VoC_test

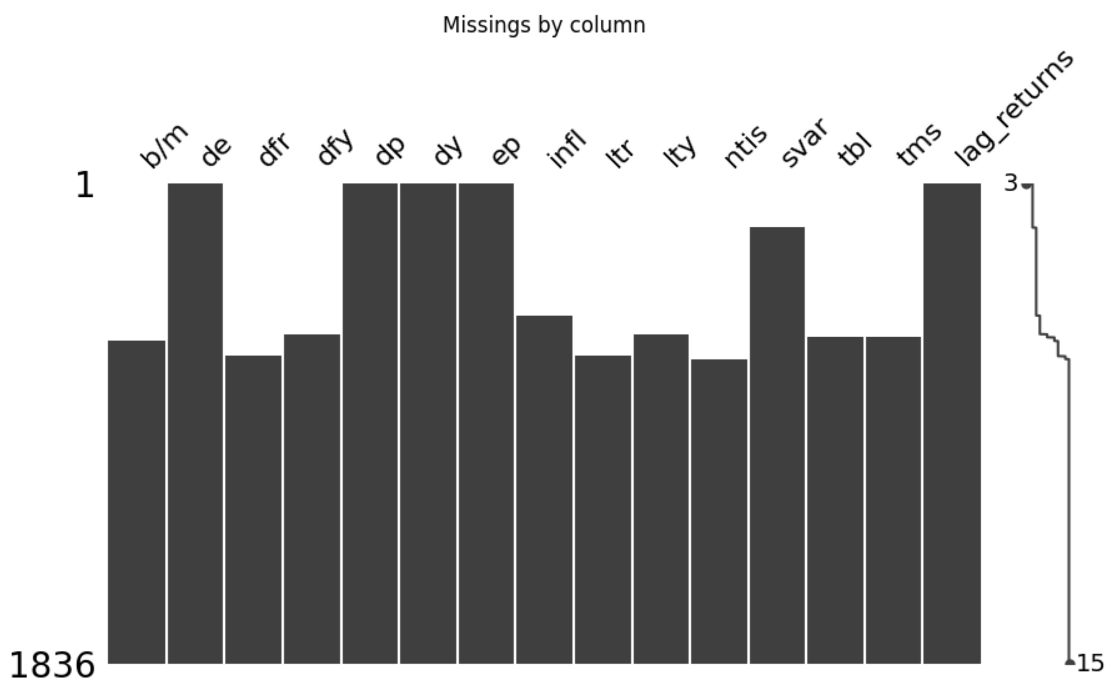
April 10, 2025

[24]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import missingno as msno
import seaborn as sns
import joblib
import os
from tqdm import tqdm
from sklearn.metrics import r2_score, precision_score, recall_score,␣
↪accuracy_score
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from scipy import stats

# Load Excel data
excel_path = "PredictorData2023.xlsx"

# --- Monthly data ---
data_raw = pd.read_excel(excel_path, sheet_name="Monthly")
data_raw["yyyymm"] = pd.to_datetime(data_raw["yyyymm"], format='%Y%m',␣
↪errors='coerce')
data_raw["Index"] = data_raw["Index"].apply(lambda x: str(x).replace(",", "")␣
↪if pd.notnull(x) else x)
data_raw = data_raw.set_index("yyyymm")
data_raw[data_raw.columns] = data_raw[data_raw.columns].astype(float)
data_raw = data_raw.rename({"Index":"prices"}, axis=1)
```

```
C:\Users\PHBS\AppData\Local\Programs\Python\Python313\Lib\site-
packages\openpyxl\worksheet\header_footer.py:48: UserWarning: Cannot parse
header or footer so it will be ignored
  warn("""Cannot parse header or footer so it will be ignored""")
```

[25]:
```
columns = ["b/m", "de", "dfr", "dfy", "dp", "dy", "ep", "infl", "ltr", "lty",␣
↪"ntis", "svar", "tbl", "tms", "lag_returns"]

