 -> Model trained, prediction made (using:
Test row date: 2012-08-30 -> Model trained, prediction made (using:
Test row date: 2012-09-30 -> Model trained, prediction made (using:
Test row date: 2012-10-30 -> Model trained, prediction made (using:
Test row date: 2012-11-30 -> Model trained, prediction made (using:
Test row date: 2012-12-30 -> Model trained, prediction made (using:
Test row date: 2013-01-30 -> Model trained, prediction made (using:
Test row date: 2013-02-28 -> Model trained, prediction made (using:
Test row date: 2013-03-30 -> Model trained, prediction made (using:
Test row date: 2013-04-30 -> Model trained, prediction made (using:
Test row date: 2013-05-30 -> Model trained, prediction made (using:
```

```
Test row date: 2013-06-30 -> Model trained, prediction made (using:
Test row date: 2013-07-30 -> Model trained, prediction made (using:
Test row date: 2013-08-30 -> Model trained, prediction made (using:
Test row date: 2013-09-30 -> Model trained, prediction made (using:
Test row date: 2013-10-30 -> Model trained, prediction made (using:
Test row date: 2013-11-30 -> Model trained, prediction made (using:
Test row date: 2013-12-30 -> Model trained, prediction made (using:
Test row date: 2014-01-30 -> Model trained, prediction made (using:
Test row date: 2014-02-28 -> Model trained, prediction made (using:
Test row date: 2014-03-30 -> Model trained, prediction made (using:
Test row date: 2014-04-30 -> Model trained, prediction made (using:
Test row date: 2014-05-30 -> Model trained, prediction made (using:
Test row date: 2014-06-30 -> Model trained, prediction made (using:
Test row date: 2014-07-30 -> Model trained, prediction made (using:
Test row date: 2014-08-30 -> Model trained, prediction made (using:
Test row date: 2014-09-30 -> Model trained, prediction made (using:
Test row date: 2014-10-30 -> Model trained, prediction made (using:
Test row date: 2014-11-30 -> Model trained, prediction made (using:
Test row date: 2014-12-30 -> Model trained, prediction made (using:
Test row date: 2015-01-30 -> Model trained, prediction made (using:
Test row date: 2015-02-28 -> Model trained, prediction made (using:
Test row date: 2015-03-30 -> Model trained, prediction made (using:
Test row date: 2015-04-30 -> Model trained, prediction made (using:
Test row date: 2015-05-30 -> Model trained, prediction made (using:
Test row date: 2015-06-30 -> Model trained, prediction made (using:
Test row date: 2015-07-30 -> Model trained, prediction made (using:
Test row date: 2015-08-30 -> Model trained, prediction made (using:
Test row date: 2015-09-30 -> Model trained, prediction made (using:
Test row date: 2015-10-30 -> Model trained, prediction made (using:
Test row date: 2015-11-30 -> Model trained, prediction made (using:
Test row date: 2015-12-30 -> Model trained, prediction made (using:
Test row date: 2016-01-30 -> Model trained, prediction made (using:
Test row date: 2016-02-29 -> Model trained, prediction made (using:
Test row date: 2016-03-30 -> Model trained, prediction made (using:
Test row date: 2016-04-30 -> Model trained, prediction made (using:
Test row date: 2016-05-30 -> Model trained, prediction made (using:
Test row date: 2016-06-30 -> Model trained, prediction made (using:
Test row date: 2016-07-30 -> Model trained, prediction made (using:
Test row date: 2016-08-30 -> Model trained, prediction made (using:
Test row date: 2016-09-30 -> Model trained, prediction made (using:
Test row date: 2016-10-30 -> Model trained, prediction made (using:
Test row date: 2016-11-30 -> Model trained, prediction made (using:
Test row date: 2016-12-30 -> Model trained, prediction made (using:
Test row date: 2017-01-30 -> Model trained, prediction made (using:
Test row date: 2017-02-28 -> Model trained, prediction made (using:
Test row date: 2017-03-30 -> Model trained, prediction made (using:
Test row date: 2017-04-30 -> Model trained, prediction made (using:
Test row date: 2017-05-30 -> Model trained, prediction made (using:
Test row date: 2017-06-30 -> Model trained, prediction made (using:
Test row date: 2017-07-30 -> Model trained, prediction made (using:
Test row date: 2017-08-30 -> Model trained, prediction made (using:
Test row date: 2017-09-30 -> Model trained, prediction made (using:
Test row date: 2017-10-30 -> Model trained, prediction made (using:
Test row date: 2017-11-30 -> Model trained, prediction made (using:
```

```
Test row date: 2017-12-30 -> Model trained, prediction made (using:
Test row date: 2018-01-30 -> Model trained, prediction made (using:
Test row date: 2018-02-28 -> Model trained, prediction made (using:
Test row date: 2018-03-30 -> Model trained, prediction made (using:
Test row date: 2018-04-30 -> Model trained, prediction made (using:
Test row date: 2018-05-30 -> Model trained, prediction made (using:
Test row date: 2018-06-30 -> Model trained, prediction made (using:
Test row date: 2018-07-30 -> Model trained, prediction made (using:
Test row date: 2018-08-30 -> Model trained, prediction made (using:
Test row date: 2018-09-30 -> Model trained, prediction made (using:
Test row date: 2018-10-30 -> Model trained, prediction made (using:
Test row date: 2018-11-30 -> Model trained, prediction made (using:
Test row date: 2018-12-30 -> Model trained, prediction made (using:
Test row date: 2019-01-30 -> Model trained, prediction made (using:
Test row date: 2019-02-28 -> Model trained, prediction made (using:
Test row date: 2019-03-30 -> Model trained, prediction made (using:
Test row date: 2019-04-30 -> Model trained, prediction made (using:
Test row date: 2019-05-30 -> Model trained, prediction made (using:
Test row date: 2019-06-30 -> Model trained, prediction made (using:
Test row date: 2019-07-30 -> Model trained, prediction made (using:
Test row date: 2019-08-30 -> Model trained, prediction made (using:
Test row date: 2019-09-30 -> Model trained, prediction made (using:
Test row date: 2019-10-30 -> Model trained, prediction made (using:
Test row date: 2019-11-30 -> Model trained, prediction made (using:
Test row date: 2019-12-30 -> Model trained, prediction made (using:
Test row date: 2020-01-30 -> Model trained, prediction made (using:
Test row date: 2020-02-29 -> Model trained, prediction made (using:
Test row date: 2020-03-30 -> Model trained, prediction made (using:
Test row date: 2020-04-30 -> Model trained, prediction made (using:
Test row date: 2020-05-30 -> Model trained, prediction made (using:
Test row date: 2020-06-30 -> Model trained, prediction made (using:
Test row date: 2020-07-30 -> Model trained, prediction made (using:
Test row date: 2020-08-30 -> Model trained, prediction made (using:
Test row date: 2020-09-30 -> Model trained, prediction made (using:
Test row date: 2020-10-30 -> Model trained, prediction made (using:
Test row date: 2020-11-30 -> Model trained, prediction made (using:
Test row date: 2020-12-30 -> Model trained, prediction made (using:
Test row date: 2021-01-30 -> Model trained, prediction made (using:
Test row date: 2021-02-28 -> Model trained, prediction made (using:
Test row date: 2021-03-30 -> Model trained, prediction made (using:
Test row date: 2021-04-30 -> Model trained, prediction made (using:
Test row date: 2021-05-30 -> Model trained, prediction made (using:
Test row date: 2021-06-30 -> Model trained, prediction made (using:
Test row date: 2021-07-30 -> Model trained, prediction made (using:
Test row date: 2021-08-30 -> Model trained, prediction made (using:
Test row date: 2021-09-30 -> Model trained, prediction made (using:
Test row date: 2021-10-30 -> Model trained, prediction made (using:
Test row date: 2021-11-30 -> Model trained, prediction made (using:
Test row date: 2021-12-30 -> Model trained, prediction made (using:
Test row date: 2022-01-30 -> Model trained, prediction made (using:
Test row date: 2022-02-28 -> Model trained, prediction made (using:
Test row date: 2022-03-30 -> Model trained, prediction made (using:
Test row date: 2022-04-30 -> Model trained, prediction made (using:
```

```
Test row date: 2022-06-30 -> Model trained, prediction made (using:
Test row date: 2022-07-30 -> Model trained, prediction made (using:
Test row date: 2022-08-30 -> Model trained, prediction made (using:
Test row date: 2022-09-30 -> Model trained, prediction made (using:
Test row date: 2022-10-30 -> Model trained, prediction made (using:
Test row date: 2022-11-30 -> Model trained, prediction made (using:
Test row date: 2022-12-30 -> Model trained, prediction made (using:
Test row date: 2023-01-30 -> Model trained, prediction made (using:
Test row date: 2023-02-28 -> Model trained, prediction made (using:
Test row date: 2023-03-30 -> Model trained, prediction made (using:
Test row date: 2023-04-30 -> Model trained, prediction made (using:
Test row date: 2023-05-30 -> Model trained, prediction made (using:
Test row date: 2023-06-30 -> Model trained, prediction made (using:
Test row date: 2023-07-30 -> Model trained, prediction made (using:
Test row date: 2023-08-30 -> Model trained, prediction made (using:
Test row date: 2023-09-30 -> Model trained, prediction made (using:
Test row date: 2023-10-30 -> Model trained, prediction made (using:
Test row date: 2023-11-30 -> Model trained, prediction made (using:
Test row date: 2023-12-30 -> Model trained, prediction made (using:
Test row date: 2024-01-30 -> Model trained, prediction made (using:
Test row date: 2024-02-29 -> Model trained, prediction made (using:
Test row date: 2024-03-30 -> Model trained, prediction made (using:
Test row date: 2024-04-30 -> Model trained, prediction made (using:
Test row date: 2024-05-30 -> Model trained, prediction made (using:
Test row date: 2024-06-30 -> Model trained, prediction made (using:
Test row date: 2024-07-30 -> Model trained, prediction made (using:
Test row date: 2024-08-30 -> Model trained, prediction made (using:
Test row date: 2024-09-30 -> Model trained, prediction made (using:
Test row date: 2024-10-30 -> Model trained, prediction made (using:
Test row date: 2024-11-30 -> Model trained, prediction made (using:
Final results_df_gb columns: ['Regime', 'Predicted_month', 'Train_St
```

	Regime	Predicted_month	Train_Start_Date	Train_End_Date	Tra
622	NoRegime	2024-02-29	2019-02-28	2024-01-30	
623	NoRegime	2024-03-30	2019-03-30	2024-02-29	
624	NoRegime	2024-04-30	2019-04-30	2024-03-30	
625	NoRegime	2024-05-30	2019-05-30	2024-04-30	
626	NoRegime	2024-06-30	2019-06-30	2024-05-30	

627	NoRegime	2024-07-30	2019-07-30	2024-06-30
628	NoRegime	2024-08-30	2019-08-30	2024-07-30
629	NoRegime	2024-09-30	2019-09-30	2024-08-30
630	NoRegime	2024-10-30	2019-10-30	2024-09-30
631	NoRegime	2024-11-30	2019-11-30	2024-10-30

10 rows × 23 columns

```
Cumulative returns 2000-01-30 - 2024-11-30 - ML strategy: 1.6938 / E
Cumulative returns 1972-04-30 - 2024-11-30 - ML strategy: 7.9123 / E
```

Gradint boosting loop

```
if gb_loop:
  import pandas as pd
  import numpy as np
  import random
  from datetime import datetime
  from xgboost import XGBClassifier
  from IPython.display import display, HTML
  import optuna
  import optuna.visualization as vis
  import time

  # ------------------
  # User-specified globals
  # ------------------
  seed = 39
  n_trials = 350

  # reproducibility
  np.random.seed(seed)
  random.seed(seed)
```

```
min_months_train    = 60    # Minimum months of data needed (5 ye
min_obs_regime      = 50    # Minimum observations per regime if
min_obs_train       = 0     # Minimum total observations after dr
use_regime_split    = False # Toggle regime-based training
use_fixed_window    = True  # True for rolling window, False for
rolling_window_size = 60    # When using a fixed window, use this

# Ensure the data is sorted by Date
df_sorted = df.sort_values('Date').reset_index(drop=True)

# Global list to store logging entries for Optuna iterations
gb_optuna_log = []

# Prepare results filename
now = datetime.now()
results_filename = f"gp_loop_results_{now.strftime('%Y%m%d')}_{nc

# --------------------
# Objective Function for Optuna
# --------------------
def objective(trial):
    start_time = time.time()

    # 1) choose number of trees (either 100 or 200)
    n_trees = trial.suggest_categorical('n_estimators', [100, 200

    # 2) single unconditional learning-rate range
    lr = trial.suggest_float('learning_rate', 0.005, 0.2, log=Tru

    # 3) other hyperparameters, including subsample now being tun
    xgb_params = {
        'n_estimators':     n_trees,
        'learning_rate':    lr,
        'max_depth':        20,
        'subsample':        trial.suggest_float('subsample', 0.5,
        'colsample_bytree': trial.suggest_float('colsample_bytree
        'min_child_weight': trial.suggest_int('min_child_weight',
        'gamma':            trial.suggest_float('gamma', 0.0, 0.7
        'n_jobs':           -1,
        'random_state':     seed,
        'eval_metric':      'mlogloss'
    }

    results = []

    # Loop through each test point
    for i in range(1, len(df_sorted)):
        test_row     = df_sorted.iloc[i]
```

```
            Predicted_month = test_row['Date']

            # Build training window
            if use_fixed_window:
                start_idx    = max(0, i - rolling_window_size)
                train_window = df_sorted.iloc[start_idx:i].copy()
            else:
                train_window = df_sorted.iloc[:i].copy()

            # Skip if not enough data
            if len(train_window) < min_months_train:
                continue

            train_start_date = train_window['Date'].iloc[0]
            train_end_date   = train_window['Date'].iloc[-1]

            # Regime-based filtering (if enabled)
            if use_regime_split:
                regime_counts = train_window[REGIMES_COLUMN].value_co
                if (regime_counts < min_obs_regime).any():
                    continue
                current_regime = test_row[REGIMES_COLUMN]
                train_window   = train_window[train_window[REGIMES_CO
                if len(train_window) < min_obs_regime:
                    continue
                regime_used = regime_short_mapping.get(current_regime
            else:
                regime_used = 'NoRegime'

            # Ensure last training date is before test date
            last_train = train_window['Date'].iloc[-1]
            if (last_train.year == Predicted_month.year and
                last_train.month >= Predicted_month.month):
                continue

            # Prepare training data
            X_train = train_window[FEATURES].dropna()
            y_train = train_window['Winning Factor'].loc[X_train.inde
            if len(X_train) < min_obs_train:
                continue

            # Encode labels
            y_cat = y_train.astype('category')
            mapping = dict(enumerate(y_cat.cat.categories))
            y_num = y_cat.cat.codes

            # Fit model
            model = XGBClassifier(**xgb_params)
```

```
            model.fit(X_train, y_num)

            # Prepare test
            X_test = train_window[FEATURES].iloc[[-1]].dropna()
            if X_test.empty:
                continue

            probs        = model.predict_proba(X_test)[0]
            pred_code    = model.classes_[probs.argmax()]
            predicted_winner = mapping[pred_code]

            # Map back to full FACTORS
            full_probs = np.zeros(len(FACTORS))
            for code, p in zip(model.classes_, probs):
                factor = mapping[code]
                if factor in FACTORS:
                    full_probs[FACTORS.index(factor)] = p

            allocated_return    = (full_probs * test_row[FACTORS].val
            equal_weight_return = np.mean(test_row[FACTORS].values)

            # Record feature levels if desired
            feature_levels = {f"Feature_Level_{f}": X_test[f].iloc[0]

            results.append({
                'Predicted_month':     Predicted_month,
                'Allocated_Return':    allocated_return,
                'Equal_Weight_Return': equal_weight_return,
                'Train_Start_Date':    train_start_date,
                'Train_End_Date':      train_end_date,
                **feature_levels
            })

        # If no valid predictions:
        if not results:
            for attr in ["duration", "total_cum_return", "filtered_cu
                trial.set_user_attr(attr, 0)
            return 0

        # Build DataFrame of results
        res_df = pd.DataFrame(results)

        # Compute cumulative returns
        total_cum    = (1 + res_df['Allocated_Return']).prod() - 1
        filt_df      = res_df[res_df['Predicted_month'] >= pd.Timesta
        filtered_cum = (1 + filt_df['Allocated_Return']).prod() - 1 i

        # Sharpe ratio function
```

```
        def sharpe(returns):
            sd = np.std(returns)
            return (np.mean(returns) / sd) * np.sqrt(12) if sd else 0

        total_sharpe    = sharpe(res_df['Allocated_Return'])
        filtered_sharpe = sharpe(filt_df['Allocated_Return']) if not

        # Log attributes
        duration = time.time() - start_time
        trial.set_user_attr("duration", duration)
        trial.set_user_attr("total_cum_return", total_cum)
        trial.set_user_attr("filtered_cum_return", filtered_cum)
        trial.set_user_attr("total_sharpe", total_sharpe)
        trial.set_user_attr("filtered_sharpe", filtered_sharpe)

        return filtered_cum  # objective: maximize post-2000 cumulati


    # --------------------
    # Callback for logging after each trial
    # --------------------
    def logging_callback(study, trial):
        ua   = trial.user_attrs
        mins, secs = divmod(int(ua.get("duration", 0)), 60)
        msg = (
            f"Trial {trial.number} in {mins:02d}:{secs:02d} | "
            f"Best: {study.best_trial.value:.4f} | "
            f"TotRet: {ua.get('total_cum_return',0):.4f} | "
            f"Ret2000+: {ua.get('filtered_cum_return',0):.4f} | "
            f"TotSharpe: {ua.get('total_sharpe',0):.4f} | "
            f"Sharpe2000+: {ua.get('filtered_sharpe',0):.4f}"
        )
        print(msg)

        log_entry = {
            'Trial': trial.number,
            'Duration': f"{mins:02d}:{secs:02d}",
            'Best_Run_So_Far': study.best_trial.value,
            'Total_Cum_Return': ua.get('total_cum_return',0),
            'Cum_Return_after_2000': ua.get('filtered_cum_return',0),
            'Total_Sharpe': ua.get('total_sharpe',0),
            'Sharpe_after_2000': ua.get('filtered_sharpe',0),
            **trial.params
        }
        gb_optuna_log.append(log_entry)
        pd.DataFrame(gb_optuna_log).to_csv(results_filename, sep=";",
```

```
    # --------------------
    # Run the Optuna study
    # --------------------
    study = optuna.create_study(
        direction="maximize",
        sampler=optuna.samplers.TPESampler(seed=seed)
    )
    study.optimize(objective, n_trials=n_trials, callbacks=[logging_c
    print("Optuna optimization completed.")

    # --------------------
    # Visualize the results
    # --------------------
    opt_history_fig    = vis.plot_optimization_history(study)
    opt_importance_fig = vis.plot_param_importances(study)
    opt_history_fig.show()
    opt_importance_fig.show()

    # Display the best trial
    best = study.best_trial
    print("Best trial:")
    print(f"  Value: {best.value:.4f}")
    for k, v in best.params.items():
        print(f"    {k}: {v}")
```

Hybrid / tää säilyttää random forestin dataframen mut averagee painot ja laskee allocated returns nistä

```
if Hybrid:
    import numpy as np
    import pandas as pd
    from functools import reduce

    # names of the two result-DataFrames in your namespace
    MODEL_DF_NAMES = ['results_df_rf', 'results_df_rf2']

    # 1) load raw factor-return sheet & rename its date column to m
    df_returns = xls_file.parse(SHEET_NAME)
    df_returns.rename(columns={'Date': 'Predicted_month'}, inplace=
    # FACTORS should already be defined as your list of factor-retu
    df_factor = df_returns[['Predicted_month'] + FACTORS].copy()

    # 2) extract each model’s month + prob vector
```

```
prob_dfs = []
for name in MODEL_DF_NAMES:
    tmp = globals()[name][['Predicted_month', 'Predicted_Probab
    tmp.rename(
        columns={'Predicted_Probabilities': f'Prob_{name}'},
        inplace=True
    )
    prob_dfs.append(tmp)

# 3) inner-join on Predicted_month
df_probs = reduce(
    lambda left, right: pd.merge(left, right, on='Predicted_mon
    prob_dfs
)

# 4) bring in the Actual_Winner from your first RF run
df_probs = pd.merge(
    df_probs,
    results_df_rf[['Predicted_month', 'Actual_Winner']],
    on='Predicted_month',
    how='left'
)

# 5) compute the averaged (“hybrid”) probabilities
df_probs['Hybrid_Predicted_Probabilities'] = df_probs.apply(
    lambda row: np.mean([row[f'Prob_{n}'] for n in MODEL_DF_NAM
    axis=1
)
# rename to match the other DataFrames’ column
df_probs.rename(
    columns={'Hybrid_Predicted_Probabilities': 'Predicted_Proba
    inplace=True
)

# 6) merge in the factor returns
df_hybrid = pd.merge(
    df_probs,
    df_factor,
    on='Predicted_month',
    how='inner'
)

# 7) compute the hybrid allocated return
df_hybrid['Allocated_Return'] = df_hybrid.apply(
    lambda row: np.dot(row['Predicted_Probabilities'], row[FACT
    axis=1
)
```

```
    # 8) keep only date, actual winner, predicted probabilities, an
    df_hybrid = df_hybrid[
        ['Predicted_month', 'Actual_Winner', 'Predicted_Probabiliti
    ]
```

Model evaluation

```
# Cell 3 — collect both RF and RF2 (and others) into your dict
results_dfs = {}

if RF:
    results_dfs["Random Forest"] = results_df_rf.copy()
    print("Results from Random Forest added.")

if RF2:
    results_dfs["Random Forest 2"] = results_df_rf2.copy()
    print("Results from Random Forest 2 added.")

if GB:
    results_dfs["Gradient Boosting"] = results_df_gb.copy()
    print("Results from Gradient Boosting added.")

if Hybrid:
    results_dfs["Hybrid"] = df_hybrid.copy()
    print("Results from Hybrid Model added.")

if not results_dfs:
    raise ValueError("No valid model was selected; set at least one

print("\nAvailable model results:")
for name, df in results_dfs.items():
    print(f" • {name}: {df.shape[0]} rows × {df.shape[1]} cols")

from IPython.display import display
for name, df in results_dfs.items():
    print(f"\n=== {name} (first 5 rows) ===")
    display(df.head())
```

```
Results from Random Forest added.
Results from Gradient Boosting added.

Available model results:
 • Random Forest: 632 rows × 23 cols
 • Gradient Boosting: 632 rows × 23 cols

=== Random Forest (first 5 rows) ===
```

	Regime	Predicted_month	Train_Start_Date	Train_End_Date	Train
0	NoRegime	1972-04-30	1967-04-30	1972-03-30	
1	NoRegime	1972-05-30	1967-05-30	1972-04-30	
2	NoRegime	1972-06-30	1967-06-30	1972-05-30	
3	NoRegime	1972-07-30	1967-07-30	1972-06-30	
4	NoRegime	1972-08-30	1967-08-30	1972-07-30	

5 rows × 23 columns

```
=== Gradient Boosting (first 5 rows) ===
```

	Regime	Predicted_month	Train_Start_Date	Train_End_Date	Train
0	NoRegime	1972-04-30	1967-04-30	1972-03-30	
1	NoRegime	1972-05-30	1967-05-30	1972-04-30	
2	NoRegime	1972-06-30	1967-06-30	1972-05-30	
3	NoRegime	1972-07-30	1967-07-30	1972-06-30	
4	NoRegime	1972-08-30	1967-08-30	1972-07-30	

5 rows × 23 columns

```
import pandas as pd
from IPython.display import display

# Increase column width so no text is truncated
```

```
pd.set_option('display.max_colwidth', None)

# Define the date range
start_date = pd.to_datetime('1968-08-01')
end_date   = pd.to_datetime('2025-01-01')

# Dictionary to store filtered results for each model using the new
filtered_results_dfs = {}

# Loop through each model's results dataframe in results_dfs and ad
for i, (model_name, df) in enumerate(results_dfs.items(), 1):
    new_model_name = f"ML{i}: {model_name}"

    # Convert 'Predicted_month' to datetime if not already
    df['Predicted_month'] = pd.to_datetime(df['Predicted_month'])

    # Filter the DataFrame within the specified date range and sort
    filtered_df = df[(df['Predicted_month'] >= start_date) & (df['P

    # Store the filtered dataframe in our new dictionary using the
    filtered_results_dfs[new_model_name] = filtered_df

    # Display the filtered results with a header showing the new mo
    print(f"\n=== Filtered Results for Model '{new_model_name}' ===
    display(filtered_df)

# Reset column width option to default after display.
pd.reset_option('display.max_colwidth')
```

```
=== Filtered Results for Model 'ML1: Random Forest' ===
```

	Regime	Predicted_month	Train_Start_Date	Train_End_Date	Tra
0	NoRegime	1972-04-30	1967-04-30	1972-03-30	
1	NoRegime	1972-05-30	1967-05-30	1972-04-30	

2	NoRegime	1972-06-30	1967-06-30	1972-05-30
3	NoRegime	1972-07-30	1967-07-30	1972-06-30
4	NoRegime	1972-08-30	1967-08-30	1972-07-30
...	...	...	...	...
627	NoRegime	2024-07-30	2019-07-30	2024-06-30
628	NoRegime	2024-08-30	2019-08-30	2024-07-30
629	NoRegime	2024-09-30	2019-09-30	2024-08-30

630	NoRegime	2024-10-30	2019-10-30	2024-09-30	
631	NoRegime	2024-11-30	2019-11-30	2024-10-30	

632 rows × 23 columns

```
=== Filtered Results for Model 'ML2: Gradient Boosting' ===
```

	Regime	Predicted_month	Train_Start_Date	Train_End_Date	Tra
0	NoRegime	1972-04-30	1967-04-30	1972-03-30	
1	NoRegime	1972-05-30	1967-05-30	1972-04-30	
2	NoRegime	1972-06-30	1967-06-30	1972-05-30	
3	NoRegime	1972-07-30	1967-07-30	1972-06-30	
4	NoRegime	1972-08-30	1967-08-30	1972-07-30	
...	...	...	...	...	
627	NoRegime	2024-07-30	2019-07-30	2024-06-30	

628	NoRegime	2024-08-30	2019-08-30	2024-07-30
629	NoRegime	2024-09-30	2019-09-30	2024-08-30
630	NoRegime	2024-10-30	2019-10-30	2024-09-30
631	NoRegime	2024-11-30	2019-11-30	2024-10-30

632 rows × 23 columns

PDP

