```
pip install arch
```

```
Collecting arch
  Downloading arch-7.2.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (13 kB)
Requirement already satisfied: numpy>=1.22.3 in /usr/local/lib/python3.11/dist-packages (from arch) (2.0.2)
Requirement already satisfied: scipy>=1.8 in /usr/local/lib/python3.11/dist-packages (from arch) (1.14.1)
Requirement already satisfied: pandas>=1.4 in /usr/local/lib/python3.11/dist-packages (from arch) (2.2.2)
Requirement already satisfied: statsmodels>=0.12 in /usr/local/lib/python3.11/dist-packages (from arch) (0.14.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas>=1.4->arch) (2
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas>=1.4->arch) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas>=1.4->arch) (2025.2)
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.11/dist-packages (from statsmodels>=0.12->arch) (1.0.1
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.11/dist-packages (from statsmodels>=0.12->arch) (24
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2->pandas>=1.4
Downloading arch-7.2.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (985 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 985.3/985.3 kB 11.8 MB/s eta 0:00:00
Installing collected packages: arch
Successfully installed arch-7.2.0
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from arch import arch_model
```

Skip this if already have the data file

↳ 10 cells hidden

Retrieve data from csv files

```
from google.colab import drive
drive.mount('/content/drive/')
```

```
Mounted at /content/drive/
```

```
cd drive/MyDrive/Math583/FinalProject
```

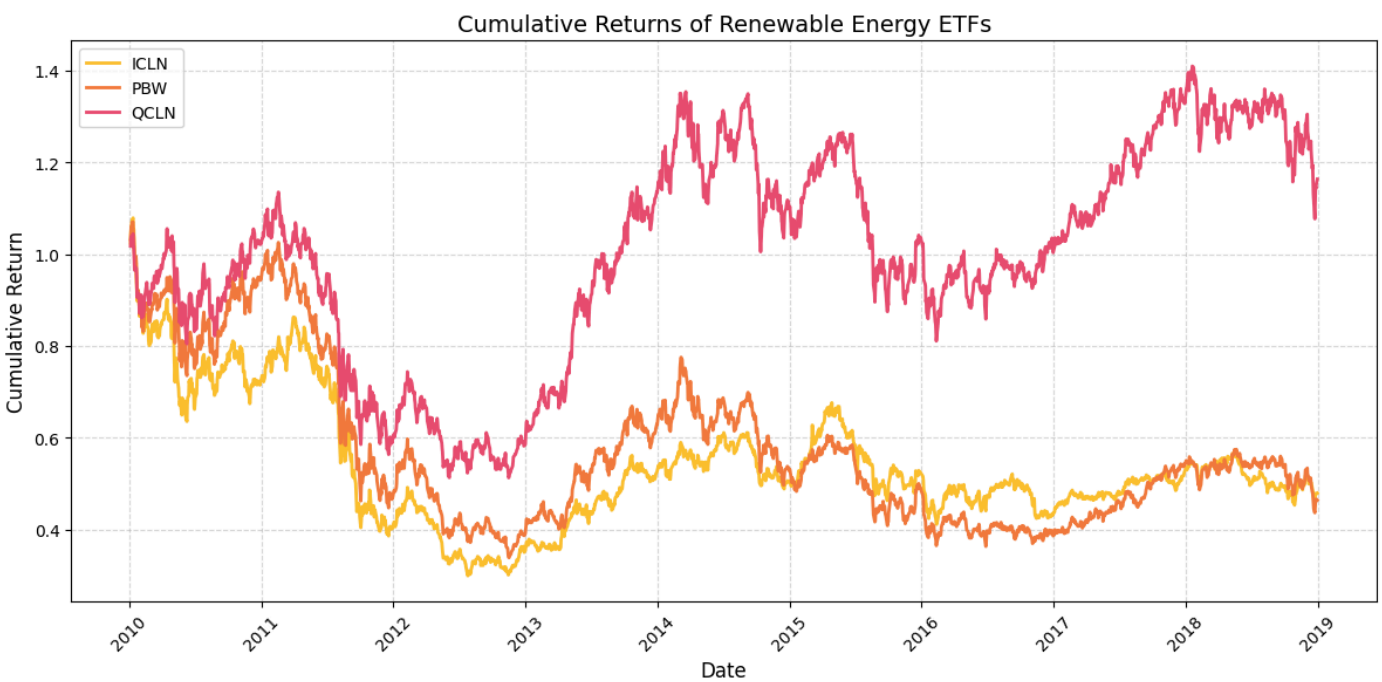
```
/content/drive/MyDrive/Math583/FinalProject
```

```
# nonrenewable_df = pd.read_csv('data/nonrenewable_etfs_returns.csv', index_col=0, parse_dates=True)
# renewable_df = pd.read_csv('data/renewable_etfs_returns.csv', index_col=0, parse_dates=True)
returns_df = pd.read_csv('daily_returns_df.csv', index_col=0, parse_dates=True)

renewable_df = returns_df[['ICLN', 'PBW', 'QCLN']]
nonrenewable_df = returns_df[['XLE', 'SPY']]

bond_returns = pd.read_csv("bond_returns.csv", index_col=0, parse_dates=True)
bond_df = bond_returns[bond_returns.index.isin(renewable_df.index)]
```

Exploratory analysis

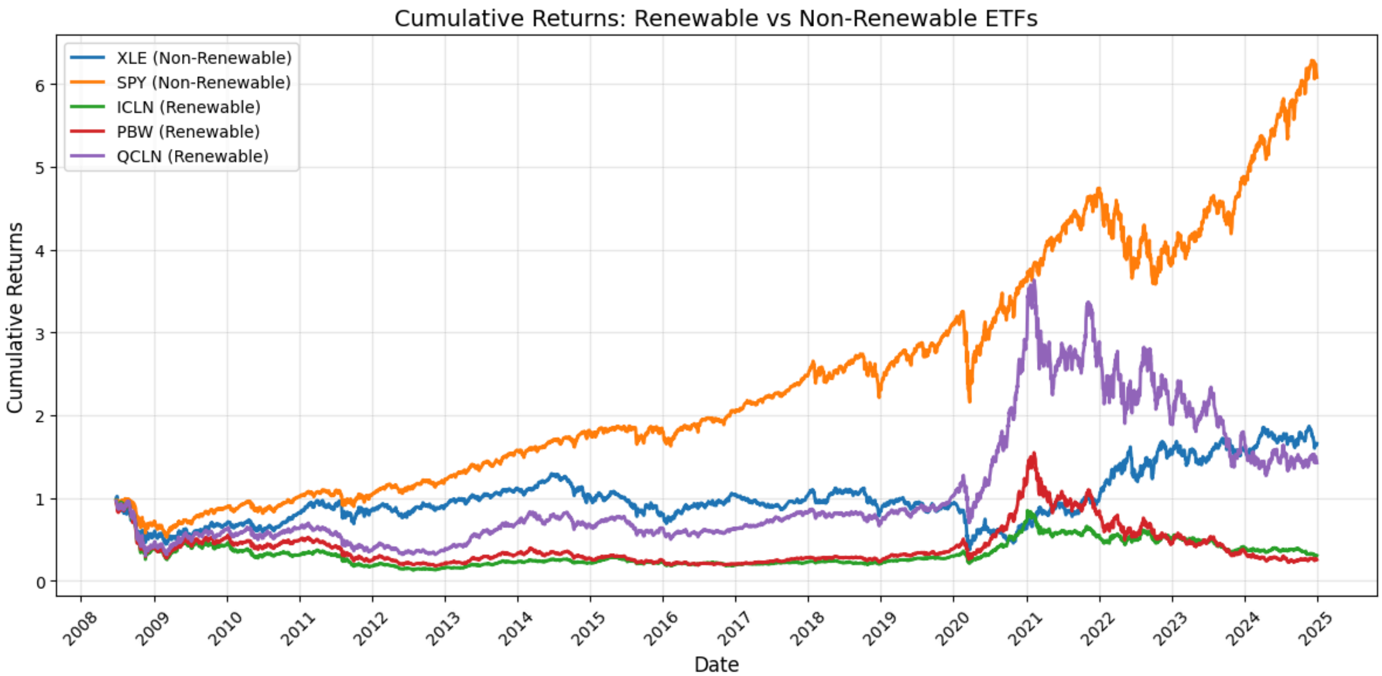
```
import matplotlib.dates as mdates

start_date = "2010-01-01"   # or pd.Timestamp("2015-01-01")
end_date   = "2018-12-31"


# Compute cumulative returns
cum_returns_r = (1 + renewable_df[start_date:end_date]).cumprod()
cum_returns_nr = (1 + nonrenewable_df[start_date:end_date]).cumprod()


dates_nr = pd.to_datetime(cum_returns_nr.index)
dates_r = pd.to_datetime(cum_returns_r.index)

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

```
plt.figure(figsize=(12, 6))

# Set custom colors
colors = {
    "ICLN": "#FDBF2D",  # golden
    "PBW": "#F47C3C",   # orange
    "QCLN": "#E94B6E",  # red-pink
}

for col in cum_returns_r.columns:
    plt.plot(cum_returns_r.index, cum_returns_r[col], label=col, color=colors.get(col, None), linewidth=2)

# Styling
plt.title("Cumulative Returns of Renewable Energy ETFs", fontsize=14)
plt.ylabel("Cumulative Return", fontsize=12)
plt.xlabel("Date", fontsize=12)
plt.grid(True, which='major', linestyle='--', alpha=0.5)

# Set x-ticks to show only every 100th date
ax = plt.gca()
ax.xaxis.set_major_locator(mdates.YearLocator())
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
plt.xticks(rotation=45)

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




```
plt.figure(figsize=(12, 6))

# Convert indices to datetime if they're not already
dates_nr = pd.to_datetime(cum_returns_nr.index)
dates_r = pd.to_datetime(cum_returns_r.index)

# Plot both cumulative returns series
plt.plot(dates_nr, cum_returns_nr['XLE'], label='XLE (Non-Renewable)', linewidth=2)
plt.plot(dates_nr, cum_returns_nr['SPY'], label='SPY (Non-Renewable)', linewidth=2)
plt.plot(dates_r, cum_returns_r['ICLN'], label='ICLN (Renewable)', linewidth=2)
plt.plot(dates_r, cum_returns_r['PBW'], label='PBW (Renewable)', linewidth=2)
plt.plot(dates_r, cum_returns_r['QCLN'], label='QCLN (Renewable)', linewidth=2)

# Customize the plot
plt.title('Cumulative Returns: Renewable vs Non-Renewable ETFs', fontsize=14)
plt.xlabel('Date', fontsize=12)
plt.ylabel('Cumulative Returns', fontsize=12)
plt.grid(True, alpha=0.3)
```

```
plt.grid(True, alpha=0.3)

# Set x-ticks to show only every 100th date
ax = plt.gca()
ax.xaxis.set_major_locator(mdates.YearLocator())
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
plt.xticks(rotation=45)

# Add legend
plt.legend(fontsize=10)

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

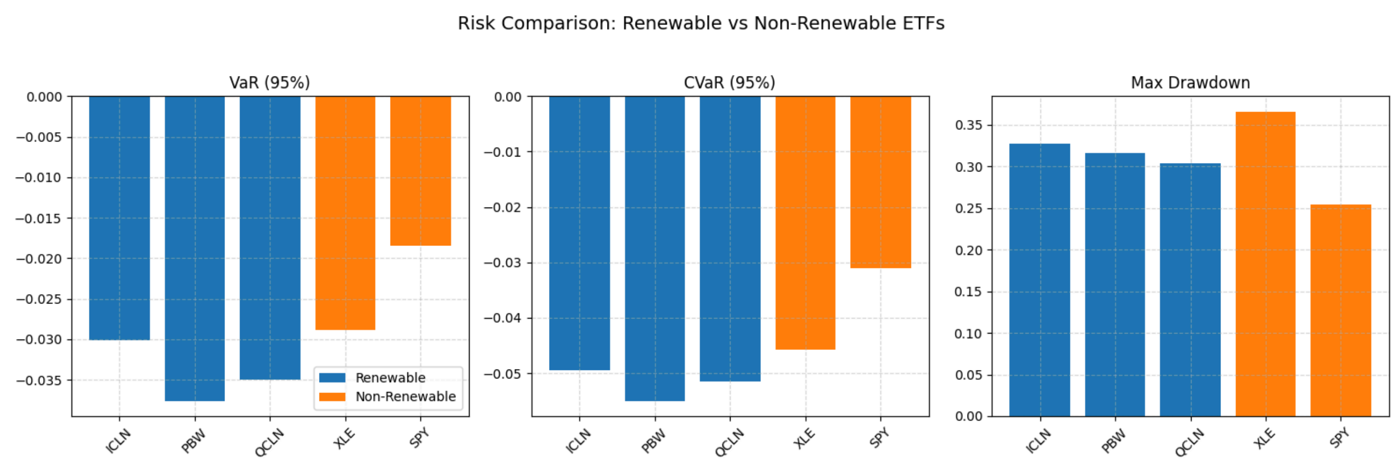


Compute VaR, CVaR and Max Drawdown for ETFs

```
import warnings
warnings.filterwarnings("ignore")

# 1. Compute risk metrics
def compute_risk_metrics(df):
    metrics = {}
    for col in df.columns:
        returns = df[col].dropna()
        var_95 = returns.quantile(0.05)
        cvar_95 = returns[returns <= var_95].mean()
        mdd = (returns.cummax() - returns).max()
        metrics[col] = {
            'VaR (95%)': round(var_95, 4),
            'CVaR (95%)': round(cvar_95, 4),
            'Max Drawdown': round(mdd, 4)
        }
    return pd.DataFrame(metrics).T

# 2. Generate risk metric tables
renewable_risk = compute_risk_metrics(renewable_df)
nonrenewable_risk = compute_risk_metrics(nonrenewable_df)

print(renewable_risk)
print(nonrenewable_risk)

# 3. Combine with group labels
combined_risk = pd.concat(
```

```
    [renewable_risk, nonrenewable_risk],
    keys=["Renewable", "Non-Renewable"]
)
combined_risk.index.names = ["Group", "ETF"]

# 4. Melt into tidy format for plotting
melted = combined_risk.reset_index().melt(
    id_vars=["Group", "ETF"],
    var_name="Metric",
    value_name="Value"
)

# 5. Plot
plt.figure(figsize=(15, 5))

metrics = melted["Metric"].unique()
for i, metric in enumerate(metrics):
    ax = plt.subplot(1, 3, i + 1)
    subset = melted[melted["Metric"] == metric]

    # Create ordered list of ETFs
    etf_order = ['ICLN', 'PBW', 'QCLN', 'XLE', 'SPY']

    # Filter and sort the data according to the ETF order
    subset = subset[subset['ETF'].isin(etf_order)]
    subset['ETF'] = pd.Categorical(subset['ETF'], categories=etf_order, ordered=True)
    subset = subset.sort_values('ETF')

    for group in subset["Group"].unique():
        group_data = subset[subset["Group"] == group]
        ax.bar(group_data["ETF"], group_data["Value"], label=group)

    ax.set_title(metric)
    ax.set_xticklabels(etf_order, rotation=45)
    ax.grid(True, linestyle="--", alpha=0.5)
    if i == 0:
        ax.legend()

plt.suptitle("Risk Comparison: Renewable vs Non-Renewable ETFs", fontsize=14)
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```

```
       VaR (95%)  CVaR (95%)  Max Drawdown
ICLN    -0.0301     -0.0495        0.3275
PBW     -0.0376     -0.0550        0.3162
QCLN    -0.0350     -0.0515        0.3040
      VaR (95%)  CVaR (95%)  Max Drawdown
XLE    -0.0288     -0.0458        0.3662
SPY    -0.0185     -0.0310        0.2546
```