# Calculate missing columns according to the explaination in m Welch and Goyal␣
↪(2008)
```

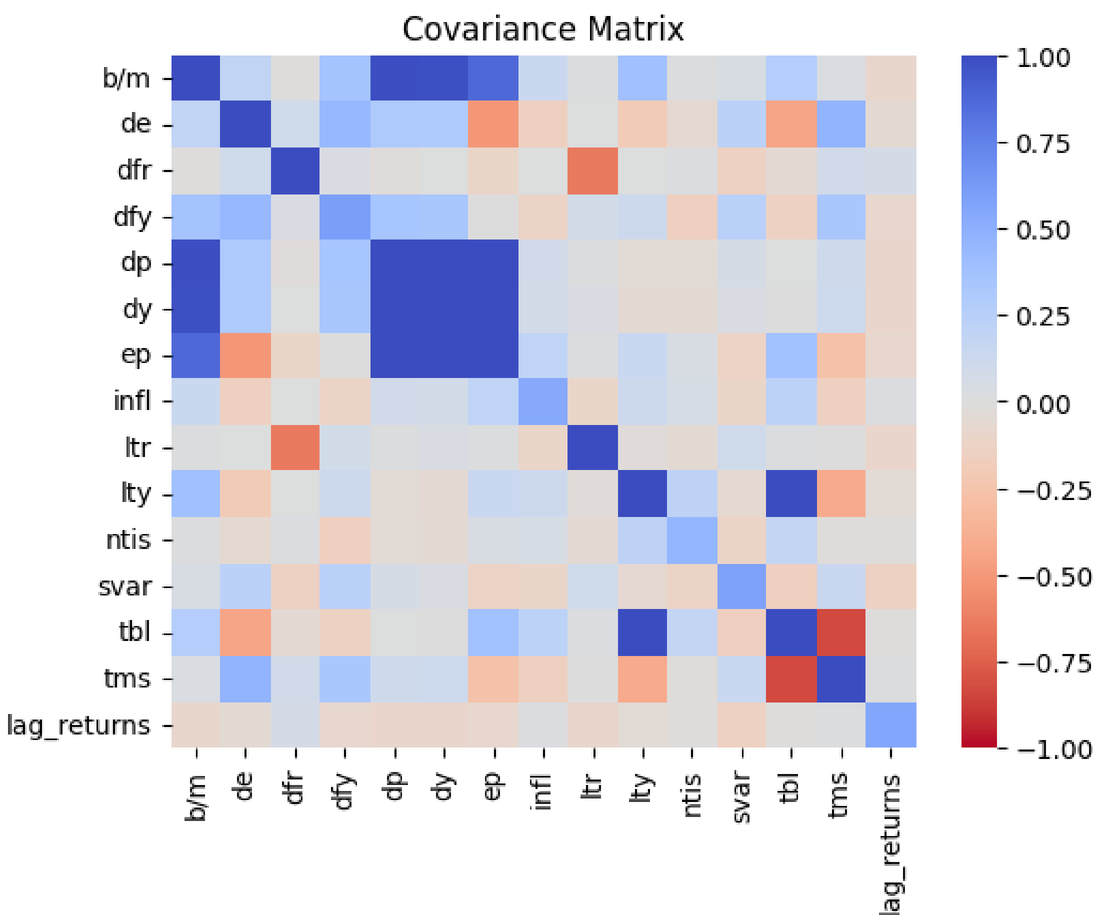
```
data_raw["dfy"] = data_raw["BAA"] - data_raw["AAA"]
data_raw["tms"] = data_raw["lty"] - data_raw["tbl"]
data_raw["de"] = np.log(data_raw["D12"]) - np.log(data_raw["E12"])
data_raw["dfr"] = data_raw["corpr"] - data_raw["ltr"]
data_raw["lag_price"] = data_raw["prices"].shift()
data_raw["dp"] = np.log(data_raw["D12"]) - np.log(data_raw["prices"])
data_raw["dy"] = np.log(data_raw["D12"]) - np.log(data_raw["lag_price"])
data_raw["ep"] = np.log(data_raw["E12"])  - np.log(data_raw["prices"])

data_raw["returns"] = data_raw["prices"].pct_change()
data_raw["lag_returns"] = data_raw["returns"].shift()

returns = data_raw["returns"].copy()
prices = data_raw["prices"].copy()

msno.matrix(data_raw[columns], figsize=(10,5))
plt.title("Missings by column")
plt.savefig("missing_pattern.jpg")
plt.show()
data = data_raw[columns].dropna()
returns = returns[returns.index.isin(data.index)]
```



[26]:
```
# Standardize predictors using expanding window of 36 months
for col in columns:
    rolling_mean = data[col].expanding(36).mean()
```

```
    rolling_std = data[col].expanding(36).std()
    data[col] = (data[col] - rolling_mean) / rolling_std