```
# # ── Cell: combined PDP heatmaps for RF and GB ──
# if RF and GB and compute_pdp:

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

#   # 1) Settings
#   z_grid    = np.array([-2, -1, 0, 1, 2])
#   eq_w      = 1.0 / len(FACTORS)    # e.g. 0.25 if 4 factors
#   eq_w_pct  = eq_w * 100            # e.g. 25.0%

# # ── RF PDP ──
#     # prepare empty DataFrames: one per factor
#   pdp_dfs = {
#         fac: pd.DataFrame(index=FEATURES, columns=z_grid, dtype=f
#         for fac in FACTORS
```

```
#     }

#     # compute raw PDP probabilities
#     for feat in FEATURES:
#         μ, σ = X_train[feat].mean(), X_train[feat].std()
#         for z in z_grid:
#             x_val = μ + z * σ
#             X_tmp = X_train.copy()
#             X_tmp[feat] = x_val
#             avg_p = rf_model.predict_proba(X_tmp).mean(axis=0)
#             for cls, p in zip(rf_model.classes_, avg_p):
#                 pdp_dfs[cls].at[feat, z] = p

#     # convert to %-point deviations
#     pdp_dev = {
#         fac: (df.subtract(eq_w)) * 100
#         for fac, df in pdp_dfs.items()
#     }
#     annot_dfs = {
#         fac: df_dev.applymap(lambda v: f"{v:+.1f}%")
#         for fac, df_dev in pdp_dev.items()
#     }
#     vlim = max(df_dev.abs().values.max() for df_dev in pdp_dev.va

#     # plot 2×2 heatmaps
#     fig, axes = plt.subplots(2, 2, figsize=(12, 8), sharex=True,
#     axes = axes.flatten()
#     for ax, fac in zip(axes, FACTORS):
#         sns.heatmap(
#             pdp_dev[fac], ax=ax,
#             cmap="vlag", center=0.0,
#             vmin=-vlim, vmax=+vlim,
#             annot=annot_dfs[fac], fmt="",
#             cbar=(ax is axes[-1]),
#             annot_kws={"fontsize":"small"}
#         )
#         ax.set_aspect(1 / 1.5)
#         ax.set_title(f"{fac}\nDeviation from {eq_w_pct:.0f}% equa
#         ax.set_xlabel("Feature z-value")
#         ax.set_ylabel("Feature")
#     plt.tight_layout()
#     plt.show()

# # ── GB PDP ──
# if GB and compute_pdp:
#     # build mapping from numeric code → factor name
#     y_cat   = y_train.astype('category')
#     mapping = dict(enumerate(y_cat.cat.categories))
```

```
#     # prepare empty DataFrames: one per factor
#     pdp_dfs = {
#         fac: pd.DataFrame(index=FEATURES, columns=z_grid, dtype=f
#         for fac in mapping.values()
#     }

#     # compute raw PDP probabilities
#     for feat in FEATURES:
#         μ, σ = X_train[feat].mean(), X_train[feat].std()
#         for z in z_grid:
#             x_val = μ + z * σ
#             X_tmp = X_train.copy()
#             X_tmp[feat] = x_val
#             avg_p = gb_model.predict_proba(X_tmp).mean(axis=0)
#             for ci, code in enumerate(gb_model.classes_):
#                 fac = mapping[code]
#                 pdp_dfs[fac].at[feat, z] = avg_p[ci]

#     # convert to %-point deviations
#     pdp_dev = {
#         fac: (df.subtract(eq_w)) * 100
#         for fac, df in pdp_dfs.items()
#     }
#     annot_dfs = {
#         fac: df_dev.applymap(lambda v: f"{v:+.1f}%")
#         for fac, df_dev in pdp_dev.items()
#     }
#     vlim = max(df_dev.abs().values.max() for df_dev in pdp_dev.va

#     # plot 2×2 heatmaps
#     fig, axes = plt.subplots(2, 2, figsize=(12, 8), sharex=True,
#     axes = axes.flatten()
#     for ax, fac in zip(axes, pdp_dfs.keys()):
#         sns.heatmap(
#             pdp_dev[fac], ax=ax,
#             cmap="vlag", center=0.0,
#             vmin=-vlim, vmax=+vlim,
#             annot=annot_dfs[fac], fmt="",
#             cbar=(ax is axes[-1]),
#             annot_kws={"fontsize":"small"}
#         )
#         ax.set_aspect(1 / 1.5)
#         ax.set_title(f"{fac}\nDeviation from {eq_w_pct:.0f}% equa
#         ax.set_xlabel("Feature z-value")
#         ax.set_ylabel("Feature")
#     plt.tight_layout()
#     plt.show()
```

By feature plots

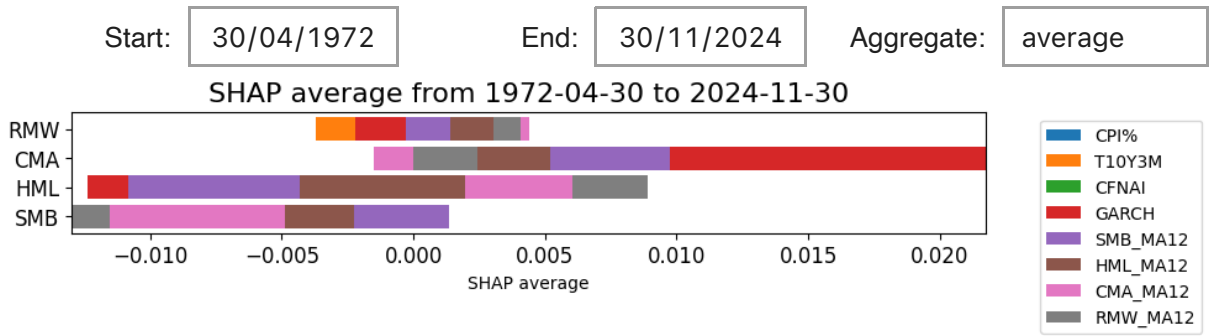
```
if RF and compute_pdp:
  fig, axes = plt.subplots(2, 4, figsize=(16, 8), sharex=True, shar
  axes = axes.flatten()
  for ax, feat in zip(axes, FEATURES):
      for fac, df in pdp_dfs.items():
          # Convert to %-pt dev:
          dev = (df.loc[feat] - eq_w) * 100
          ax.plot(z_grid, dev, marker='o', label=fac)
      ax.set_title(feat)
      ax.axhline(0, color='gray', lw=0.8)
      if feat == FEATURES[0]:
          ax.legend(fontsize='small', ncol=2)
  plt.tight_layout()
```

SHAP

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display

# ——————————————————
# 1) Build the long DataFrame from shap_data
# ——————————————————
records = []
for rec in shap_data:
    month = rec['month']
    mat   = rec['shap_values']  # shape = (n_factors, n_features)
    for i, fac in enumerate(FACTORS):
        for j, feat in enumerate(FEATURES):
            records.append({
                'month':      pd.to_datetime(month),
                'factor':     fac,
                'feature':    feat,
                'shap_value': mat[i, j]
            })
shap_df = pd.DataFrame(records)

# ——————————————————
# 2) Interactive controls
```

```
# ——————————————
start_picker = widgets.DatePicker(
    description="Start:",
    value=shap_df['month'].min().date()
)
end_picker = widgets.DatePicker(
    description="End:",
    value=shap_df['month'].max().date()
)
agg_dropdown = widgets.Dropdown(
    options=['average', 'mean', 'sum'],
    value='average',
    description='Aggregate:'
)

controls = widgets.HBox([start_picker, end_picker, agg_dropdown])

# ——————————————
# 3) Plot function
# ——————————————
def update_plot(start_date, end_date, agg):
    df = shap_df[
        (shap_df['month'] >= pd.to_datetime(start_date)) &
        (shap_df['month'] <= pd.to_datetime(end_date))
    ]
    if df.empty:
        print("No data in that range.")
        return

    grouped = df.groupby(['factor','feature'], observed=True)['shap
    if agg in ('average', 'mean'):
        df_agg = grouped.mean().reset_index()
    else:  # 'sum'
        df_agg = grouped.sum().reset_index()

    pivot = (
        df_agg
          .pivot(index='factor', columns='feature', values='shap_va
          .reindex(index=FACTORS, columns=FEATURES)
          .fillna(0)
    )

    fig, ax = plt.subplots(figsize=(10, len(FACTORS)*0.5))
    left = np.zeros(len(pivot))
    for feat in FEATURES:
        vals = pivot[feat].values
        ax.barh(pivot.index, vals, left=left, label=feat)
        left += vals
```

```
    ax.set_xlabel(f"SHAP {agg}")
    ax.set_title(f"SHAP {agg} from {start_date} to {end_date}")
    ax.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.tight_layout()
    plt.show()

# ——————————————————
# 4) Display everything
# ——————————————————
out = widgets.interactive_output(
    update_plot,
    {
      'start_date': start_picker,
      'end_date':   end_picker,
      'agg':        agg_dropdown
    }
)

display(controls, out)
```



Permutation

```
# ——————————————————————————————————————————————
# Cell 2: Interactive Permutation Importance (choose date-range & a
# ——————————————————————————————————————————————
if RF and compute_permutation_importance:
  import pandas as pd
  import matplotlib.pyplot as plt
  import ipywidgets as widgets
```

```
from IPython.display import display

# build DataFrame of permutation importances
perm_df = pd.DataFrame(
    perm_importances_list,
    index=months_list,
    columns=RF1_FEATURES
)
perm_df.sort_index(inplace=True)

# widgets: start date, end date, aggregation
date_options   = list(perm_df.index)
start_widget   = widgets.Dropdown(options=date_options, descripti
end_widget     = widgets.Dropdown(options=date_options,   descrip
agg_widget     = widgets.RadioButtons(options=['mean','median'],

def plot_perm_range(start, end, agg):
    # select slice (inclusive)
    try:
        block = perm_df.loc[start:end]
    except KeyError:
        print("Invalid range.")
        return
    if block.empty:
        print("No data for this range.")
        return

    if agg == 'mean':
        imp  = block.mean(axis=0)
        title = f'Mean Permutation Importance\n{start} → {end}'
    else:
        imp  = block.median(axis=0)
        title = f'Median Permutation Importance\n{start} → {end}'

    plt.figure(figsize=(10,6))
    imp.sort_values(ascending=True).plot.barh()
    plt.title(title)
    plt.xlabel('Importance')
    plt.tight_layout()
    plt.show()

ui = widgets.interactive(
    plot_perm_range,
    start=start_widget,
    end=end_widget,
    agg=agg_widget
)
display(ui)
```

Confusion Matrix

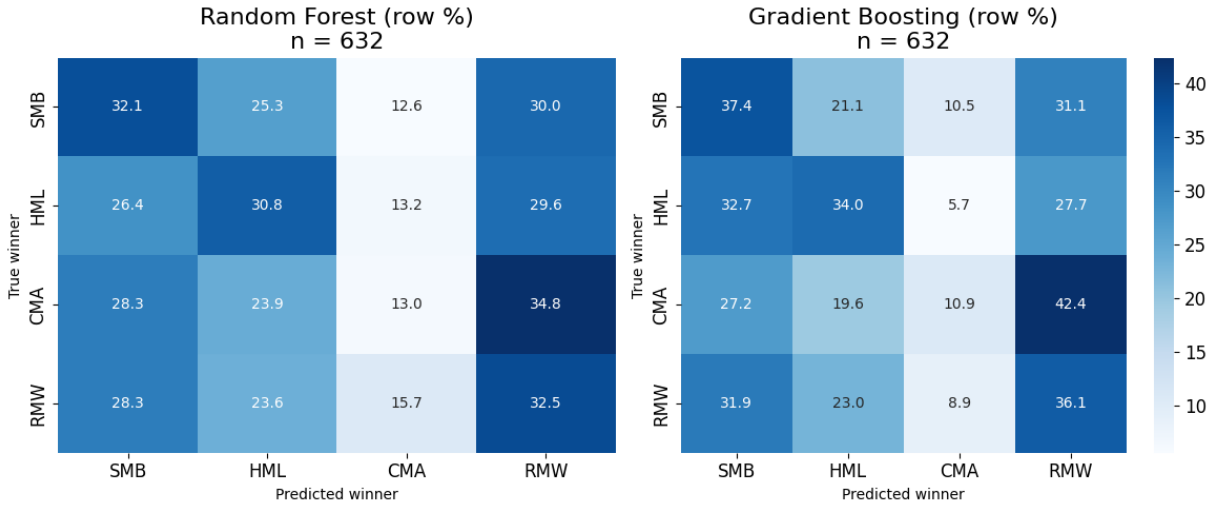
```
"""
Comprehensive evaluation script
==============================

Outputs
-------
1.  Row-normalised **confusion-matrix heat-maps** + a printed numer
    for each model
2.  **Single combined per-factor table** with a merged header row
    (Random Forest vs Gradient Boosting)
3.  Headline **Overall-Performance Summary** table
"""

# ──────────────────────────────────────────────────────────────────
# 0. Imports and expected inputs
# ──────────────────────────────────────────────────────────────────
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.metrics import (
    confusion_matrix,
    classification_report,
    accuracy_score,
    precision_score,
    recall_score,
    f1_score
)
from IPython.display import display

# The notebook/environment must already provide:
#   FACTORS      : list[str]   – the four factor names in display o
#   results_dfs  : dict[str → DataFrame] with columns
#                    • “Actual_Winner”
#                    • “Predicted_Winner”
# ──────────────────────────────────────────────────────────────────

labels = FACTORS
items  = list(results_dfs.items())             # [('Random Forest'

# ──────────────────────────────────────────────────────────────────
# 1. Confusion-matrix heat-maps **and** printed tables
```

```
# ──────────────────────────────────────────────────────────────────────
fig, axes = plt.subplots(1, len(items), figsize=(6 * len(items), 5)
axes = axes[0]

for idx, (model_name, df_res) in enumerate(items):
    cm     = confusion_matrix(df_res.Actual_Winner, df_res.Predicte
    cm_pct = (cm / cm.sum(axis=1, keepdims=True)) * 100

    # Heat-map
    sns.heatmap(
        cm_pct, annot=True, fmt=".1f", cmap="Blues",
        cbar=(idx == len(items) - 1),
        xticklabels=labels, yticklabels=labels,
        ax=axes[idx]
    )
    axes[idx].set_title(f"{model_name} (row %)\nn = {len(df_res)}")
    axes[idx].set_xlabel("Predicted winner")
    axes[idx].set_ylabel("True winner")

    # Printed numeric table
    print(f"\nConfusion matrix for {model_name} (row %):")
    print(pd.DataFrame(cm_pct, index=labels, columns=labels)
            .to_string(float_format=lambda x: f"{x:.1f}"))

plt.tight_layout()
plt.show()

# ──────────────────────────────────────────────────────────────────────
# 2. Combined per-factor metrics table (RF vs GB)
# ──────────────────────────────────────────────────────────────────────
per_factor_tables = {}

for model_name, df_res in items:
    y_true, y_pred = df_res.Actual_Winner, df_res.Predicted_Winner

    rep = classification_report(
        y_true, y_pred,
        labels=labels, target_names=labels,
        output_dict=True, zero_division=0
    )

    df_cr = (pd.DataFrame(rep).T
               .drop('accuracy', errors='ignore')
               .rename(columns={'support': 'Samples'}))

    for col in ['precision', 'recall', 'f1-score']:
        df_cr[col] = (df_cr[col] * 100).round(1)
    df_cr['Samples'] = df_cr['Samples'].astype(int)
```

```
    df_cr = (df_cr.rename(columns={'precision':'Precision',
                                   'recall':'Recall',
                                   'f1-score':'F1-score'})
                  [['Precision', 'Recall', 'F1-score', 'Samples']]

    per_factor_tables[model_name] = df_cr

# Concatenate side-by-side → MultiIndex columns: level-0=model, lev
combined = pd.concat(
    {m: tbl.drop(columns='Samples') for m, tbl in per_factor_tables
    axis=1
)

# Single shared Samples column
combined[('', 'Samples')] = next(iter(per_factor_tables.values()))[
combined = combined.reindex(
    columns=[c for c in combined.columns if c[1] != 'Samples'] + [(
)
combined.columns.set_names(['Model', 'Metric'], inplace=True)

print("\n## Per-Factor Metrics: Random Forest vs Gradient Boosting\
display(
    combined.style
            .format(lambda x: f"{x:.1f}%" if isinstance(x, (float,
                              else "{:d}".format(x))
            .set_caption("Per-Factor Metrics (Combined)")
            .set_table_styles(
                [
                    {'selector': 'th', 'props': [('text-align', 'ce
                    {'selector': 'th.col_heading.level0',
                     'props': [('border-bottom', '1pt solid black')
                ]
            )
)

# ──────────────────────────────────────────────────────────────────
# 3. Overall-performance summary
# ──────────────────────────────────────────────────────────────────
summary_rows = []
for model_name, df_res in items:
    y_true, y_pred = df_res.Actual_Winner, df_res.Predicted_Winner
    summary_rows.append({
        'Model':              model_name,
        'Accuracy (%)':       round(accuracy_score(y_true, y_pred)
        'Precision (wtd %)':  round(precision_score(y_true, y_pred,
        'Recall  (wtd %)':    round(recall_score   (y_true, y_pred,
        'F1 (wtd %)':         round(f1_score       (y_true, y_pred,
```

```
        'Samples':            len(y_true)
    })

summary_df = pd.DataFrame(summary_rows).set_index('Model')

print("\n### Overall Performance Summary ###\n")
display(
    summary_df.style
              .format({
                  'Accuracy (%)':      '{:.1f}%',
                  'Precision (wtd %)': '{:.1f}%',
                  'Recall  (wtd %)':   '{:.1f}%',
                  'F1 (wtd %)':        '{:.1f}%',
                  'Samples':           '{:d}'
              })
              .set_caption("Overall Performance Summary")
)
```

```
Confusion matrix for Random Forest (row %):
     SMB   HML   CMA   RMW
SMB 32.1 25.3 12.6 30.0
HML 26.4 30.8 13.2 29.6
CMA 28.3 23.9 13.0 34.8
RMW 28.3 23.6 15.7 32.5

Confusion matrix for Gradient Boosting (row %):
     SMB   HML   CMA   RMW
SMB 37.4 21.1 10.5 31.1
HML 32.7 34.0  5.7 27.7
CMA 27.2 19.6 10.9 42.4
RMW 31.9 23.0  8.9 36.1
```



Per-Factor Metrics: Random Forest vs Gradient Boosting

Per-Factor Metrics (Combined)

Model	Random Forest	Gradient Boosting

Metric	Precision	Recall	F1-score	Precision	Recall	F1-score	Samples
SMB	33.3%	32.1%	32.7%	34.0%	37.4%	35.6%	190
HML	29.9%	30.8%	30.3%	34.6%	34.0%	34.3%	159
CMA	13.8%	13.0%	13.4%	17.9%	10.9%	13.5%	92
RMW	31.3%	32.5%	31.9%	32.7%	36.1%	34.3%	191
macro avg	27.1%	27.1%	27.1%	29.8%	29.6%	29.4%	632
weighted avg	29.0%	29.1%	29.1%	31.4%	32.3%	31.7%	632

```
### Overall Performance Summary ###
```

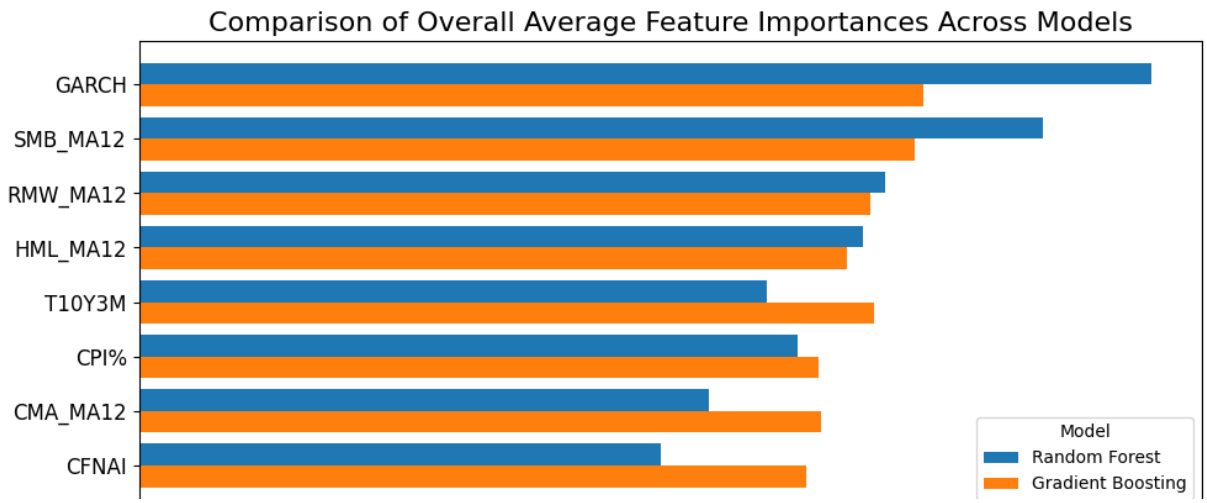
Overall Performance Summary

Model	Accuracy (%)	Precision (wtd %)	Recall (wtd %)	F1 (wtd %)	Samples
Random Forest	29.1%	29.0%	29.1%	29.1%	632
Gradient Boosting	32.3%	31.4%	32.3%	31.7%	632

Feature importance

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Assume FEATURES is defined (e.g., FEATURES = ['CPI%', 'T10YFF', '
# and results_dfs is a dictionary with keys like "RF", "GB", (and m
# where each value is a DataFrame that has a column "Feature_Import

# 1. Compute overall average feature importances for each model.
model_importances = {}
for model_key, df in results_dfs.items():
    # Stack the arrays from the "Feature_Importances" column and av
```

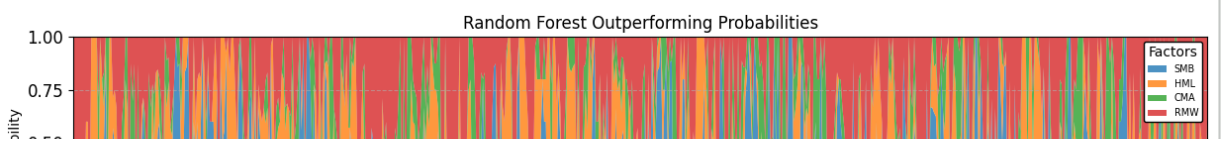
```
    model_importances[model_key] = np.vstack(df['Feature_Importance

# 2. Create a DataFrame from the computed importances.
# Rows: features, Columns: model keys.
importance_df = pd.DataFrame(model_importances, index=FEATURES)

# Optional: sort features by overall mean importance (averaged acro
# the most important features appear on top.
importance_df['Mean'] = importance_df.mean(axis=1)
importance_df = importance_df.sort_values(by='Mean', ascending=Fals
sorted_features = importance_df.index.tolist()
importance_df = importance_df.drop(columns=['Mean'])

# 3. Plot a grouped horizontal bar chart.
models = importance_df.columns.tolist()
n_models = len(models)
n_features = len(sorted_features)
y = np.arange(n_features)  # base positions for each feature group
bar_height = 0.8 / n_models  # total group thickness is 0.8

fig, ax = plt.subplots(figsize=(10, max(4, n_features * 0.6)))
for i, model in enumerate(models):
    # Calculate an offset for each model in the group.
    offset = (i - n_models/2) * bar_height + bar_height/2
    ax.barh(y + offset, importance_df.loc[sorted_features, model],

ax.set_yticks(y)
ax.set_yticklabels(sorted_features)
ax.invert_yaxis()  # so the top feature is at the top
ax.set_xlabel("Average Feature Importance")
ax.set_title("Comparison of Overall Average Feature Importances Acr
ax.legend(title="Model")
plt.tight_layout()
plt.show()
```

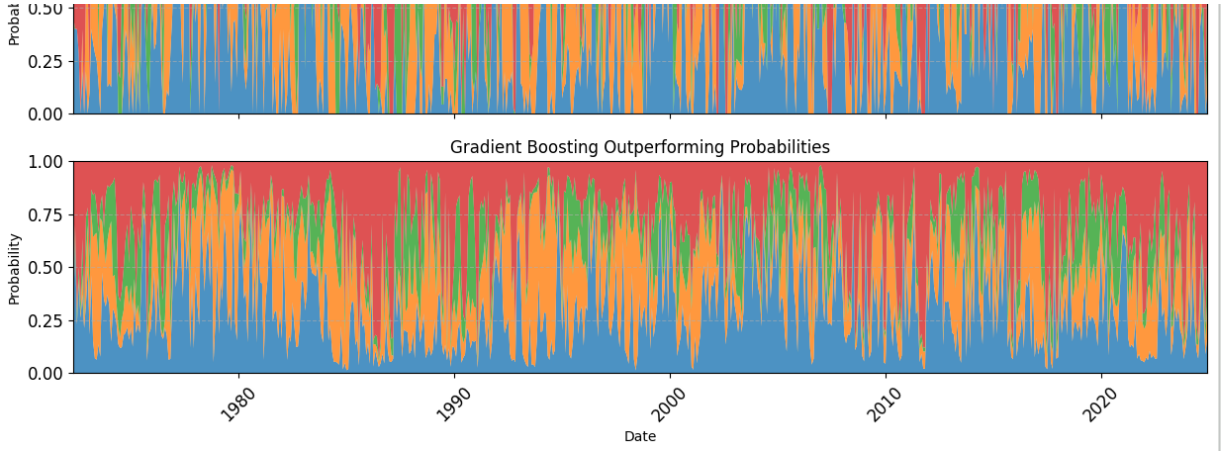


Allocation chart

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ── 0.  Figure parameters ──────────────────────────────────────
panel_height = 3.0
fig_width    = 12
num_models   = len(results_dfs)
fig_height   = panel_height * num_models

fig, axes = plt.subplots(
    nrows=num_models, ncols=1,
    figsize=(fig_width, fig_height),
    sharex=True,
)

if num_models == 1:      # keep iterable
    axes = [axes]

# ── 1.  Get the GLOBAL date span so every axis uses identical limi
all_dates = pd.concat(
    [pd.to_datetime(df["Predicted_month"], errors="coerce") for df
).dropna()
date_min, date_max = all_dates.min(), all_dates.max()

# ── 2.  Plot each model ────────────────────────────────────────
for ax, (model_key, df_model) in zip(axes, results_dfs.items()):
    df_temp = df_model.copy()

    # Parse dates & clean
    df_temp["Predicted_month"] = pd.to_datetime(df_temp["Predicted_
```

```
        df_temp = df_temp.dropna(subset=["Predicted_month"]).sort_value

        # Probability matrix → DataFrame
        full_probs = np.vstack(df_temp["Predicted_Probabilities"].value
        probability_df         = pd.DataFrame(full_probs, columns=FACTO
        probability_df["Date"] = df_temp["Predicted_month"].values

        # Stackplot
        ax.stackplot(
            probability_df["Date"],
            [probability_df[col] for col in FACTORS],
            labels=FACTORS,
            alpha=0.8,
        )

        # Aesthetics
        ax.set_title(f"{model_key} Outperforming Probabilities", fontsi
        ax.set_ylabel("Probability", fontsize=10)
        ax.set_ylim(0, 1)
        ax.set_yticks([0, 0.25, 0.5, 0.75, 1.0])
        ax.grid(axis="y", linestyle="--", alpha=0.6)

    # ── 3.  Remove x-axis padding & apply shared limits ──────────────
    for ax in axes:
        ax.set_xlim(date_min, date_max)  # exact span
        ax.margins(x=0)                  # turn off the default 5% padd

    # Legend only on the first axis (or wherever you prefer)
    axes[0].legend(
        loc="upper right",
        fontsize="x-small",
        title="Factors",
        frameon=True,
        framealpha=1.0,
        facecolor="white",
        edgecolor="black",
    )

    # ── 4.  Final touches ────────────────────────────────────────────
    axes[-1].set_xlabel("Date", fontsize=10)
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.subplots_adjust(left=0.04, right=0.995)   # trims the white bor
    plt.show()
```