USE: GARCH Volatility signals

make training and test set to later test performance of volatility signals

```
from arch import arch_model
import pandas as pd
import numpy as np

def compute_weekly_forecast_and_realized_vol(
    returns_df: pd.DataFrame,
    ticker: str,
    start: str,
    end: str,
    rolling_window: int = 500,
    forecast_horizon: int = 5,        # 5 trading days ≈ 1 week
) -> tuple[pd.Series, pd.Series]:
    """
    Weekly GARCH(1,1) σ forecast vs. realised σ (5-day window).

    • Fits a new model every Friday (or last trading day of the week)
      using the previous `rolling_window` daily returns.
    • Forecasts the variance of the next 5 trading days, converts to
      weekly σ in *decimal* units.
    • Computes the realised 5-day σ for comparison.

    Returns
    -------
    forecast_vol : pd.Series  (index = forecast dates, name = 'Forecast 1W σ')
    realized_vol : pd.Series  (same index, name = 'Realised 1W σ')
    """
    # 1) prep ---------------------------------------------------------------
    df = returns_df.copy()
    df.index = pd.to_datetime(df.index)
    series = df[ticker].loc[start:end].dropna()

    pct_returns = series * 100         # arch library expects percent scale

    f_dates, f_vols = [], []

    # 2) rolling weekly forecasts ------------------------------------------
    for i in range(rolling_window, len(pct_returns) - forecast_horizon):
        date_i = pct_returns.index[i]

        # run only on Friday (weekday() == 4)  —— or use the last day of the week
        if date_i.weekday() != 4:
            continue

        window = pct_returns.iloc[i - rolling_window : i]
        res = arch_model(window, vol='Garch', p=1, q=1).fit(disp='off')

        fc_var_pct2 = res.forecast(horizon=forecast_horizon, reindex=False).variance.values[-1]
        weekly_var_dec = fc_var_pct2.sum() / 10000     # percent² → decimal²
        f_vols.append(np.sqrt(weekly_var_dec))
        f_dates.append(date_i)

    forecast_vol = pd.Series(f_vols, index=f_dates, name='Forecast 1W vol')

    # 3) realised 5-day σ over the same horizons ----------------------------
    r_vols = []
    for dt in f_dates:
        # position of dt in *original* decimal series
        pos = series.index.get_loc(dt)
        window = series.iloc[pos + 1 : pos + 1 + forecast_horizon]   # next 5 days
        realised_var = (window ** 2).sum()
        r_vols.append(np.sqrt(realised_var))

    realized_vol = pd.Series(r_vols, index=f_dates, name='Realised 1W vol')
    return forecast_vol, realized_vol



def plot_forecast_vs_realized_vol(
    forecast_vol: pd.Series,
    realized_vol: pd.Series,
```

```python
    ticker: str,
    start_date: str,
    end_date: str,
    *,
    horizon_label: str = "1W",
    figsize: tuple = (12, 6),
    linewidth: float = 2.0,
    alpha: float = 0.8,
):
    """
    Plot forecasted vs. realised volatility for a chosen horizon.

    Parameters
    ----------
    forecast_vol, realized_vol : pd.Series
        Forecasted and realised volatilities (decimal σ), indexed by date.
    ticker        : str        – ETF symbol used in the title.
    start_date,end_date : str/Timestamp – inclusive date window.
    horizon_label : str        – text that appears in legend/title ("1W","1M",…).
    figsize       : tuple      – figure size in inches.
    linewidth     : float      – line width.
    alpha         : float      – transparency for realised line.
    """
    # date window
    start_date = pd.to_datetime(start_date)
    end_date   = pd.to_datetime(end_date)

    fcast = forecast_vol.loc[start_date:end_date]
    real  = realized_vol.loc[start_date:end_date]

    plt.figure(figsize=figsize)
    plt.plot(fcast.index, fcast, label=f"Forecasted {horizon_label} Vol", linewidth=linewidth)
    plt.plot(real.index,  real,  label=f"Realised {horizon_label} Vol",  linewidth=linewidth, alpha=alpha)

    plt.title(
        f"{ticker}: Forecasted vs Realised {horizon_label} Volatility\n"
        f"{start_date.strftime('%Y-%m-%d')} to {end_date.strftime('%Y-%m-%d')}"
    )
    plt.xlabel("Date")
    plt.ylabel("Volatility (σ)")
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.4)
    plt.gcf().autofmt_xdate()
    plt.tight_layout()
    plt.show()
```

```python
forecast_ICLN, realized_ICLN = compute_weekly_forecast_and_realized_vol(renewable_df, 'ICLN', '2008-01-01', '2024-12-31')
forecast_PBW, realized_PBW = compute_weekly_forecast_and_realized_vol(renewable_df, 'PBW', '2008-01-01', '2024-12-31')
forecast_QCLN, realized_QCLN = compute_weekly_forecast_and_realized_vol(renewable_df, 'QCLN', '2008-01-01', '2024-12-31')

# combine forecat_ICLN, forecast_PBW, and forecast_QCLN
# combine on dates
garch_forecast = pd.concat([forecast_ICLN, forecast_PBW, forecast_QCLN], axis=1)
# rename to ICLN, PBW, and QCLN
garch_forecast.columns = ['ICLN', 'PBW', 'QCLN']
```

```python
hi_mask = garch_forecast > garch_forecast.quantile(0.90)   # top 10 %
overlaps = hi_mask.sum(1)          # 0–3 ETFs in regime
overlaps.value_counts()
```

	count
0	628
3	41
2	34
1	31

dtype: int64

```
garch_forecast.corr()
corr_roll   = garch_forecast.rolling(52).corr().unstack().loc[:, 'ICLN']  # 1-yr rolling
corr_roll
```

	ICLN	PBW	QCLN
2010-06-25	NaN	NaN	NaN
2010-07-02	NaN	NaN	NaN
2010-07-09	NaN	NaN	NaN
2010-07-16	NaN	NaN	NaN
2010-07-23	NaN	NaN	NaN
...	...	...	...
2024-11-22	1.0	0.809644	0.793055
2024-11-29	1.0	0.806783	0.769033
2024-12-06	1.0	0.799197	0.753145
2024-12-13	1.0	0.795487	0.746606
2024-12-20	1.0	0.753513	0.694672

734 rows × 3 columns

Next steps: Generate code with corr_roll View recommended plots New interactive sheet

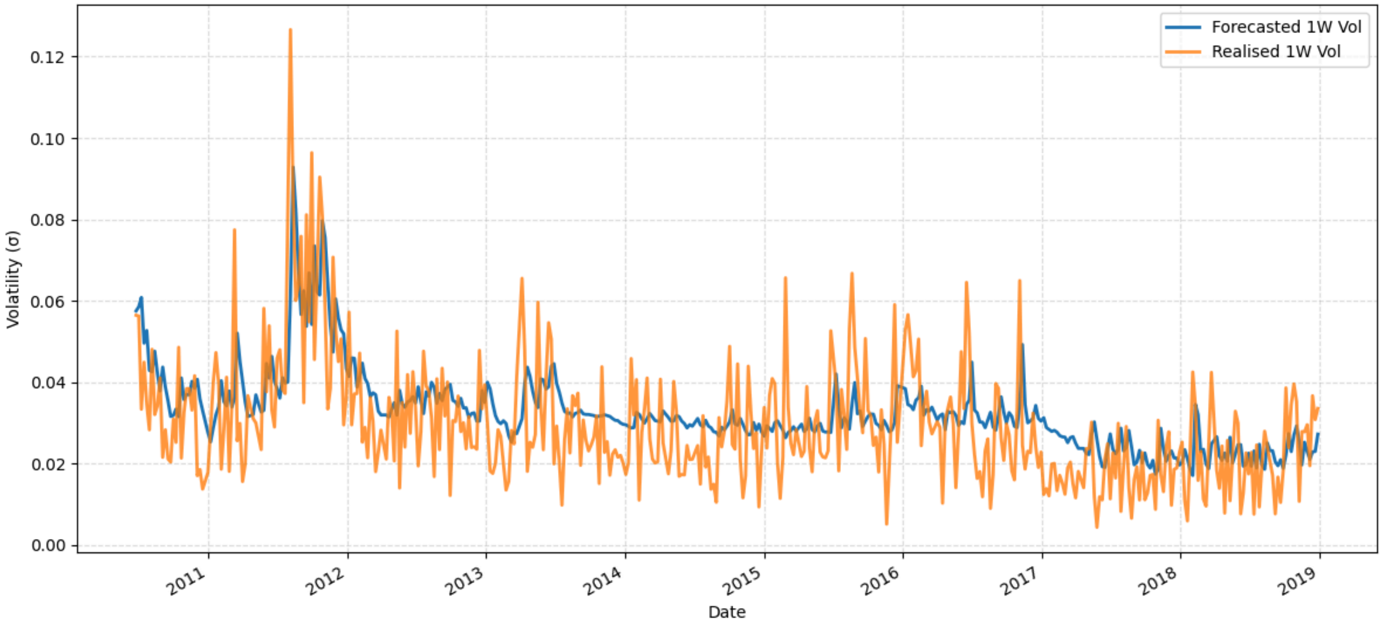
```
(forecast - realized)**2).mean(), .pow(2).mean()**0.5
```

```
  File "<ipython-input-21-181e5be57d20>", line 1
    (forecast - realized)**2).mean(), .pow(2).mean()**0.5
                            ^
SyntaxError: unmatched ')'
```

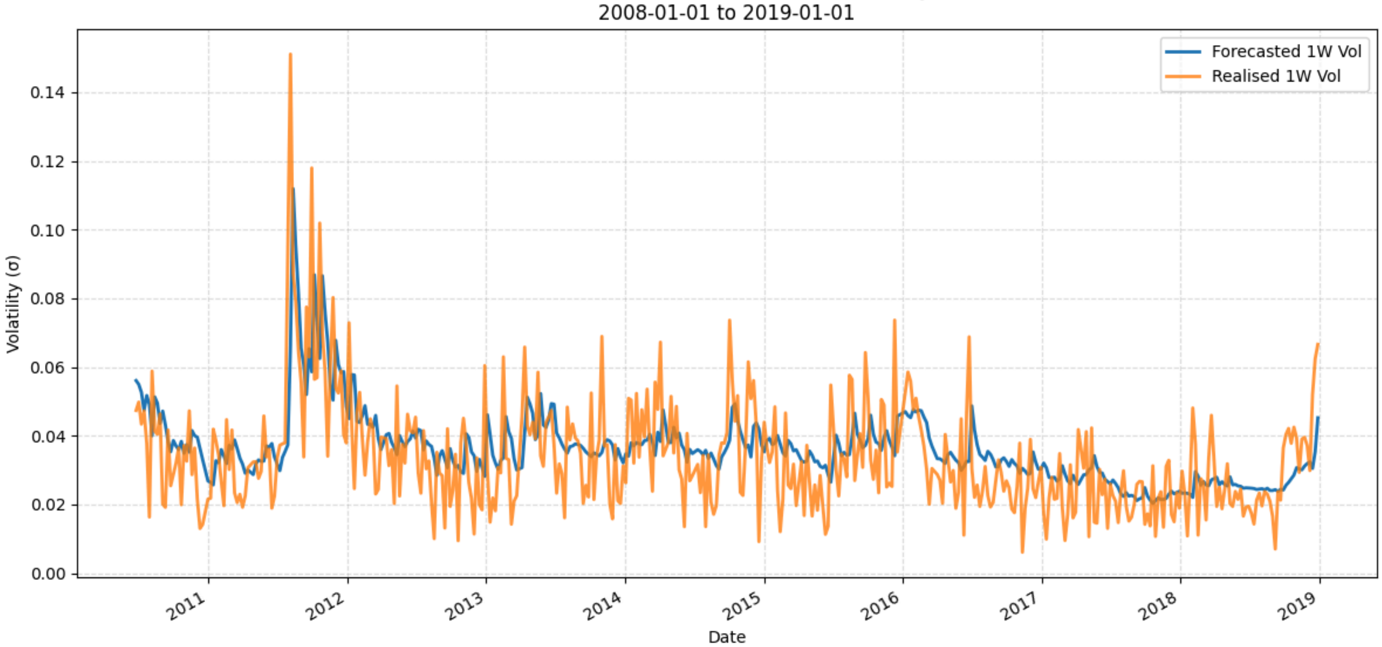
Next steps: Fix error

```
plot_forecast_vs_realized_vol(forecast_ICLN, realized_ICLN, 'ICLN', '2008', '2019')
plot_forecast_vs_realized_vol(forecast_PBW, realized_PBW, 'PBW', '2008', '2019')
plot_forecast_vs_realized_vol(forecast_QCLN, realized_QCLN, 'QCLN', '2008', '2019')
```

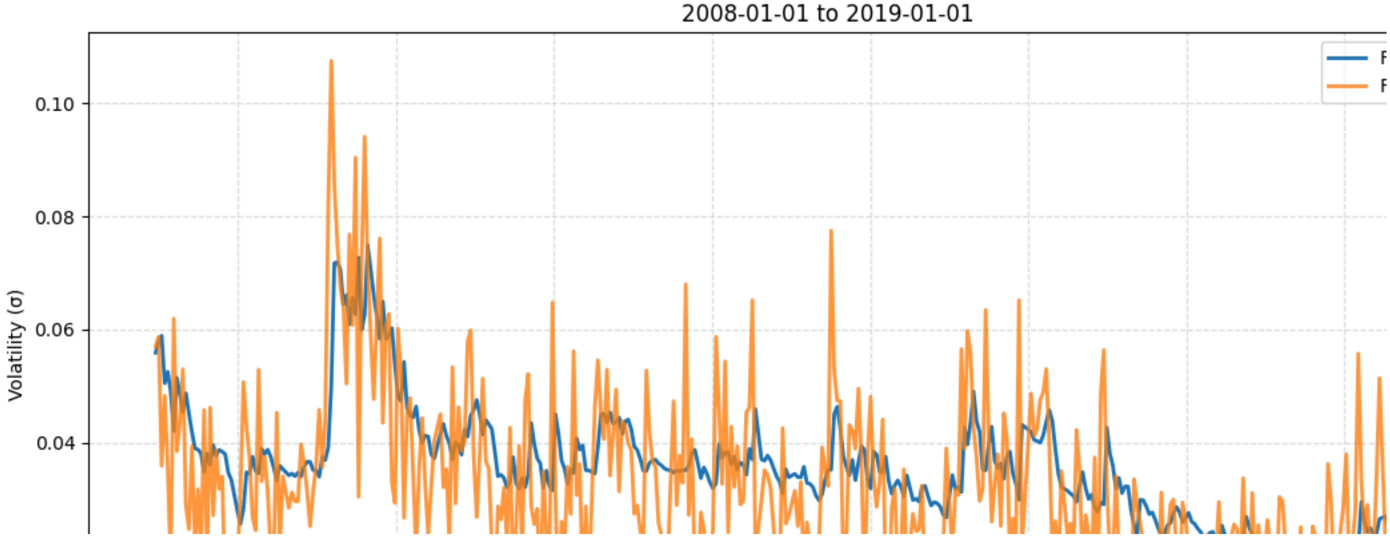
ICLN: Forecasted vs Realised 1W Volatility
2008-01-01 to 2019-01-01

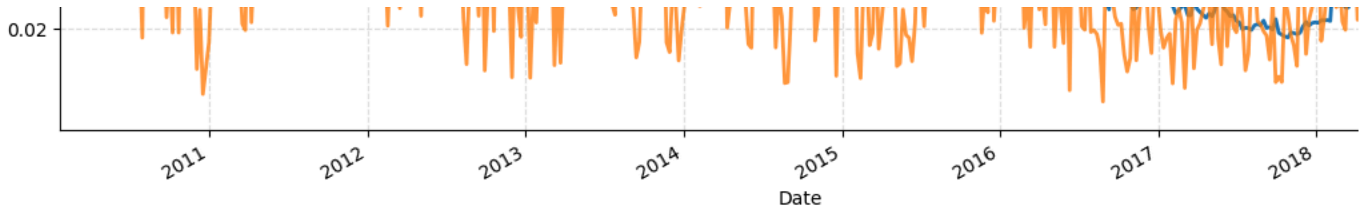


PBW: Forecasted vs Realised 1W Volatility
2008-01-01 to 2019-01-01



QCLN: Forecasted vs Realised 1W Volatility
2008-01-01 to 2019-01-01





USE: GARCH Hedging

Sensitivity analysis to parameters (what are "ideal" parameters) VaR vs return (relative to no hedge)

```
def compute_hedged_returns(
    equity_ret: pd.Series,
    vol_forecast: pd.Series,
    safe_ret: pd.Series,
    vol_thr: float,
    de_risk: float
) -> pd.Series:
    """
    Blend equity and safe-asset returns on high-vol days.

    Parameters
    ----------
    equity_ret : pd.Series
        Daily equity returns (decimal), e.g. renewable_df['PBW'].
    vol_forecast : pd.Series
        1-month ahead forecast volatilities (decimal), same dates as equity_ret subset.
    safe_ret : pd.Series
        Daily “safe” asset returns (decimal), e.g. bond_returns['SHY'].
    vol_thr : float
        Volatility percentile threshold (0-1), e.g. 0.9.
    de_risk : float
        Fraction to move into safe asset on high-vol days (0-1), e.g. 0.5.

    Returns
    -------
    pd.Series
        Hedged daily returns, indexed by date.  On days where
        vol_forecast > vol_forecast.quantile(vol_thr),
        it returns (1-de_risk)*equity_ret + de_risk*safe_ret,
        otherwise just equity_ret.
    """
    # align all three series
    idx = equity_ret.index.intersection(vol_forecast.index).intersection(safe_ret.index)
    eq = equity_ret.loc[idx]
    vf = vol_forecast.loc[idx]
    sf = safe_ret.loc[idx]

    # threshold
    cutoff = vf.quantile(vol_thr)

    # start from pure equity
    hedged = eq.copy()

    # on high-vol days, blend into safe asset
    high_vol = vf > cutoff
    hedged.loc[high_vol] = (
        eq.loc[high_vol] * (1 - de_risk)
      + sf.loc[high_vol] * de_risk
    )

    return hedged
```

```python
def compute_hedged_returns(
    equity_ret: pd.Series,
    vol_forecast: pd.Series,
    safe_ret: pd.Series,
    vol_thr: float,
    de_risk: float,
) -> pd.Series:
    """
    Hedge rule: if σ̂_t exceeds its `vol_thr` percentile, move `de_risk`
    of the portfolio into the safe asset *until the next forecast arrives*.

    Works for daily, weekly, or monthly forecast series.

    Parameters
    ----------
    equity_ret   : pd.Series  – daily equity returns (decimal).
    vol_forecast : pd.Series  – forecast σ, any regular freq (D, W, M, …).
    safe_ret     : pd.Series  – daily safe-asset returns (decimal).
    vol_thr      : float      – percentile threshold (0–1) applied to σ̂.
    de_risk      : float      – share shifted into bonds when above threshold.

    Returns
    -------
    pd.Series – hedged daily returns, aligned with `equity_ret`.
    """
    # Align everything on dates present in equity & safe series
    idx = equity_ret.index.intersection(safe_ret.index)
    eq  = equity_ret.loc[idx]
    sf  = safe_ret.loc[idx]

    # 1. Build a daily series of weights based on the latest forecast
    # ----------------------------------------------------------------
    # Cut-off on forecast scale
    cutoff = vol_forecast.quantile(vol_thr)

    # Weight (equity share) on the dates forecasts actually exist
    w_equity_on_fcst = pd.Series(1.0, index=vol_forecast.index)
    w_equity_on_fcst.loc[vol_forecast > cutoff] = 1.0 - de_risk

    # Reindex to full daily calendar by forward-filling
    w_equity = w_equity_on_fcst.reindex(idx).ffill().fillna(1.0)

    # 2. Compute hedged daily return
    # ----------------------------------------------------------------
    hedged = w_equity * eq + (1.0 - w_equity) * sf
    hedged.name = "hedged_return"
    return hedged


hedged_daily_returns = compute_hedged_returns(
    equity_ret = renewable_df['PBW'],
    vol_forecast= garch_forecast["PBW"],
    safe_ret    = bond_returns['SHY'],
    vol_thr     = 0.9,    # 90th percentile
    de_risk     = 0.5     # move 50% into bonds
)
```