# Standardize returns by their past 12-month rolling standard deviation
returns_std = returns.rolling(12).std().shift()
returns = returns / returns_std

# Drop first 36 months (burn-in for expanding stats)
data = data[36:]
returns = returns[36:]
```

[27]:
```
sns.heatmap(data[columns].cov(), center=0, vmin=-1, vmax=1, cmap=sns.
 ↪color_palette("coolwarm_r", as_cmap=True))
fig = plt.gcf()
fig.figsize = (10,10)
plt.title("Covariance Matrix")
plt.show()
```



[28]:
```
import numpy as np
import pandas as pd
from tqdm import tqdm

# Setup
nr_features = 6000
rff_names = []
rff_features = []
omegas = []

print("Generating Random Fourier Features (not saving to disk)...")

# Generate omegas and apply projections
for i in tqdm(range(nr_features)):
    omega = np.random.normal(loc=0.0, scale=2.0, size=len(columns))  # shape:␣
↪(n_features,)
    projection = data.values @ omega  # (n_obs,)

    rff_features.append(np.sin(projection))
    rff_features.append(np.cos(projection))
    rff_names.append(f"sin_{i}")
    rff_names.append(f"cos_{i}")
    omegas.append(omega)

# Stack to (n_obs, 2*nr_features)
rff_array = np.vstack(rff_features).T
rff_df = pd.DataFrame(rff_array, columns=rff_names, index=data.index)

# Combine original and RFF features
data_full = pd.concat([data, rff_df], axis=1)

print("Shape of data after RFF transformation:", data_full.shape)
```

```
Generating Random Fourier Features (not saving to disk)…

100%|
| 6000/6000 [00:00<00:00, 7711.60it/s]

Shape of data after RFF transformation: (1129, 12015)
```

[30]:
```
from sklearn.linear_model import Ridge
import numpy as np
import pandas as pd
from tqdm import tqdm
import time

# Make sure the RFF names are defined
```

```
rff_names = [f"sin_{i}" for i in range(nr_features)] + [f"cos_{i}" for i in␣
↪range(nr_features)]
regression_data = data_full[rff_names]

# Setup
z_values = [10**-3, 10**2, 10**3, 10**4, 10**5, 10**6, 10**7, 10**8, 10**9]
t_values = list(range(12, data.shape[0]))  # t starts from 12

# Begin fresh backtest
backtest = []
print("Running backtest from scratch...")
start_time = time.time()

for t in tqdm(t_values[:-1]):
    for z in z_values:
        try:
            # Define training and test sets
            R = returns[t-12+1:t+1].values
            R_s = returns[t+1:t+2].values
            R_s_index = returns[t+1:t+2].index
            S = regression_data.iloc[t-12:t].values
            S_t = regression_data.iloc[t:t+1].values

            if np.any(np.isnan(S)) or np.any(np.isnan(S_t)) or np.any(np.
↪isnan(R)) or np.any(np.isnan(R_s)):
                continue  # skip if any NA

            # Fit ridge regression
            beta = Ridge(alpha=z, fit_intercept=False).fit(S, R).coef_
            beta_norm = np.sqrt(np.sum(beta**2))

            # Forecast & strategy return
            forecast = (S_t @ beta).item()
            timing_strategy = forecast * R_s.item()

            backtest.append({
                "z": z,
                "t": t,
                "beta_norm": beta_norm,
                "index": R_s_index[0],
                "forecast": forecast,
                "timing_strategy_index": R_s_index[0],
                "timing_strategy": timing_strategy,
                "return": R_s.item()
            })

        except Exception as e:
```

```
            print(f"Error at t={t}, z={z}: {e}")
            continue

# Convert to DataFrame
backtest = pd.DataFrame(backtest).set_index("index")

# Add denormalized returns
backtest["return_denorm"] = backtest["return"] * returns_std.loc[backtest.index]
backtest["forecast_denorm"] = backtest["forecast"] * returns_std.loc[backtest.
 ↪index]

print("Backtest completed in", round(time.time() - start_time, 2), "seconds.")
```

```
Running backtest from scratch…

100%|
  | 1116/1116 [00:47<00:00, 23.51it/s]