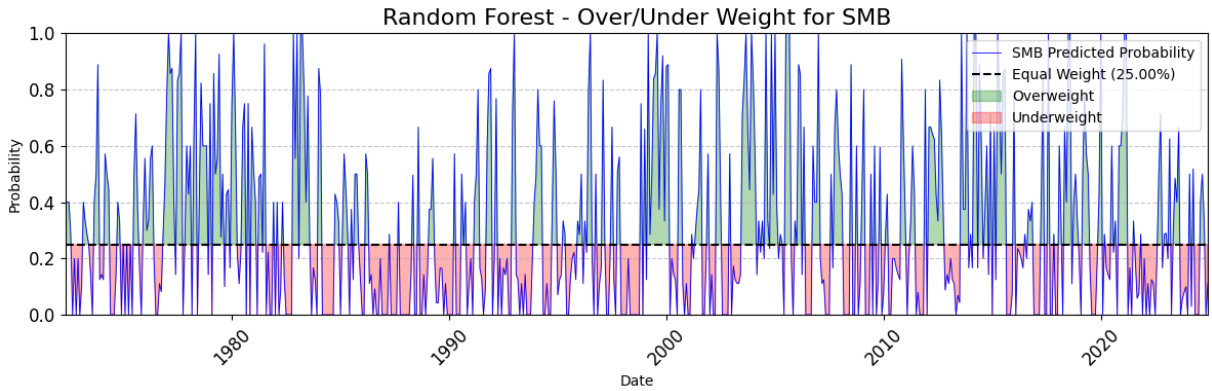


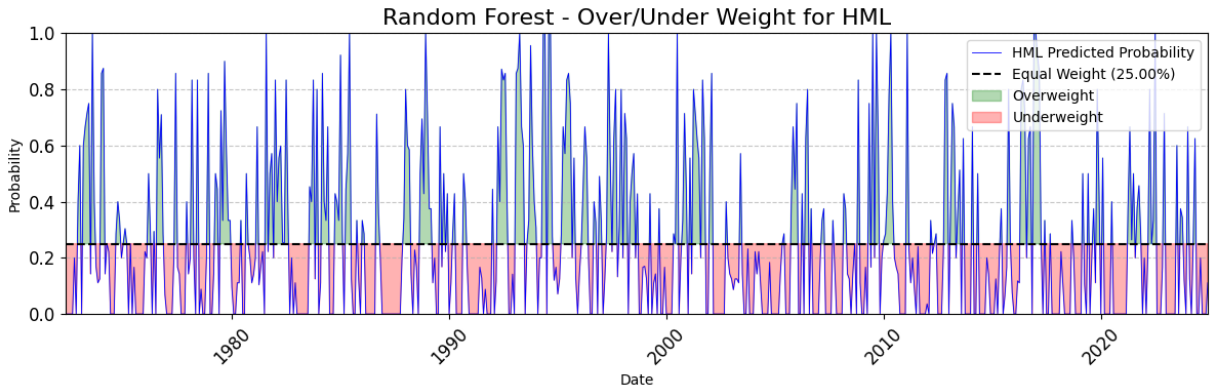
Factor weight analysis

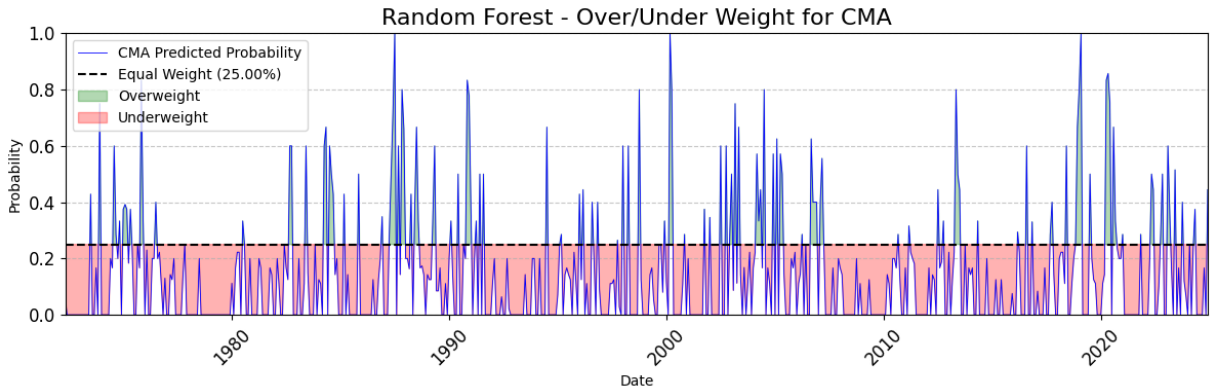
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Toggle:
combine_all_models = False  # Set to True to combine all models int

# Set the date range for viewing.
start_date = pd.to_datetime("1968-07-30")
end_date   = pd.to_datetime("2024-11-30")

# Define static equal weight value.
equal_weight = 1 / len(FACTORS)  # e.g., for 5 factors equal_weight

if combine_all_models:
    # Combined charts: One set of subplots (one per factor) for all
    n_factors = len(FACTORS)
    fig, axs = plt.subplots(n_factors, 1, figsize=(12, 4 * n_factor
    if n_factors == 1:
        axs = [axs]  # ensure axs is iterable

    for i, factor in enumerate(FACTORS):
```

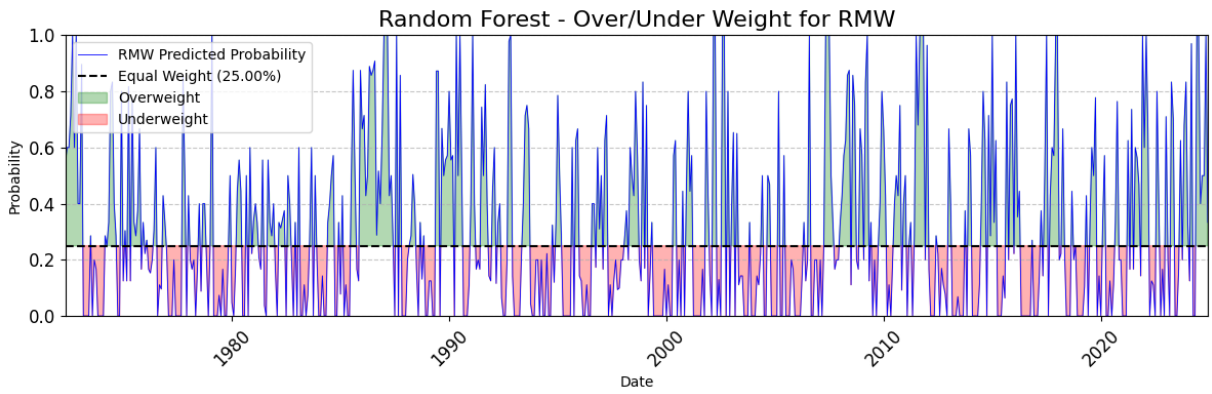
```
            ax = axs[i]
            min_dates = []
            max_dates = []

            # Loop through each model's results
            for model_key, df_model in results_dfs.items():
                df_temp = df_model.copy()
                df_temp["Predicted_month"] = pd.to_datetime(df_temp["Pr
                df_temp = df_temp.dropna(subset=["Predicted_month"]).so

                # Stack predicted probabilities into a DataFrame.
                full_probs = np.vstack(df_temp["Predicted_Probabilities
                probability_df = pd.DataFrame(full_probs, columns=FACTO
                probability_df["Date"] = df_temp["Predicted_month"]

                # Filter to desired date range.
                mask = (probability_df["Date"] >= start_date) & (probab
                filtered_df = probability_df.loc[mask].reset_index(drop

                if filtered_df.empty:
                    continue

                ax.plot(filtered_df["Date"], filtered_df[factor],
                        label=f"{factor}_{model_key}", linewdth=0.6)

                min_dates.append(filtered_df["Date"].min())
                max_dates.append(filtered_df["Date"].max())

            # Set x-axis limits to exactly the data span (if any data e
            if min_dates and max_dates:
                ax.set_xlim(min(min_dates), max(max_dates))

            # Draw the static equal weight horizontal line.
            ax.axhline(equal_weight, color='black', linestyle='--',
                       label=f"Equal Weight ({equal_weight:.2%})")

            ax.set_title(f"{factor} Predicted Probabilities Across Mode
            ax.set_ylabel("Probability", fontsize=12)
            ax.set_ylim(0, 1)
            ax.set_yticks([0, 0.25, 0.5, 0.75, 1.0])
            ax.grid(axis='y', linestyle='--', alpha=0.7)
            ax.legend(loc='best', fontsize='small')

        plt.xlabel("Date", fontsize=12)
        plt.xticks(rotation=45)
        plt.tight_layout()
        plt.show()
```

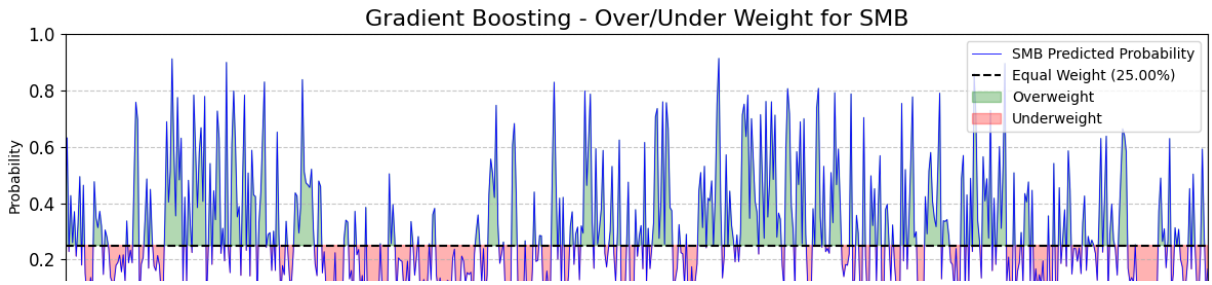
```
    else:
        # Separate charts: Loop over each model and for each factor cre
        for model_key, df_model in results_dfs.items():
            for factor in FACTORS:
                df_temp = df_model.copy()
                df_temp["Predicted_month"] = pd.to_datetime(df_temp["Pr
                df_temp = df_temp.dropna(subset=["Predicted_month"]).so

                full_probs = np.vstack(df_temp["Predicted_Probabilities
                probability_df = pd.DataFrame(full_probs, columns=FACTO
                probability_df["Date"] = df_temp["Predicted_month"]

                # Filter to desired date range.
                mask = (probability_df["Date"] >= start_date) & (probab
                filtered_df = probability_df.loc[mask].reset_index(drop

                if filtered_df.empty:
                    continue

                plt.figure(figsize=(12, 4))
                plt.plot(filtered_df["Date"], filtered_df[factor],
                         label=f"{factor} Predicted Probability", color

                plt.axhline(equal_weight, color='black', linestyle='--'
                            label=f"Equal Weight ({equal_weight:.2%})")

                plt.fill_between(filtered_df["Date"],
                                 filtered_df[factor],
                                 equal_weight,
                                 where=(filtered_df[factor] > equal_wei
                                 interpolate=True, color='green', alpha
                plt.fill_between(filtered_df["Date"],
                                 filtered_df[factor],
                                 equal_weight,
                                 where=(filtered_df[factor] < equal_wei
                                 interpolate=True, color='red', alpha=0

                plt.title(f"{model_key} - Over/Under Weight for {factor
                plt.xlabel("Date")
                plt.ylabel("Probability")
                plt.ylim(0, 1)
                # Set x-axis limits to exactly where data exists.
                plt.xlim(filtered_df["Date"].min(), filtered_df["Date"]
                plt.legend(loc='best')
                plt.grid(axis='y', linestyle='--', alpha=0.7)
                plt.xticks(rotation=45)
                plt.tight_layout()
                plt.show()
```

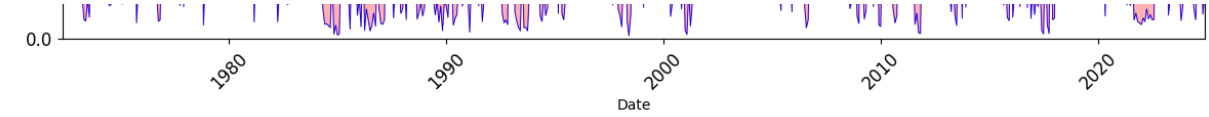

Random Forest - Over/Under Weight for SMB
SMB Predicted Probability
Equal Weight (25.00%)
Overweight
Underweight
Probability
Date
1980
1990
2000
2010
2020

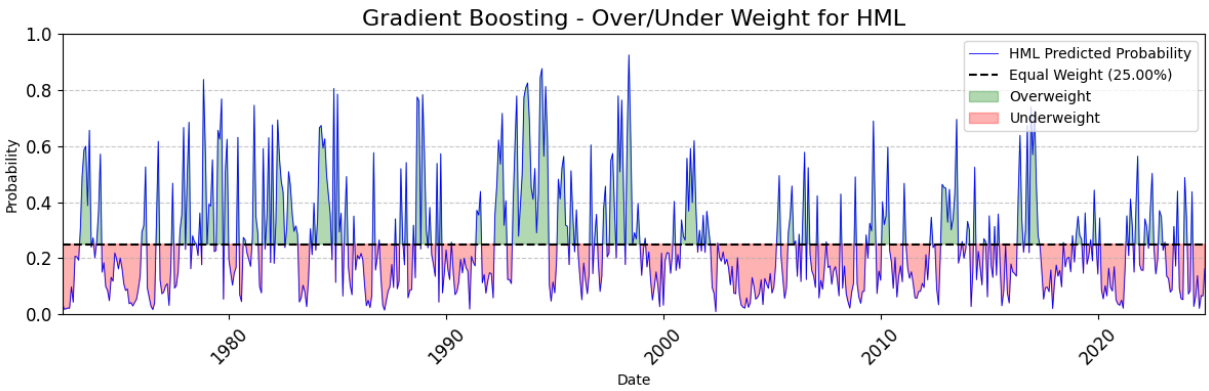

Random Forest - Over/Under Weight for HML
HML Predicted Probability
Equal Weight (25.00%)
Overweight
Underweight
Probability
Date

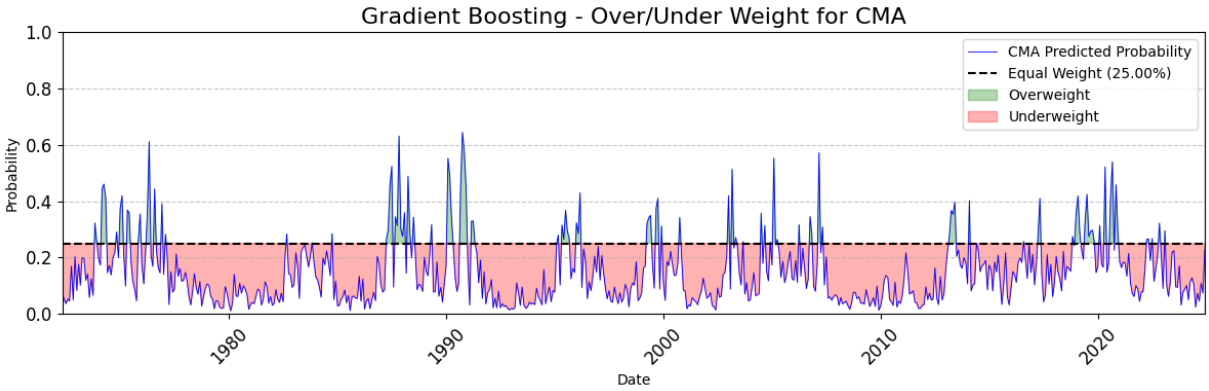

Random Forest - Over/Under Weight for CMA
CMA Predicted Probability
Equal Weight (25.00%)
Overweight
Underweight
Probability
Date

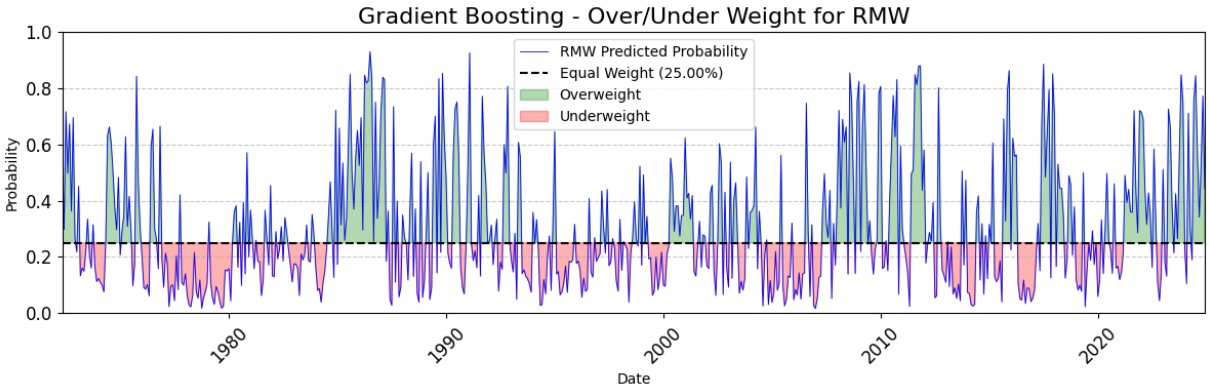

Random Forest - Over/Under Weight for RMW
RMW Predicted Probability
Equal Weight (25.00%)
Overweight
Underweight
Probability
Date


Gradient Boosting - Over/Under Weight for SMB
SMB Predicted Probability
Equal Weight (25.00%)
Overweight
Underweight
Probability


0.0
1980
1990
2000
2010
2020
Date


Gradient Boosting - Over/Under Weight for HML
HML Predicted Probability
Equal Weight (25.00%)
Overweight
Underweight
Probability
1.0
0.8
0.6
0.4
0.2
0.0
1980
1990
2000
2010
2020
Date


Gradient Boosting - Over/Under Weight for CMA
CMA Predicted Probability
Equal Weight (25.00%)
Overweight
Underweight
Probability
1.0
0.8
0.6
0.4
0.2
0.0
1980
1990
2000
2010
2020
Date


Gradient Boosting - Over/Under Weight for RMW
RMW Predicted Probability
Equal Weight (25.00%)
Overweight
Underweight
Probability
1.0
0.8
0.6
0.4
0.2
0.0
1980
1990
2000
2010
2020
Date

Total outperforming probabilities

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import HTML, display

# ------------------------------------------------
# 1. Compute average predicted probabilities per model
```

```
# ----------------------------------------------------
avg_probs_dict = {}
avg_highest_factor_weight_dict = {}

for model_key, df_model in results_dfs.items():
    full_probs = np.vstack(df_model["Predicted_Probabilities"].valu

    # 1a. Compute the average probabilities across all rows
    avg = full_probs.mean(axis=0)
    avg_probs_dict[model_key] = pd.Series(avg, index=FACTORS)

    # 1b. Compute the average of the highest weight factor
    #     (for each time step, pick the max factor weight, then ave
    avg_highest_factor_weight_dict[model_key] = full_probs.max(axis

# Create a DataFrame where rows = models, columns = factors
avg_probs_df = pd.DataFrame(avg_probs_dict).T
avg_probs_df.index.name = "Model"
avg_probs_df = avg_probs_df.round(4)

# ----------------------------------------------------
# 2. Generate consistent factor colors from stackplot
# ----------------------------------------------------
# Use a dummy stackplot to extract the assigned factor colors
_, ax_dummy = plt.subplots()
dummy_data = np.random.rand(10, len(FACTORS))
dummy_dates = pd.date_range("2000-01-01", periods=10)
stack = ax_dummy.stackplot(dummy_dates, dummy_data.T, labels=FACTOR
plt.close()  # We don't want to display this

# Build color map: factor name → RGBA color
factor_colors = {factor: poly.get_facecolor()[0] for factor, poly i

# ----------------------------------------------------
# 3. Display HTML Table of average probabilities
# ----------------------------------------------------
html_table = avg_probs_df.reset_index().to_html(index=False, classe
display(HTML("<h3>Average Outperforming Probabilities by Model</h3>

# ----------------------------------------------------
# 4. Print average of the highest factor weight by model
# ----------------------------------------------------
display(HTML("<h4>Average of the Highest Factor Weight by Model</h4
for model_key, avg_highest in avg_highest_factor_weight_dict.items(
    display(HTML(f"<p><strong>{model_key}:</strong> {avg_highest:.4

# ----------------------------------------------------
# 5. Stacked Bar Chart with Consistent Colors and Labels
```

```
# ----------------------------------------------------
fig, ax = plt.subplots(figsize=(10, 6))

bottom = np.zeros(len(avg_probs_df))
x = np.arange(len(avg_probs_df))

for factor in FACTORS:
    values = avg_probs_df[factor].values
    bars = ax.bar(x, values, bottom=bottom,
                  label=factor,
                  color=factor_colors[factor],
                  edgecolor="white",
                  linewidth=0.5)

    # Centered labels
    for bar, val in zip(bars, values):
        if val > 0.03:
            ax.text(
                bar.get_x() + bar.get_width() / 2,
                bar.get_y() + bar.get_height() / 2,
                f"{val * 100:.1f}%",
                ha="center", va="center", fontsize=9, color="white"
            )

    bottom += values

# Final touches
ax.set_xticks(x)
ax.set_xticklabels(avg_probs_df.index)
ax.set_ylabel("Strategy average factor weights")
ax.set_title("Strategy average factor weights")
ax.legend(title="Factor", loc="upper right")
plt.tight_layout()
plt.show()
```

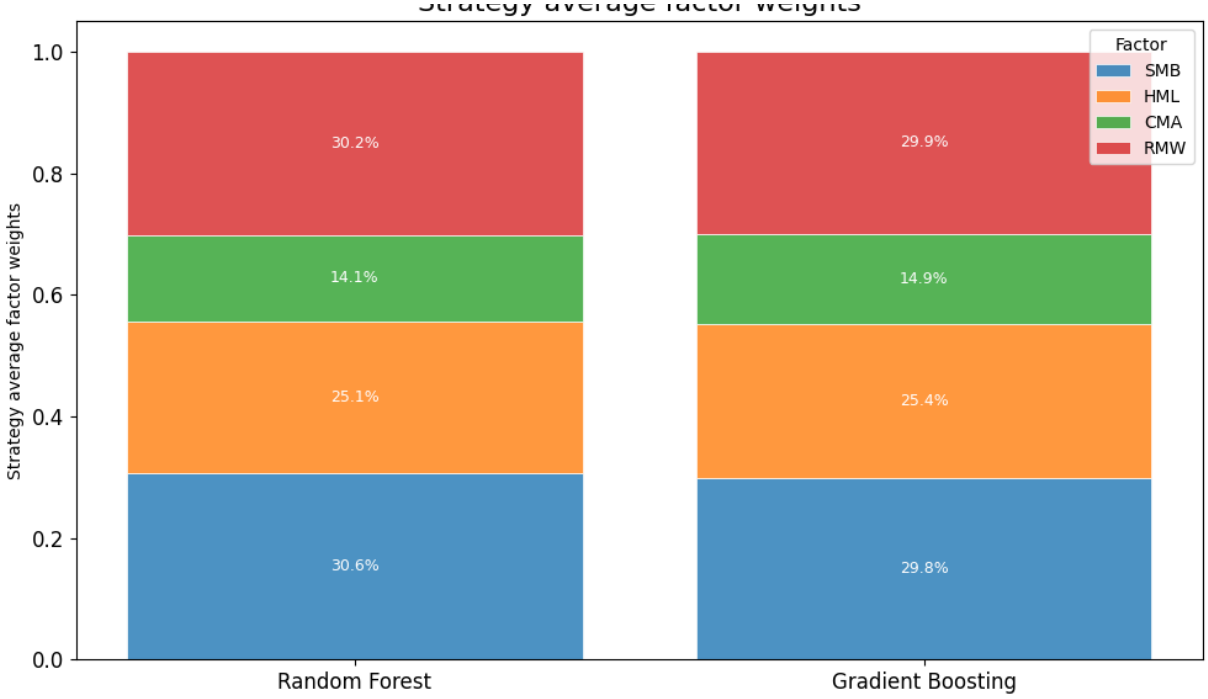
Average Outperforming Probabilities by Model

Model	SMB	HML	CMA	RMW
Random Forest	0.3059	0.2509	0.1414	0.3018
Gradient Boosting	0.2984	0.2536	0.1486	0.2994