Testing multiple one-year periods

```python
def single_year_metrics(
    returns: pd.Series,
    forecast: pd.Series,
    bond_returns: pd.Series,
    year: int,
    vol_thr: float,
    de_risk: float,
    min_obs: int = 50,
):
    """
    • Use *all* daily returns in the calendar year.
    • Pass the (weekly) forecast series directly to compute_hedged_returns,
      which will forward-fill the hedge weight between forecast dates.
    """
        # –– slice DAILY equity & bond returns for the whole calendar year ––––––
    mask_year = returns.index.year == year
```

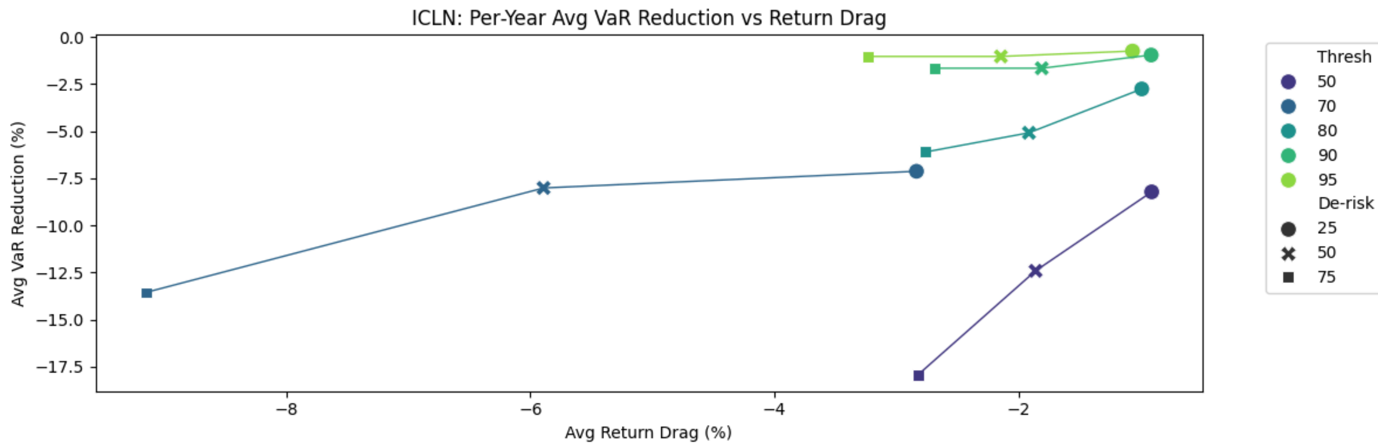
```
    raw  = returns.loc[mask_year]                    # equity
    bond = bond_returns.reindex(raw.index)           #  align by date

    if len(raw) < min_obs:
        return None                       # skip if too little data

    # —— slice the forecast up to (and including) that year ————————————————
    vol_year = forecast.loc[forecast.index.year <= year]

    # build hedged DAILY returns (forward-fills hedge weight between forecasts)
    hedged = compute_hedged_returns(
        equity_ret   = raw,
        vol_forecast = vol_year,
        safe_ret     = bond,
        vol_thr      = vol_thr,
        de_risk      = de_risk,
    )

    # —— performance & risk deltas (unchanged) ————————————————————————————————
    R_raw = (1 + raw).cumprod().iloc[-1]
    R_hdg = (1 + hedged).cumprod().iloc[-1]
    return_drag = (R_raw - R_hdg) / R_raw * 100

    var_raw, var_hdg = raw.quantile(0.05), hedged.quantile(0.05)
    cvar_raw = raw[raw <= var_raw].mean()
    cvar_hdg = hedged[hedged <= var_hdg].mean()

    def _mdd(x):
        path = (1 + x).cumprod()
        return (path / path.cummax() - 1).min()
    mdd_raw, mdd_hdg = _mdd(raw), _mdd(hedged)

    sharpe_raw = (raw  - bond).mean() / (raw  - bond).std(ddof=0) * np.sqrt(252)
    sharpe_hdg = (hedged - bond).mean() / (hedged - bond).std(ddof=0) * np.sqrt(252)
    sharpe_delta = (sharpe_hdg - sharpe_raw) / abs(sharpe_raw) * 100 if sharpe_raw else np.nan

    vol_delta = (raw.std(ddof=0) - hedged.std(ddof=0)) / raw.std(ddof=0) * 100

    return {
        "Return Drag":           return_drag,
        "VaR Reduction":         (var_raw - var_hdg)   / abs(var_raw)  * 100,
        "CVaR Reduction":        (cvar_raw - cvar_hdg) / abs(cvar_raw) * 100,
        "Max Drawdown Reduction":(mdd_raw - mdd_hdg)   / abs(mdd_raw)  * 100,
        "Sharpe Ratio Change":   sharpe_delta,
        "Volatility Change":     vol_delta,
    }



def yearly_sensitivity_averages(
    rets: pd.Series,
    fc: pd.Series,
    bond_rets: pd.Series,
    years: list,
    threshold_list: list,
    de_risk_list: list
) -> pd.DataFrame:
    """
    For a given returns series and matching GARCH-forecast series,
    compute the per-year average Return Drag, VaR Reduction, CVaR Reduction,
    Max Drawdown Reduction, and Sharpe Ratio across a grid of (volatility threshold, de-risk %) pairs.

    Parameters
    ----------
    rets : pd.Series
        Daily decimal returns, indexed by date.
    fc : pd.Series
        1-week ahead forecast volatilities (decimal), same index subset as `rets`.
    bond_rets : pd.Series
        Daily bond returns, same index as `rets`.
    years : list of int
        Calendar years to include, e.g. list(range(2010, 2020)).
    threshold_list : list of float
        Percentile thresholds for vol, e.g. [0.8, 0.9, 0.95].
    de_risk_list : list of float
        Fraction to de-risk on high-vol days, e.g. [0.25, 0.5, 0.75].
```

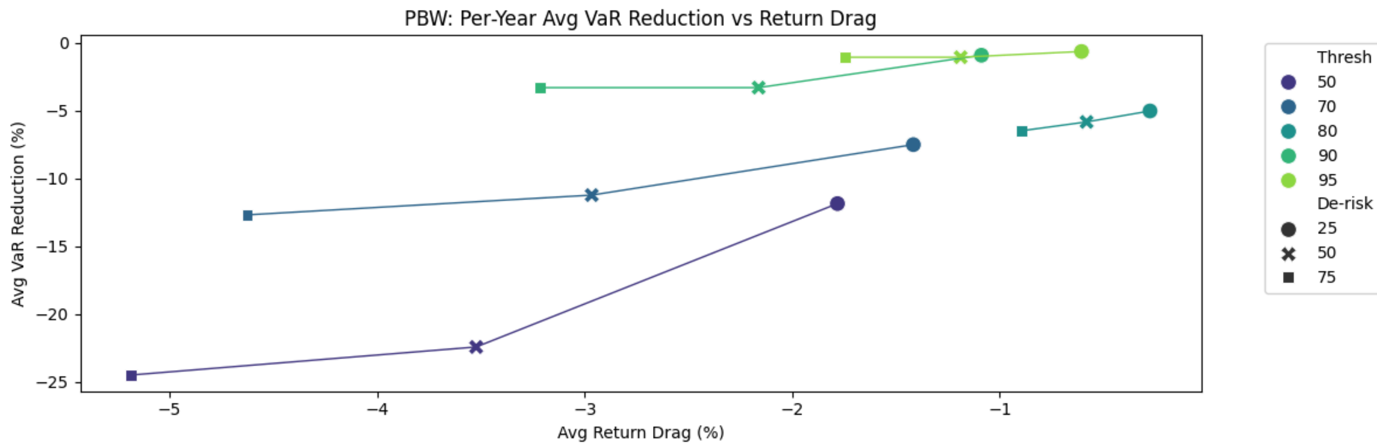
```
    Returns
    -------
    pd.DataFrame
        Columns: ['Thresh', 'De-risk', 'Avg Drag', 'Avg VaR Red', 'Avg CVaR Red', 'Avg MDD Red', 'Avg Sharpe', 'Years Used']
    """
    records = []

    for thr in threshold_list:
        for dr in de_risk_list:
            yearly = []
            for y in years:
                m = single_year_metrics(rets, fc, bond_rets, y, thr, dr)
                if m is not None:
                    yearly.append(m)

            if not yearly:
                continue

            dfy = pd.DataFrame(yearly)
            records.append({
                'Thresh':       int(thr * 100),
                'De-risk':      int(dr * 100),
                'Avg Drag':     dfy['Return Drag'].mean(),
                'Avg VaR Red':  dfy['VaR Reduction'].mean(),
                'Avg CVaR Red': dfy['CVaR Reduction'].mean(),
                'Avg MDD Red':  dfy['Max Drawdown Reduction'].mean(),
                'Avg Sharpe Change':  dfy['Sharpe Ratio Change'].mean(),
                'Avg Volatility Change':  dfy['Volatility Change'].mean(),
                'Years Used':   len(dfy)
            })

    return pd.DataFrame(records)



years = list(range(2011, 2015))
threshold_list = [0.5, 0.7, 0.8, 0.9, 0.95]
de_risk_list   = [0.25, 0.5, 0.75]

# rets and fc should already be defined:

sens_yearly_ICLN = yearly_sensitivity_averages(renewable_df['ICLN'], garch_forecast["ICLN"], bond_returns["SHY"],years, threshol
sens_yearly_PBW = yearly_sensitivity_averages(renewable_df['PBW'], garch_forecast["PBW"], bond_returns["SHY"], years, threshold_
sens_yearly_QCLN = yearly_sensitivity_averages(renewable_df['QCLN'], garch_forecast["QCLN"], bond_returns["SHY"], years, thresho

print("==============ICLN==============")
print(sens_yearly_ICLN)
print("==============PBW==============")
print(sens_yearly_PBW)
print("==============QCLN==============")
print(sens_yearly_QCLN)
```

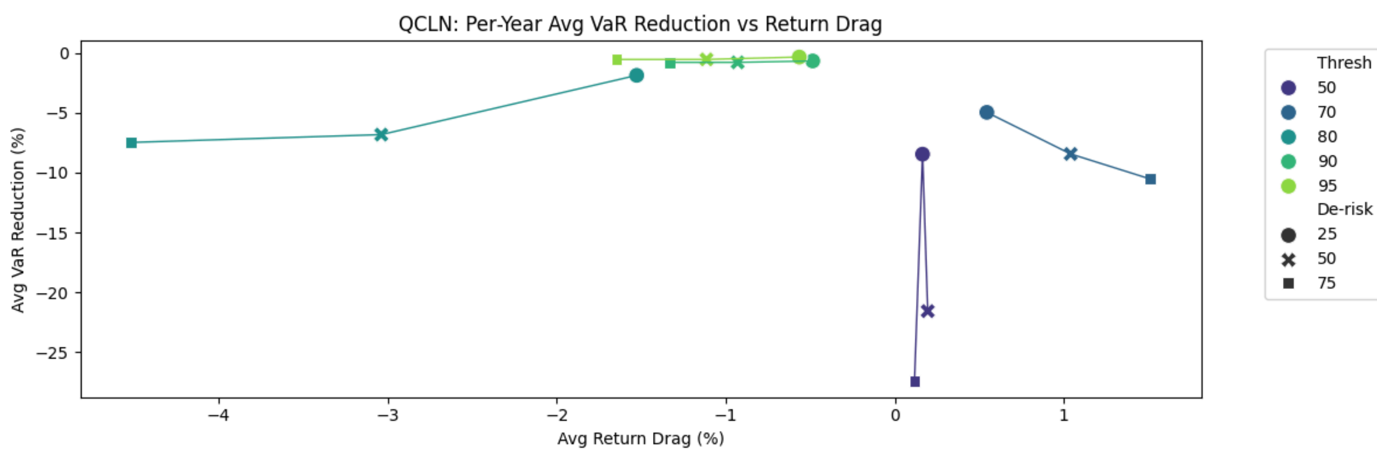
```
4          -24.470797              14.492913          4
```

```
5         -34.954611             18.863560           4
6          -6.994641              4.624405           4
7         -13.672438              8.263927           4
8         -19.210296             10.623826           4
9           0.028186              1.345870           4
10          0.430231              2.350606           4
11          1.287775              2.967487           4
12         -0.115670              0.858581           4
13          0.003391              1.487495           4
14          0.392279              1.868004           4
==============QCLN==============
    Thresh  De-risk  Avg Drag  Avg VaR Red  Avg CVaR Red  Avg MDD Red  \
0       50       25  0.164243    -8.454490    -12.629532    -6.756588
1       50       50  0.194879   -21.557561    -19.383902   -10.525507
2       50       75  0.116602   -27.387191    -21.110502   -12.493732
3       70       25  0.544634    -4.961667     -6.286570    -4.348230
4       70       50  1.040578    -8.435754     -9.423623    -6.191107
5       70       75  1.510044   -10.520810     -9.700530    -6.570793
6       80       25 -1.526934    -1.873847     -3.914066    -2.925068
7       80       50 -3.037896    -6.821397     -5.990613    -4.652373
8       80       75 -4.518522    -7.473627     -6.033921    -5.819819
9       90       25 -0.485429    -0.672040     -1.240393    -0.479079
10      90       50 -0.929632    -0.789178     -1.511459    -0.917471
11      90       75 -1.330277    -0.789178     -1.511459    -1.098157
12      95       25 -0.564872    -0.344743     -0.974657    -0.557483
13      95       50 -1.113909    -0.543877     -1.232048    -1.099337
14      95       75 -1.645696    -0.543877     -1.232048    -1.624168

    Avg Sharpe Change  Avg Volatility Change  Years Used
0          437.934908              12.337981           4
```

```python
# scatter of the “efficient frontier”
def plot_efficiency_hedge(etf, sens_yearly):
    # start fresh
    plt.figure(figsize=(12,4))

    # get sorted list of unique thresholds
    thr_values = sorted(sens_yearly['Thresh'].unique())
    # pick distinct colors from the same “viridis” palette
    colors = sns.color_palette("viridis", len(thr_values))
    thr_color = dict(zip(thr_values, colors))

    # scatter each point, mapping threshold → color
    sns.scatterplot(
        data=sens_yearly,
        x="Avg Drag", y="Avg VaR Red",
        hue="Thresh", style="De-risk",
        palette=thr_color, s=100
    )

    # now draw lines connecting the points of each threshold
    for thr in thr_values:
        df_thr = sens_yearly[sens_yearly['Thresh'] == thr] \
                         .sort_values("Avg Drag")
        plt.plot(
            df_thr["Avg Drag"], df_thr["Avg VaR Red"],
            color=thr_color[thr], linewidth=1
        )

    plt.title(f"{etf}: Per-Year Avg VaR Reduction vs Return Drag")
    plt.xlabel("Avg Return Drag (%)")
    plt.ylabel("Avg VaR Reduction (%)")

    # move legend outside
    plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left")
    plt.tight_layout()
    plt.show()
    plt.close()

plot_efficiency_hedge("ICLN", sens_yearly_ICLN)
plot_efficiency_hedge("PBW", sens_yearly_PBW)
plot_efficiency_hedge("QCLN", sens_yearly_QCLN)
```