Backtest completed in 47.54 seconds.
```

[31]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, precision_score, recall_score,␣
 ↪accuracy_score

result = []
time_factor = 12  # Annualization factor

for z in z_values:
    df = backtest[backtest["z"] == z].dropna()

    # Calculate regression of strategy return on market return
    market_reg = LinearRegression().fit(df[["timing_strategy"]].values,␣
 ↪df["return"].values)
    beta = market_reg.coef_[0]
    alpha = market_reg.intercept_

    # Avoid division by zero if beta is very small
    if np.abs(beta) < 1e-6:
        continue

    mean = df["timing_strategy"].mean() * time_factor
    std = df["timing_strategy"].std() * np.sqrt(time_factor)
    mean_return = df["return"].mean() / beta

    # Forecast vs actual sign for classification metrics
    actual_up = (df["return"] > 0).astype(int)
```

```python
    forecast_up = (df["forecast"] > 0).astype(int)

    result.append({
        "log10(z)": np.log10(z),
        "beta_norm_mean": df["beta_norm"].mean(),
        "Market Sharpe Ratio": (df["return"].mean() * time_factor) /␣
↪(df["return"].std() * np.sqrt(time_factor)),
        "Expected Return": mean,
        "Volatility": std,
        "R²": r2_score(df["return"].values / beta, df["timing_strategy"].
↪values),
        "Sharpe Ratio": mean / std,
        "Information Ratio": (mean - mean_return) / std,
        "Alpha": alpha,
        "Precision": precision_score(actual_up, forecast_up, zero_division=0),
        "Recall": recall_score(actual_up, forecast_up, zero_division=0),
        "Accuracy": accuracy_score(actual_up, forecast_up),
    })

result = pd.DataFrame(result)
print(result.round(5))
```

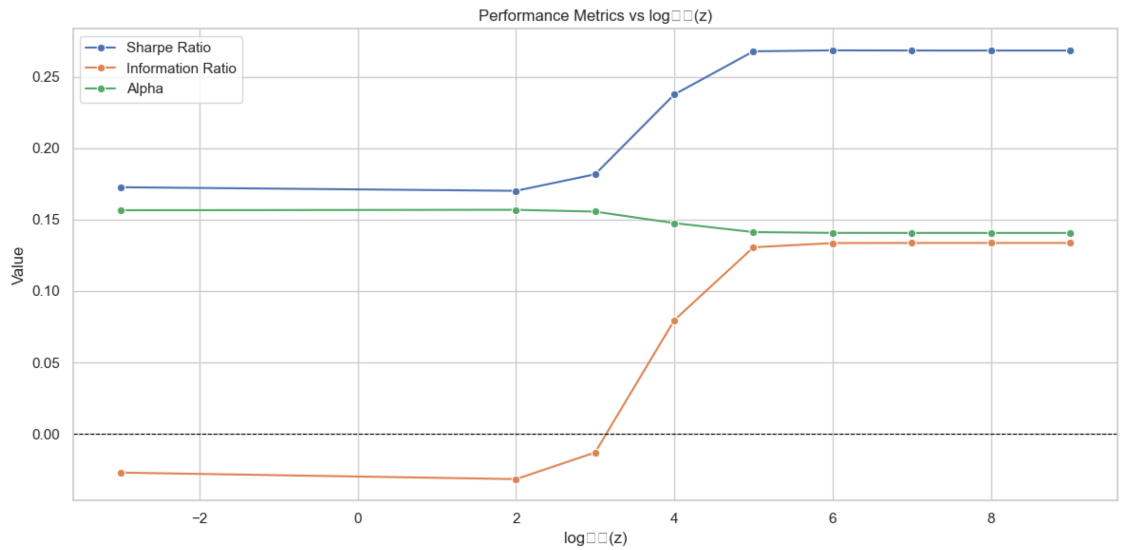
```
   log10(z)  beta_norm_mean  Market Sharpe Ratio  Expected Return  Volatility  \
0      -3.0         0.05260              0.51029          0.18522     1.07286
1       2.0         0.05139              0.51029          0.17728     1.04205
2       3.0         0.04358              0.51029          0.15962     0.87851
3       4.0         0.01859              0.51029          0.09614     0.40474
4       5.0         0.00285              0.51029          0.02006     0.07494
5       6.0         0.00030              0.51029          0.00227     0.00846
6       7.0         0.00003              0.51029          0.00023     0.00086
7       8.0         0.00000              0.51029          0.00002     0.00009
8       9.0         0.00000              0.51029          0.00000     0.00001

        R²  Sharpe Ratio  Information Ratio    Alpha  Precision   Recall  \
0  0.02666       0.17265           -0.02701  0.15654    0.58401  0.54079
1  0.02567       0.17013           -0.03159  0.15684    0.58401  0.54079
2  0.02926       0.18170           -0.01293  0.15558    0.58682  0.55136
3  0.05590       0.23754            0.07962  0.14755    0.60128  0.56949
4  0.08100       0.26773            0.13064  0.14125    0.59718  0.57553
5  0.08438       0.26840            0.13358  0.14071    0.59875  0.57704
6  0.08468       0.26828            0.13366  0.14068    0.59875  0.57704
7  0.08471       0.26827            0.13366  0.14068    0.59875  0.57704
8  0.08471       0.26827            0.13367  0.14068    0.59875  0.57704

   Accuracy
0   0.49910
1   0.49910
```

```
2    0.50358
3    0.52061
4    0.51792
5    0.51971
6    0.51971
7    0.51971
8    0.51971
```