Average of the Highest Factor Weight by Model

Random Forest: 0.6780

Gradient Boosting: 0.5364



Return data

```
import pandas as pd
from IPython.display import display

# ================================================================
# Step 0: Define column orders based on your requirements
# ----------------------------------------------------------------
# Common columns that are identical across all models.
common_cols = ['Predicted_month', 'Mkt', 'RF', 'Mkt-RF', 'Us_standa

# Model-specific columns that will be renamed.
model_specific_cols = ['Allocated_Return', 'Predicted_Winner']

# ================================================================
```

```
# Step 1. Build a base common DataFrame from one model's merged res
# ----------------------------------------------------------------
# Take the first model as the base to extract common columns.
base_key, base_df = list(results_dfs.items())[0]
base_df = base_df.copy()
base_df['Predicted_month'] = pd.to_datetime(base_df['Predicted_mont

# Merge with df_sorted (the master DataFrame sorted by date) on dat
base_df_local = base_df.merge(df_sorted, left_on='Predicted_month',
common_df = base_df_local[[c for c in common_cols if c in base_df_l

# ================================================================
# Step 2. Process each model individually to extract the model-spec
# ----------------------------------------------------------------
# We'll assign a new display name using numbering such that each mo
# "ML{number}: {Model Name}"
model_dfs = []         # Will hold one DataFrame per model.
new_model_names = []   # To store new model names.
for i, (model_key, df_model) in enumerate(results_dfs.items(), 1):
    new_model_name = f"ML{i}: {model_key}"  # New display name.
    new_model_names.append(new_model_name)

    df_temp = df_model.copy()
    df_temp['Predicted_month'] = pd.to_datetime(df_temp['Predicted_

    # Merge with df_sorted on 'Predicted_month' = 'Date'
    df_temp_local = df_temp.merge(df_sorted, left_on='Predicted_mon

    # Keep only the 'Predicted_month' plus the model-specific colum
    subset_cols = ['Predicted_month'] + [col for col in model_speci
    df_subset = df_temp_local[subset_cols].copy()

    # Rename model-specific columns with the new model name.
    rename_dict = {}
    for col in model_specific_cols:
        if col in df_subset.columns:
            rename_dict[col] = f"{new_model_name} {col}"
    df_subset.rename(columns=rename_dict, inplace=True)

    model_dfs.append(df_subset)

# ================================================================
# Step 3. Merge each model-specific DataFrame with the common DataF
# ----------------------------------------------------------------
combined_df = common_df.copy()
for df_sub in model_dfs:
    combined_df = combined_df.merge(df_sub, on='Predicted_month', h
```

```
# ================================================================
# Step 4. Reorder the columns to match the desired order.
# ----------------------------------------------------------------
benchmark_cols = ['Mkt', 'RF', 'Mkt-RF', 'Us_standard']
common_order = ['Predicted_month'] + benchmark_cols + FACTORS
# Model-specific allocated return columns.
allocated_cols = [f"{name} Allocated_Return" for name in new_model_
# Model-specific predicted winner columns.
predicted_cols = [f"{name} Predicted_Winner" for name in new_model_

final_order = common_order + ['Equal_Weight_Return'] + allocated_co
final_order = [col for col in final_order if col in combined_df.col

combined_df = combined_df[final_order].sort_values('Predicted_month

# ================================================================
# Step 5. Display the final combined results table.
# ----------------------------------------------------------------
display(combined_df)
print("\nFirst date in 'Predicted_month':", pd.to_datetime(combined
print("Last date in 'Predicted_month':", pd.to_datetime(combined_df
```

	Predicted_month	Mkt	RF	Mkt-RF	SMB	HML	CMA	
0	1972-04-30	0.0058	0.0029	0.0029	0.0023	0.0012	-0.0103	-0.0
1	1972-05-30	0.0155	0.0030	0.0125	-0.0310	-0.0270	-0.0195	0.0
2	1972-06-30	-0.0214	0.0029	-0.0243	-0.0043	-0.0248	-0.0036	0.0
3	1972-07-30	-0.0049	0.0031	-0.0080	-0.0277	0.0066	-0.0066	0.0
4	1972-08-30	0.0355	0.0029	0.0326	-0.0348	0.0454	0.0285	-0.0
...	...	...	...	...	...	...	...	
627	2024-07-30	0.0169	0.0045	0.0124	0.0828	0.0574	0.0043	0.0
628	2024-08-30	0.0209	0.0048	0.0161	-0.0365	-0.0113	0.0086	0.0
629	2024-09-30	0.0214	0.0040	0.0174	-0.0102	-0.0259	-0.0026	0.0
630	2024-10-30	-0.0058	0.0039	-0.0097	-0.0088	0.0089	0.0103	-0.0
631	2024-11-30	0.0691	0.0040	0.0651	0.0478	-0.0005	-0.0217	-0.0

632 rows × 14 columns

```
First date in 'Predicted_month': 1972-04-30 00:00:00
Last date in 'Predicted_month': 2024-11-30 00:00:00
```

Last date in `Predicted_month`: 2024-11-30 00:00:00

Next steps: Generate code with `combined_df` New interactive sheet

```
import pandas as pd
import numpy as np
import textwrap

# Define your date range
start_date = pd.to_datetime('2000-01-01')
end_date   = pd.to_datetime('2024-12-30')

# Create a filtered copy of combined_df (so the original data isn't
filtered_df = combined_df.loc[
    (combined_df['Predicted_month'] >= start_date) &
    (combined_df['Predicted_month'] <= end_date)
].copy()

# Ensure "Year" column exists in filtered_df
if 'Year' not in filtered_df.columns:
    filtered_df['Year'] = filtered_df['Predicted_month'].dt.year

# Identify ML return columns and create a name map to remove "Alloc
ml_return_cols = [c for c in filtered_df.columns if 'Allocated_Retu
ml_name_map = {ml: ml.replace("Allocated_Return", "").strip() for m

# Helper: Compute annual metrics (RF-adjusted)
def compute_annual_metrics(returns: pd.Series, rf: pd.Series):
    returns = returns.dropna()
    if returns.empty:
        return np.nan, np.nan, np.nan
    rf = rf.reindex(returns.index)
    ann_ret = (1 + returns).prod() - 1
    ann_rf  = (1 + rf).prod() - 1
    ann_ex_ret = ann_ret - ann_rf
    ann_vol = (returns - rf).std() * np.sqrt(12)
    ann_sharpe = ann_ex_ret / ann_vol if ann_vol else np.nan
    return ann_ret, ann_vol, ann_sharpe

# Build df_metrics (raw values) and df_excess (excess over Equal We
metrics_rows, excess_rows = [], []
for year, grp in filtered_df.groupby('Year'):
    ew_ret, ew_vol, ew_sharpe = compute_annual_metrics(grp['Equal_W
    row_m = {'Year': year,
             'Equal_Weight Return': ew_ret,
             'Equal_Weight Vol':    ew_vol,
```

```
                'Equal_Weight Sharpe': ew_sharpe}
        row_e = {'Year': year}

        for ml in ml_return_cols:
            ml_short = ml_name_map[ml]
            ml_ret, ml_vol, ml_sharpe = compute_annual_metrics(grp[ml],
            row_m[f"{ml_short} Return"]  = ml_ret
            row_m[f"{ml_short} Vol"]     = ml_vol
            row_m[f"{ml_short} Sharpe"]  = ml_sharpe

            row_e[f"{ml_short} Excess Return"]  = ml_ret - ew_ret
            row_e[f"{ml_short} Excess Vol"]     = ml_vol - ew_vol
            row_e[f"{ml_short} Excess Sharpe"]  = ml_sharpe - ew_sharpe

        metrics_rows.append(row_m)
        excess_rows.append(row_e)

    df_metrics = pd.DataFrame(metrics_rows).set_index('Year').sort_inde
    df_excess  = pd.DataFrame(excess_rows).set_index('Year').sort_index

    # Function: Insert newline breaks for column names longer than max_
    def wrap_colname(colname, max_width=15):
        lines = textwrap.wrap(colname, width=max_width)
        return "\n".join(lines)

    # Apply wrapping to all column names
    df_metrics.columns = [wrap_colname(col) for col in df_metrics.colum
    df_excess.columns  = [wrap_colname(col) for col in df_excess.column

    # Function: Style dataframe to enable multiline headers and format
    def style_with_wrapping_and_format(df):
        styled = df.style.set_table_styles([
            {
                'selector': 'th',
                'props': [
                    ('white-space', 'pre-wrap'),  # allow multiline
                    ('word-wrap', 'break-word')   # break long words
                ]
            }
        ]).format(lambda x: f"{x:.3f}" if isinstance(x, float) else x)
        return styled

    # Display the raw metrics and excess metrics with formatted output.
    display(style_with_wrapping_and_format(df_metrics))
    display(style_with_wrapping_and_format(df_excess))

    # ------------------------------------------------------------------
    # Summary: Count how often each ML strategy "beats" Equal Weight.
```

```
#  - Excess Return is "better" if > 0.
#  - Excess Volatility is "better" if < 0.
#  - Excess Sharpe is "better" if > 0.
#
# Average excess metrics are now calculated from all observations.
# ----------------------------------------------------------------
summary_rows = []
total_years = len(df_excess)

def w(ml_short, suffix):
    return wrap_colname(f"{ml_short} {suffix}")

for ml in ml_return_cols:
    ml_short = ml_name_map[ml]
    ret_series    = df_excess[w(ml_short, "Excess Return")]
    vol_series    = df_excess[w(ml_short, "Excess Vol")]
    sharpe_series = df_excess[w(ml_short, "Excess Sharpe")]

    ret_pos_count    = (ret_series > 0).sum()
    vol_neg_count    = (vol_series < 0).sum()
    sharpe_pos_count = (sharpe_series > 0).sum()

    # Average excess metrics now computed over all observations:
    avg_ret    = ret_series.mean()
    avg_vol    = vol_series.mean()
    avg_sharpe = sharpe_series.mean()

    summary_rows.append({
        "Strategy": ml_short,
        "Excess Return (Positive) Count": f"{ret_pos_count}/{total_
        "Avg Excess Return":   avg_ret,
        "Excess Vol (Negative) Count":    f"{vol_neg_count}/{total_
        "Avg Excess Vol":      avg_vol,
        "Excess Sharpe (Positive) Count": f"{sharpe_pos_count}/{tot
        "Avg Excess Sharpe":   avg_sharpe
    })

summary_df = pd.DataFrame(summary_rows)
summary_df.columns = [wrap_colname(col) for col in summary_df.colum

display(style_with_wrapping_and_format(summary_df))
```

	Equal_Weight Return	Equal_Weight Vol	Equal_Weight Sharpe	ML1: Random Forest Return	ML1: Random Forest Vol	MI Rand Fore Shar
Year						

Year						
2000	0.273	0.112	1.919	0.216	0.175	0.8
2001	0.178	0.114	1.226	0.112	0.129	0.5
2002	0.156	0.079	1.783	0.242	0.122	1.8
2003	0.044	0.022	1.548	0.106	0.056	1.7
2004	0.035	0.026	0.905	-0.003	0.048	-0.3
2005	0.012	0.026	-0.669	0.030	0.052	0.0
2006	0.058	0.017	0.594	0.105	0.055	1.0
2007	-0.068	0.025	-4.571	-0.034	0.038	-2.1
2008	0.093	0.040	1.951	0.102	0.035	2.4
2009	0.011	0.073	0.142	0.073	0.104	0.6
2010	0.040	0.046	0.856	0.063	0.053	1.1
2011	-0.001	0.022	-0.079	0.100	0.052	1.9
2012	0.028	0.019	1.406	-0.004	0.043	-0.1
2013	0.015	0.018	0.848	0.001	0.032	0.0
2014	-0.022	0.035	-0.625	-0.081	0.035	-2.2
2015	-0.060	0.035	-1.728	-0.039	0.073	-0.5
2016	0.104	0.050	2.040	0.194	0.096	2.0
2017	-0.054	0.039	-1.572	-0.065	0.054	-1.3
2018	-0.044	0.027	-2.280	-0.070	0.036	-2.4
2019	-0.029	0.047	-1.094	-0.071	0.052	-1.7
2020	-0.097	0.077	-1.313	0.008	0.081	0.0
2021	0.139	0.075	1.847	0.332	0.126	2.6
2022	0.169	0.086	1.807	0.079	0.111	0.5
2023	-0.066	0.070	-1.653	0.008	0.072	-0.5
2024	-0.040	0.060	-1.483	-0.017	0.078	-0.8

Year	ML1: Random Forest Excess Return	ML1: Random Forest Excess Vol	ML1: Random Forest Excess Sharpe	ML2: Gradient Boosting Excess Return	ML2: Gradient Boosting Excess Vol	ML2: Gradient Boosting Excess Sharpe

2000	-0.057	0.063	-1.020	0.059	0.014	0.251
2001	-0.065	0.016	-0.653	-0.041	0.025	-0.517
2002	0.086	0.044	0.067	0.010	0.011	-0.104
2003	0.061	0.034	0.162	0.021	0.025	-0.371
2004	-0.039	0.022	-1.223	-0.020	0.016	-0.826
2005	0.017	0.026	0.670	0.000	0.012	0.220
2006	0.047	0.037	0.455	-0.007	0.019	-0.509
2007	0.034	0.012	2.425	-0.000	0.009	1.156
2008	0.009	-0.005	0.505	0.011	0.002	0.138
2009	0.062	0.032	0.552	0.046	0.006	0.581
2010	0.023	0.007	0.321	0.015	-0.009	0.602
2011	0.101	0.030	1.994	0.086	0.018	2.154
2012	-0.031	0.024	-1.508	-0.020	0.019	-1.212
2013	-0.014	0.014	-0.821	-0.003	0.002	-0.226
2014	-0.059	0.000	-1.661	-0.057	0.001	-1.586
2015	0.021	0.038	1.195	-0.025	0.021	0.199
2016	0.090	0.046	-0.040	0.056	0.028	-0.023
2017	-0.011	0.015	0.218	-0.023	0.005	-0.348
2018	-0.025	0.008	-0.179	-0.007	0.003	-0.030
2019	-0.042	0.006	-0.681	-0.016	0.004	-0.236
2020	0.104	0.004	1.353	0.044	0.016	0.699
2021	0.193	0.051	0.783	0.100	0.023	0.589
2022	-0.090	0.026	-1.226	-0.032	0.013	-0.553
2023	0.074	0.002	1.076	0.077	-0.005	1.058
2024	0.023	0.018	0.647	-0.012	0.001	-0.182

Strategy	Excess Return (Positive) Count	Avg Excess Return	Excess Vol (Negative) Count	Avg Excess Vol	Excess Sharpe (Positive) Count	Exc Sha

Cumulative returns table

```
import pandas as pd
import numpy as np

# ------------------------------------------------------------------
# PRELIMINARY: Use the merged multi-model table (combined_df)
# ------------------------------------------------------------------
start_date = pd.to_datetime('1970-01-01')
end_date   = pd.to_datetime('2024-12-30')
df_filtered = combined_df[(combined_df['Predicted_month'] >= start_
                          (combined_df['Predicted_month'] <= end_

# Rename benchmark column if present (using the first element of BE
rename_dict = {}
if BENCHMARK[0] in df_filtered.columns:
    rename_dict[BENCHMARK[0]] = 'Benchmark Return'
df_filtered.rename(columns=rename_dict, inplace=True)

# ------------------------------------------------------------------
# Remove RF from factors (we exclude it)
factors_to_use = [fac for fac in FACTORS if fac.upper() != 'RF']

# Define possible benchmark columns (for cumulative returns).
possible_bench = ["Benchmark Return", "Mkt", "Mkt-RF", "Us_standard
benchmark_cols = [col for col in possible_bench if col in df_filter

# ------------------------------------------------------------------
# Calculate Equal-Weighted Returns based on factors_to_use.
if all(f in df_filtered.columns for f in factors_to_use):
    df_filtered['Equal Factor Weight Strategy Return'] = df_filtere
    equal_ret_col_list = ['Equal Factor Weight Strategy Return']
else:
    equal_ret_col_list = []

# ------------------------------------------------------------------
# Predicted Winner Weighted Strategy Return:
# Use the base model's predicted winner column using new naming.
base_model_key = list(results_dfs.keys())[0]
base_model_new = f"ML1: {base_model_key}"  # First model is ML1.
base_model_pred_col = f"{base_model_new} Predicted_Winner"
if base_model_pred_col in df_filtered.columns:
    df_filtered['Predicted_Winner'] = df_filtered[base_model_pred_c

def calc_winner_strategy(row):
    pred = row['Predicted_Winner']
```

```
        if pred in factors_to_use:
            other_factors = [f for f in factors_to_use if f != pred]
            if other_factors:
                return 0.5 * row[pred] + 0.5 * row[other_factors].mean(
            else:
                return row[pred]
        else:
            return row[factors_to_use].mean()

    df_filtered['Predicted Winner Weighted Strategy Return'] = df_filte

    # ------------------------------------------------------------------
    # Compute cumulative returns for each return series.
    # We'll work on a copy for cumulative computations.
    cum = df_filtered.copy()

    # 1. For each model's allocated return (using new names).
    allocated_cum_cols = []
    new_model_names = [f"ML{i}: {model_key}" for i, model_key in enumer
    for name in new_model_names:
        col_alloc = f"{name} Allocated_Return"
        if col_alloc in cum.columns:
            new_cum_col = col_alloc.replace("Allocated_Return", "Cumula
            cum[new_cum_col] = (1 + cum[col_alloc]).cumprod() - 1
            allocated_cum_cols.append(new_cum_col)

    # 2. For equal factor weight returns.
    if 'Equal Factor Weight Strategy Return' in cum.columns:
        cum['Equal Factor Weight Cumulative Return'] = (1 + cum['Equal
        equal_cum_cols = ['Equal Factor Weight Cumulative Return']
    else:
        equal_cum_cols = []

    # 3. For each benchmark column.
    bench_cum_cols = []
    # If "Benchmark Return" is available, use the actual benchmark name
    if "Benchmark Return" in cum.columns:
        new_bench_col = f"{BENCHMARK[0]} Cumulative Return"
        cum[new_bench_col] = (1 + cum["Benchmark Return"]).cumprod() -
        bench_cum_cols.append(new_bench_col)
    # Process any other benchmark columns
    for col in benchmark_cols:
        if col != "Benchmark Return":
            new_col = col + " Cumulative Return"
            cum[new_col] = (1 + cum[col]).cumprod() - 1
            bench_cum_cols.append(new_col)

    # 4. For Predicted Winner Weighted Strategy Return.
```

```
if 'Predicted Winner Weighted Strategy Return' in cum.columns:
    cum['Predicted Winner Weighted Cumulative Return'] = (1 + cum['

# 5. For each factor in factors_to_use: compute cumulative returns.
factor_cum_cols = []
for fac in factors_to_use:
    if fac in cum.columns:
        new_name = fac + " Cumulative"
        cum[new_name] = (1 + cum[fac]).cumprod() - 1
        factor_cum_cols.append(new_name)

# ------------------------------------------------------------------
# Build the final cumulative returns table.
# Final order:
#   a. Common columns: Predicted_month, then benchmark cumulative r
#   b. Then Equal Factor Weight Cumulative Return.
#   c. Then each model's Cumulative Allocated Return.
#   d. Then Predicted Winner Weighted Cumulative Return.
# ------------------------------------------------------------------
final_common_order = ['Predicted_month'] + bench_cum_cols + factor_
final_order = final_common_order + equal_cum_cols + allocated_cum_c
if 'Predicted Winner Weighted Cumulative Return' in cum.columns:
    final_order.append('Predicted Winner Weighted Cumulative Return

final_order = [col for col in final_order if col in cum.columns]
cumulative_table = cum[final_order].sort_values('Predicted_month').

# ------------------------------------------------------------------
# Display the Final Cumulative Returns Table.
# ------------------------------------------------------------------
print("Cumulative Returns Table:")
display(cumulative_table)
print("\nFirst date in 'Predicted_month':", pd.to_datetime(cumulati
print("Last date in 'Predicted_month':", pd.to_datetime(cumulative_
```

```
Cumulative Returns Table:
```

	Predicted_month	Mkt Cumulative Return	Mkt-RF Cumulative Return	SMB Cumulative	HML Cumulative
0	1972-04-30	0.005800	0.002900	0.002300	0.001200
1	1972-05-30	0.021390	0.015436	-0.028771	-0.025832
2	1972-06-30	-0.000468	-0.009239	-0.032948	-0.049992
3	1972-07-30	-0.005366	-0.017165	-0.059735	-0.043722

4	1972-08-30	0.029944	0.014875	-0.092456	-0.000307
...	...	...	...	...	...
627	2024-07-30	218.141845	22.118468	0.787200	3.728210
628	2024-08-30	222.721910	22.490675	0.721967	3.674781
629	2024-09-30	227.509558	22.899413	0.704403	3.553704
630	2024-10-30	226.184203	22.667588	0.689404	3.594232
631	2024-11-30	241.882631	24.208348	0.770158	3.591935

632 rows × 11 columns

```
First date in 'Predicted_month': 1972-04-30 00:00:00
Last date in 'Predicted_month': 2024-11-30 00:00:00
```

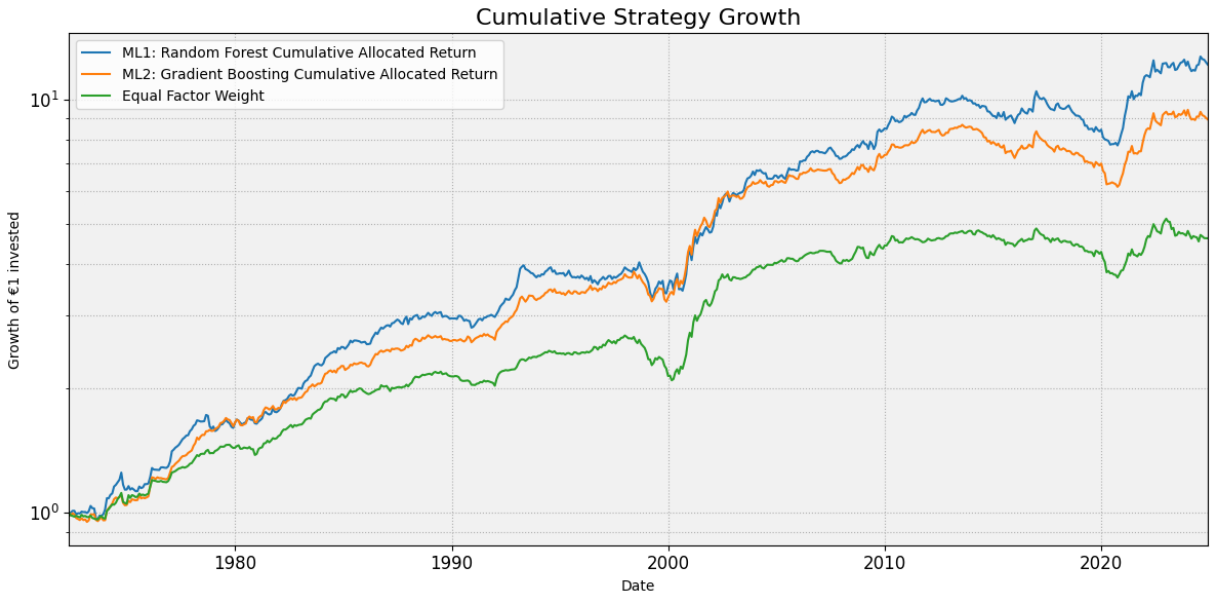
Next steps: Generate code with cumulative_table | New interactive sheet

Cumulative returns chart

```
import matplotlib.pyplot as plt

# ──────────────────────────────────────────────────────────────────────
# User-configurable toggles
# ──────────────────────────────────────────────────────────────────────
cut_off_date        = pd.to_datetime("1970-01-01")   # ① where to
show_50_50_strategy = False                          # keep your ol
show_benchmark      = False
use_log_scale       = True                           # ← log scale
shade_out_of_sample = True                           # ② grey shad

# ──────────────────────────────────────────────────────────────────────
# Prep the data
# ──────────────────────────────────────────────────────────────────────
df_plot  = cumulative_table.copy()
start_dt = df_plot['Predicted_month'].min()
end_dt   = df_plot['Predicted_month'].max()

# Helper: rebase any cumulative-return series so it equals 1 at cut
def rebase(series, cut_date):
    """Convert cumulative-return series to wealth curve rebased to
    wealth = 1 + series            # turn cum-return into wealth
    base   = wealth.loc[series.index == cut_date]
```

```
    if not base.empty and base.iloc[0] != 0:          # protect aga
        return wealth / base.iloc[0]
    else:
        return wealth / wealth.iloc[0]              # fallback: re

# Apply rebasing to every cumulative series we intend to plot
rebased_cols = {}   # {col_name: rebased_series}
for col in df_plot.columns:
    if ("Cumulative" in col) and (col != "Predicted_month"):
        rebased_cols[col] = rebase(
            df_plot.set_index('Predicted_month')[col],
            cut_off_date
        )

# ──────────────────────────────────────────────────────────────────
# Plot
# ──────────────────────────────────────────────────────────────────
plt.figure(figsize=(12, 6))
plt.clf()

# 1. ML models
for col, series in rebased_cols.items():
    if "Cumulative Allocated Return" in col:
        plt.plot(series.index, series, label=col)

# 2. Equal-weight factor
if 'Equal Factor Weight Cumulative Return' in rebased_cols:
    plt.plot(rebased_cols['Equal Factor Weight Cumulative Return'].
             rebased_cols['Equal Factor Weight Cumulative Return'],
             label='Equal Factor Weight')

# 3. Benchmarks (optional)
if show_benchmark:
    for col in rebased_cols:
        if ("Benchmark" in col or "Mkt" in col or "Us_standard" in
            plt.plot(rebased_cols[col].index, rebased_cols[col], la

# 4. 50/50 predicted-winner strategy (optional)
if show_50_50_strategy and 'Predicted Winner Weighted Cumulative Re
    plt.plot(rebased_cols['Predicted Winner Weighted Cumulative Ret
             rebased_cols['Predicted Winner Weighted Cumulative Ret
             label='50%/50% Predicted Winner')

# ──────────────────────────────────────────────────────────────────
# Cosmetics (log-scale, cut-off line, shading)
# ──────────────────────────────────────────────────────────────────
plt.xlabel("Date")
plt.ylabel("Growth of €1 invested")
```

```
plt.title("Cumulative Strategy Growth")
plt.xlim(start_dt, end_dt)
plt.yscale('log' if use_log_scale else 'linear')
plt.grid(True, which='both', linestyle=':')
plt.legend()

# Vertical cut-off marker
plt.axvline(cut_off_date, color='gray', linestyle='--', linewidth=1

# Shaded out-of-sample region
if shade_out_of_sample:
    plt.axvspan(cut_off_date, end_dt, color='lightgrey', alpha=0.3,

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



PERFORMANCE METRICS