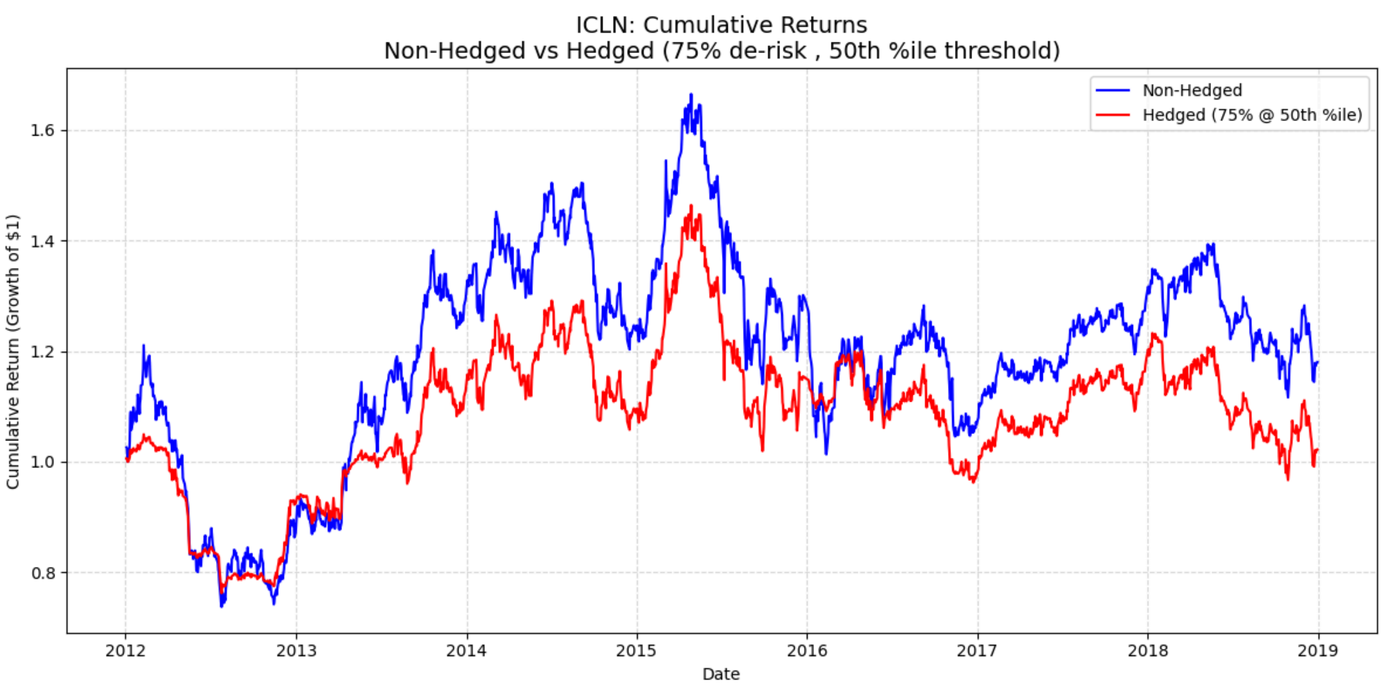






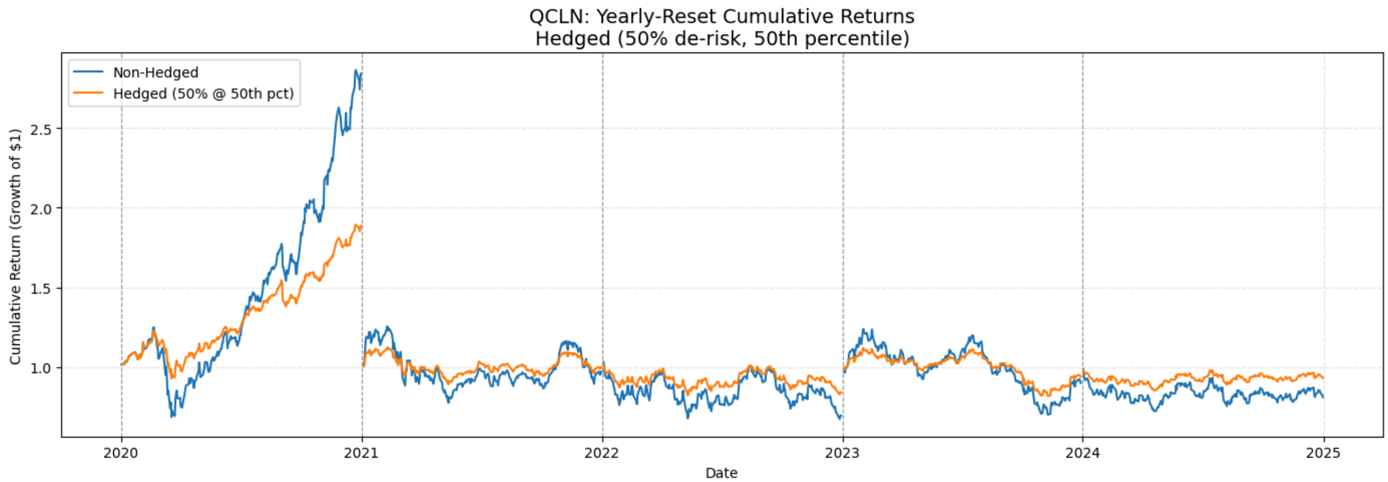

```
def plot_cumulative_returns(etf_name,
                            renewable_df,
                            garch_forecast,
                            start_date=None,
                            end_date=None,
                            vol_thr = 0,
                            de_risk=0):

    hedged_ret = compute_hedged_returns(
      renewable_df[etf_name],
      garch_forecast[etf_name],
      bond_returns["SHY"],
      vol_thr,
```

```
        de_risk
    )

    raw_ret_cropped = renewable_df[etf_name].loc[start_date:end_date]
    hedged_ret_cropped = hedged_ret.loc[start_date:end_date]

    # Compute cumulative returns
    cum_raw   = (1 + raw_ret_cropped).cumprod()
    cum_hedged= (1 + hedged_ret_cropped).cumprod()

    # Plot
    plt.figure(figsize=(12, 6))
    plt.plot(cum_raw.index, cum_raw, label="Non-Hedged", color="blue")

    plt.plot(cum_hedged.index, cum_hedged,
         label=f"Hedged ({de_risk*100:.0f}% @ {vol_thr*100:.0f}th %ile)",color="red")

    plt.title(f"{etf_name}: Cumulative Returns\nNon-Hedged vs Hedged ({de_risk*100:.0f}% de-risk , {vol_thr*100:.0f}th %ile thre
          fontsize=14)

    plt.xlabel("Date")
    plt.ylabel("Cumulative Return (Growth of $1)")
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.5)
    plt.tight_layout()
    plt.show()


plot_cumulative_returns("ICLN",
                          renewable_df,
                          garch_forecast,
                          start_date="2012",
                          end_date="2018",
                          vol_thr = 0.5,
                          de_risk=0.75)
```



```
hedged_daily = compute_hedged_returns(
    equity_ret = renewable_df['QCLN'],
    vol_forecast= garch_forecast["QCLN"],
    safe_ret    = bond_returns['SHY'],
    vol_thr     = 0.5,    # 90th percentile
    de_risk     = 0.25     # move 50% into bonds
)
pbw_plain = renewable_df['PBW'].loc['2011-01-01':'2019-12-31']
pbw_hedge = hedged_daily.loc['2011-01-01':'2019-12-31']


def plot_cumulative_returns_yearly(
    etf_name,
    renewable_df,
    garch_forecast,
    start_date=None,
    end_date=None,
    vol_thr=0.0,
    de_risk=0.0,
    figsize=(14,5)
):
    """
    Plot yearly-resetting cumulative returns for raw vs hedged strategies,
    with a vertical line at the start of each calendar year.
    """
    # build hedged series
    hedged = compute_hedged_returns(
        renewable_df[etf_name],
        garch_forecast[etf_name],
        bond_returns["SHY"],
        vol_thr,
        de_risk
    )
    raw = renewable_df[etf_name].loc[hedged.index]

    # crop to date window if requested
    if start_date:
        raw, hedged = raw.loc[start_date:], hedged.loc[start_date:]
    if end_date:
        raw, hedged = raw.loc[:end_date], hedged.loc[:end_date]

    years = sorted(raw.index.year.unique())
    fig, ax = plt.subplots(figsize=figsize)

    # plot each year's cum-prod
    for y in years:
        mask = raw.index.year == y
        dates = raw.index[mask]
        cum_raw = (1 + raw[mask]).cumprod()
        cum_hdg = (1 + hedged[mask]).cumprod()

        # only label the first year
        lbl_raw = "Non-Hedged" if y == years[0] else ""
        lbl_hdg = f"Hedged ({de_risk*100:.0f}% @ {vol_thr*100:.0f}th pct)" if y == years[0] else ""

        ax.plot(dates, cum_raw, color="C0", lw=1.5, label=lbl_raw)
        ax.plot(dates, cum_hdg, color="C1", lw=1.5, label=lbl_hdg)

    # vertical lines at Jan 1 of each year (including first)
    for y in years:
        ax.axvline(pd.Timestamp(f"{y}-01-01"), color="0.6", linestyle="--", lw=0.8)

    ax.set_title(
        f"{etf_name}: Yearly-Reset Cumulative Returns\n"
        f"Hedged ({de_risk*100:.0f}% de-risk, {vol_thr*100:.0f}th percentile)",
        fontsize=14
    )
    ax.set_xlabel("Date")
    ax.set_ylabel("Cumulative Return (Growth of $1)")
    ax.legend(loc="upper left")
    ax.grid(alpha=0.3, linestyle="--")
    plt.tight_layout()
    plt.show()


plot_cumulative_returns_yearly("QCLN",
                          renewable_df,
                          garch_forecast,
```

```
                    start_date="2020",
                    end_date="2025",
                    vol_thr = 0.5,
                    de_risk=0.5)
```