[32]:

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

# Set plot style
sns.set(style="whitegrid")
plt.figure(figsize=(12, 6))

# Plot key metrics
for metric in ["Sharpe Ratio", "Information Ratio", "Alpha"]:
    sns.lineplot(data=result, x="log10(z)", y=metric, marker="o", label=metric)

plt.title("Performance Metrics vs log (z)")
plt.xlabel("log (z)")
plt.ylabel("Value")
plt.legend()
plt.axhline(0, color="black", linewidth=0.8, linestyle="--")
plt.tight_layout()
plt.show()
```

```
C:\Users\PHBS\AppData\Local\Temp\ipykernel_10540\3705762621.py:17: UserWarning:
Glyph 8321 (\N{SUBSCRIPT ONE}) missing from font(s) Arial.
  plt.tight_layout()
C:\Users\PHBS\AppData\Local\Temp\ipykernel_10540\3705762621.py:17: UserWarning:
Glyph 8320 (\N{SUBSCRIPT ZERO}) missing from font(s) Arial.
  plt.tight_layout()
C:\Users\PHBS\AppData\Local\Programs\Python\Python313\Lib\site-
packages\IPython\core\pylabtools.py:170: UserWarning: Glyph 8321 (\N{SUBSCRIPT
ONE}) missing from font(s) Arial.