```
import pandas as pd
```

```
import numpy as np
from IPython.display import display, HTML

# ╔═══════════════════════════════════════════════════════════════════════
# ║                  U S E R   C O N F I G
# ╚═══════════════════════════════════════════════════════════════════════
METRIC_START_DATE = '1972-04-30'     # inclusive — set None for "ea
METRIC_END_DATE   = '2024-12-30'     # inclusive — set None for "la

# ──────────────────────────────────────────────────────────────────────
#  1)  S U B S E T   D A T A
# ──────────────────────────────────────────────────────────────────────
df_metrics = combined_df.copy()
df_metrics['Predicted_month'] = pd.to_datetime(df_metrics['Predicte

if METRIC_START_DATE is not None:
    df_metrics = df_metrics[df_metrics['Predicted_month'] >= METRIC
if METRIC_END_DATE   is not None:
    df_metrics = df_metrics[df_metrics['Predicted_month'] <= METRIC

print(f"\n=== PERFORMANCE METRICS "
      f"({df_metrics['Predicted_month'].min().date()} → "
      f"{df_metrics['Predicted_month'].max().date()}) ===")

# ──────────────────────────────────────────────────────────────────────
#  2)  H E L P E R   F N S
# ──────────────────────────────────────────────────────────────────────
def annualized_metrics(monthly_returns):
    """Annualised return, volatility & Sharpe from monthly returns.
    monthly_returns = monthly_returns.fillna(0)
    mean_m, std_m = monthly_returns.mean(), monthly_returns.std()
    ann_ret = mean_m * 12
    ann_vol = std_m * np.sqrt(12)
    sharpe  = ann_ret / ann_vol if ann_vol != 0 else np.nan
    return ann_ret, ann_vol, sharpe

def max_drawdown(monthly_returns):
    wealth = (1 + monthly_returns.fillna(0)).cumprod()
    return (wealth / wealth.cummax() - 1).min()

# ──────────────────────────────────────────────────────────────────────
#  3)  P I C K   S T R A T E G Y   C O L U M N S
#      (same logic you had, just on the *filtered* frame)
# ──────────────────────────────────────────────────────────────────────
all_columns = df_metrics.columns.tolist()
strategy_cols = [
    col for col in all_columns
    if (
```

```
        (("Return" in col) or (col == BENCHMARK[0]) or (col == "Ben
        and ("Cumulative" not in col)
        and (col not in ["Actual_Winner", "Predicted_month"])
    )
]

print("Strategy columns used:")
print(strategy_cols)

# ──────────────────────────────────────────────────────────────────
#  4)  C A L C U L A T E   M E T R I C S
# ──────────────────────────────────────────────────────────────────
metrics = []
for col in strategy_cols:
    monthly = df_metrics[col]
    ann_ret, ann_vol, sharpe = annualized_metrics(monthly)
    mdd = max_drawdown(monthly)
    metrics.append({
        "Strategy": col,
        "Annualised Return":   f"{ann_ret*100:.2f}%",
        "Annualised Volatility": f"{ann_vol*100:.2f}%",
        "Sharpe Ratio":        f"{sharpe:.2f}",
        "Max Drawdown":        f"{mdd*100:.2f}%"
    })

metrics_df = (
    pd.DataFrame(metrics)
      .sort_values("Strategy")
      .reset_index(drop=True)
)

# ──────────────────────────────────────────────────────────────────
#  5)  D I S P L A Y   (with title above the table)   ←  updated
# ──────────────────────────────────────────────────────────────────
title = f"PERFORMANCE METRICS ({df_metrics['Predicted_month'].min()
        f"{df_metrics['Predicted_month'].max().date()})"

html = metrics_df.to_html(index=False)                # the table i
display(HTML(f"<h3 style='margin-bottom:8px'>{title}</h3>{html}"))
```

```
=== PERFORMANCE METRICS (1972-04-30 → 2024-11-30) ===
Strategy columns used:
['Mkt', 'Equal_Weight_Return', 'ML1: Random Forest Allocated_Return'
```

PERFORMANCE METRICS (1972-04-30 → 2024-11-30)

Strategy	Annualised Return	Annualised Volatility	Sharpe Ratio	Max Drawdown

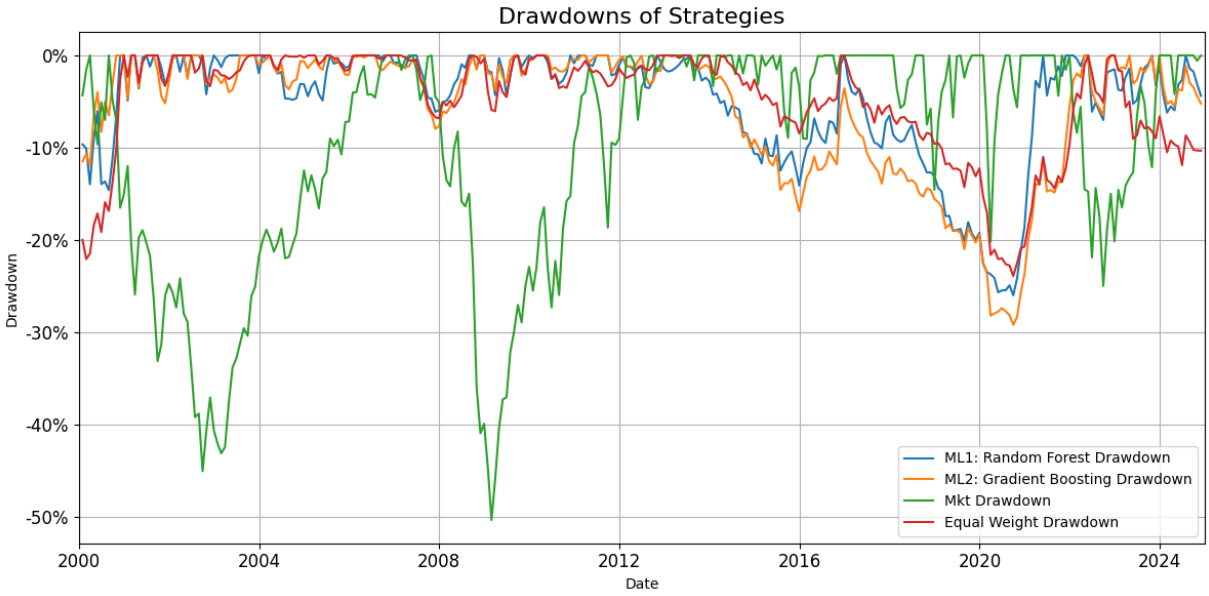
Equal_Weight_Return	3.03%	5.08%	0.60	-23.93%
ML1: Random Forest Allocated_Return	5.00%	7.13%	0.70	-26.02%
ML2: Gradient Boosting	4.34%	6.04%	0.72	-29.22%

Drawdown chart

```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker

# ---------------------------------------------------------------------
# 1. TOGGLE OPTIONS
# ---------------------------------------------------------------------
show_benchmark_drawdown = True           # Toggle benchmark drawdow
show_equal_weight_drawdown = True        # Toggle Equal Weight (sin
show_winner_weighted_drawdown = False     # Toggle Winner Weighted

# ---------------------------------------------------------------------
# 2. COPY cumulative_table (assumed computed previously)
# ---------------------------------------------------------------------
drawdown_df = cumulative_table.copy()

# ---------------------------------------------------------------------
# 3. CALCULATE DRAWDOWNS USING WEALTH INDEX (Wealth = 1 + Cumulativ
# ---------------------------------------------------------------------

# a) For each ML model's cumulative allocated return column:
ml_alloc_cols = [col for col in drawdown_df.columns if "Cumulative
for col in ml_alloc_cols:
    wealth = 1 + drawdown_df[col]
    drawdown_name = col.replace("Cumulative Allocated Return", "Dra
    drawdown_df[drawdown_name] = wealth / wealth.cummax() - 1

# b) For Benchmark:
# Use the benchmark name from BENCHMARK[0]
benchmark_name = BENCHMARK[0]  # for example, "Mkt"
benchmark_cum_col = f"{benchmark_name} Cumulative Return"
if benchmark_cum_col in drawdown_df.columns:
    wealth = 1 + drawdown_df[benchmark_cum_col]
    drawdown_df[f"{benchmark_name} Drawdown"] = wealth / wealth.cum
elif "Mkt Cumulative Return" in drawdown_df.columns:
    wealth = 1 + drawdown_df["Mkt Cumulative Return"]
    drawdown_df["Mkt Drawdown"] = wealth / wealth.cummax() - 1
```

```
else:
    print("WARNING: No benchmark cumulative return column found.")

# c) For Equal Weight (single version):
if "Equal Factor Weight Cumulative Return" in drawdown_df.columns:
    wealth = 1 + drawdown_df["Equal Factor Weight Cumulative Return
    drawdown_df["Equal Weight Drawdown"] = wealth / wealth.cummax()

# d) For Predicted Winner Weighted:
if "Predicted Winner Weighted Cumulative Return" in drawdown_df.col
    wealth = 1 + drawdown_df["Predicted Winner Weighted Cumulative
    drawdown_df["Winner Weighted Drawdown"] = wealth / wealth.cumma

# ------------------------------------------------------------------
# 4. FILTER BY DATE RANGE
# ------------------------------------------------------------------
start_date = pd.to_datetime("2000-01-01")
end_date   = pd.to_datetime("2024-12-31")
drawdown_df["Predicted_month"] = pd.to_datetime(drawdown_df["Predic
plot_df = drawdown_df[(drawdown_df["Predicted_month"] >= start_date
                      (drawdown_df["Predicted_month"] <= end_date)]

# ------------------------------------------------------------------
# 5. PLOT THE DRAWDOWNS
# ------------------------------------------------------------------
plt.figure(figsize=(12, 6))
plt.clf()  # Clear any existing figure

# a) Plot each ML model's drawdown (those columns that start with "
for col in plot_df.columns:
    if col.startswith("ML") and "Drawdown" in col:
        plt.plot(plot_df["Predicted_month"], plot_df[col], label=co

# b) Plot Benchmark Drawdown if toggled on
if show_benchmark_drawdown:
    # Look for the benchmark drawdown column with the actual benchm
    bench_drawdown_col = f"{benchmark_name} Drawdown"
    if bench_drawdown_col in plot_df.columns:
        plt.plot(plot_df["Predicted_month"], plot_df[bench_drawdown

# c) Plot Equal Weight Drawdown if available
if show_equal_weight_drawdown and "Equal Weight Drawdown" in plot_d
    plt.plot(plot_df["Predicted_month"], plot_df["Equal Weight Draw

# d) Plot Winner Weighted Drawdown if toggled on
if show_winner_weighted_drawdown and "Winner Weighted Drawdown" in
    plt.plot(plot_df["Predicted_month"], plot_df["Winner Weighted D
```

```
# Format the y-axis as percentages.
plt.gca().yaxis.set_major_formatter(
    mticker.FuncFormatter(lambda val, _: f"{val*100:.0f}%")
)

# ------------------------------------------------------------------
# 6. ENSURE THE X-AXIS SPANS EXACTLY THE DEFINED DATE RANGE
# ------------------------------------------------------------------
plt.xlabel("Date")
plt.ylabel("Drawdown")
plt.title("Drawdowns of Strategies")
plt.xlim(start_date, end_date)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Regression

```
import pandas as pd
import numpy as np
import statsmodels.api as sm
from IPython.display import display

# ------------------------------------------------------------------
# CONFIGURATION
# ------------------------------------------------------------------
subtract_rf    = True
reg_start_date = pd.to_datetime('1972-04-30')
reg_end_date   = pd.to_datetime('2024-11-30')

# ------------------------------------------------------------------
# 1. LOAD FAMA-FRENCH DATA
# ------------------------------------------------------------------
xls_file = pd.ExcelFile("/content/Gradu/THE_2ND_latest.xlsx")
df_factors = xls_file.parse("FF5", dtype=str)
df_factors["Date"] = pd.to_datetime(df_factors["Date"])

factors      = ['Mkt-RF', 'SMB', 'HML', 'RMW', 'CMA']
all_ff5_cols = factors + ['RF']

for col in all_ff5_cols:
    df_factors[col] = (
        df_factors[col]
          .str.replace(",", ".", regex=False)
          .astype(float)
          .div(100)      # convert from percent to decimal
    )

df_factors = df_factors.sort_values("Date").reset_index(drop=True)

# ------------------------------------------------------------------
# 2. PREPARE MODEL NAMES AND STORAGE
# ------------------------------------------------------------------
# `results_dfs` and `combined_df` must already exist in your sessio
new_model_names         = [f"ML{i}: {k}" for i, k in enumerate(resu
regression_summary_list = []

# ------------------------------------------------------------------
# 3. RUN REGRESSIONS FOR EACH MODEL
# ------------------------------------------------------------------
for new_model_name in new_model_names:
    col = f"{new_model_name} Allocated_Return"
    if col not in combined_df.columns:
        print(f"Skipping {new_model_name}: no column '{col}'")
        continue
```

```
        # filter & rename
        df_model = (
            combined_df.loc[
                (pd.to_datetime(combined_df['Predicted_month']) >= reg_
                (pd.to_datetime(combined_df['Predicted_month']) <= reg_
                ['Predicted_month', col]
            ]
            .rename(columns={col: "Allocated_Return"})
        )
        df_model['Predicted_month'] = pd.to_datetime(df_model['Predicte

        # merge factors
        merged = (
            pd.merge(
                df_model,
                df_factors[['Date'] + all_ff5_cols],
                left_on="Predicted_month", right_on="Date", how="inner"
              .drop(columns=["Date"])
        )

        # subtract RF?
        merged['Y'] = merged['Allocated_Return'] - merged['RF'] if subt

        # regress
        X     = sm.add_constant(merged[factors])
        y     = merged['Y']
        model = sm.OLS(y, X, missing='drop').fit()

        # extract
        alpha_dec   = model.params.get('const', np.nan)
        alpha_t     = model.tvalues.get('const', np.nan)
        beta_mkt    = model.params.get('Mkt-RF', np.nan)
        beta_smb    = model.params.get('SMB',    np.nan)
        beta_hml    = model.params.get('HML',    np.nan)
        beta_rmw    = model.params.get('RMW',    np.nan)
        beta_cma    = model.params.get('CMA',    np.nan)
        r2          = model.rsquared
        r2_adj      = model.rsquared_adj
        pval        = model.f_pvalue

        # print in percent
        print(f"\n=== {new_model_name} vs. FF5 ===")
        print(model.summary())
        print(f"Monthly Alpha:    {alpha_dec*100:.3f}%")
        print(f"Alpha t-stat:     {alpha_t:.3f}")
        print(f"Market Beta:      {beta_mkt:.3f}")
        print("="*80)
```

```
        # store decimals for table
        regression_summary_list.append({
            "Model":           new_model_name,
            "Alpha (dec)":     alpha_dec,
            "Alpha t-stat":    alpha_t,
            "Market Beta":     beta_mkt,
            "SMB Beta":        beta_smb,
            "HML Beta":        beta_hml,
            "RMW Beta":        beta_rmw,
            "CMA Beta":        beta_cma,
            "R-squared":       r2,
            "Adj. R-squared":  r2_adj,
            "p-value":         pval
        })

    # ------------------------------------------------------------------
    # 4. EQUAL-WEIGHT STRATEGY
    # ------------------------------------------------------------------
    eq = (
        combined_df.loc[
            (pd.to_datetime(combined_df['Predicted_month']) >= reg_star
            (pd.to_datetime(combined_df['Predicted_month']) <= reg_end_
            ['Predicted_month', 'Equal_Weight_Return']
        ]
        .rename(columns={'Equal_Weight_Return':'EW_Return'})
    )
    eq['Predicted_month'] = pd.to_datetime(eq['Predicted_month'])

    eqm = (
        pd.merge(
            eq,
            df_factors[['Date'] + all_ff5_cols],
            left_on="Predicted_month", right_on="Date", how="inner")
          .drop(columns=["Date"])
    )
    eqm['Y'] = eqm['EW_Return'] - eqm['RF'] if subtract_rf else eqm['EW

    X_eq     = sm.add_constant(eqm[factors])
    y_eq     = eqm['Y']
    model_eq = sm.OLS(y_eq, X_eq, missing='drop').fit()

    alpha_eq_dec = model_eq.params.get('const', np.nan)
    alpha_eq_t   = model_eq.tvalues.get('const', np.nan)
    beta_eq      = model_eq.params.get('Mkt-RF', np.nan)
    beta_eq_smb  = model_eq.params.get('SMB',    np.nan)
    beta_eq_hml  = model_eq.params.get('HML',    np.nan)
    beta_eq_rmw  = model_eq.params.get('RMW',    np.nan)
    beta_eq_cma  = model_eq.params.get('CMA',    np.nan)
```

```
print("\n=== Equal-Weight vs. FF5 ===")
print(model_eq.summary())
print(f"Monthly Alpha:    {alpha_eq_dec*100:.3f}%")
print(f"Alpha t-stat:     {alpha_eq_t:.3f}")
print(f"Market Beta:      {beta_eq:.3f}")
print("="*80)

regression_summary_list.append({
    "Model":           "Equal-Weight",
    "Alpha (dec)":     alpha_eq_dec,
    "Alpha t-stat":    alpha_eq_t,
    "Market Beta":     beta_eq,
    "SMB Beta":        beta_eq_smb,
    "HML Beta":        beta_eq_hml,
    "RMW Beta":        beta_eq_rmw,
    "CMA Beta":        beta_eq_cma,
    "R-squared":       model_eq.rsquared,
    "Adj. R-squared":  model_eq.rsquared_adj,
    "p-value":         model_eq.f_pvalue
})

# ------------------------------------------------------------------
# 5. SUMMARY TABLE (with % conversions)
# ------------------------------------------------------------------
df_sum = pd.DataFrame(regression_summary_list)

# convert decimals to percent in new columns
df_sum['Alpha (%)']           = df_sum['Alpha (dec)'] * 100
df_sum['Ann. Alpha (%) comp'] = (1 + df_sum['Alpha (dec)'])**12 - 1
df_sum['Ann. Alpha (%) comp'] = df_sum['Ann. Alpha (%) comp'] * 100
df_sum['Ann. Alpha (%) lin']  = df_sum['Alpha (dec)'] * 12 * 100

# round
df_sum = (
    df_sum.rename(columns={
        'Alpha t-stat':   'Alpha t-stat',
        'Market Beta':    'Market Beta',
        'SMB Beta':       'SMB Beta',
        'HML Beta':       'HML Beta',
        'RMW Beta':       'RMW Beta',
        'CMA Beta':       'CMA Beta',
        'R-squared':      'R²',
        'Adj. R-squared': 'Adj. R²',
        'p-value':        'p-value'
    })
    .round({
        'Alpha (%)':           3,
```

```
        'Alpha t-stat':        3,
        'Market Beta':         3,
        'SMB Beta':            3,
        'HML Beta':            3,
        'RMW Beta':            3,
        'CMA Beta':            3,
        'R²':                  3,
        'Adj. R²':             3,
        'p-value':             6,
        'Ann. Alpha (%) comp': 3,
        'Ann. Alpha (%) lin':  3
    })
)

# ---------------- NEW, FRIENDLIER HEADLINE WITH DATE WINDOW ------
date_window = f"{reg_start_date.strftime('%b %Y')} – {reg_end_date.
print(f"\n=== Summary of All Models ({date_window}, in percent) ===
# -----------------------------------------------------------------

display(df_sum[[
    'Model',
    'Alpha (%)',
    'Alpha t-stat',
    'Market Beta',
    'SMB Beta',
    'HML Beta',
    'RMW Beta',
    'CMA Beta',
    'R²',
    'Adj. R²',
    'p-value',
    'Ann. Alpha (%) comp',
    'Ann. Alpha (%) lin'
]])
```

```
=== ML1: Random Forest vs. FF5 ===
                            OLS Regression Results
==============================================================================
Dep. Variable:                          Y   R-squared:
Model:                                OLS   Adj. R-squared:
Method:                     Least Squares   F-statistic:
Date:                    Sun, 30 Nov 2025   Prob (F-statistic):
Time:                            16:48:05   Log-Likelihood:
No. Observations:                     632   AIC:
Df Residuals:                         626   BIC:
Df Model:                               5
Covariance Type:                nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025
```

```
                                                                       
------------------------------------------------------------------------
const         -0.0020      0.001     -3.206      0.001      -0.003
Mkt-RF        -0.0138      0.015     -0.942      0.347      -0.042
SMB            0.2797      0.022     12.769      0.000       0.237
HML            0.1469      0.027      5.492      0.000       0.094
RMW            0.2842      0.029      9.899      0.000       0.228
CMA            0.3525      0.042      8.379      0.000       0.270
========================================================================
Omnibus:                       58.515   Durbin-Watson:
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
Skew:                           0.223   Prob(JB):
Kurtosis:                       6.231   Cond. No.
========================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors
Monthly Alpha:    -0.201%
Alpha t-stat:     -3.206
Market Beta:      -0.014
========================================================================

=== ML2: Gradient Boosting vs. FF5 ===
                            OLS Regression Results
========================================================================
Dep. Variable:                      Y   R-squared:
Model:                            OLS   Adj. R-squared:
Method:                 Least Squares   F-statistic:
Date:                Sun, 30 Nov 2025   Prob (F-statistic):
Time:                        16:48:05   Log-Likelihood:
No. Observations:                 632   AIC:
Df Residuals:                     626   BIC:
Df Model:                           5
Covariance Type:            nonrobust
========================================================================
                 coef    std err          t      P>|t|      [0.025
------------------------------------------------------------------------
const         -0.0026      0.000     -6.049      0.000      -0.003
Mkt-RF        -0.0049      0.010     -0.492      0.623      -0.024
SMB            0.2945      0.015     19.815      0.000       0.265
HML            0.1757      0.018      9.682      0.000       0.140
RMW            0.2857      0.019     14.663      0.000       0.247
CMA            0.2994      0.029     10.490      0.000       0.243
========================================================================
Omnibus:                       68.780   Durbin-Watson:
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
Skew:                           0.411   Prob(JB):
Kurtosis:                       6.140   Cond. No.
========================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors
Monthly Alpha:    -0.257%
Alpha t-stat:     -6.049
Market Beta:      -0.005
```

```
==============================================================================

=== Equal-Weight vs. FF5 ===
                            OLS Regression Results
==============================================================================
Dep. Variable:                      Y   R-squared:
Model:                            OLS   Adj. R-squared:
Method:                 Least Squares   F-statistic:
Date:                Sun, 30 Nov 2025   Prob (F-statistic):
Time:                        16:48:05   Log-Likelihood:
No. Observations:                 632   AIC:
Df Residuals:                     626   BIC:
Df Model:                           5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025
------------------------------------------------------------------------------
const         -0.0036      0.000    -31.138      0.000      -0.004
Mkt-RF         0.0045      0.003      1.650      0.099      -0.001
SMB            0.2530      0.004     61.815      0.000       0.245
HML            0.2428      0.005     48.571      0.000       0.233
RMW            0.2556      0.005     47.641      0.000       0.245
CMA            0.2543      0.008     32.344      0.000       0.239
==============================================================================
Omnibus:                       38.418   Durbin-Watson:
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
Skew:                          -0.633   Prob(JB):
Kurtosis:                       3.252   Cond. No.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors
Monthly Alpha:    -0.364%
Alpha t-stat:     -31.138
Market Beta:      0.005
==============================================================================

=== Summary of All Models (Apr 1972 – Nov 2024, in percent) ===
```

	Model	Alpha (%)	Alpha t-stat	Market Beta	SMB Beta	HML Beta	RMW Beta	CMA Beta	R^2	Adj. R^2
0	ML1: Random Forest	-0.201	-3.206	-0.014	0.280	0.147	0.284	0.353	0.479	0.475
1	ML2: Gradient	-0.257	-6.049	-0.005	0.294	0.176	0.286	0.299	0.666	0.664

```
import pandas as pd
import numpy as np
```

```
from IPython.display import display, HTML

print("\n=== PERFORMANCE METRICS ===")

def annual_sharpe(series):
    """
    Calculate the annual Sharpe ratio from a monthly return Series.
    - Annual return = (product(1 + monthly returns)) - 1
    - Annual volatility = std(monthly returns) * sqrt(12)
    - Sharpe = annual return / annual volatility (assuming risk-fre
    """
    annual_return = (1 + series).prod() - 1
    annual_vol = series.std() * np.sqrt(12)
    if annual_vol == 0:
        return np.nan
    return annual_return / annual_vol

# ------------------------------------------------------------------
# 1) Define the date range and filter combined_df accordingly.
# ------------------------------------------------------------------
start_date = pd.to_datetime("2000-01-01")
end_date   = pd.to_datetime("2024-12-31")

# Filter the merged DataFrame for the desired date range.
df_filtered = combined_df[(combined_df['Predicted_month'] >= start_
                          (combined_df['Predicted_month'] <= end_

# Create a "Year" column from Predicted_month.
df_filtered["Year"] = df_filtered["Predicted_month"].dt.year

# ------------------------------------------------------------------
# 2) Identify Strategy Columns for Regression Metrics:
#    - ML strategy columns: Expected to have names like "ML1: Randc
#    - Benchmark: "Benchmark Return"
#    - Equal Weight: "Equal_Weight_Return"
# ------------------------------------------------------------------
ml_strategy_cols = [col for col in df_filtered.columns
                    if col.startswith("ML") and "Allocated_Return"

bench_col = "Benchmark Return" if "Benchmark Return" in df_filtered
eq_col    = "Equal_Weight_Return" if "Equal_Weight_Return" in df_fi

print("Strategy columns used for annual Sharpe calculation:")
print("ML Strategy Columns:", ml_strategy_cols)
if bench_col:
    print("Benchmark Column:", bench_col)
if eq_col:
    print("Equal Weight Column:", eq_col)
```

```
# ------------------------------------------------------------------
# 3) Compute Annual Sharpe Ratios for Each Strategy
# ------------------------------------------------------------------
annual_sharpe_dict = {}

# Compute for each ML model column.
for col in ml_strategy_cols:
    annual_sharpe_dict[col] = df_filtered.groupby("Year")[col].appl

# Compute for benchmark (if available).
if bench_col is not None:
    annual_sharpe_dict[bench_col] = df_filtered.groupby("Year")[ben

# Compute for equal weight (if available).
if eq_col is not None:
    annual_sharpe_dict[eq_col] = df_filtered.groupby("Year")[eq_col

# ------------------------------------------------------------------
# 4) Combine the Results into One DataFrame and Round to 3 Decimals
# ------------------------------------------------------------------
annual_sharpe_table = pd.DataFrame(annual_sharpe_dict)
annual_sharpe_table = annual_sharpe_table.round(3)

# ------------------------------------------------------------------
# 5) Display the Annual Sharpe Ratios as a Neatly Formatted HTML Ta
# ------------------------------------------------------------------
display(HTML("<h3>Annual Sharpe Ratios by Year and Strategy</h3>" +
```

```
=== PERFORMANCE METRICS ===
Strategy columns used for annual Sharpe calculation:
ML Strategy Columns: ['ML1: Random Forest Allocated_Return', 'ML2: G
Equal Weight Column: Equal_Weight_Return
```