› old code

[ ] ↳ 10 cells hidden

› Attempts at Hidden Markov Model on Realized Volatilities

[ ] ↳ 15 cells hidden

⌄ USE: HMM Calculations

```
"""
def make_realized_vol_from_returns(ret_series: pd.Series,
                                   window: int = 20,
                                   trading_days: int = 252) -> pd.Series:
    rolling_sigma = ret_series.rolling(window=window).std(ddof=0)
    return rolling_sigma * np.sqrt(trading_days)
"""


def realised_vol_weekly(ret_series, window=20):
    # 1) daily realised σ
    daily_sigma = ret_series.rolling(window).std(ddof=0)
    # 2) pick last trading day of each ISO week
    wk_end = daily_sigma.groupby(daily_sigma.index.to_series().dt.isocalendar().week).tail(1)
    # 3) annualise to compare with weekly GARCH forecasts
    return wk_end * np.sqrt(252)
```

```
pip install hmmlearn
```

```
Collecting hmmlearn
  Downloading hmmlearn-0.3.3-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (3.0 kB)
Requirement already satisfied: numpy>=1.10 in /usr/local/lib/python3.11/dist-packages (from hmmlearn) (2.0.2)
Requirement already satisfied: scikit-learn!=0.22.0,>=0.16 in /usr/local/lib/python3.11/dist-packages (from hmmlearn) (1.6.1
Requirement already satisfied: scipy>=0.19 in /usr/local/lib/python3.11/dist-packages (from hmmlearn) (1.14.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn!=0.22.0,>=0.16->h
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn!=0.22.0,>=
Downloading hmmlearn-0.3.3-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (165 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 165.9/165.9 kB 2.8 MB/s eta 0:00:00
```

```
     Installing collected packages: hmmlearn
     Successfully installed hmmlearn-0.3.3
```

```
"""
import numpy as np
import pandas as pd
from hmmlearn.hmm import GaussianHMM

def predict_hmm_states(
    vol_series: pd.Series,
    window: int = 500,
    n_states: int = 3,
    hmm_iterations: int = 100,
    hmm_tolerance: float = 1e-4,
) -> pd.DataFrame:
    """
    Rolling-window HMM state classification, one day ahead.

    Parameters
    ----------
    vol_series : pd.Series
        Realised volatility (or any 1-D signal), indexed by datetime.
    window : int, default 500
        Number of look-back days used to fit the model each step.
    n_states : int, default 3
        Hidden-state count.
    hmm_iterations : int, default 100
        Maximum EM iterations per fit.
    hmm_tolerance : float, default 1e-4
        Convergence threshold for the HMM.

    Returns
    -------
    pd.DataFrame
        Columns: ["date", "state_pred"] – one row per day that was predicted.
    """
    # Ensure chronological ordering and drop missing / inf values
    vol_series = vol_series.sort_index()
    valid_mask = vol_series.replace([np.inf, -np.inf], np.nan).notna()
    vol_series = vol_series[valid_mask]

    dates   = vol_series.index
    values  = vol_series.values.reshape(-1, 1)

    pred_dates, state_preds = [], []

    # Step through the series, always using the *previous* `window` obs to fit
    for i in range(window, len(values)):
        train_X = values[i - window : i]        # 500-day window
        next_X  = values[i].reshape(-1, 1)      # obs we want to classify

        # Fit HMM on the rolling window
        hmm = GaussianHMM(
            n_components=n_states,
            covariance_type="diag",
            n_iter=hmm_iterations,
            tol=hmm_tolerance,
        )
        hmm.fit(train_X)

        # Predict hidden state for the "next day" observation
        state = hmm.predict(next_X)[0]

        pred_dates.append(dates[i])
        state_preds.append(state)

    return pd.DataFrame({"date": pred_dates, "state_pred": state_preds})
  """



"""
def map_states_by_mean_vol(

    vol_series: pd.Series,
    state_series: pd.Series | pd.DataFrame,
    labels: tuple[str, str, str] = ("low", "med", "high"),
`
```

```python
):
    """
    Identify which HMM state is Low / Med / High volatility.

    Parameters
    ----------
    vol_series   : pd.Series
        Realised volatility for ONE ETF (decimal or %), indexed by date.
    state_series : pd.Series | pd.DataFrame
        Hidden-state labels for the same ETF, indexed by date.
        If a DataFrame is passed it must have exactly one column.
    labels       : tuple[str,str,str]
        The qualitative labels in ascending-vol order.

    Returns
    -------
    label2id : dict[str, int]   e.g. {'low':0, 'med':2, 'high':1}
    id2label : dict[int, str]   e.g. {0:'low', 2:'med', 1:'high'}
    sorted_ids : list[int]      e.g. [0, 2, 1]  (ascending mean σ)
    """
    # --- accept either Series or single-col DataFrame ----------------------
    if isinstance(state_series, pd.DataFrame):
        if state_series.shape[1] != 1:
            raise ValueError("state_series DataFrame must have exactly one column.")
        state_series = state_series.iloc[:, 0]

    # --- align & clean -----------------------------------------------------
    idx = vol_series.index.intersection(state_series.index)
    vol  = vol_series.loc[idx]
    st   = state_series.loc[idx]

    mask = vol.replace([np.inf, -np.inf], np.nan).notna() & st.notna()
    vol, st = vol[mask], st[mask]

    if vol.empty:
        raise ValueError("No overlapping, non-NaN observations to evaluate.")

    # --- compute mean σ by state, then rank -------------------------------
    means = vol.groupby(st).mean().sort_values()   # ascending: low → high
    sorted_ids = means.index.to_list()

    label2id = {lbl: sid for lbl, sid in zip(labels, sorted_ids)}
    id2label = {sid: lbl for lbl, sid in label2id.items()}

    return label2id, id2label, sorted_ids
    """


import numpy as np
import pandas as pd
from hmmlearn.hmm import GaussianHMM

def predict_hmm_states_weekly(
    vol_series: pd.Series,
    *,
    window_days: int = 500,        # daily look-back window
    horizon_days: int = 5,         # one trading week ahead
    n_states: int = 3,
    hmm_iterations: int = 100,
    hmm_tolerance: float = 1e-4,
):
    """
    Fit a rolling Gaussian HMM on the most-recent `window_days` daily
    realised-vol observations and classify the hidden state *one week*
    (≈ `horizon_days`) ahead – but only once per ISO week.

    ►  The “week-end” anchor date is defined as the **last trading day
       appearing in that ISO calendar week**.
       If Friday is a holiday, Thursday (or Wednesday, …) is used.

    Returns
    -------
    state_df : pd.DataFrame
        Columns ['date','state_pred'] – one row per forecasted week.
        Each row’s `date` is the *target* trading day (t + horizon_days).
    label2id : dict[str, int]
        Mapping {'low': id_low, 'med': id_med, 'high': id_high}.
    id2label : dict[int, str]
```

```
        Reverse mapping {id_low:'low', …}.
    """

    # ─────────────────────────────────────────────────────────────────────
    # 0)  Clean input
    # ─────────────────────────────────────────────────────────────────────
    vol_series = (
        vol_series.sort_index()
                  .replace([np.inf, -np.inf], np.nan)
                  .dropna()
    )
    dates  = vol_series.index
    values = vol_series.values.reshape(-1, 1)

    # helper: is this index position the last trading day of its ISO week?
    def _is_week_end(idx: int) -> bool:
        if idx == len(dates) - 1:          # very last observation
            return True
        this_week = dates[idx].isocalendar()[:2]   # (ISO year, ISO week)
        next_week = dates[idx + 1].isocalendar()[:2]
        return this_week != next_week              # change ⇒ new week

    pred_dates, state_preds = [], []
    last_trainable = len(values) - horizon_days

    # ─────────────────────────────────────────────────────────────────────
    # 1)  Rolling estimation & week-ahead prediction
    # ─────────────────────────────────────────────────────────────────────
    for i in range(window_days, last_trainable):
        if not _is_week_end(i):                    # skip mid-week rows
            continue

        X_train   = values[i - window_days : i]                  # 500-day window
        X_target  = values[i + horizon_days].reshape(1, -1)      # vol at t+5

        model = GaussianHMM(
            n_components=n_states,
            covariance_type="diag",
            n_iter=hmm_iterations,
            tol=hmm_tolerance,
        ).fit(X_train)

        state = model.predict(X_target)[0]

        pred_dates.append(dates[i + horizon_days])  # stamp at target date
        state_preds.append(state)

    state_df = pd.DataFrame({"date": pred_dates, "state_pred": state_preds})

    # ─────────────────────────────────────────────────────────────────────
    # 2)  Map numeric states → 'low' / 'med' / 'high'
    # ─────────────────────────────────────────────────────────────────────
    joined = state_df.set_index("date").join(vol_series.rename("σ"))
    mean_by_state = joined.groupby("state_pred")["σ"].mean().sort_values()

    sorted_ids = mean_by_state.index.to_list()      # ascending σ
    label2id = {"low": sorted_ids[0],
                "med": sorted_ids[1],
                "high": sorted_ids[2]}
    id2label = {sid: lbl for lbl, sid in label2id.items()}

    return state_df, label2id, id2label


import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
import numpy as np
import pandas as pd

def plot_hmm_states_scatter(
    vol_series: pd.Series,
    state_series: pd.Series | pd.DataFrame,
    *,
```

```
    id2label: dict[int, str] | None = None,     # numeric-to-text map
    state_colors: dict[str, str] | None = None, # keyed by 'low','med','high'
    figsize: tuple[int, int] = (12, 4),
    title: str | None = None,
    ylabel: str = "Realised Volatility",
    ticker: str | None = None,
):
    """
    Grey daily volatility line + coloured weekly points.

    Works whether `state_series` is:
        • a Series indexed by weekday dates, or
        • the 2-column DataFrame returned by predict_hmm_states_weekly.
    """

    # Accept the 2-column DF shape
    if isinstance(state_series, pd.DataFrame):
        state_series = state_series.set_index("date")["state_pred"]

    # Make sure both indexes are datetime and sorted
    vol_series = vol_series.sort_index()
    state_series = state_series.sort_index()

    # Default colours
    default_colours = {"low": "green", "med": "orange", "high": "red"}

    # --- Colour & label helpers -------------------------------------------
    if id2label is None:                    # stick with numeric labels
        colour_cycle = plt.rcParams["axes.prop_cycle"].by_key()["color"]
        colour_func  = lambda s: colour_cycle[int(s) % len(colour_cycle)]
        legend_elems = [
            Line2D([0], [0], marker='o', lw=0, markersize=8,
                   markerfacecolor=colour_func(s), label=f"State {int(s)}")
            for s in np.sort(state_series.unique())
        ]
    else:                                   # use qualitative labels
        if state_colors is None:
            state_colors = default_colours
        colour_func = lambda s: state_colors[id2label[s]]
        legend_elems = [
            Line2D([0], [0], marker='o', lw=0, markersize=8,
                   markerfacecolor=state_colors[lbl], label=lbl.capitalize())
            for lbl in ["low", "med", "high"] if lbl in state_colors
        ]

    # --- Plot ---------------------------------------------------------------
    fig, ax = plt.subplots(figsize=figsize)

    # Grey daily line
    ax.plot(vol_series.index, vol_series, color="grey", lw=1, label="Realised Vol")

    # Coloured weekly scatter
    common_idx = state_series.index.intersection(vol_series.index)
    ax.scatter(
        common_idx,
        vol_series.loc[common_idx],
        c=[colour_func(s) for s in state_series.loc[common_idx]],
        s=8,
    )

    legend_elems.append(Line2D([0], [0], color="grey", lw=1, label="Realised Vol"))
    ax.legend(handles=legend_elems, frameon=False)

    ax.set_ylabel(ylabel)
    ax.set_xlabel("Date")
    if title is None:
        base = f"{ticker}: " if ticker else ""
        title = base + "HMM Regimes on Realised Volatility"
    ax.set_title(title)
    ax.grid(alpha=0.3)

    return ax


import pandas as pd

def hedge_by_hmm_state(
```

```python
    equity_ret: pd.Series,
    state_series: pd.Series | pd.DataFrame,
    safe_ret: pd.Series,
    *,
    de_risk_med: float = 0.30,
    de_risk_high: float = 0.60,
    id2label: dict[int, str] | None = None,
) -> pd.Series:
    """
    Blend equity & safe-asset returns according to HMM regime labels.

    * Works when `state_series` is daily, weekly, or monthly.
    * Forward-fills the most recent state so every trading day gets a weight.

    Parameters
    ----------
    equity_ret : pd.Series   – daily equity returns (decimal).
    state_series : pd.Series | pd.DataFrame
        HMM numeric IDs or already-text labels, indexed by date.
        If a DataFrame (from predict_hmm_states*), it must have ['date','state_pred'].
    safe_ret : pd.Series     – daily safe-asset returns (decimal).
    de_risk_med, de_risk_high : float   – % shifted into bonds in MED / HIGH states.
    id2label : dict[int,str] | None
        Mapping {numeric_id : 'low'/'med'/'high'} if `state_series` is numeric.

    Returns
    -------
    pd.Series – hedged daily returns, aligned with `equity_ret`.
    """
    # Accept the 2-column DF shape
    if isinstance(state_series, pd.DataFrame):
        state_series = state_series.set_index("date")["state_pred"]

    # —— 1.  Align daily equity & bond returns ——————————————————————————————
    idx = equity_ret.index.intersection(safe_ret.index)
    eq  = equity_ret.loc[idx]
    sf  = safe_ret.loc[idx]

    # —— 2.  Prepare the state labels at daily frequency ————————————————————
    st = state_series

    # Map numeric → text if needed
    if id2label is not None:
        st = st.map(id2label)

    # Reindex to daily calendar, forward-filling the last known state
    st_daily = (
        st.reindex(idx, method="ffill")      # carry regime forward
          .fillna(method="bfill")            # in case the first few days miss
    )

    # Sanity check
    if not {"low", "med", "high"}.issubset(set(st_daily.unique())):
        raise ValueError("State labels must resolve to 'low', 'med', 'high'.")

    # —— 3.  Build dynamic equity weight per day ————————————————————————————
    w_eq = pd.Series(1.0, index=idx)              # default 100 % equity
    w_eq[st_daily == "med"]  = 1.0 - de_risk_med
    w_eq[st_daily == "high"] = 1.0 - de_risk_high

    # —— 4.  Blend returns ——————————————————————————————————————————————————
    hedged = w_eq * eq + (1.0 - w_eq) * sf
    hedged.name = "hedged_return"
    return hedged



# example
hedged_icln = hedge_by_hmm_state(
    equity_ret   = renewable_df["ICLN"],   # daily returns
    state_series = states_hmm["ICLN"],     # weekly IDs (0/1/2)
    safe_ret     = bond_returns["SHY"],    # daily bond returns
    de_risk_med  = 0.40,
    de_risk_high = 0.80,
    id2label     = id2label_ICLN,
)
```

```python
def plot_cumulative_returns_HMM(
    etf_name: str,
    renewable_df: pd.DataFrame,
    hedged_returns: pd.Series,
    *,
    start_date: str | pd.Timestamp = None,
    end_date:   str | pd.Timestamp = None,
    de_risk_med:  float | None = None,
    de_risk_high: float | None = None,
):
    """
    Plot cumulative-return paths for an ETF vs. its HMM-hedged version.

    Parameters
    ----------
    etf_name        : str               – column name in `renewable_df`.
    renewable_df    : pd.DataFrame      – daily returns (decimal).
    hedged_returns  : pd.Series         – hedged daily returns (decimal).
    start_date,end_date : optional      – date window (inclusive).
    de_risk_med     : float | None      – % shifted into bonds in MED regime.
    de_risk_high    : float | None      – % shifted into bonds in HIGH regime.
    """
    # slice to window
    raw_ret_cropped   = renewable_df[etf_name].loc[start_date:end_date]
    hedged_cropped    = hedged_returns.loc[start_date:end_date]

    # cumulative growth of $1
    cum_raw    = (1 + raw_ret_cropped).cumprod()
    cum_hedged = (1 + hedged_cropped).cumprod()

    # build the title with de-risk info if provided
    title = f"{etf_name}: Cumulative Returns – Non-Hedged vs. HMM-Hedged"
    if (de_risk_med is not None) and (de_risk_high is not None):
        title += f"\n(Hedge: {int(de_risk_med*100)}% in MED, {int(de_risk_high*100)}% in HIGH)"

    # plot
    plt.figure(figsize=(12, 6))
    plt.plot(cum_raw.index, cum_raw, label="Non-Hedged", color="blue")
    plt.plot(cum_hedged.index, cum_hedged, label="HMM-Hedged", color="red", linestyle="--")

    plt.title(title, fontsize=14)
    plt.xlabel("Date")
    plt.ylabel("Growth of $1")
    plt.legend()
    plt.grid(alpha=0.4, linestyle="--")
    plt.tight_layout()
    plt.show()


def compute_risk_metrics(
    test_returns: pd.Series,
    test_hedged:  pd.Series,
    start: str | pd.Timestamp,
    end:   str | pd.Timestamp
) -> pd.DataFrame:
    # –– 1. slice to the desired window ––––––––––––––––––––––––––––––––––––––
    start = pd.to_datetime(start)
    end   = pd.to_datetime(end)

    unhedged = test_returns.loc[start:end]
    hedged   = test_hedged .loc[start:end]

    if unhedged.empty or hedged.empty:
        raise ValueError("No data in the specified date range.")

    # ensure equal dates
    idx = unhedged.index.intersection(hedged.index)
    unhedged, hedged = unhedged.loc[idx], hedged.loc[idx]

    # –– 2. helper for the three metrics –––––––––––––––––––––––––––––––––––––
    def _metrics(r):
        var  = r.quantile(0.05)
        cvar = r[r <= var].mean()
        # Max drawdown on cumulative return path
        cum = (1 + r).cumprod()
        mdd = (cum / cum.cummax() - 1).min()
        return var, cvar, mdd
```

```python
    orig_var, orig_cvar, orig_mdd = _metrics(unhedged)
    hedg_var, hedg_cvar, hedg_mdd = _metrics(hedged)

    metrics_orig   = np.array([orig_var,  orig_cvar,  orig_mdd])
    metrics_hedged = np.array([hedg_var, hedg_cvar, hedg_mdd])
    pct_change     = (metrics_hedged - metrics_orig) / metrics_orig * 100.0

    out = pd.DataFrame(
        {
            "Original":         metrics_orig.round(4),
            "Hedged":           metrics_hedged.round(4),
            "Percent Change (%)": pct_change.round(2),
        },
        index=["VaR (95%)", "CVaR (95%)", "Max Drawdown"],
    )
    return out


def single_year_metrics_HMM(
    unhedged_returns: pd.Series,
    hedged_returns: pd.Series,
    year: int,
    min_obs: int = 50,
) -> dict | None:
    """
    Same metrics as before, but all values are *improvements*:
        • +Drag  -> hedged return is higher
        • +VaR/ CVaR/ MDD Reductions -> smaller tail risk
        • +Sharpe  -> higher risk-adjusted return
        • +Vol Change -> lower volatility
    """
    # 1) align & slice calendar year ----------------------------------------
    idx = unhedged_returns.index.intersection(hedged_returns.index)
    raw   = unhedged_returns.loc[idx]
    hedg  = hedged_returns.loc[idx]

    mask = raw.index.year == year
    raw, hedg = raw[mask], hedg[mask]

    if len(raw) < min_obs:
        return None

    # 2) Return improvement -------------------------------------------------
    R_raw = (1 + raw ).cumprod().iloc[-1]
    R_hdg = (1 + hedg).cumprod().iloc[-1]
    drag_improve = (R_hdg - R_raw) / abs(R_raw) * 100.0   # + is good

    # 3) VaR & CVaR improvements --------------------------------------------
    var_raw, var_hdg = raw.quantile(0.05), hedg.quantile(0.05)
    cvar_raw = raw [raw  <= var_raw ].mean()
    cvar_hdg = hedg[hedg <= var_hdg].mean()

    var_improve  = (abs(var_raw)  - abs(var_hdg))  / abs(var_raw)  * 100.0
    cvar_improve = (abs(cvar_raw) - abs(cvar_hdg)) / abs(cvar_raw) * 100.0

    # 4) Max-drawdown improvement -------------------------------------------
    def _mdd(x):
        path = (1 + x).cumprod()
        return (path / path.cummax() - 1).min()

    mdd_raw, mdd_hdg = _mdd(raw), _mdd(hedg)
    mdd_improve = (abs(mdd_raw) - abs(mdd_hdg)) / abs(mdd_raw) * 100.0

    # 5) Sharpe improvement (rf = 0) ----------------------------------------
    sharpe_raw = raw.mean()  / raw.std(ddof=0)  * np.sqrt(252)
    sharpe_hdg = hedg.mean() / hedg.std(ddof=0) * np.sqrt(252)
    sharpe_improve = (
        (sharpe_hdg - sharpe_raw) / abs(sharpe_raw) * 100.0
        if sharpe_raw != 0 else np.nan
    )

    # 6) Volatility improvement ---------------------------------------------
    vol_improve = (raw.std(ddof=0) - hedg.std(ddof=0)) / raw.std(ddof=0) * 100.0

    return {
        "Return Improvement":       drag_improve,     # + better
```

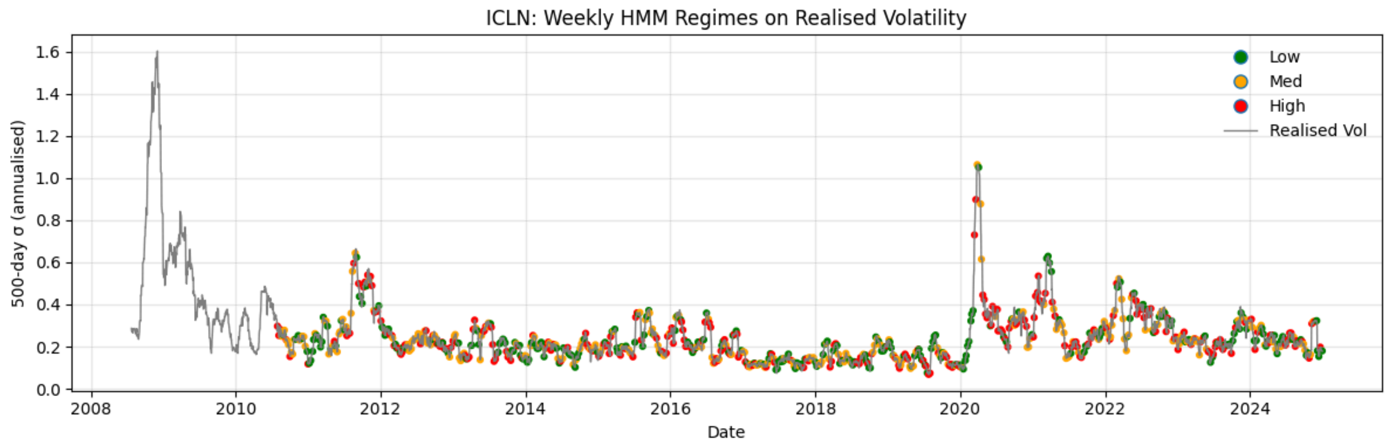
```python
        "VaR Reduction":            var_improve,      # + better
        "CVaR Reduction":           cvar_improve,     # + better
        "Max Drawdown Reduction":   mdd_improve,      # + better
        "Sharpe Ratio Improvement": sharpe_improve,   # + better
        "Volatility Reduction":     vol_improve,      # + better
    }


import pandas as pd
from itertools import product

def yearly_sensitivity_averages_HMM(
    rets: pd.Series,
    states: pd.Series,
    bond_rets: pd.Series,
    years: list[int],
    med_list: list[float],
    high_list: list[float],
    *,
    id2label: dict[int, str] | None = None,
    min_obs: int = 50,
) -> pd.DataFrame:
    """
    Grid-search (de_risk_med, de_risk_high) and report the
    *average yearly improvements* vs. the unhedged series.
    Positive numbers are good (hedge is better).

    Returns
    -------
    pd.DataFrame
        One row per (med, high) pair with average improvements.
    """

    # Align once ------------------------------------------------------------
    idx = rets.index.intersection(states.index).intersection(bond_rets.index)
    rets   = rets.loc[idx]
    states = states.loc[idx]
    bonds  = bond_rets.loc[idx]

    records = []

    for med, high in product(med_list, high_list):

        # Build the hedged series for this parameter pair
        hedged = hedge_by_hmm_state(
            equity_ret   = rets,
            state_series = states,
            safe_ret     = bonds,
            de_risk_med  = med,
            de_risk_high = high,
            id2label     = id2label,
        )

        yearly = []
        for y in years:
            m = single_year_metrics_HMM(
                unhedged_returns = rets,
                hedged_returns   = hedged,
                year             = y,
                min_obs          = min_obs,
            )
            if m is not None:
                yearly.append(m)

        if not yearly:        # no valid years
            continue

        dfy = pd.DataFrame(yearly)

        records.append({
            "De-risk Med (%)":        int(med * 100),
            "De-risk High (%)":       int(high * 100),
            "Avg Return Improve":     dfy["Return Improvement"].mean(),
            "Avg VaR Reduction":      dfy["VaR Reduction"].mean(),
            "Avg CVaR Reduction":     dfy["CVaR Reduction"].mean(),
            "Avg MDD Reduction":      dfy["Max Drawdown Reduction"].mean(),
            "Avg Sharpe Improve":     dfy["Sharpe Ratio Improvement"].mean(),
            "Avg Vol Reduction":      dfy["Volatility Reduction"].mean(),
```