  fig.canvas.print_figure(bytes_io, **kw)
C:\Users\PHBS\AppData\Local\Programs\Python\Python313\Lib\site-
packages\IPython\core\pylabtools.py:170: UserWarning: Glyph 8320 (\N{SUBSCRIPT
ZERO}) missing from font(s) Arial.
  fig.canvas.print_figure(bytes_io, **kw)
```



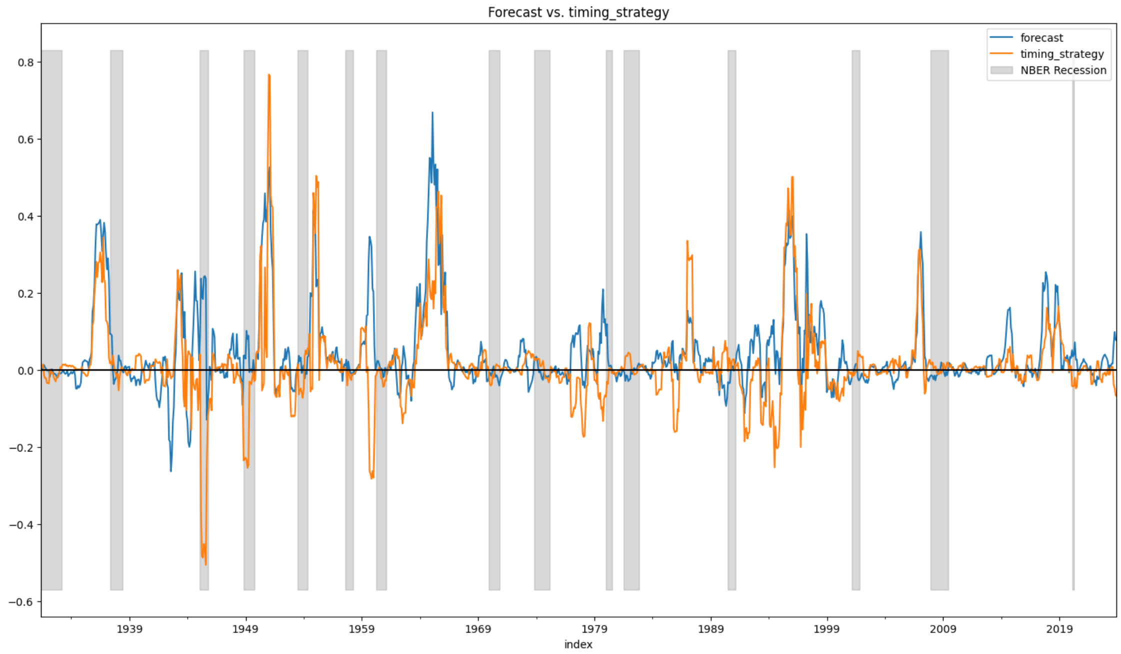
[17]:
```
nber = pd.read_csv("NBER_20210719_cycle_dates_pasted.csv")[1:]
nber["peak"] = pd.to_datetime(nber["peak"])
nber["trough"] = pd.to_datetime(nber["trough"])
nber.head()
```

[17]:

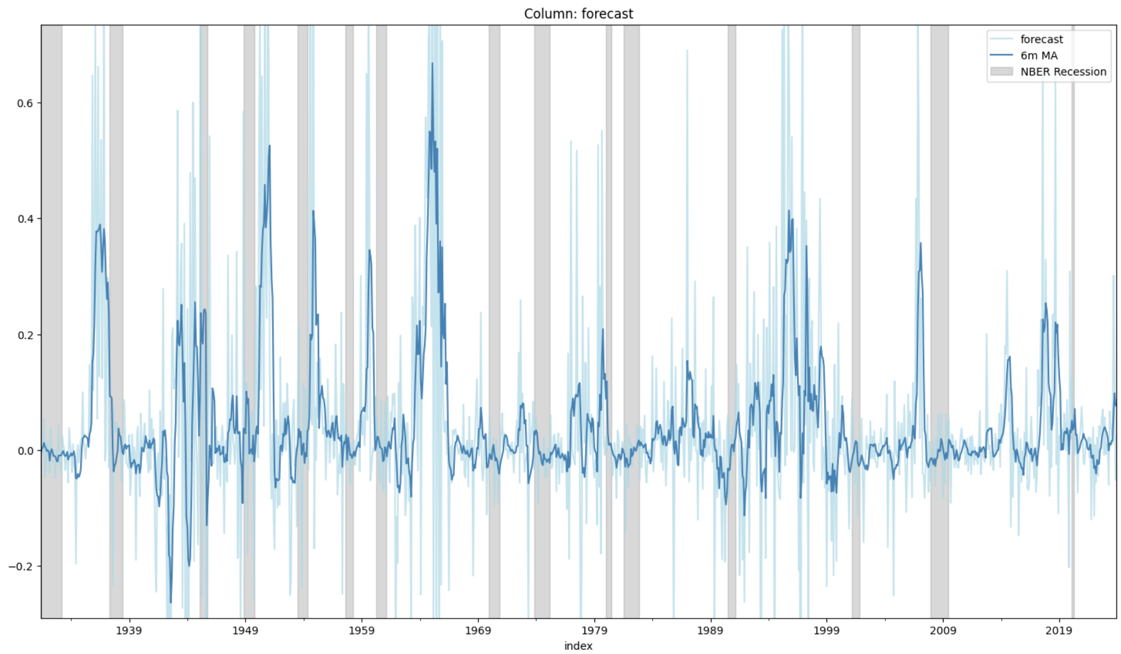
	peak	trough
1	1857-06-01	1858-12-01
2	1860-10-01	1861-06-01
3	1865-04-01	1867-12-01
4	1869-06-01	1870-12-01
5	1873-10-01	1879-03-01

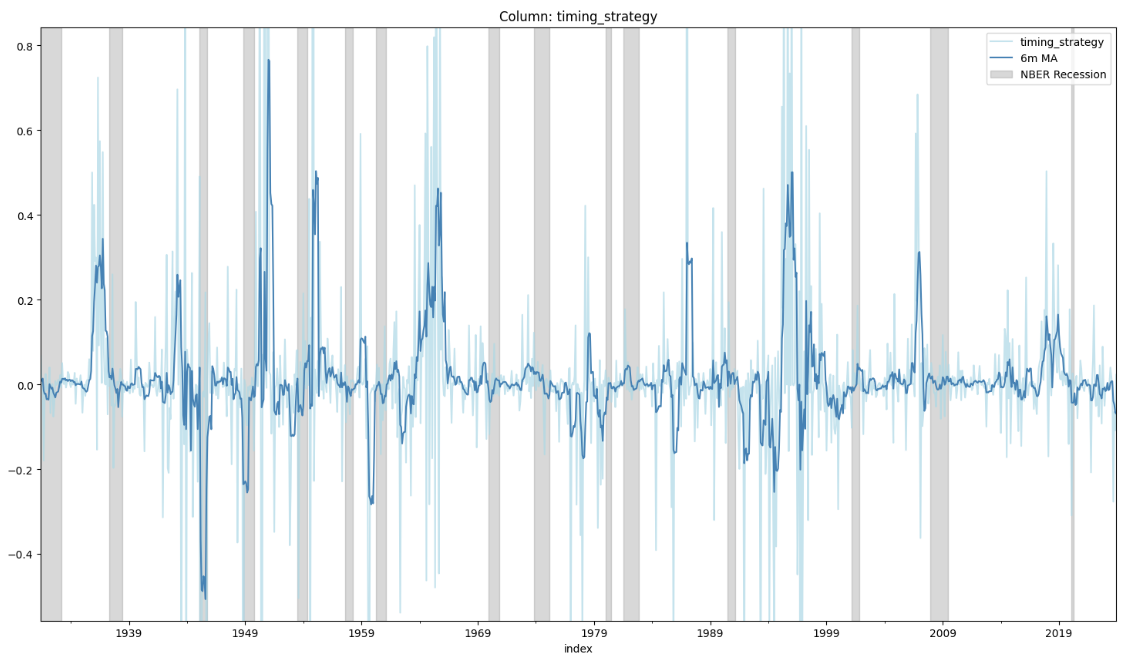
[21]:
```
fig, ax = plt.subplots(figsize=(18,10))
for col in ["forecast", "timing_strategy"]:
    plot_data = pd.DataFrame()
    plot_data[col] = backtest.loc[backtest["z"] == 1000, col]
    plot_data["6m MA"] = plot_data[col].rolling(6).mean()