Annual Sharpe Ratios by Year and Strategy

Year	ML1: Random Forest Allocated_Return	ML2: Gradient Boosting Allocated_Return	Equal_Weight_Return
2000	1.229	2.618	2.423
2001	0.869	0.986	1.560
2002	1.984	1.861	1.991
2003	1.902	1.404	2.047

2004	-0.071	0.364	1.364
2005	0.572	0.345	0.496
2006	1.939	1.420	3.510
2007	-0.889	-2.005	-2.670
2008	2.937	2.452	2.310
2009	0.702	0.734	0.154
2010	1.197	1.487	0.878
2011	1.920	2.082	-0.062
2012	-0.089	0.209	1.427
2013	0.027	0.621	0.848
2014	-2.286	-2.211	-0.625
2015	-0.532	-1.527	-1.726
2016	2.023	2.045	2.084
2017	-1.202	-1.730	-1.360
2018	-1.955	-1.704	-1.617
2019	-1.367	-0.903	-0.631
2020	0.097	-0.574	-1.278
2021	2.630	2.437	1.848
2022	0.710	1.402	1.979
2023	0.111	0.164	-0.936

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import display, HTML

# ================================================================
# 1) Prepare Annual Return Data
# ================================================================

# Define the date range for annual analysis.
start_date = pd.to_datetime("2000-01-01")
end_date   = pd.to_datetime("2024-12-31")

# Filter the merged DataFrame (combined_df) for the desired date ra
```

```
# (combined_df comes from your earlier multi-model merging steps.)
df_annual = combined_df[(combined_df['Predicted_month'] >= start_da
                         (combined_df['Predicted_month'] <= end_da

# Create a "Year" column.
df_annual['Year'] = df_annual['Predicted_month'].dt.year

# ------------------------------------------------------------------
# Identify columns:
#   • ML strategy columns: they contain "Allocated_Return" (e.g., "
#   • Benchmark: assume the column is "Benchmark Return" (or use wh
#   • Equal Weight: assume "Equal_Weight_Return"
#   • Factor columns: using the global FACTORS (e.g., ['SMB', 'HML'
# ------------------------------------------------------------------
ml_cols = [col for col in df_annual.columns if ("Allocated_Return"
benchmark_col = "Benchmark Return" if "Benchmark Return" in df_annu
equal_weight_col = "Equal_Weight_Return" if "Equal_Weight_Return" i
factor_cols = [fac for fac in FACTORS if fac in df_annual.columns]

print("ML Strategy Columns:", ml_cols)
print("Benchmark Column:", benchmark_col)
print("Equal Weight Column:", equal_weight_col)
print("Factor Columns:", factor_cols)

# ------------------------------------------------------------------
# Function to compute annual return as the compounded return over t
# ------------------------------------------------------------------
def annual_return(group, col):
    """Compound return over the group: product(1 + r) - 1."""
    return (1 + group[col]).prod() - 1

# ------------------------------------------------------------------
# Compute annual returns for each strategy.
# We'll build a dictionary where keys are strategy names and values
annual_returns = {}

# For each ML model column.
for col in ml_cols:
    annual_returns[col] = df_annual.groupby("Year").apply(lambda gr

# For benchmark.
if benchmark_col is not None:
    annual_returns[benchmark_col] = df_annual.groupby("Year").apply

# For equal weight strategy.
if equal_weight_col is not None:
    annual_returns[equal_weight_col] = df_annual.groupby("Year").ap
```

```
# For each factor.
for fac in factor_cols:
    annual_returns[fac] = df_annual.groupby("Year").apply(lambda gr

# Combine the computed annual returns into one DataFrame.
annual_returns_df = pd.DataFrame(annual_returns)
annual_returns_df = annual_returns_df.round(3)

# Display the Annual Returns Table.
display(HTML("<h3>Annual Returns Table</h3>" + annual_returns_df.to

# ================================================================
# 2) Compute Excess Returns and Plot by Factor
# ================================================================

# For each factor, compute excess return for each ML model relative
# We define excess return as: (ML Annual Return) - (Factor Annual R
excess_returns = {}
for fac in factor_cols:
    # Create a DataFrame of excess returns for all ML models for fa
    excess_df = pd.DataFrame()
    for ml in ml_cols:
        excess_df[ml] = annual_returns_df[ml] - annual_returns_df[f
    excess_returns[fac] = excess_df

# Now plot excess returns by factor.
n_factors = len(factor_cols)
if n_factors > 0:
    # Create one subplot per factor.
    fig, axes = plt.subplots(1, n_factors, figsize=(6 * n_factors,
    if n_factors == 1:
        axes = [axes]
    for i, fac in enumerate(factor_cols):
        ax = axes[i]
        # Plot a line for each ML model excess return for this fact
        for ml in ml_cols:
            ax.plot(excess_returns[fac].index, excess_returns[fac][
                    marker='o', label=f'{ml} - {fac}')
        ax.set_title(f'Excess Return: ML - {fac}')
        ax.set_xlabel("Year")
        ax.set_ylabel("Excess Return")
        ax.grid(True, linestyle='--', alpha=0.7)
        ax.legend(fontsize='small')
    plt.tight_layout()
    plt.show()

# ================================================================
# 3) (Optional) Compute and Plot Annual Returns for Benchmark and E
```

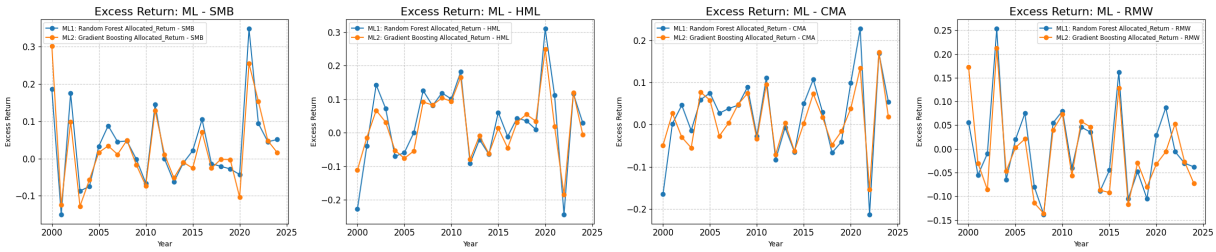
```
# ================================================================

# For convenience, let's display the annual returns for benchmark a
if benchmark_col is not None:
    print("\nAnnual Returns - Benchmark:")
    display(annual_returns_df[[benchmark_col]])
if equal_weight_col is not None:
    print("\nAnnual Returns - Equal Weight Strategy:")
    display(annual_returns_df[[equal_weight_col]])
```

```
ML Strategy Columns: ['ML1: Random Forest Allocated_Return', 'ML2: G
Benchmark Column: None
Equal Weight Column: Equal_Weight_Return
Factor Columns: ['SMB', 'HML', 'CMA', 'RMW']
/tmp/ipython-input-1377566255.py:53: DeprecationWarning: DataFrameGr
  annual_returns[col] = df_annual.groupby("Year").apply(lambda grp:
/tmp/ipython-input-1377566255.py:53: DeprecationWarning: DataFrameGr
  annual_returns[col] = df_annual.groupby("Year").apply(lambda grp:
/tmp/ipython-input-1377566255.py:61: DeprecationWarning: DataFrameGr
  annual_returns[equal_weight_col] = df_annual.groupby("Year").apply
/tmp/ipython-input-1377566255.py:65: DeprecationWarning: DataFrameGr
  annual_returns[fac] = df_annual.groupby("Year").apply(lambda grp:
/tmp/ipython-input-1377566255.py:65: DeprecationWarning: DataFrameGr
  annual_returns[fac] = df_annual.groupby("Year").apply(lambda grp:
/tmp/ipython-input-1377566255.py:65: DeprecationWarning: DataFrameGr
  annual_returns[fac] = df_annual.groupby("Year").apply(lambda grp:
/tmp/ipython-input-1377566255.py:65: DeprecationWarning: DataFrameGr
  annual_returns[fac] = df_annual.groupby("Year").apply(lambda grp:
```

Annual Returns Table

Year	ML1: Random Forest Allocated_Return	ML2: Gradient Boosting Allocated_Return	Equal_Weight_Return	SMB
2000	0.216	0.332	0.273	0.030
2001	0.112	0.137	0.178	0.262
2002	0.242	0.166	0.156	0.067
2003	0.106	0.065	0.044	0.193
2004	-0.003	0.015	0.035	0.072
2005	0.030	0.013	0.012	-0.003
2006	0.105	0.051	0.058	0.017
2007	-0.034	-0.068	-0.068	-0.079
2008	0.102	0.104	0.093	0.055

2009	0.073	0.058	0.011	0.075
2010	0.063	0.056	0.040	0.129
2011	0.100	0.084	-0.001	-0.045
2012	-0.004	0.008	0.028	-0.003
2013	0.001	0.012	0.015	0.064
2014	-0.081	-0.079	-0.022	-0.069
2015	-0.039	-0.086	-0.060	-0.060
2016	0.194	0.160	0.104	0.089
2017	-0.065	-0.077	-0.054	-0.051
2018	-0.070	-0.051	-0.044	-0.049
2019	-0.071	-0.046	-0.029	-0.043
2020	0.008	-0.053	-0.097	0.051
2021	0.332	0.239	0.139	-0.016
2022	0.079	0.138	0.169	-0.015
2023	0.008	0.011	-0.066	-0.037
2024	-0.017	-0.052	-0.040	-0.068



Annual Returns - Equal Weight Strategy:

Year	Equal_Weight_Return
2000	0.273
2001	0.178
2002	0.156
2003	0.044
2004	0.035
2005	0.012

2006	0.058
2007	-0.068
2008	0.093
2009	0.011
2010	0.040
2011	-0.001
2012	0.028
2013	0.015
2014	-0.022
2015	-0.060
2016	0.104
2017	-0.054
2018	-0.044
2019	-0.029
2020	-0.097
2021	0.139
2022	0.169
2023	-0.066
2024	-0.040

Return comparison by period

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import display

# ------------------------------------------------
# 0) (NEW) Choose a DATE RANGE to analyse
#     • Set start_date / end_date to datetime-like strings or pd.Ti
```

```
#     • Leave either (or both) as None to make it “open-ended”
# ------------------------------------------------
start_date = pd.to_datetime("2000-01-01")   # e.g. "2000-01-01"
end_date   = pd.to_datetime("2024-12-31")   # e.g. "2024-12-31"

# ------------------------------------------------
# 1) (NEW)  Slice the dataframe to that date range
#           and add a “Year” column
# ------------------------------------------------
combined_df['Predicted_month'] = pd.to_datetime(combined_df['Predic

mask = True
if start_date is not None:
    mask &= combined_df['Predicted_month'] >= start_date
if end_date is not None:
    mask &= combined_df['Predicted_month'] <= end_date

df_filtered = combined_df.loc[mask].copy()
df_filtered['Year'] = df_filtered['Predicted_month'].dt.year

# ------------------------------------------------
# 2) Define a Function to Calculate Annual Return
# ------------------------------------------------
def annual_return(series):
    """
    Compute the annual compounded return from a monthly return seri
    """
    return (1 + series).prod() - 1

# ------------------------------------------------
# 3) Compute Annual Returns for Each Strategy:
#    a) For each ML model's allocated return.
#    b) For Benchmark Return.
#    c) For Equal Weight Return.
# ------------------------------------------------
# a) For ML models: we assume these columns start with "ML" and con
ml_cols = [col for col in df_filtered.columns
           if col.startswith("ML") and "Allocated_Return" in col an
annual_returns_ml = {col: df_filtered.groupby("Year")[col].apply(an

# b) For Benchmark:
annual_return_bench = None
if "Benchmark Return" in df_filtered.columns:
    annual_return_bench = df_filtered.groupby("Year")["Benchmark Re

# c) For Equal Weight:
annual_return_eq = None
if "Equal_Weight_Return" in df_filtered.columns:
```

```
    annual_return_eq = df_filtered.groupby("Year")["Equal_Weight_Re

# ------------------------------------------------
# 4) Compute Annual Returns for Each Factor in FACTORS:
# ------------------------------------------------
annual_returns_factors = {}
for factor in FACTORS:
    if factor in df_filtered.columns:
        annual_returns_factors[factor] = df_filtered.groupby("Year"

# ------------------------------------------------
# 5) Compute Excess Returns for each ML model relative to each fact
#    Excess = (ML Model Annual Return) - (Factor Annual Return)
# ------------------------------------------------
excess_returns = {}
for ml_col, ml_series in annual_returns_ml.items():
    df_excess = pd.DataFrame(index=ml_series.index)
    for factor, factor_series in annual_returns_factors.items():
        df_excess[f"Excess ({ml_col} - {factor})"] = ml_series - fa
    excess_returns[ml_col] = df_excess

# ------------------------------------------------
# 6) Build a Summary Table of Annual Returns for all Strategies
# ------------------------------------------------
years = sorted(df_filtered['Year'].unique())
annual_summary = pd.DataFrame(index=years)

# Add ML model returns.
for ml_col, series in annual_returns_ml.items():
    annual_summary[ml_col] = series

# Add benchmark return if available.
if annual_return_bench is not None:
    annual_summary["Benchmark Return"] = annual_return_bench

# Add equal weight return if available.
if annual_return_eq is not None:
    annual_summary["Equal_Weight_Return"] = annual_return_eq

# ------------------------------------------------
# 7) Display the Summary Table and (optionally) Excess Returns for
# ------------------------------------------------
print("Annual Returns Summary:")
display(annual_summary.round(3))

# Optionally, display excess returns for the first ML model.
first_ml_col = ml_cols[0] if ml_cols else None
if first_ml_col is not None:
```

```
    print(f"\nExcess Returns for {first_ml_col}:")
    display(excess_returns[first_ml_col].round(3))

# ------------------------------------------------
# 8) Plot annual *excess* returns for each ML model
#     relative to the Equal-Weight benchmark
# ------------------------------------------------
# --- choose the baseline you want to subtract ---
baseline_series = annual_return_eq          # <-- equal-weight
baseline_label  = "Equal-Weight"

# guard-rail in case Equal-Weight isn't available
if baseline_series is None:
    raise ValueError("Equal-Weight returns not found; pick another

# --- build a DataFrame whose columns are the excess returns we'll
excess_plot_df = pd.concat(
    {ml_name.replace("Allocated_Return", "").strip(): ml_series - b
     for ml_name, ml_series in annual_returns_ml.items()},
    axis=1
)

# --- make the chart ----------------------------------------------
plt.figure(figsize=(11,6))

for col in excess_plot_df.columns:
    plt.plot(
        excess_plot_df.index,           # years on the x-axis
        excess_plot_df[col],            # excess returns
        marker="o",                     # a dot on each year
        label=col                       # legend label = strategy
    )

# horizontal line at 0 %
plt.axhline(0, linestyle="--", linewidth=1, alpha=0.7)

# title reflects the selected date range
title_start = start_date.strftime("%Y-%m-%d") if start_date is not
title_end   = end_date.strftime("%Y-%m-%d")   if end_date   is not

plt.title(f"Annual Excess Returns ({title_start} → {title_end})")
plt.xlabel("Year")
plt.ylabel(f"Excess Return vs {baseline_label}")
plt.legend(title="Strategy")
plt.tight_layout()
plt.show()
```

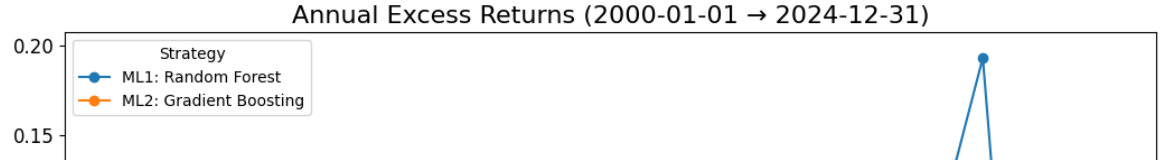
```
Annual Returns Summary:
```

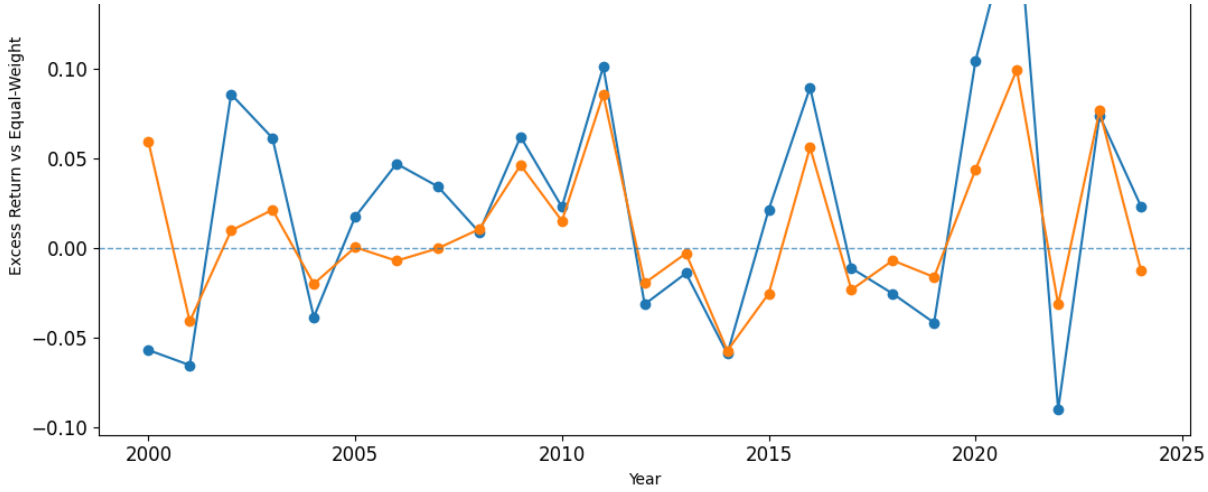
	ML1: Random Forest Allocated_Return	ML2: Gradient Boosting Allocated_Return	Equal_Weight_Return
2000	0.216	0.332	0.273
2001	0.112	0.137	0.178
2002	0.242	0.166	0.156
2003	0.106	0.065	0.044
2004	-0.003	0.015	0.035
2005	0.030	0.013	0.012
2006	0.105	0.051	0.058
2007	-0.034	-0.068	-0.068
2008	0.102	0.104	0.093
2009	0.073	0.058	0.011
2010	0.063	0.056	0.040
2011	0.100	0.084	-0.001
2012	-0.004	0.008	0.028
2013	0.001	0.012	0.015
2014	-0.081	-0.079	-0.022
2015	-0.039	-0.086	-0.060
2016	0.194	0.160	0.104
2017	-0.065	-0.077	-0.054
2018	-0.070	-0.051	-0.044
2019	-0.071	-0.046	-0.029
2020	0.008	-0.053	-0.097
2021	0.332	0.239	0.139
2022	0.079	0.138	0.169
2023	0.008	0.011	-0.066
2024	-0.017	-0.052	-0.040

Excess Returns for ML1: Random Forest Allocated_Return:

	Excess (ML1: Random Forest Allocated_Return	Excess (ML1: Random Forest Allocated_Return	Excess (ML1: Random Forest Allocated_Return	Exces Randor Allocated

Year	Allocated_Return - SMB)	Allocated_Return - HML)	Allocated_Return - CMA)	Allocate
2000	0.186	-0.227	-0.165	
2001	-0.150	-0.038	0.002	
2002	0.175	0.143	0.046	
2003	-0.087	0.073	-0.014	
2004	-0.075	-0.071	0.059	
2005	0.033	-0.059	0.075	
2006	0.089	0.001	0.027	
2007	0.045	0.126	0.038	
2008	0.047	0.081	0.046	
2009	-0.001	0.120	0.089	
2010	-0.066	0.101	-0.027	
2011	0.145	0.182	0.111	
2012	-0.001	-0.091	-0.083	
2013	-0.063	-0.020	-0.007	
2014	-0.012	-0.064	-0.065	
2015	0.021	0.061	0.050	
2016	0.104	-0.011	0.106	
2017	-0.014	0.044	0.028	
2018	-0.020	0.037	-0.067	
2019	-0.028	0.010	-0.041	
2020	-0.043	0.309	0.099	
2021	0.348	0.112	0.227	
2022	0.094	-0.243	-0.213	
2023	0.045	0.117	0.169	
2024	0.052	0.031	0.055	



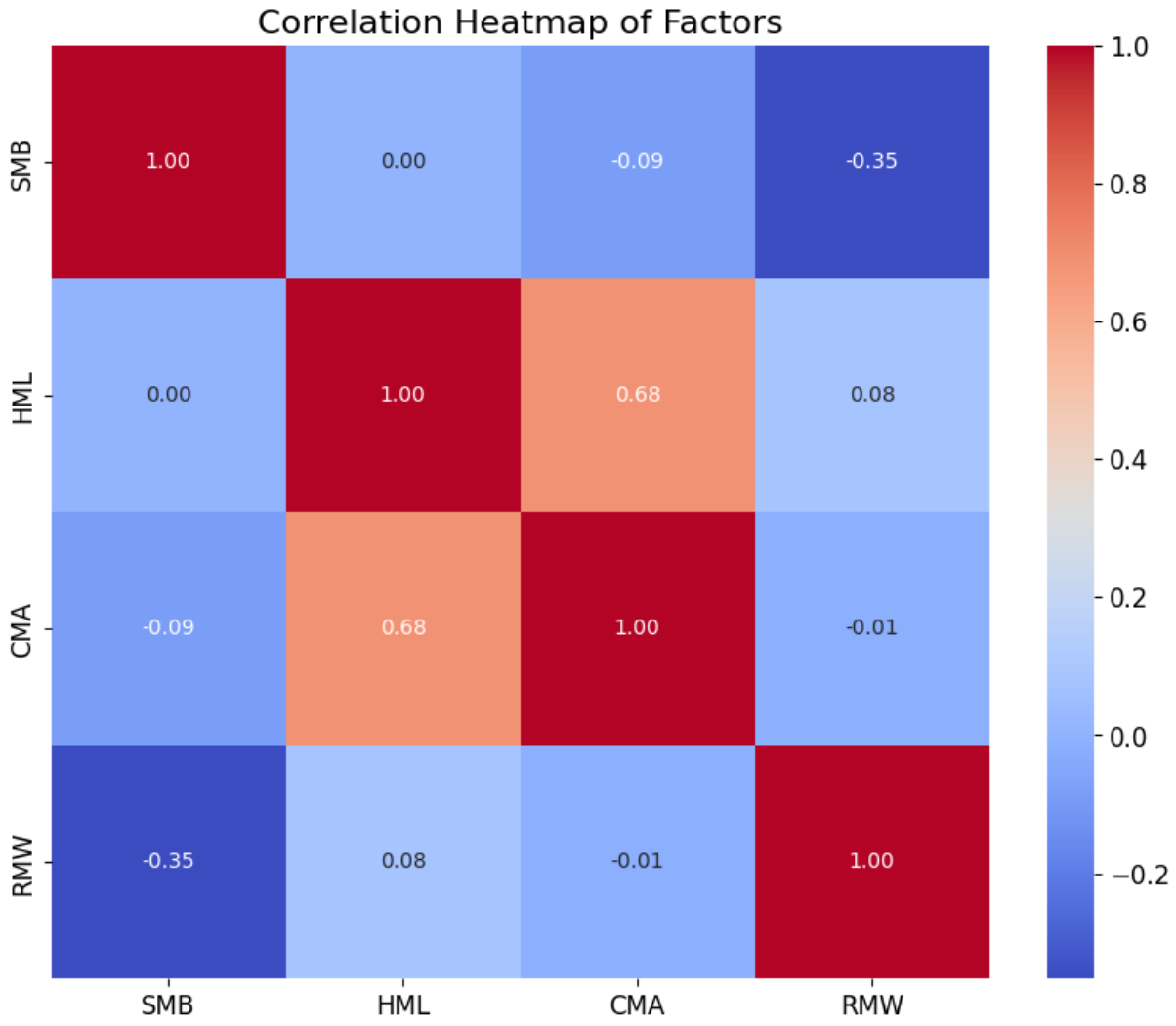


Corr Heat map & regiimi sharpet

```
import seaborn as sns
import matplotlib.pyplot as plt

df = xls_file.parse(SHEET_NAME)
df = df[["Date"] + FACTORS]

# Calculate correlations
correlation_matrix = df[FACTORS].corr()

# Show regular correlation table
print(correlation_matrix)

# Plot heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap="coolwa
plt.title("Correlation Heatmap of Factors")
plt.show()
```

```
           SMB       HML       CMA       RMW
SMB   1.000000  0.002825 -0.087740 -0.353244
HML   0.002825  1.000000  0.684483  0.083180
CMA  -0.087740  0.684483  1.000000 -0.012315
```

```
RMW -0.353244  0.083180 -0.012315  1.000000
```