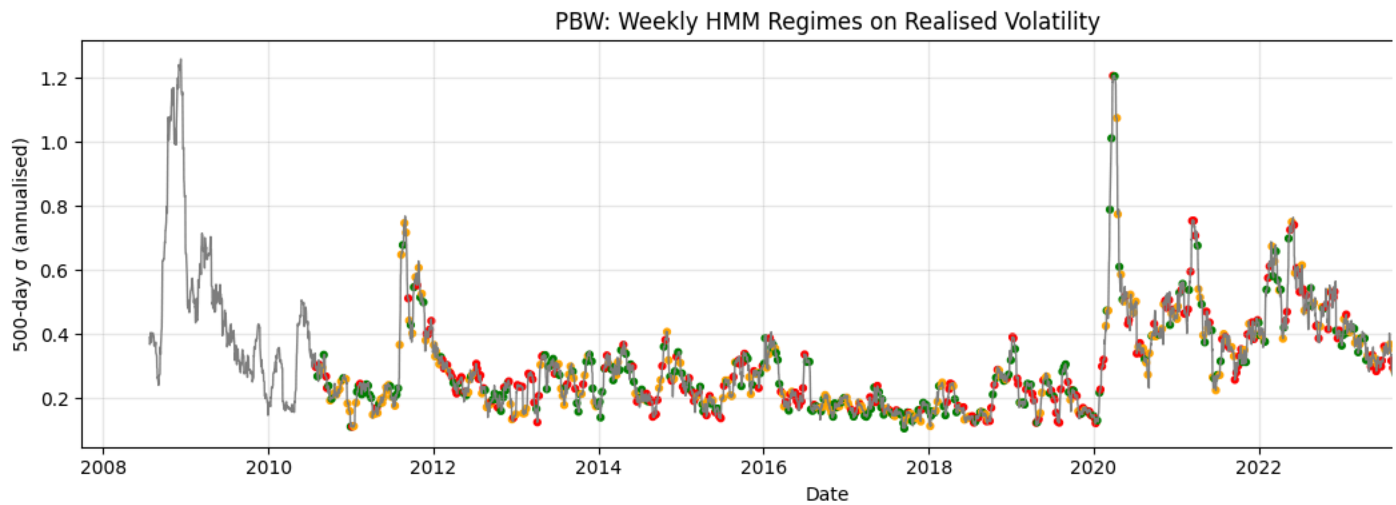
```
            "Years Used":             len(dfy),
        })

    return pd.DataFrame(records)



import numpy as np
import pandas as pd
from itertools import product

def overall_sensitivity_HMM(
    rets: pd.Series,
    states: pd.Series,
    bond_rets: pd.Series,
    med_list: list[float],
    high_list: list[float],
    *,
    start: str | pd.Timestamp = None,
    end:   str | pd.Timestamp = None,
    id2label: dict[int, str] | None = None,
    min_obs: int = 250,          # require at least ~1 year of data
) -> pd.DataFrame:
    """
    Evaluate hedging effectiveness over a chosen date window.

    Parameters
    ----------
    rets, states, bond_rets : pd.Series
        Daily data (decimal) indexed by date.
    med_list, high_list     : list[float]
        Grid of de-risk % for MED and HIGH volatility regimes.
    start, end              : str | pd.Timestamp | None
        Date window (inclusive).  Pass None for full range.
    id2label                : mapping numeric ID → 'low/med/high' (optional).
    min_obs                 : int
        Minimum overlapping observations; raises if fewer.

    Returns
    -------
    pd.DataFrame  – one row per (med, high) pair with overall risk deltas.
    """
    # ── 1. align & slice to window ──────────────────────────────────────────
    idx = rets.index.intersection(states.index).intersection(bond_rets.index)

    if start is not None:
        idx = idx[idx >= pd.to_datetime(start)]
    if end is not None:
        idx = idx[idx <= pd.to_datetime(end)]

    if len(idx) < min_obs:
        raise ValueError("Too few overlapping dates in the specified window.")

    r     = rets.loc[idx]
    s     = states.loc[idx]
    bonds = bond_rets.loc[idx]

    # ── 2. helper to compute risk-metric deltas ─────────────────────────────
    def _risk_metrics(u, h):
        drag = ((1 + u).cumprod().iloc[-1] - (1 + h).cumprod().iloc[-1]) \
               / (1 + u).cumprod().iloc[-1] * 100.0
        var_u, var_h = u.quantile(0.05), h.quantile(0.05)
        cvar_u = u[u <= var_u].mean()
        cvar_h = h[h <= var_h].mean()

        def _mdd(x):
            cum = (1 + x).cumprod()
            return (cum / cum.cummax() - 1).min()
        mdd_u, mdd_h = _mdd(u), _mdd(h)

        sharpe_u = u.mean() / u.std(ddof=0) * np.sqrt(252)
        sharpe_h = h.mean() / h.std(ddof=0) * np.sqrt(252)
        sharpe_delta = (sharpe_h - sharpe_u) / abs(sharpe_u) * 100.0 \
                       if sharpe_u != 0 else np.nan

        vol_delta = (u.std(ddof=0) - h.std(ddof=0)) / u.std(ddof=0) * 100.0
```

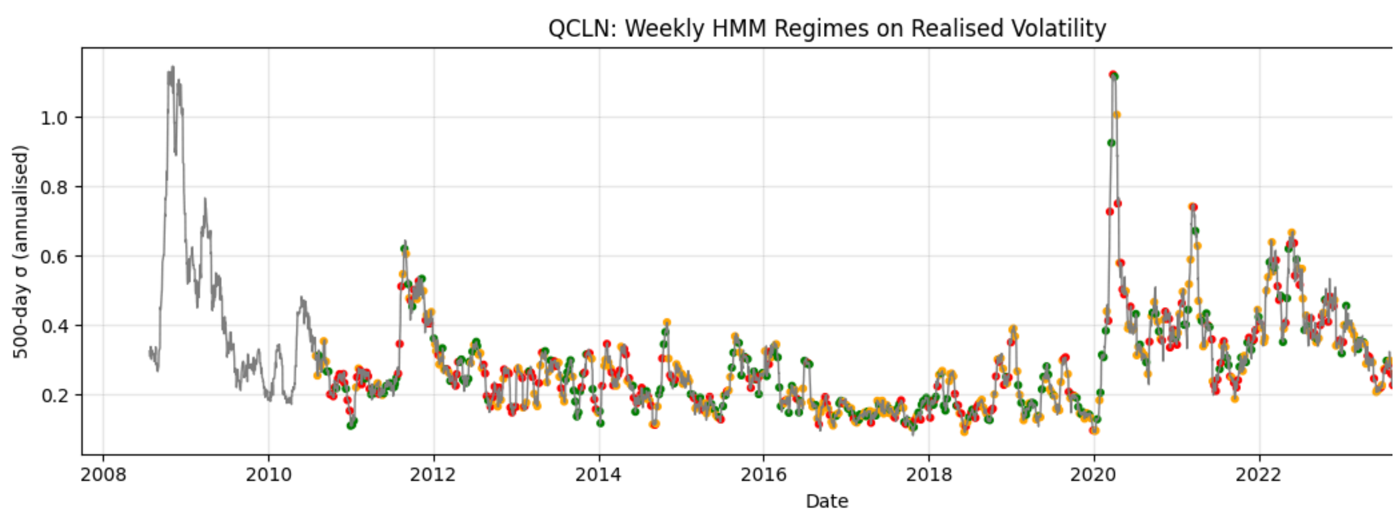
```
        return {
            "Drag": drag,
            "VaR Red": (var_u - var_h) / abs(var_u) * 100.0,
            "CVaR Red": (cvar_u - cvar_h) / abs(cvar_u) * 100.0,
            "MDD Red": (mdd_u - mdd_h) / abs(mdd_u) * 100.0,
            "Sharpe Δ": sharpe_delta,
            "Vol Δ": vol_delta,
        }

    # ── 3. grid-search over (med, high) pairs ──────────────────────────────
    rows = []
    for med, high in product(med_list, high_list):
        hedged = hedge_by_hmm_state(
            equity_ret   = r,
            state_series = s,
            safe_ret     = bonds,
            de_risk_med  = med,
            de_risk_high = high,
            id2label     = id2label,
        )
        metrics = _risk_metrics(r, hedged)
        rows.append({
            "De-risk Med (%)":   int(med * 100),
            "De-risk High (%)":  int(high * 100),
            **metrics,
            "Obs Used": len(idx)
        })

    return pd.DataFrame(rows)


def realised_vol_daily(ret_series, window=20, annualise=True):
    sigma = ret_series.rolling(window).std(ddof=0)
    return sigma * np.sqrt(252) if annualise else sigma

realised_vol_daily_ICLN  = realised_vol_daily(renewable_df["ICLN"]).dropna()
realised_vol_daily_PBW   = realised_vol_daily(renewable_df["PBW"]).dropna()
realised_vol_daily_QCLN  = realised_vol_daily(renewable_df["QCLN"]).dropna()

# Predict weekly states one week ahead
state_df_ICLN, label2id_ICLN, id2label_ICLN = predict_hmm_states_weekly(realised_vol_daily_ICLN)
state_df_PBW , label2id_PBW , id2label_PBW = predict_hmm_states_weekly(realised_vol_daily_PBW)
state_df_QCLN, label2id_QCLN, id2label_QCLN = predict_hmm_states_weekly(realised_vol_daily_QCLN)
```

```
WARNING:hmmlearn.base:Model is not converging.  Current: 732.325952970463 is not greater than 732.3260447994006. Delta is -9
WARNING:hmmlearn.base:Model is not converging.  Current: 734.3425687533911 is not greater than 734.342638747576. Delta is -6
WARNING:hmmlearn.base:Model is not converging.  Current: 724.3743651825123 is not greater than 724.374500764277. Delta is -0
WARNING:hmmlearn.base:Model is not converging.  Current: 723.2747697357344 is not greater than 723.2806536227769. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 718.1074805303662 is not greater than 718.1081390450115. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 716.3703405658262 is not greater than 716.3827086621479. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 723.4022273601724 is not greater than 723.4027801930108. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 727.3286602437908 is not greater than 727.3309158872785. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 822.2117494747085 is not greater than 822.2137685281517. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 829.6025058428884 is not greater than 829.6132335875628. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 832.7005980600132 is not greater than 832.7192618634954. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 698.4491032269303 is not greater than 698.4492006576463. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 879.0916604756782 is not greater than 879.1024029247127. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 872.1642922614382 is not greater than 872.1674079843893. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 878.1057538081496 is not greater than 878.1117370110051. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 677.2573166764116 is not greater than 677.2573641188328. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 658.8443220270095 is not greater than 658.8446192322012. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 842.5138988440569 is not greater than 842.5528546962447. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 838.3496584754843 is not greater than 838.3587787891801. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 640.8625956577954 is not greater than 640.8631950155874. Delta is -
WARNING:hmmlearn.base:Model is not converging.  Current: 818.3475343690438 is not greater than 818.3598535567135. Delta is -
```

```
# combine on dates
state_df_ICLN = state_df_ICLN.set_index('date')
state_df_PBW = state_df_PBW.set_index('date')
state_df_QCLN = state_df_QCLN.set_index('date')
realized_vols = pd.concat([realised_vol_daily_ICLN, realised_vol_daily_PBW, realised_vol_daily_QCLN], axis=1)
states_hmm = pd.concat([state_df_ICLN, state_df_PBW, state_df_QCLN], axis=1)

# rename to ICLN, PBW, and QCLN
realized_vols.columns = ['ICLN', 'PBW', 'QCLN']
states_hmm.columns = ['ICLN', 'PBW', 'QCLN']

states_hmm.index     = pd.to_datetime(states_hmm.index)
realized_vols.index  = pd.to_datetime(realized_vols.index)


id2label = {
    "ICLN": id2label_ICLN,
    "PBW" : id2label_PBW,
    "QCLN": id2label_QCLN,
}


# save states_hmm to a csv file so we don´t have to run the HMM every time
states_hmm.to_csv("states_hmm.csv", index=True)


# read states_hmm csv file if we haven´t ran it here:
states_hmm = pd.read_csv("states_hmm.csv").set_index("date")


plot_hmm_states_scatter(
    realised_vol_daily_ICLN,
    states_hmm["ICLN"],
    id2label=id2label_ICLN,
    ticker="ICLN",
    ylabel="500-day σ (annualised)"
)

plot_hmm_states_scatter(
    realised_vol_daily_PBW,
    states_hmm["PBW"],
    id2label=id2label_PBW,
    ticker="PBW",
    ylabel="500-day σ (annualised)"
)

plot_hmm_states_scatter(
    realised_vol_daily_QCLN,
    states_hmm["QCLN"],
    id2label=id2label_QCLN,
    ticker="QCLN",
    ylabel="500-day σ (annualised)"
)
```

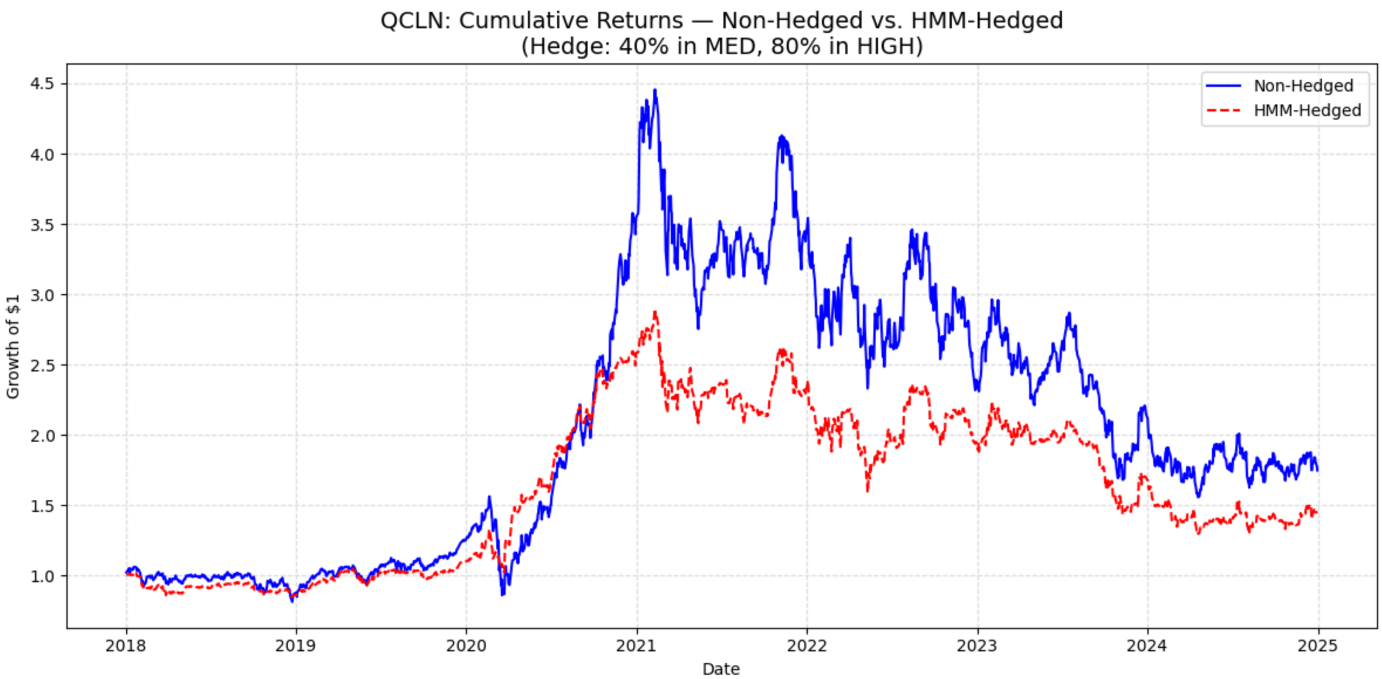







```
etf = "QCLN"
de_risk_med = 0.4
de_risk_high = 0.8

hedged = hedge_by_hmm_state(
    equity_ret   = renewable_df[etf],   # daily returns
    state_series = states_hmm[etf],     # weekly IDs (0/1/2)
    safe_ret     = bond_returns["SHY"],    # daily bond returns
    de_risk_med  = de_risk_med,
    de_risk_high = de_risk_high,
    id2label     = id2label[etf],          # {0:'high',1:'med',2:'low'} for example
)
```

```
plot_cumulative_returns_HMM(etf,
                            renewable_df,
                            hedged,
                            start_date="2018",
                            end_date="2025",
                            de_risk_med = de_risk_med,
                            de_risk_high = de_risk_high
                            )
```




```
single_year_metrics_HMM(renewable_df["ICLN"],hedged_icln,2012)
```

```
{'Return Drag': np.float64(-3.805342871674179),
 'VaR Reduction': np.float64(-24.00299304654105),
 'CVaR Reduction': np.float64(-19.286465173953246),
 'Max Drawdown Reduction': -25.425001525372455,
 'Sharpe Ratio Change': np.float64(-20.682222970486126),
 'Volatility Change': 33.7197227881433}
```

```
grid_df_ICLN_yearly_avg = yearly_sensitivity_averages_HMM(
    rets       = renewable_df["ICLN"],
    states     = states_hmm["ICLN"],
    bond_rets  = bond_returns["SHY"],
    years      = list(range(2018, 2025)),
    med_list   = [0.25, 0.40, 0.60],
    high_list  = [0.50, 0.75, 0.90],
    id2label   = id2label_ICLN  # from map_states_by_mean_vol
)

grid_df_ICLN_yearly_avg
```