    recessions = [t for date_list in nber.apply(lambda x: pd.
 ↪date_range(x["peak"], x["trough"]), axis=1).values for t in date_list]
    plot_data["NBER Recession"] = plot_data.index.isin(recessions).astype(int)
    plot_data = plot_data.dropna()
    plot_data["6m MA"].plot(ax=ax, label=col)
ax.fill_between(plot_data.index, ax.get_ylim()[0], ax.get_ylim()[1],
                where=plot_data["NBER Recession"] == 1 ,color='grey', alpha=0.
 ↪3,  label="NBER Recession")
ax.legend(loc="upper right")
ax.axhline(0, c="black")
```

```
ax.set_title("Forecast vs. timing_strategy")
plt.show()
```



[23]:
```
for col in ["forecast", "timing_strategy"]:
    plot_data = pd.DataFrame()
    plot_data[col] = backtest.loc[backtest["z"] == 1000, col]
    plot_data["6m MA"] = plot_data[col].rolling(6).mean()

    recessions = [t for date_list in nber.apply(lambda x: pd.
 ↪date_range(x["peak"], x["trough"]), axis=1).values for t in date_list]
    plot_data["NBER Recession"] = plot_data.index.isin(recessions).astype(int)

    plot_data = plot_data.dropna()

    fig, ax = plt.subplots(figsize=(18,10))
    plot_data[col].plot(ax=ax, alpha=0.7, c="lightblue")
    plot_data["6m MA"].plot(ax=ax, c="steelblue")
    ax.set_ylim(plot_data["6m MA"].min()*1.1, plot_data["6m MA"].max()*1.1)
    ax.fill_between(plot_data.index, ax.get_ylim()[0], ax.get_ylim()[1],
                    where=plot_data["NBER Recession"] == 1 ,color='grey',␣
 ↪alpha=0.3,  label="NBER Recession")
    ax.legend(loc="upper right")
    ax.set_title(f"Column: {col}")
```


Column: forecast
forecast
6m MA
NBER Recession
0.6
0.4
0.2
0.0
−0.2
1939
1949
1959
1969
1979
1989
1999
2009
2019
index


Column: timing_strategy
timing_strategy
6m MA
NBER Recession
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
1939
1949
1959
1969
1979
1989
1999
2009
2019
index

```
[ ]:
```