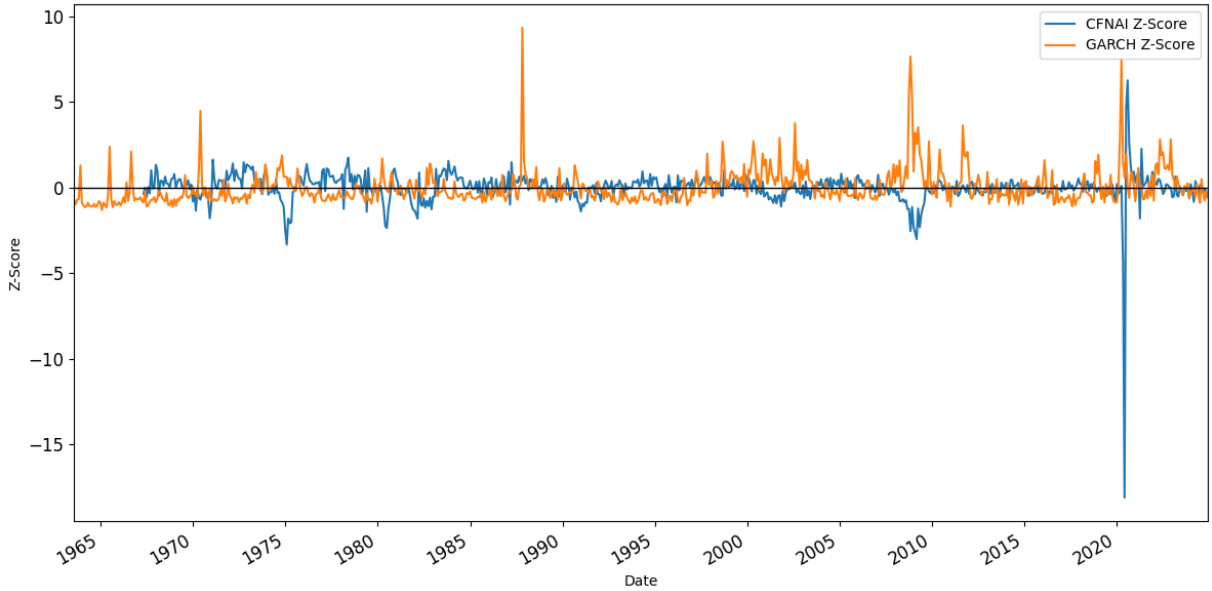
```
import os
import subprocess
import pandas as pd

# --- 1) Clone or pull your repo ---
repo_url  = "https://github.com/Elkkujou/Gradu.git"
repo_name = "Gradu"

if os.path.exists(repo_name):
    subprocess.run(["git", "-C", repo_name, "pull"], check=True)
else:
    subprocess.run(["git", "clone", repo_url], check=True)

# --- 2) Load the Excel sheet into data_ff5 ---
xlsx_path = os.path.join(repo_name, "THE_2ND_latest.xlsx")
xls_file  = pd.ExcelFile(xlsx_path)

SHEET_NAME  = "ajodata_FF5"
data_ff5    = xls_file.parse(SHEET_NAME)

# --- 3) Parse the date column (assumed to be the first column) ---
date_col = data_ff5.columns[0]
data_ff5[date_col] = pd.to_datetime(data_ff5[date_col])

# --- 4) Compute z-scores for your four features ---
FEATURES = ['CPI%','T10Y3M', 'CFNAI', 'GARCH']
z_cols   = [f + '_z' for f in FEATURES]

# Cross-sample z-score: (x − mean) / std
data_ff5[z_cols] = data_ff5[FEATURES].apply(lambda x: (x - x.mean()

# --- 5) Quick check ---
print(data_ff5.loc[:, FEATURES + z_cols].head(10))
```

```
      CPI%  T10Y3M  CFNAI     GARCH    CPI%_z  T10Y3M_z  CFNAI_z   G
0  0.32776    0.82    NaN  0.409987  0.030707 -0.466658      NaN -0.
1  0.26135    0.70    NaN  0.281066 -0.178386 -0.560180      NaN -0.
2  0.19550    0.69    NaN  0.431188 -0.385715 -0.567974      NaN -0.
3 -0.09756    0.70    NaN  0.444504 -1.308417 -0.560180      NaN -0.
4  0.09766    0.56    NaN  1.526543 -0.693765 -0.669289      NaN  1.
5  0.09756    0.62    NaN  0.316026 -0.694080 -0.622528      NaN -0.
6  0.32489    0.63    NaN  0.250391  0.021670 -0.614735      NaN -1.
7  0.19430    0.65    NaN  0.183620 -0.389493 -0.599148      NaN -1.
8 -0.09696    0.69    NaN  0.217851 -1.306528 -0.567974      NaN -1.
9  0.09706    0.75    NaN  0.326239 -0.695654 -0.521213      NaN -0.
```

```
import os
```

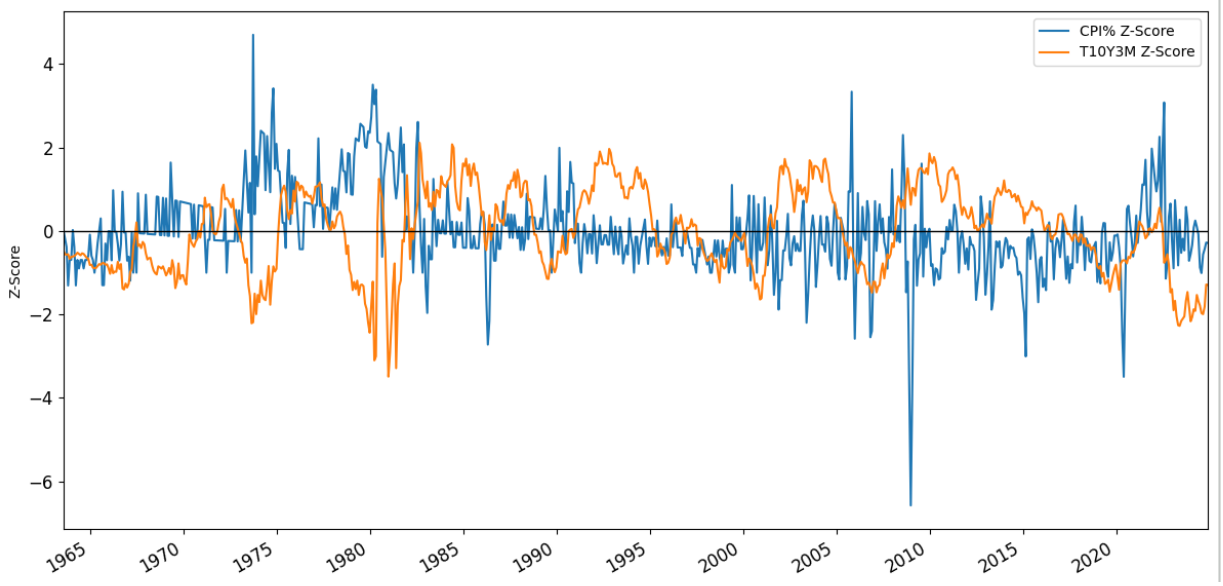
```
import subprocess
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

# 1) Clone or pull your repo
repo_url, repo_name = "https://github.com/Elkkujou/Gradu.git", "Gra
if os.path.exists(repo_name):
    subprocess.run(["git", "-C", repo_name, "pull"], check=True)
else:
    subprocess.run(["git", "clone", repo_url], check=True)

# 2) Load the Excel sheet
xlsx_path = os.path.join(repo_name, "THE_2ND_latest.xlsx")
data_ff5  = pd.read_excel(xlsx_path, sheet_name="ajodata_FF5")

# 3) Parse the date column
date_col = data_ff5.columns[0]
data_ff5[date_col] = pd.to_datetime(data_ff5[date_col])
data_ff5.set_index(date_col, inplace=True)
data_ff5.sort_index(inplace=True)

# 4) Compute z-scores for CFNAI & GARCH
FEATURES = ["CFNAI", "GARCH"]
for feat in FEATURES:
    data_ff5[f"{feat}_z"] = (data_ff5[feat] - data_ff5[feat].mean()

# 5) Plot
fig, ax = plt.subplots(figsize=(12, 6))
for feat in FEATURES:
    ax.plot(data_ff5.index, data_ff5[f"{feat}_z"], label=f"{feat} Z

ax.axhline(0, color="black", lw=1)

ax.set_xlabel("Date")
ax.set_ylabel("Z-Score")
ax.legend()

# 6) Fix x-axis from first to last date, no padding
x0, x1 = data_ff5.index.min(), data_ff5.index.max()
ax.set_xlim(x0, x1)
ax.margins(x=0)

# 7) Tick every 5 years, label “YYYY”
ax.xaxis.set_major_locator(mdates.YearLocator(5))
ax.xaxis.set_major_formatter(mdates.DateFormatter("%Y"))

fig.autofmt_xdate()
```

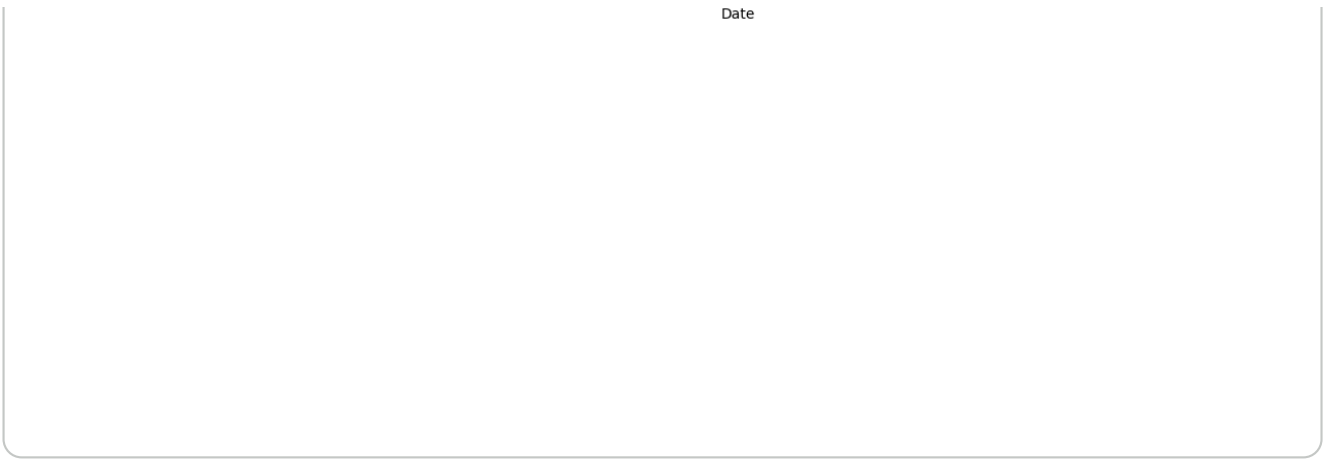
```
plt.tight_layout()
plt.show()
```




```
import os
import subprocess
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

# 1) Clone or pull your repo
repo_url, repo_name = "https://github.com/Elkkujou/Gradu.git", "Gra
if os.path.exists(repo_name):
    subprocess.run(["git", "-C", repo_name, "pull"], check=True)
else:
    subprocess.run(["git", "clone", repo_url], check=True)

# 2) Load the Excel sheet
xlsx_path = os.path.join(repo_name, "THE_2ND_latest.xlsx")
data_ff5  = pd.read_excel(xlsx_path, sheet_name="ajodata_FF5")

# 3) Parse the date column
```

```
date_col = data_ff5.columns[0]
data_ff5[date_col] = pd.to_datetime(data_ff5[date_col])
data_ff5.set_index(date_col, inplace=True)
data_ff5.sort_index(inplace=True)

# 4) Compute z-scores for CPI% & T10Y3M
FEATURES = ["CPI%", "T10Y3M"]
for feat in FEATURES:
    data_ff5[f"{feat}_z"] = (data_ff5[feat] - data_ff5[feat].mean()

# 5) Plot
fig, ax = plt.subplots(figsize=(12, 6))
for feat in FEATURES:
    ax.plot(data_ff5.index, data_ff5[f"{feat}_z"], label=f"{feat} Z

ax.axhline(0, color="black", lw=1)

ax.set_xlabel("Date")
ax.set_ylabel("Z-Score")
ax.legend()

# 6) Fix x-axis from first to last date, no padding
x0, x1 = data_ff5.index.min(), data_ff5.index.max()
ax.set_xlim(x0, x1)
ax.margins(x=0)

# 7) Tick every 5 years, label “YYYY”
ax.xaxis.set_major_locator(mdates.YearLocator(5))
ax.xaxis.set_major_formatter(mdates.DateFormatter("%Y"))

fig.autofmt_xdate()
plt.tight_layout()
plt.show()
```






```
import numpy as np
import pandas as pd
from IPython.display import HTML, display

# 1) Define your features and factors

ZCOLS    = [f + '_z' for f in FEATURES]
FACTORS  = ['SMB', 'HML', 'CMA', 'RMW']

# 2) Prepare an empty DataFrame to hold Sharpe ratios
cols = []
for feat in FEATURES:
    cols += [f"{feat} > 0 SR", f"{feat} < 0 SR"]
sharpe_df = pd.DataFrame(index=FACTORS, columns=cols, dtype=float)

# 3) Compute annualized Sharpe = √12 * mean(return) / std(return)
for feat, zcol in zip(FEATURES, ZCOLS):
    for fac in FACTORS:
        mask_pos = data_ff5[zcol] > 0
        mask_neg = ~mask_pos

        r_pos = data_ff5.loc[mask_pos, fac]
        r_neg = data_ff5.loc[mask_neg, fac]

        # avoid division by zero
        sr_pos = np.sqrt(12) * r_pos.mean() / r_pos.std() if r_pos.
        sr_neg = np.sqrt(12) * r_neg.mean() / r_neg.std() if r_neg.

        sharpe_df.loc[fac, f"{feat} > 0 SR"] = sr_pos
        sharpe_df.loc[fac, f"{feat} < 0 SR"] = sr_neg

# 4) Round
sharpe_df = sharpe_df.round(3)

# 5) Convert to HTML and display in-notebook
html = sharpe_df.to_html(
```

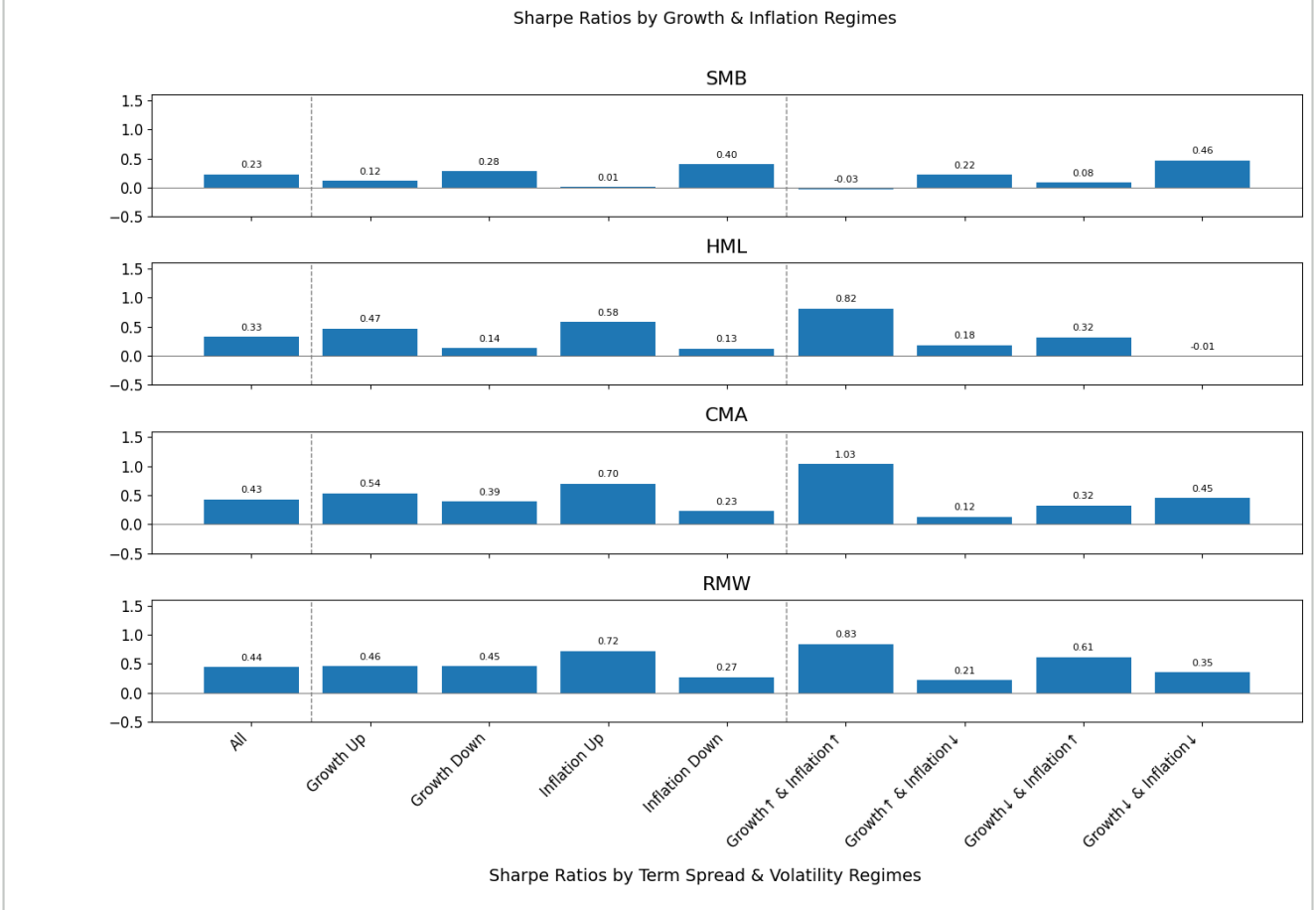
```
    border=1,
    classes="dataframe",
    escape=False
)
display(HTML(html))
```

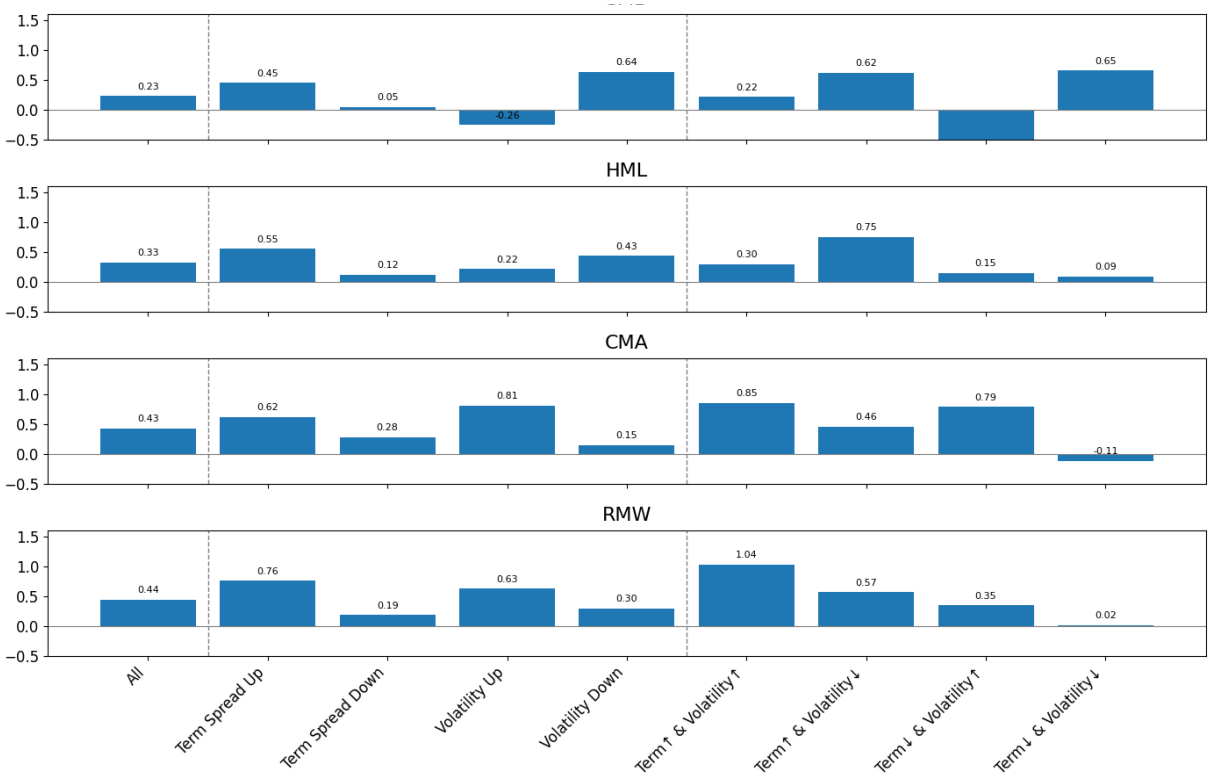
	CPI% > 0 SR	CPI% < 0 SR	T10Y3M > 0 SR	T10Y3M < 0 SR
SMB	0.007	0.396	0.454	0.050
HML	0.583	0.128	0.552	0.117
CMA	0.698	0.225	0.617	0.284
RMW	0.717	0.271	0.759	0.189

```
import os
import subprocess
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# 1) (Optional) Clone or pull your repo
repo_url, repo_name = "https://github.com/Elkkujou/Gradu.git", "Gra
if os.path.exists(repo_name):
    subprocess.run(["git", "-C", repo_name, "pull"], check=True)
else:
    subprocess.run(["git", "clone", repo_url], check=True)

# 2) Load & prepare the data
xlsx_path = os.path.join(repo_name, "THE_2ND_latest.xlsx")
data = pd.read_excel(
    xlsx_path,
    sheet_name="ajodata_FF5",
    parse_dates=[0]
)
# set the date column as a proper DateTimeIndex and sort
data.set_index(data.columns[0], inplace=True)
data.sort_index(inplace=True)

# 3) Define your four macro “features” and the FF5 factor returns
FEATURES = ["CPI%", "T10Y3M", "CFNAI", "GARCH"]
FACTORS  = ["SMB", "HML", "CMA", "RMW"]

# 4) Compute z-scores for each feature
for feat in FEATURES:
    data[f"{feat}_z"] = (data[feat] - data[feat].mean()) / data[fea
```

```
# 5) Helper to compute annualized Sharpe by regime
def compute_sharpe(regimes):
    sharpe = pd.DataFrame(index=FACTORS, columns=regimes.keys(), dt
    for fac in FACTORS:
        for name, mask in regimes.items():
            ret = data.loc[mask, fac]
            sharpe.loc[fac, name] = (
                np.sqrt(12) * ret.mean() / ret.std()
                if ret.std() != 0 else np.nan
            )
    return sharpe.round(3)

# 6) Helper to plot a 4×1 bar-chart grid
def plot_grid(sharpe_df, title):
    regimes = sharpe_df.columns.tolist()
    fig, axes = plt.subplots(len(FACTORS), 1, figsize=(14, 10), sha
    for ax, fac in zip(axes, FACTORS):
        vals = sharpe_df.loc[fac].values.astype(float)
        bars = ax.bar(regimes, vals)
        ax.axhline(0, color="gray", linewidth=0.8)
        # annotate heights
        for b in bars:
            h = b.get_height()
            ax.annotate(f"{h:.2f}",
                        xy=(b.get_x() + b.get_width()/2, h),
                        xytext=(0, 4), textcoords="offset points",
                        ha="center", va="bottom", fontsize=8)
        ax.set_ylim(-0.5, 1.6)
        ax.set_yticks(np.arange(-0.5, 1.6, 0.5))
        ax.set_title(fac, pad=8)
        # optional separators
        for sep in (1, 5):
            if sep < len(regimes):
                ax.axvline(sep - 0.5, color="gray", linestyle="--",
    axes[-1].set_xticks(range(len(regimes)))
    axes[-1].set_xticklabels(regimes, rotation=45, ha="right")
    plt.suptitle(title, y=0.98, fontsize=14)
    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

# 7) Build & plot regimes

# a) Growth (CFNAI) & Inflation (CPI%)
mask_all = pd.Series(True, index=data.index)
regimes1 = {
    "All":                          mask_all,
    "Growth Up":                    data["CFNAI_z"] > 0,
    "Growth Down":                  data["CFNAI_z"] <= 0,
```

```
    "Inflation Up":                data["CPI%_z"] > 0,
    "Inflation Down":              data["CPI%_z"] <= 0,
    "Growth↑ & Inflation↑":        (data["CFNAI_z"] > 0) & (data["CP
    "Growth↑ & Inflation↓":        (data["CFNAI_z"] > 0) & (data["CP
    "Growth↓ & Inflation↑":        (data["CFNAI_z"] <= 0) & (data["C
    "Growth↓ & Inflation↓":        (data["CFNAI_z"] <= 0) & (data["C
}
sh1 = compute_sharpe(regimes1)
plot_grid(sh1, "Sharpe Ratios by Growth & Inflation Regimes")

# b) Term Spread (T10Y3M) & Volatility (GARCH)
regimes2 = {
    "All":                         mask_all,
    "Term Spread Up":              data["T10Y3M_z"] > 0,
    "Term Spread Down":            data["T10Y3M_z"] <= 0,
    "Volatility Up":               data["GARCH_z"] > 0,
    "Volatility Down":             data["GARCH_z"] <= 0,
    "Term↑ & Volatility↑":         (data["T10Y3M_z"] > 0) & (data["G
    "Term↑ & Volatility↓":         (data["T10Y3M_z"] > 0) & (data["G
    "Term↓ & Volatility↑":         (data["T10Y3M_z"] <= 0) & (data["
    "Term↓ & Volatility↓":         (data["T10Y3M_z"] <= 0) & (data["
}
sh2 = compute_sharpe(regimes2)
plot_grid(sh2, "Sharpe Ratios by Term Spread & Volatility Regimes")
```