	De-risk Med (%)	De-risk High (%)	Avg Return Improve	Avg VaR Reduction	Avg CVaR Reduction	Avg MDD Reduction	Avg Sharpe Improve	Avg Vol Reduction	Years Used
0	25	50	0.698974	24.159238	18.068008	23.596131	42.652206	23.753795	7
1	25	75	1.184621	30.385141	20.446515	29.737356	117.232585	29.271562	7
2	25	90	1.488975	31.111317	20.582263	29.747511	165.149089	31.013177	7
3	40	50	0.552206	26.788400	21.287623	25.238223	-8.010084	27.268481	7
4	40	75	1.022835	33.754486	23.666130	31.864953	67.997277	33.092765	7
5	40	90	1.317985	35.214380	23.801878	32.493927	118.814142	34.937306	7
6	60	50	0.355064	31.141941	24.161766	27.409864	-86.559452	30.976647	7
7	60	75	0.805874	39.132085	27.088115	34.305530	-12.678171	37.171607	7
8	60	90	1.088875	41.409799	27.223863	35.992416	40.615460	39.142044	7

Next steps: Generate code with grid_df_ICLN_yearly_avg | View recommended plots | New interactive sheet

Start coding or generate with AI.

	QCLN
date	
2010-07-30	0
2010-08-06	0
2010-08-13	1
2010-08-20	0
2010-08-27	1
...	...
2024-12-02	2
2024-12-06	1
2024-12-13	1
2024-12-20	1
2024-12-30	2

753 rows × 1 columns

dtype: int64

```
grid_df_ICLN_overall = overall_sensitivity_HMM(
    rets       = renewable_df["ICLN"],
    states     = states_hmm["ICLN"],
    bond_rets  = bond_returns["SHY"],
    med_list   = [0.25, 0.40, 0.60],
    high_list  = [0.50, 0.75, 0.90],
    start      = "2020-01-01",
    end        = "2024-12-31",
    id2label   = id2label["ICLN"]
)
grid_df_ICLN_overall
```

	De-risk Med (%)	De-risk High (%)	Drag	VaR Red	CVaR Red	MDD Red	Sharpe Δ	Vol Δ	Obs Used
0	25	50	-5.408088	-23.925839	-23.022010	-31.149039	-7.574373	28.068087	261
1	25	75	-8.212051	-29.744647	-26.465520	-33.859079	4.686507	34.765787	261
2	25	90	-9.804387	-29.744647	-26.465520	-33.847682	16.067366	36.788556	261
3	40	50	-4.899682	-32.259755	-25.898193	-31.078555	-20.534962	31.926436	261
4	40	75	-7.690121	-36.182625	-30.194261	-35.959143	-9.670343	39.045374	261
5	40	90	-9.274776	-36.182625	-30.194261	-35.947672	1.800437	41.214434	261
6	60	50	-4.148917	-32.259755	-28.151024	-30.607869	-37.870354	35.967407	261
7	60	75	-6.919385	-36.182625	-32.447091	-37.344764	-29.554296	43.592248	261
8	60	90	-8.492700	-36.182625	-32.447091	-38.374868	-18.311913	45.942309	261

Next steps: Generate code with grid_df_ICLN_overall | View recommended plots | New interactive sheet

Comparison to "placebo" (de-risking at random days)

Double-click (or enter) to edit

```
import random

# ------------------------------------------------------------------ helpers
def _extract_blocks(label_series, active_set):
    """Return [(length, label), ...] for consecutive active blocks."""
    blocks, current_len = [], 0
    current_lbl = None

    for lbl in label_series:
        if lbl in active_set:
            if lbl == current_lbl:
                current_len += 1
            else:
                if current_len:
                    blocks.append((current_len, current_lbl))
                current_len, current_lbl = 1, lbl
        else:
            if current_len:
                blocks.append((current_len, current_lbl))
                current_len = 0
                current_lbl = None
    if current_len:
        blocks.append((current_len, current_lbl))
    return blocks


def _place_blocks_once(n_weeks, blocks, rng):
    """
    Attempt to lay down all blocks in random order. Returns a char array
    or None if no non-overlapping placement is possible.
    """
    out = np.full(n_weeks, "low", dtype=object)
    occupied = np.zeros(n_weeks, dtype=bool)

    # make a copy and shuffle in-place with stdlib.random (avoids dtype issues)
    blocks_shuffled = blocks.copy()
    random.shuffle(blocks_shuffled)

    for length, lbl in blocks_shuffled:
        length = int(length)                 # ensure numeric
        possible = np.where(~occupied)[0]
        possible = possible[possible <= n_weeks - length]
        possible = [p for p in possible if not occupied[p : p + length].any()]
        if not possible:
            return None
        start = rng.choice(possible)
        out[start : start + length] = lbl
        occupied[start : start + length] = True
```

```python
    return out


def _place_blocks_randomly(n_weeks, blocks, rng):
    """
    Place blocks into an empty calendar of length n_weeks without overlap.
    Returns an array of labels ('low','med','high'), default 'low'.
    """
    out = np.full(n_weeks, "low", dtype=object)
    occupied = np.zeros(n_weeks, dtype=bool)

    for _, length, lbl in blocks:
        # find all start positions that fit
        possible = np.where(~occupied)[0]
        possible = possible[possible <= n_weeks - length]
        possible = [p for p in possible if not occupied[p : p + length].any()]
        start = rng.choice(possible)
        out[start : start + length] = lbl
        occupied[start : start + length] = True

    return out

def random_timing_schedules(label_series, n_draws=1000, seed=0):
    """
    Produce placebo schedules that (a) keep block lengths if possible,
    else (b) fall back to a simple random permutation of the weekly labels.
    """
    rng = np.random.default_rng(seed)
    is_numeric = np.issubdtype(label_series.dtype, np.number)
    active = {1, 2} if is_numeric else {"med", "high"}     # active states

    blocks = _extract_blocks(label_series, active)
    n_weeks = len(label_series)

    schedules = []

    for _ in range(n_draws):
        sched = None
        # up to 100 placement attempts
        for _try in range(100):
            sched = _place_blocks_once(n_weeks, blocks, rng)
            if sched is not None:
                break
        if sched is None:  # still no fit → fallback: permute whole vector
            sched = label_series.sample(frac=1, random_state=rng).values
        schedules.append(pd.Series(sched, index=label_series.index))

    return schedules



grid_df_ICLN_yearly_avg = yearly_sensitivity_averages_HMM(
    rets       = renewable_df["ICLN"],
    states     = states_hmm["ICLN"],
    bond_rets  = bond_returns["SHY"],
    years      = list(range(2018, 2025)),
    med_list   = [0.25, 0.40, 0.60],
    high_list  = [0.50, 0.75, 0.90],
    id2label   = id2label_ICLN  # from map_states_by_mean_vol
)

grid_df_ICLN_yearly_avg
```

	De-risk Med (%)	De-risk High (%)	Avg Return Improve	Avg VaR Reduction	Avg CVaR Reduction	Avg MDD Reduction	Avg Sharpe Improve	Avg Vol Reduction	Years Used
0	25	50	0.698974	24.159238	18.068008	23.596131	42.652206	23.753795	7
1	25	75	1.184621	30.385141	20.446515	29.737356	117.232585	29.271562	7
2	25	90	1.488975	31.111317	20.582263	29.747511	165.149089	31.013177	7
3	40	50	0.552206	26.788400	21.287623	25.238223	-8.010084	27.268481	7
4	40	75	1.022835	33.754486	23.666130	31.864953	67.997277	33.092765	7
5	40	90	1.317985	35.214380	23.801878	32.493927	118.814142	34.937306	7
6	60	50	0.355064	31.141941	24.161766	27.409864	-86.559452	30.976647	7
7	60	75	0.805874	39.132085	27.088115	34.305530	-12.678171	37.171607	7
8	60	90	1.088875	41.409799	27.223863	35.992416	40.615460	39.142044	7

Next steps: Generate code with grid_df_ICLN_yearly_avg | View recommended plots | New interactive sheet

```
def overall_metrics(
    unhedged_returns: pd.Series,
    hedged_returns: pd.Series,
) -> dict:
    """
    Full-period hedge effectiveness.  All outputs are IMPROVEMENTS:
        +Return  +VaR/CVaR/MDD  +Sharpe  +Vol  ⇒ hedge is better.

    Parameters
    ----------
    unhedged_returns, hedged_returns : pd.Series
        Daily returns (decimal) with a common DatetimeIndex.

    Returns
    -------
    dict
        {
          'Return Improvement', 'VaR Reduction', 'CVaR Reduction',
          'Max Drawdown Reduction', 'Sharpe Ratio Improvement',
          'Volatility Reduction'
        }   # all in percent
    """
    # –– align ––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
    idx   = unhedged_returns.index.intersection(hedged_returns.index)
    raw   = unhedged_returns.loc[idx]
    hedg  = hedged_returns.loc[idx]

    # –– Return improvement (cum) ––––––––––––––––––––––––––––––––––––––
    R_raw = (1 + raw ).cumprod().iloc[-1]
    R_hdg = (1 + hedg).cumprod().iloc[-1]
    ret_improve = (R_hdg - R_raw) / abs(R_raw) * 100.0

    # –– VaR & CVaR (95 %) improvements ––––––––––––––––––––––––––––––––
    var_raw, var_hdg = raw.quantile(0.05), hedg.quantile(0.05)
    cvar_raw = raw [raw  <= var_raw ].mean()
    cvar_hdg = hedg[hedg <= var_hdg].mean()

    var_improve  = (abs(var_raw)  - abs(var_hdg))  / abs(var_raw)  * 100.0
    cvar_improve = (abs(cvar_raw) - abs(cvar_hdg)) / abs(cvar_raw) * 100.0

    # –– Max-drawdown improvement ––––––––––––––––––––––––––––––––––––––
    def _mdd(r):
        path = (1 + r).cumprod()
        return (path / path.cummax() - 1).min()

    mdd_raw, mdd_hdg = _mdd(raw), _mdd(hedg)
    mdd_improve = (abs(mdd_raw) - abs(mdd_hdg)) / abs(mdd_raw) * 100.0

    # –– Sharpe improvement (rf = 0) –––––––––––––––––––––––––––––––––––
    sharpe_raw = raw.mean()  / raw.std(ddof=0)  * np.sqrt(252)
    sharpe_hdg = hedg.mean() / hedg.std(ddof=0) * np.sqrt(252)
    sharpe_improve = (
        (sharpe_hdg - sharpe_raw) / abs(sharpe_raw) * 100.0
        if sharpe_raw != 0 else np.nan
    )
```

```python
    # —— Volatility improvement ——————————————————————————————————————————
    vol_improve = (raw.std(ddof=0) - hedg.std(ddof=0)) / raw.std(ddof=0) * 100.0

    return {
        "Return Improvement":       ret_improve,
        "VaR Reduction":            var_improve,
        "CVaR Reduction":           cvar_improve,
        "Max Drawdown Reduction":   mdd_improve,
        "Sharpe Ratio Improvement": sharpe_improve,
        "Volatility Reduction":     vol_improve,
    }


grid_df_PBW_yearly_avg = yearly_sensitivity_averages_HMM(
        rets       = renewable_df["PBW"],
        states     = states_hmm["PBW"],                   # << placebo timing
        bond_rets  = bond_returns["SHY"],
        years      = list(range(2018, 2025)),
        med_list   = med_grid,
        high_list  = high_grid,
        id2label   = id2label_PBW                     # labels already text
    )


from itertools import product
import pandas as pd

# ------------------------------------------------------------------
# Parameter grid
med_grid  = [0.25, 0.40, 0.60]
high_grid = [0.50, 0.75, 0.90]
years_eval = list(range(2018, 2025))
n_draws = 1000                 # placebo schedules per fund
# ------------------------------------------------------------------

# Mapping from numeric IDs to 'low/med/high' for each ETF
id2label = {
    "ICLN": id2label_ICLN,
    "PBW" : id2label_PBW,
    "QCLN": id2label_QCLN,
}

# Containers
emp_dist = {}        # will hold a DataFrame per ticker
real_grid = {}       # optional: store the real grid for each ticker

for ticker in ["ICLN", "PBW", "QCLN"]:
    # real grid (model timing) – useful for later comparison
    real_grid[ticker] = yearly_sensitivity_averages_HMM(
        rets      = renewable_df[ticker],
        states    = states_hmm[ticker],          # weekly HMM states
        bond_rets = bond_returns["SHY"],
        years     = years_eval,
        med_list  = med_grid,
        high_list = high_grid,
        id2label  = id2label[ticker],            # numeric → text
    )

    # numeric → 'low/med/high' for placebo generation
    state_str = states_hmm[ticker].map(id2label[ticker])

    placebo_tables = []
    for sched in random_timing_schedules(state_str, n_draws=n_draws):
        grid = yearly_sensitivity_averages_HMM(
            rets      = renewable_df[ticker],
            states    = sched,                   # placebo timing
            bond_rets = bond_returns["SHY"],
            years     = years_eval,
            med_list  = med_grid,
            high_list = high_grid,
            id2label  = None                     # already 'low/med/high'
        )
        grid["run_id"] = len(placebo_tables)
        placebo_tables.append(grid)

    # concatenate to one big empirical distribution
    emp_dist[ticker] = pd.concat(placebo_tables, ignore_index=True)
```

```python
emp_dist["ICLN"].to_csv("random_derisk_ICLN.csv", index=True)
emp_dist["PBW"].to_csv("random_derisk_PBW.csv", index=True)
emp_dist["QCLN"].to_csv("random_derisk_QCLN.csv", index=True)
```

```python
emp_dist["ICLN"]
```

	De-risk Med (%)	De-risk High (%)	Avg Return Improve	Avg VaR Reduction	Avg CVaR Reduction	Avg MDD Reduction	Avg Sharpe Improve	Avg Vol Reduction	Years Used	run_id
0	25	50	1.744258	35.017799	29.456899	37.084934	80.225182	26.901221	7	0
1	25	75	2.440674	44.592182	33.497951	46.523624	136.752740	33.897319	7	0
2	25	90	2.853494	45.625295	33.706215	49.887273	172.383452	36.060094	7	0
3	40	50	1.942983	38.671400	32.717606	40.047986	79.810287	29.682890	7	0
4	40	75	2.643767	48.224946	37.153090	49.510804	140.220363	36.938964	7	0
...	...	...	...	...	...	...	...	...	...	...
8995	40	75	1.473192	38.950777	19.904562	35.317760	-7.036691	30.452370	7	999
8996	40	90	1.732557	38.973451	19.904562	36.911688	-17.755431	31.944964	7	999
8997	60	50	1.083145	45.473701	20.982198	37.134963	35.763814	29.662539	7	999
8998	60	75	1.514127	49.216194	23.443650	40.974508	19.202432	34.672818	7	999
8999	60	90	1.769226	50.432891	23.451548	42.562271	8.909609	36.233798	7	999

```python
import pandas as pd

def add_pvalues(real_grid, null_dist, metrics):
    out = real_grid.copy()
    for m in metrics:
        pvals = []
        for _, row in real_grid.iterrows():
            med, high = row["De-risk Med (%)"], row["De-risk High (%)"]
            real_val  = row[m]
            null_vals = null_dist.query(
                "`De-risk Med (%)`==@med and `De-risk High (%)`==@high"
            )[m]
            pvals.append((null_vals >= real_val).mean())   # one-sided
        out[f"{m} p"] = pvals
    return out

metrics = ["Avg Return Improve",
           "Avg VaR Reduction",
           "Avg CVaR Reduction",
           "Avg MDD Reduction",
           "Avg Vol Reduction"]
```

```python
grid_icln_p = add_pvalues(real_grid["ICLN"], emp_dist["ICLN"], metrics)

# Show nicely
display(grid_icln_p.style.format("{:.2f}", subset=metrics)
                        .format("{:.2%}", subset=[f"{m} p" for m in metrics])
                        .background_gradient(subset=[f"{m} p" for m in metrics],
                                             cmap="RdYlGn_r", vmin=0, vmax=0.1))
```

	De-risk Med (%)	De-risk High (%)	Avg Return Improve	Avg VaR Reduction	Avg CVaR Reduction	Avg MDD Reduction	Avg Sharpe Improve	Avg Vol Reduction	Years Used	Avg Return Improve p	Avg VaR Reduction p	Avg CVaR Reduction p	Avg MDD Reduction p
0	25	50	0.70	24.16	18.07	23.60	42.652206	23.75	7	53.90%	83.60%	65.70%	43.00%
1	25	75	1.18	30.39	20.45	29.74	117.232585	29.27	7	44.10%	69.70%	56.40%	36.40%
2	25	90	1.49	31.11	20.58	29.75	165.149089	31.01	7	41.70%	67.20%	56.10%	46.70%
3	40	50	0.55	26.79	21.29	25.24	-8.010084	27.27	7	66.30%	93.80%	72.90%	65.00%
4	40	75	1.02	33.75	23.67	31.86	67.997277	33.09	7	54.80%	84.30%	67.70%	52.20%
5	40	90	1.32	35.21	23.80	32.49	118.814142	34.94	7	49.30%	78.50%	67.40%	56.30%
6	60	50	0.36	31.14	24.16	27.41	-86.559452	30.98	7	75.30%	94.90%	73.80%	76.00%
7	60	75	0.81	39.13	27.09	34.31	-12.678171	37.17	7	66.00%	87.20%	72.70%	67.40%

```
grid_PBW_p = add_pvalues(real_grid["PBW"], emp_dist["PBW"], metrics)

# Show nicely
display(grid_PBW_p.style.format("{:.2f}", subset=metrics)
                        .format("{:.2%}", subset=[f"{m} p" for m in metrics])
                        .background_gradient(subset=[f"{m} p" for m in metrics],
                                             cmap="RdYlGn_r", vmin=0, vmax=0.1))
```

	De-risk Med (%)	De-risk High (%)	Avg Return Improve	Avg VaR Reduction	Avg CVaR Reduction	Avg MDD Reduction	Avg Sharpe Improve	Avg Vol Reduction	Years Used	Avg Return Improve p	Avg VaR Reduction p	Avg CVaR Reduction p	Avg MDD Reduction p
0	25	50	2.90	26.18	20.54	27.71	43.254572	23.24	7	31.50%	17.70%	30.80%	12.60%
1	25	75	3.96	28.47	21.25	34.41	50.076947	27.82	7	31.70%	20.50%	31.50%	11.40%
2	25	90	4.60	28.47	21.25	36.48	53.484155	29.13	7	31.70%	20.90%	31.50%	12.80%
3	40	50	3.38	34.99	24.60	32.27	70.866950	27.54	7	30.10%	4.40%	30.50%	12.50%
4	40	75	4.44	38.85	26.21	40.00	80.868097	32.40	7	30.80%	4.70%	27.50%	9.20%
5	40	90	5.08	38.85	26.21	42.11	85.443470	33.80	7	31.50%	4.80%	27.80%	10.60%
6	60	50	4.01	41.42	26.38	35.87	113.326545	32.15	7	29.90%	3.30%	37.00%	21.00%
7	60	75	5.07	48.22	29.02	45.92	129.597198	37.35	7	30.20%	2.90%	31.70%	7.70%

```
grid_QCLN_p = add_pvalues(real_grid["QCLN"], emp_dist["QCLN"], metrics)

# Show nicely
display(grid_QCLN_p.style.format("{:.2f}", subset=metrics)
                        .format("{:.2%}", subset=[f"{m} p" for m in metrics])
                        .background_gradient(subset=[f"{m} p" for m in metrics],
                                             cmap="RdYlGn_r", vmin=0, vmax=0.1))
```

	De-risk Med (%)	De-risk High (%)	Avg Return Improve	Avg VaR Reduction	Avg CVaR Reduction	Avg MDD Reduction	Avg Sharpe Improve	Avg Vol Reduction	Years Used	Avg Return Improve p	Avg VaR Reduction p	Avg CVaR Reduction p	Avg MDD Reduction p
0	25	50	-0.10	22.16	19.07	18.97	-397.304813	20.56	7	63.90%	84.90%	74.00%	60.60%
1	25	75	-0.24	24.09	19.69	19.78	-879.830903	23.90	7	64.20%	86.20%	74.70%	67.90%
2	25	90	-0.32	24.09	19.69	20.09	-1171.282632	24.87	7	64.20%	86.60%	74.70%	67.10%
3	40	50	0.03	29.78	24.22	23.73	-79.092617	25.51	7	61.50%	73.40%	71.90%	60.10%
4	40	75	-0.12	32.29	25.06	24.72	-581.730068	29.07	7	62.70%	77.10%	73.80%	71.30%
5	40	90	-0.20	32.29	25.06	25.01	-886.807235	30.11	7	62.70%	78.20%	73.80%	72.90%
6	60	50	0.18	36.49	27.05	27.65	389.636515	30.75	7	57.30%	68.80%	76.70%	66.70%
7	60	75	0.03	41.91	27.89	30.29	-134.194916	34.59	7	61.20%	61.60%	80.20%	70.90%

Start coding or generate with AI.

Even simpler placebo (no clustering)

```
def shuffle_timing_schedules(label_series, n_draws=1000, seed=0):
    """
    Fast placebo generator that keeps the same number of 'low','med','high'
    labels but randomly permutes their order (one draw per permutation).
    """
    rng = np.random.default_rng(seed)
    arr = label_series.values
    schedules = []
    for _ in range(n_draws):
        shuffled = rng.permutation(arr)
        schedules.append(pd.Series(shuffled, index=label_series.index))
    return schedules



# -------------------------------------------------------------------
# Parameter grid
med_grid  = [0.25, 0.40, 0.60]
high_grid = [0.50, 0.75, 0.90]
years_eval = list(range(2018, 2025))
n_draws = 100                 # placebo schedules per fund
# -------------------------------------------------------------------

# Containers
emp_dist_shuffle = {}        # will hold a DataFrame per ticker

for ticker in ["ICLN", "PBW", "QCLN"]:
    placebo_tables = []
    for sched in shuffle_timing_schedules(state_str, n_draws=100):
        grid = yearly_sensitivity_averages_HMM(
            rets      = renewable_df["ICLN"],
            states    = sched,
            bond_rets = bond_returns["SHY"],
            years     = list(range(2018, 2025)),
            med_list  = med_grid,
            high_list = high_grid,
            id2label  = None
        )
        grid["run_id"] = len(placebo_tables)
        placebo_tables.append(grid)

    # concatenate to one big empirical distribution
    emp_dist_shuffle[ticker] = pd.concat(placebo_tables, ignore_index=True)
```

```
grid_ICLN_p_shuffle = add_pvalues(real_grid["ICLN"], emp_dist_shuffle["ICLN"], metrics)

# Show nicely
display(grid_ICLN_p_shuffle.style.format("{:.2f}", subset=metrics)
                        .format("{:.2%}", subset=[f"{m} p" for m in metrics])
                        .background_gradient(subset=[f"{m} p" for m in metrics],
                                             cmap="RdYlGn_r", vmin=0, vmax=0.1))
```

	De-risk Med (%)	De-risk High (%)	Avg Return Improve	Avg VaR Reduction	Avg CVaR Reduction	Avg MDD Reduction	Avg Sharpe Improve	Avg Vol Reduction	Years Used	Avg Return Improve p	Avg VaR Reduction p	Avg CVaR Reduction p	Avg MDD Reduction p
0	25	50	0.70	24.16	18.07	23.60	42.652206	23.75	7	50.00%	76.00%	60.00%	41.00%
1	25	75	1.18	30.39	20.45	29.74	117.232585	29.27	7	37.00%	67.00%	56.00%	34.00%
2	25	90	1.49	31.11	20.58	29.75	165.149089	31.01	7	35.00%	62.00%	56.00%	44.00%
3	40	50	0.55	26.79	21.29	25.24	-8.010084	27.27	7	67.00%	94.00%	73.00%	61.00%
4	40	75	1.02	33.75	23.67	31.86	67.997277	33.09	7	51.00%	80.00%	64.00%	50.00%
5	40	90	1.32	35.21	23.80	32.49	118.814142	34.94	7	45.00%	76.00%	65.00%	53.00%
6	60	50	0.36	31.14	24.16	27.41	-86.559452	30.98	7	74.00%	96.00%	78.00%	80.00%
7	60	75	0.81	39.13	27.09	34.31	-12.678171	37.17	7	67.00%	90.00%	73.00%	63.00%

```
grid_PBW_p_shuffle = add_pvalues(real_grid["PBW"], emp_dist_shuffle["PBW"], metrics)

# Show nicely
display(grid_PBW_p_shuffle.style.format("{:.2f}", subset=metrics)
```

```
                          .format("{:.2%}", subset=[f"{m} p" for m in metrics])
                          .background_gradient(subset=[f"{m} p" for m in metrics],
                                               cmap="RdYlGn_r", vmin=0, vmax=0.1))
```

	De-risk Med (%)	De-risk High (%)	Avg Return Improve	Avg VaR Reduction	Avg CVaR Reduction	Avg MDD Reduction	Avg Sharpe Improve	Avg Vol Reduction	Years Used	Avg Return Improve p	Avg VaR Reduction p	Avg CVaR Reduction p	Avg MDD Reduction p
0	25	50	2.90	26.18	20.54	27.71	43.254572	23.24	7	2.00%	60.00%	42.00%	14.00%
1	25	75	3.96	28.47	21.25	34.41	50.076947	27.82	7	2.00%	77.00%	51.00%	16.00%
2	25	90	4.60	28.47	21.25	36.48	53.484155	29.13	7	3.00%	77.00%	51.00%	17.00%
3	40	50	3.38	34.99	24.60	32.27	70.866950	27.54	7	1.00%	35.00%	43.00%	17.00%
4	40	75	4.44	38.85	26.21	40.00	80.868097	32.40	7	2.00%	54.00%	51.00%	13.00%
5	40	90	5.08	38.85	26.21	42.11	85.443470	33.80	7	2.00%	58.00%	51.00%	17.00%
6	60	50	4.01	41.42	26.38	35.87	113.326545	32.15	7	2.00%	26.00%	57.00%	24.00%
7	60	75	5.07	48.22	29.02	45.92	129.597198	37.35	7	1.00%	35.00%	63.00%	11.00%

```
grid_QCLN_p_shuffle = add_pvalues(real_grid["QCLN"], emp_dist_shuffle["QCLN"], metrics)

# Show nicely
display(grid_QCLN_p_shuffle.style.format("{:.2f}", subset=metrics)
                          .format("{:.2%}", subset=[f"{m} p" for m in metrics])
                          .background_gradient(subset=[f"{m} p" for m in metrics],
                                               cmap="RdYlGn_r", vmin=0, vmax=0.1))
```

	De-risk Med (%)	De-risk High (%)	Avg Return Improve	Avg VaR Reduction	Avg CVaR Reduction	Avg MDD Reduction	Avg Sharpe Improve	Avg Vol Reduction	Years Used	Avg Return Improve p	Avg VaR Reduction p	Avg CVaR Reduction p	Avg MDD Reduction p
0	25	50	-0.10	22.16	19.07	18.97	-397.304813	20.56	7	81.00%	91.00%	51.00%	80.00%
1	25	75	-0.24	24.09	19.69	19.78	-879.830903	23.90	7	81.00%	95.00%	61.00%	86.00%
2	25	90	-0.32	24.09	19.69	20.09	-1171.282632	24.87	7	79.00%	95.00%	62.00%	85.00%
3	40	50	0.03	29.78	24.22	23.73	-79.092617	25.51	7	82.00%	80.00%	47.00%	72.00%
4	40	75	-0.12	32.29	25.06	24.72	-581.730068	29.07	7	80.00%	88.00%	59.00%	89.00%
5	40	90	-0.20	32.29	25.06	25.01	-886.807235	30.11	7	80.00%	90.00%	59.00%	87.00%
6	60	50	0.18	36.49	27.05	27.65	389.636515	30.75	7	82.00%	75.00%	51.00%	79.00%
7	60	75	0.03	41.91	27.89	30.29	-134.194916	34.59	7	82.00%	77.00%	70.00%	86.00%

```
real_grid["QCLN"]
```

	De-risk Med (%)	De-risk High (%)	Avg Return Improve	Avg VaR Reduction	Avg CVaR Reduction	Avg MDD Reduction	Avg Sharpe Improve	Avg Vol Reduction	Years Used
0	25	50	-0.100396	22.162800	19.074599	18.966658	-397.304813	20.560651	7
1	25	75	-0.242481	24.094436	19.685564	19.782032	-879.830903	23.900414	7
2	25	90	-0.316324	24.094436	19.685564	20.087161	-1171.282632	24.867936	7
3	40	50	0.027014	29.776645	24.220202	23.726790	-79.092617	25.508646	7
4	40	75	-0.121748	32.286605	25.056199	24.722783	-581.730068	29.074872	7
5	40	90	-0.199416	32.286605	25.056199	25.008944	-886.807235	30.110726	7
6	60	50	0.183712	36.490971	27.050177	27.648185	389.636515	30.745772	7
7	60	75	0.025858	41.906164	27.886175	30.289108	-134.194916	34.590075	7
8	60	90	-0.057035	41.906164	27.886175	30.557424	-454.522703	35.710813	7

Start coding or generate with AI.

```
# ──────────────────────────────────────────────────────────────────────────
# 1.  helper: clustered placebo schedules
# ──────────────────────────────────────────────────────────────────────────

import numpy as np
import pandas as pd
```

```
import random
from dataclasses import dataclass

@dataclass
class Seg:
    start: int
    length: int

def random_timing_schedules(label_series: pd.Series,
                     n_draws: int = 100,
                     seed: int = 0):
    """
    Faster clustered placebo generator.
    Keeps every (run-length, label) pair but relocates them.
    """
    rng = np.random.default_rng(seed)

    # -------- extract (length, label) blocks ------------------------------
    blocks, cur_lbl, cur_len = [], None, 0
    for lbl in label_series:
        if lbl == cur_lbl:
            cur_len += 1
        else:
            if cur_len:
                blocks.append((cur_len, cur_lbl))
            cur_lbl, cur_len = lbl, 1
    blocks.append((cur_len, cur_lbl))

    n = len(label_series)
    schedules = []

    for _ in range(n_draws):
        out = np.full(n, "low", dtype=object)
        free = [Seg(0, n)]                              # one big free segment

        # shuffle block order once per draw
        blocks_shuf = blocks.copy()
        random.shuffle(blocks_shuf)

        for length, lbl in blocks_shuf:
            length = int(length)

            # pick a random *segment* that can fit the block
            big_enough = [seg for seg in free if seg.length >= length]
            if not big_enough:                          # pathological: fallback
                rng.shuffle(out)                        # scatter labels randomly
                break

            seg = rng.choice(big_enough)
            free.remove(seg)

            # choose random offset inside that segment
            offset = rng.integers(0, seg.length - length + 1)
            start = seg.start + offset
            out[start : start + length] = lbl

            # push back leftover left / right pieces
            if offset:                      # left remainder
                free.append(Seg(seg.start, offset))
            right_len = seg.length - offset - length
            if right_len:
                free.append(Seg(start + length, right_len))

        schedules.append(pd.Series(out, index=label_series.index, name="state"))

    return schedules


# ──────────────────────────────────────────────────────────────────────────────
# 2.  plotting routine
# ──────────────────────────────────────────────────────────────────────────────
def plot_placebo_vs_real(
    equity_ret   : pd.Series,
    bond_ret     : pd.Series,
    state_series : pd.Series,         # weekly HMM states (numeric or text)
    *,
    id2label     : dict[int, str],
    de_risk_mod  : float
```

```
    de_risk_med  : float,
    de_risk_high : float,
    n_draws      : int = 100,
    seed         : int = 0,
    ticker       : str = "",
    figsize      : tuple = (12, 5)
):
    """
    Draw cumulative return curves:
      • unhedged (blue)
      • HMM-timed hedge (red)
      • n placebo hedges (thin grey)
    """

    # —— ensure monotonic indexes ——————————————————————————————————————————
    equity_ret   = equity_ret.sort_index()
    bond_ret     = bond_ret.sort_index()
    state_series = state_series.sort_index()

    # —— real hedge ————————————————————————————————————————————————————————
    hedged_real = hedge_by_hmm_state(
        equity_ret   = equity_ret,
        state_series = state_series,
        safe_ret     = bond_ret,
        de_risk_med  = de_risk_med,
        de_risk_high = de_risk_high,
        id2label     = id2label,
    )
    cum_raw  = (1 + equity_ret).cumprod()
```