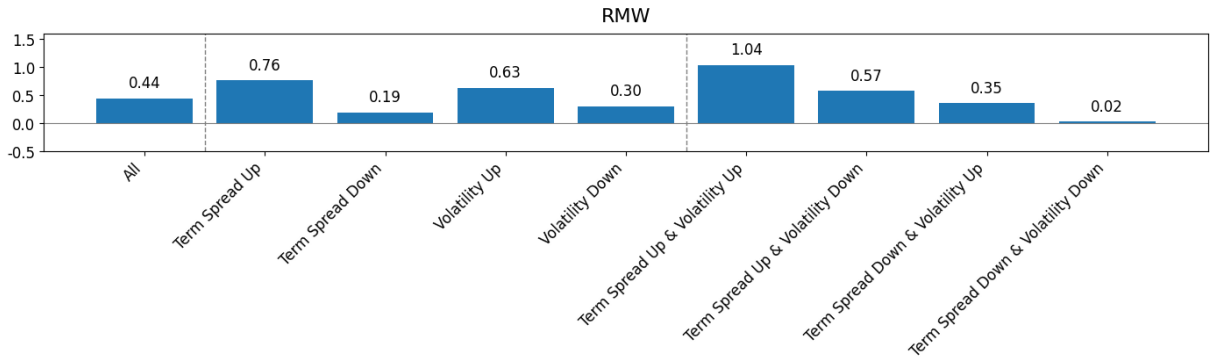



HML
CMA
RMW
0.23
0.45
0.05
-0.26
0.64
0.22
0.62
0.65
0.33
0.55
0.12
0.22
0.43
0.30
0.75
0.15
0.09
0.43
0.62
0.28
0.81
0.15
0.85
0.46
0.79
-0.11
0.44
0.76
0.19
0.63
0.30
1.04
0.57
0.35
0.02
All
Term Spread Up
Term Spread Down
Volatility Up
Volatility Down
Term↑ & Volatility↑
Term↑ & Volatility↓
Term↓ & Volatility↑
Term↓ & Volatility↓

```
import os
import subprocess
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# ——— Even larger global font tweaks ————————————————————
plt.rcParams['axes.titlesize']    = 16   # subplot title
plt.rcParams['xtick.labelsize']   = 12   # x-axis tick labels
plt.rcParams['ytick.labelsize']   = 12   # y-axis tick labels

# 1) Clone or pull your repo
repo_url, repo_name = "https://github.com/Elkkujou/Gradu.git", "Grad
if os.path.exists(repo_name):
    subprocess.run(["git", "-C", repo_name, "pull"], check=True)
else:
    subprocess.run(["git", "clone", repo_url], check=True)

# 2) Load & prepare the data
xlsx_path = os.path.join(repo_name, "THE_2ND_latest.xlsx")
data = pd.read_excel(xlsx_path, sheet_name="ajodata_FF5", parse_date
data.set_index(data.columns[0], inplace=True)
data.sort_index(inplace=True)

# 3) Define features & FF5 factors
FEATURES = ["CPI%", "T10Y3M", "CFNAI", "GARCH"]
FACTORS  = ["SMB", "HML", "CMA", "RMW"]

# 4) Compute z-scores
for feat in FEATURES:
    data[f"{feat}_z"] = (data[feat] - data[feat].mean()) / data[feat

# 5) Annualized Sharpe helper
def compute_sharpe(regimes):
    sharpe = pd.DataFrame(index=FACTORS, columns=regimes.keys(), dty
    for fac in FACTORS:
        for name, mask in regimes.items():
            ret = data.loc[mask, fac]
            sharpe.loc[fac, name] = (
                np.sqrt(12) * ret.mean() / ret.std()
                if ret.std() != 0 else np.nan
            )
    return sharpe.round(3)

# 6) Plot grid with larger annotations
def plot_grid(sharpe_df):
    regimes = sharpe_df.columns.tolist()
    fig, axes = plt.subplots(len(FACTORS), 1, figsize=(14, 10), shar
```

```
        for ax, fac in zip(axes, FACTORS):
            vals = sharpe_df.loc[fac].values.astype(float)
            bars = ax.bar(regimes, vals)
            ax.axhline(0, color="gray", linewidth=0.8)

            # annotate bars with fontsize=12
            for b in bars:
                h = b.get_height()
                ax.annotate(f"{h:.2f}",
                            xy=(b.get_x() + b.get_width()/2, h),
                            xytext=(0, 6),               # a little more
                            textcoords="offset points",
                            ha="center", va="bottom",
                            fontsize=12)

            # separators
            for sep in (1, 5):
                if sep < len(regimes):
                    ax.axvline(sep - 0.5, color="gray", linestyle="--",

            ax.set_ylim(-0.5, 1.6)
            yticks = np.arange(-0.5, 1.6, 0.5)
            ax.set_yticks(yticks)
            ax.set_yticklabels([f"{y:.1f}" for y in yticks])

            ax.set_title(fac, pad=10)  # titlesize pulled from rcParams

        axes[-1].set_xticks(range(len(regimes)))
        axes[-1].set_xticklabels(regimes, rotation=45, ha="right")
        plt.tight_layout()
        plt.show()

    # 7a) Growth & Inflation regimes
    mask_all = pd.Series(True, index=data.index)
    regimes1 = {
        "All":                          mask_all,
        "Growth Up":                    data["CFNAI_z"] > 0,
        "Growth Down":                  data["CFNAI_z"] <= 0,
        "Inflation Up":                 data["CPI%_z"] > 0,
        "Inflation Down":               data["CPI%_z"] <= 0,
        "Growth Up & Inflation Up":     (data["CFNAI_z"] > 0) & (data["C
        "Growth Up & Inflation Down":   (data["CFNAI_z"] > 0) & (data["C
        "Growth Down & Inflation Up":   (data["CFNAI_z"] <= 0) & (data["
        "Growth Down & Inflation Down": (data["CFNAI_z"] <= 0) & (data["
    }
    sh1 = compute_sharpe(regimes1)
    plot_grid(sh1)
```

```
# (b) Term Spread & Volatility regimes
regimes2 = {
    "All":                                  mask_all,
    "Term Spread Up":                       data["T10Y3M_z"] > 0,
    "Term Spread Down":                     data["T10Y3M_z"] <= 0,
    "Volatility Up":                        data["GARCH_z"] > 0,
    "Volatility Down":                      data["GARCH_z"] <= 0,
    "Term Spread Up & Volatility Up":    (data["T10Y3M_z"] > 0) & (d
    "Term Spread Up & Volatility Down":  (data["T10Y3M_z"] > 0) & (d
    "Term Spread Down & Volatility Up":  (data["T10Y3M_z"] <= 0) & (
    "Term Spread Down & Volatility Down":(data["T10Y3M_z"] <= 0) & (
}
sh2 = compute_sharpe(regimes2)
plot_grid(sh2)
```




```
#!/usr/bin/env python3
import os
import subprocess
import pandas as pd
import itertools

# 1) Clone or pull your repo
repo_url, repo_name = "https://github.com/Elkkujou/Gradu.git", "Gra
if os.path.exists(repo_name):
```

```
    subprocess.run(["git", "-C", repo_name, "pull"], check=True)
else:
    subprocess.run(["git", "clone", repo_url], check=True)

# 2) Load the Excel sheet
xlsx_path = os.path.join(repo_name, "THE_2ND_latest.xlsx")
data_ff5  = pd.read_excel(xlsx_path, sheet_name="ajodata_FF5")

# 3) Parse the date column (assumed to be the first one), set as in
date_col = data_ff5.columns[0]
data_ff5[date_col] = pd.to_datetime(data_ff5[date_col])
data_ff5.set_index(date_col, inplace=True)
data_ff5.sort_index(inplace=True)

# 4) Define your features & factors
FEATURES = ['CPI%', 'T10Y3M', 'CFNAI', 'GARCH']
FACTORS  = ['SMB', 'HML', 'CMA', 'RMW']

# 5) Compute z-scores for each feature
for feat in FEATURES:
    zcol = feat + "_z"
    data_ff5[zcol] = (data_ff5[feat] - data_ff5[feat].mean()) / dat

# 6) Build all unique factor-pairs and prepare the output table
pairs = list(itertools.combinations(FACTORS, 2))
row_index = [f"{i} – {j}" for i, j in pairs]

col_index = pd.MultiIndex.from_product(
    [FEATURES, ["> 0", "≤ 0"]],
    names=["Regime", "Sign"]
)

corr_table = pd.DataFrame(index=row_index, columns=col_index, dtype

# 7) Fill in the regime-conditional correlations
for feat in FEATURES:
    zcol = feat + "_z"
    mask_pos = data_ff5[zcol] > 0
    mask_neg = ~mask_pos

    for i, j in pairs:
        corr_pos = data_ff5.loc[mask_pos, i].corr(data_ff5.loc[mask
        corr_neg = data_ff5.loc[mask_neg, i].corr(data_ff5.loc[mask

        corr_table.loc[f"{i} – {j}", (feat, "> 0")] = corr_pos
        corr_table.loc[f"{i} – {j}", (feat, "≤ 0")] = corr_neg

# 8) Round to three decimals and display
```

```
corr_table = corr_table.round(3)
print(corr_table)
```

```
Regime      CPI%        T10Y3M         CFNAI         GARCH
Sign         > 0    ≤ 0    > 0    ≤ 0    > 0    ≤ 0    > 0    ≤ 0
SMB – HML -0.036  0.042  0.010 -0.007 -0.081  0.089 -0.002  0.006
SMB – CMA -0.062 -0.103 -0.038 -0.123 -0.081 -0.092 -0.067 -0.084
SMB – RMW -0.293 -0.392 -0.308 -0.389 -0.451 -0.256 -0.376 -0.309
HML – CMA  0.758  0.617  0.674  0.694  0.709  0.662  0.655  0.735
HML – RMW -0.024  0.153 -0.091  0.204  0.069  0.098  0.285 -0.239
CMA – RMW -0.105  0.046 -0.079  0.027 -0.057  0.029  0.167 -0.299
```

```
from IPython.display import HTML, display

html = """
<style>
  table {
    width: auto;               /* only as wide as it needs to be */
    table-layout: auto;        /* allow columns to size tightly */
    border-collapse: collapse;
    font-family: Arial, sans-serif;
    font-size: 10px;           /* smaller font */
    margin: 0;                 /* remove extra breathing outside */
  }
  th, td {
    padding: 4px 6px;          /* much less padding */
    text-align: center;
    line-height: 1.2;          /* tighter line-height */
  }
  thead tr:first-child th {
    font-weight: bold;
    border-bottom: 2px solid #333;
  }
  thead tr:nth-child(2) th {
    font-weight: bold;
    border-bottom: 1px solid #666;
  }
  tbody tr + tr td {
    border-top: 1px solid #e0e0e0;
  }
  tbody th {
    font-weight: bold;
    text-align: left;
    padding-left: 8px;         /* less indent */
  }
</style>

<table>
```

```
      <thead>
        <tr>
          <th rowspan="2"></th>
          <th colspan="2">CPI%</th>
          <th colspan="2">T10Y3M</th>
          <th colspan="2">CFNAI</th>
          <th colspan="2">GARCH</th>
        </tr>
        <tr>
          <th>&gt; 0</th><th>≤ 0</th>
          <th>&gt; 0</th><th>≤ 0</th>
          <th>&gt; 0</th><th>≤ 0</th>
          <th>&gt; 0</th><th>≤ 0</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <th>SMB – HML</th>
          <td>−0.036</td><td> 0.042</td>
          <td> 0.010</td><td>−0.007</td>
          <td>−0.081</td><td> 0.089</td>
          <td>−0.002</td><td> 0.006</td>
        </tr>
        <tr>
          <th>SMB – CMA</th>
          <td>−0.062</td><td>−0.103</td>
          <td>−0.038</td><td>−0.123</td>
          <td>−0.081</td><td>−0.092</td>
          <td>−0.067</td><td>−0.084</td>
        </tr>
        <tr>
          <th>SMB – RMW</th>
          <td>−0.293</td><td>−0.392</td>
          <td>−0.308</td><td>−0.389</td>
          <td>−0.451</td><td>−0.256</td>
          <td>−0.376</td><td>−0.309</td>
        </tr>
        <tr>
          <th>HML – CMA</th>
          <td> 0.758</td><td> 0.617</td>
          <td> 0.674</td><td> 0.694</td>
          <td> 0.709</td><td> 0.662</td>
          <td> 0.655</td><td> 0.735</td>
        </tr>
        <tr>
          <th>HML – RMW</th>
          <td>−0.024</td><td> 0.153</td>
          <td>−0.091</td><td> 0.204</td>
```

```
      <td> 0.069</td><td> 0.098</td>
      <td> 0.285</td><td>-0.239</td>
    </tr>
    <tr>
      <th>CMA - RMW</th>
      <td>-0.105</td><td> 0.046</td>
      <td>-0.079</td><td> 0.027</td>
      <td>-0.057</td><td> 0.029</td>
      <td> 0.167</td><td>-0.299</td>
    </tr>
  </tbody>
</table>
"""
display(HTML(html))
```

	CPI%		T10Y3M		CFNAI		GARCH	
	> 0	≤ 0	> 0	≤ 0	> 0	≤ 0	> 0	≤ 0
SMB − HML	−0.036	0.042	0.010	−0.007	−0.081	0.089	−0.002	0.006
SMB − CMA	−0.062	−0.103	−0.038	−0.123	−0.081	−0.092	−0.067	−0.084
SMB − RMW	−0.293	−0.392	−0.308	−0.389	−0.451	−0.256	−0.376	−0.309
HML − CMA	0.758	0.617	0.674	0.694	0.709	0.662	0.655	0.735
HML − RMW	−0.024	0.153	−0.091	0.204	0.069	0.098	0.285	−0.239
CMA − RMW	−0.105	0.046	−0.079	0.027	−0.057	0.029	0.167	−0.299

```
from IPython.display import HTML, display

html = """
<style>
  table {
    border-collapse: collapse;
    font-family: Arial, sans-serif;
    font-size: 12px;
    margin: 10px 0;
  }
  th, td {
    padding: 6px 10px;
    text-align: center;
  }
  /* Outer border only on top of header and bottom of table */
  thead th {
    font-weight: bold;
    border-bottom: 1px solid #999;
  }
  /* horizontal lines between body rows */
  tbody tr {
    border-top: 1px solid #ddd;
```

```
    }
    /* no vertical borders */
    th, td {
      border: none;
    }
    /* left-align the row labels */
    tbody th {
      font-weight: bold;
      text-align: left;
      padding-left: 12px;
    }
    /* bottom border after last row */
    tbody tr:last-child {
      border-bottom: 1px solid #999;
    }
  </style>
  <table>
    <thead>
      <tr>
        <th></th>
        <th>SMB</th>
        <th>HML</th>
        <th>CMA</th>
        <th>RMW</th>
      </tr>
    </thead>
    <tbody>
      <tr><th>SMB</th><td>1.00</td><td>0.00</td><td>−0.09</td><td>−0.
      <tr><th>HML</th><td>0.00</td><td>1.00</td><td>0.68</td><td>0.08
      <tr><th>CMA</th><td>−0.09</td><td>0.68</td><td>1.00</td><td>−0.
      <tr><th>RMW</th><td>−0.35</td><td>0.08</td><td>−0.01</td><td>1.
    </tbody>
  </table>
  """
  display(HTML(html))
```

	SMB	HML	CMA	RMW
SMB	1.00	0.00	−0.09	−0.35
HML	0.00	1.00	0.68	0.08
CMA	−0.09	0.68	1.00	−0.01
RMW	−0.35	0.08	−0.01	1.00

Double-click (or enter) to edit

```
import os
import subprocess
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# 1) Clone or pull your repo
repo_url, repo_name = "https://github.com/Elkkujou/Gradu.git", "Gra
if os.path.isdir(repo_name):
    subprocess.run(["git", "-C", repo_name, "pull"], check=True)
else:
    subprocess.run(["git", "clone", repo_url], check=True)

# 2) Load & prepare the data
xlsx_path = os.path.join(repo_name, "THE_2ND_latest.xlsx")
data = pd.read_excel(
    xlsx_path,
    sheet_name="ajodata_FF5",
    parse_dates=[0],
)
data.set_index(data.columns[0], inplace=True)
data.sort_index(inplace=True)

# 3) Define macro “features” and FF-5 factors
FEATURES = ["CPI%", "T10Y3M", "CFNAI", "GARCH"]
FACTORS  = ["SMB", "HML", "CMA", "RMW"]

# 4) Compute z-scores for each feature
for feat in FEATURES:
    data[f"{feat}_z"] = (data[feat] - data[feat].mean()) / data[fea

# 5) Compute annualized Sharpe ratios by regime for each feature
#    Sharpe = √12 * mean(r) / std(r)
sharpe = pd.DataFrame(
    index=FACTORS,
    columns=pd.MultiIndex.from_product([FEATURES, ["> 0", "≤ 0"]]),
    dtype=float,
)

for feat in FEATURES:
    for fac in FACTORS:
        mask_pos = data[f"{feat}_z"] > 0
        mask_neg = ~mask_pos
        r_pos = data.loc[mask_pos, fac]
        r_neg = data.loc[mask_neg, fac]

        sr_pos = np.sqrt(12) * r_pos.mean() / r_pos.std() if r_pos.
```

```
            sr_neg = np.sqrt(12) * r_neg.mean() / r_neg.std() if r_neg.

            sharpe.loc[fac, (feat, "> 0")] = sr_pos
            sharpe.loc[fac, (feat, "≤ 0")] = sr_neg

    sharpe = sharpe.round(3)

    # 6) Build pairwise-difference table: rows = “F1 – F2”
    rows = []
    pairs = []
    for i in range(len(FACTORS)):
        for j in range(i+1, len(FACTORS)):
            f1, f2 = FACTORS[i], FACTORS[j]
            rows.append(f"{f1} – {f2}")
            pairs.append((f1, f2))

    diffs = pd.DataFrame(index=rows, columns=sharpe.columns, dtype=floa
    for row, (f1, f2) in zip(rows, pairs):
        diffs.loc[row] = (sharpe.loc[f1] - sharpe.loc[f2]).values

    diffs = diffs.round(3)

    # 7) Display the DataFrame
    print("\nPairwise Sharpe-ratio differences by regime:\n")
    print(diffs)

    # 8) Plot as a Matplotlib table
    fig, ax = plt.subplots(figsize=(12, 3))
    ax.axis("off")

    # Flatten column labels for display
    col_labels = [f"{feat}\n{reg}" for feat, reg in diffs.columns]

    tbl = ax.table(
        cellText=diffs.values,
        rowLabels=diffs.index,
        colLabels=col_labels,
        loc="center",
        cellLoc="center",
    )
    tbl.auto_set_font_size(False)
    tbl.set_fontsize(8)
    tbl.scale(1, 1.5)

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

```
Pairwise Sharpe-ratio differences by regime:
```

```
Pairwise Sharpe-ratio differences by regime:

              CPI%          T10Y3M          CFNAI          GARCH
               > 0    ≤ 0    > 0    ≤ 0    > 0    ≤ 0    > 0    ≤ 0
SMB — HML -0.576  0.268 -0.098 -0.067 -0.353  0.174 -0.474  0.203
SMB — CMA -0.691  0.171 -0.163 -0.234 -0.422  0.021 -1.069  0.486
SMB — RMW -0.710  0.125 -0.305 -0.139 -0.341 -0.076 -0.885  0.341
HML — CMA -0.115 -0.097 -0.065 -0.167 -0.069 -0.153 -0.595  0.283
HML — RMW -0.134 -0.143 -0.207 -0.072  0.012 -0.250 -0.411  0.138
CMA — RMW -0.019 -0.046 -0.142  0.095  0.081 -0.097  0.184 -0.145
```

	CPI% > 0	CPI% ≤ 0	T10Y3M > 0	T10Y3M ≤ 0	CFNAI > 0	CFNAI ≤ 0	GARCH > 0	GARCH ≤ 0
SMB – HML	-0.576	0.268	-0.098	-0.067	-0.353	0.174	-0.474	0.203
SMB – CMA	-0.691	0.171	-0.163	-0.234	-0.422	0.021	-1.069	0.486
SMB – RMW	-0.71	0.125	-0.305	-0.139	-0.341	-0.076	-0.885	0.341
HML – CMA	-0.115	-0.097	-0.065	-0.167	-0.069	-0.153	-0.595	0.283
HML – RMW	-0.134	-0.143	-0.207	-0.072	0.012	-0.25	-0.411	0.138
CMA – RMW	-0.019	-0.046	-0.142	0.095	0.081	-0.097	0.184	-0.145

```
# # # Plot Regime-wise Correlation Heatmaps
# #
# # For the selected return columns, compute and plot the correlati
# # for each market regime as a heatmap.

# # %%
# # Use the global FACTORS instead of redefining returns_columns
# unique_regimes = df[REGIMES_COLUMN].unique()
# for regime in unique_regimes:
#     regime_data = df[df[REGIMES_COLUMN] == regime][FACTORS]
#     corr = regime_data.corr()

#     plt.figure(figsize=(10, 8))
#     sns.heatmap(corr, annot=True, fmt=".2f", cmap="coolwarm", cba
#     plt.title(f"Return Correlation Heatmap - {regime}")
#     plt.tight_layout()
#     plt.show()
```

```
# # # Plot Sharpe Ratios by Market Regime
# #
# # Compute and visualize Sharpe ratios for selected factors across
# # as well as the unconditional (all-data) values, using a bar cha
# # The numeric regime codes are converted back to their original n
```

```
# # and then further shortened using regime_short_mapping.

# # %%
# # Define factors and regime columns (using global variables if al
# factors_columns = FACTORS
# regimes_column = REGIMES_COLUMN   # Assumes REGIMES_COLUMN was de

# # Use the previously created regime_short_mapping to convert nume
# # (If a code is not in regime_short_mapping, it will default to "
# regime_short_names = {reg: regime_short_mapping.get(reg, f"Regime
#                       for reg in df[regimes_column].unique()}

# sharpe_ratios = {
#     regime_short_names[regime]: (
#         df[df[regimes_column] == regime][factors_columns].mean()
#         df[df[regimes_column] == regime][factors_columns].std()
#     )
#     for regime in df[regimes_column].unique()
# }

# # Calculate the "Unconditional" Sharpe ratios (using all data)
# sharpe_ratios["Unconditional"] = df[factors_columns].mean() / df[

# # Convert the dictionary to a DataFrame and set column names
# sharpe_ratios_df = pd.DataFrame(sharpe_ratios).T
# sharpe_ratios_df.columns = factors_columns

# # Plot the Sharpe ratios using the same styling as before.
# plt.figure(figsize=(14, 8))
# sharpe_ratios_df.plot(
#     kind="bar",
#     grid=True,
#     colormap="viridis",
#     title="Sharpe Ratios by Regime and Unconditional",
#     figsize=(14, 8)
# )
# plt.ylabel("Sharpe Ratio", fontsize=12)
# plt.xlabel("Market Regimes", fontsize=12)
# plt.xticks(rotation=45, fontsize=10)
# plt.yticks(fontsize=10)
# plt.legend(title="Factors", fontsize=10, bbox_to_anchor=(1.05, 1)
# plt.tight_layout()
# plt.show()
```

Feature importance by period

```
# import numpy as np
# import pandas as pd
# import matplotlib.pyplot as plt

# # ======= USER-DEFINED DATE RANGE =======
# # Adjust these dates to view feature importances for a specific pe
# start_date = pd.to_datetime('2020-01-01')
# end_date   = pd.to_datetime('2022-12-31')

# # ======= Filter the Data =======
# # Filter the results_df for the specified date range based on the
# filtered_results_df = results_df[
#     (results_df['Predicted_month'] >= start_date) &
#     (results_df['Predicted_month'] <= end_date)
# ]

# # ======= Get Unique Regimes and Feature Count =======
# existing_regimes = filtered_results_df['Regime'].unique()
# n_regimes = len(existing_regimes)
# n_features = len(filtered_results_df['Feature_Importances'].iloc[0

# # ======= Robust Feature Naming =======
# try:
#     # Validate if the predefined FEATURES list matches the actual
#     if len(FEATURES) != n_features:
#         print(f"⚠️ Warning: FEATURES list length ({len(FEATURES)})
#         print("Using auto-generated feature names instead.")
#         raise ValueError
#     feature_names = FEATURES
# except (NameError, ValueError):
#     # Generate default feature names if there's a mismatch or if F
#     feature_names = [f'Feature {i+1}' for i in range(n_features)]
#     print(f"Using auto-generated feature names for {n_features} fe

# # ======= Compute Overall Average Feature Importances =======
# overall_avg_fi = np.vstack(filtered_results_df['Feature_Importance

# # ======= Compute Regime-Specific Average Feature Importances ====
# regime_avg_fi = {}
# for regime_name in existing_regimes:
#     regime_df = filtered_results_df[filtered_results_df['Regime'] =
#     regime_fi_array = np.vstack(regime_df['Feature_Importances'].v
#     regime_avg_fi[regime_name] = regime_fi_array.mean(axis=0)
```

```
# # ======= Sort Features by Overall Importance (Descending) =======
# sorted_idx = overall_avg_fi.argsort()[::-1]
# sorted_idx = sorted_idx[sorted_idx < len(feature_names)]  # Ensure
# sorted_features = [feature_names[i] for i in sorted_idx]

# # ======= Plotting =======
# if n_regimes > 1:
#     total_plots = 1 + n_regimes  # One overall plot plus one for e
#     row_height = max(0.3 * n_features, 4)
#     fig, axs = plt.subplots(
#         total_plots,
#         1,
#         figsize=(19.5, total_plots * row_height),
#         gridspec_kw={'hspace': 0.4}
#     )
#     if total_plots == 1:
#         axs = [axs]

#     # --- Overall Feature Importances ---
#     axs[0].barh(
#         np.arange(n_features),
#         overall_avg_fi[sorted_idx],
#         color='steelblue',
#         edgecolor='black'
#     )
#     axs[0].set_yticks(np.arange(n_features))
#     axs[0].set_yticklabels(sorted_features)
#     axs[0].set_title("Overall Average Feature Importances", pad=12
#     axs[0].set_xlabel("Average Importance")
#     axs[0].grid(axis='x', linestyle='--', alpha=0.7)

#     # --- Regime-Specific Feature Importances ---
#     for idx, (regime_name, avg_fi) in enumerate(regime_avg_fi.item
#         sorted_regime_fi = avg_fi[sorted_idx]
#         axs[idx].barh(
#             np.arange(n_features),
#             sorted_regime_fi,
#             color='salmon',
#             edgecolor='black'
#         )
#         axs[idx].set_yticks(np.arange(n_features))
#         axs[idx].set_yticklabels(sorted_features)
#         axs[idx].set_title(f"Feature Importances: {regime_name} Re
#         axs[idx].set_xlabel("Average Importance")
#         axs[idx].grid(axis='x', linestyle='--', alpha=0.7)
# else:
#     # If zero or one regime, show only the overall chart
#     row_height = max(0.3 * n_features, 4)
```

```
#     fig, ax = plt.subplots(
#         1, 1, figsize=(19.5, row_height),
#         gridspec_kw={'hspace': 0.4}
#     )
#     ax.barh(
#         np.arange(n_features),
#         overall_avg_fi[sorted_idx],
#         color='steelblue',
#         edgecolor='black'
#     )
#     ax.set_yticks(np.arange(n_features))
#     ax.set_yticklabels(sorted_features)
#     ax.set_title("Overall Average Feature Importances (No Multiple
#     ax.set_xlabel("Average Importance")
#     ax.grid(axis='x', linestyle='--', alpha=0.7)

# plt.tight_layout(pad=4.0)
# plt.subplots_adjust(left=0.3)  # Extra space for feature labels
# plt.show()
```

```
Start coding or generate with AI.
```