Optimizing AI Trading Algorithms - Course Project

In this project you will practice optimizing various aspects of a machine learning model for predicting stock price movements. This will provide you with an opportunity to integrate the concepts covered in the course, such as data preprocessing and cleaning, hyperparameter tuning, detecting and addressing over-/under-fitting, model evaluation, and feature selection techniques. While you will use real-world data in this project, the goal is not necessarily to build a "winning" trading *strategy*. The goal of this course has been to equip you with the tools, techniques, concepts and insights you need to evaluate, optimize and monitor *your own* trading strategies.

The Scenario

You are an analyst at a boutique investment firm tasked with coming up with a novel idea for investing in specific sectors of the industry. You've heard that the Utilities, Consumer Staples and Healthcare sectors are relatively resilient to economic shocks and recessions, and that stock market investors tend to flock to these sectors in times of uncertainty. You decide to take the SPDR Healthcase Sector ETF (NYSEARCA: XLV) and try to model its returns' dynamics using a machine learning AI strategy. Your novel idea is to get data for the volatility index (INDEXCBOE: VIX) as a proxy for uncertainty in the market. You also decide to take a look at Google Trends data for the search term "recession" in the United States, in order to try and see if there is any meaningful relationship between the general public's level of concern about a recession happening and the price movements of the Health Care Select Sector SPDR Fund.

You decide to train a binary **classification** model that merely attempts to predict the **direction** of XLV's 5-day price movements. In other words, you want to see if on any given day, with the above data in hand, you could reliably predict whether the price of XLV will increase or decrease over the next 5 trading days.

Run the cell below to `import` all the Python packages and modules you will be using throughout the project.

In [1]:
```
!pip install wheel
!pip install --upgrade pip yfinance ta --quiet
```

```
Requirement already satisfied: wheel in /Users/ishanklal/miniconda3/envs/ait
snd/lib/python3.9/site-packages (0.45.1)
```

In [2]:
```
import matplotlib.dates as mdates
import matplotlib.pyplot as plt
import numpy as np
```

```
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
import seaborn as sns
import yfinance as yf
from plotly.subplots import make_subplots
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    accuracy_score,
    classification_report,
    confusion_matrix,
    f1_score,
    precision_score,
    recall_score,
)
from sklearn.model_selection import GridSearchCV, learning_curve, train_test
from ta.momentum import RSIIndicator
from ta.volatility import BollingerBands

pd.options.display.max_columns = 50
pd.options.display.max_rows = 50

RANDOM_SEED = 42
```

1. Data Acquisition, Exploration, Cleaning and Preprocessing

In this section, you will download and inspect:

- daily data for the SPDR Healthcase Sector ETF (NYSEARCA: XLV)
- daily data for the volatility index (INDEXCBOE: VIX)
- monthly data from Google Trends for the search interest in the term "recession" in the United States

The goal is to make sure the data is clean, meaningful, and usable for selecting and engineering features.

1.1. Price and Volume Data for "XLV"

We have downloded daily data from **January 1st, 2004** to **March 31st, 2024** for the ticker **XLV** using the `yfinance` library and stored it in a CSV file named `xlv_data.csv`. Load this data into a Pandas DataFrame named `xlv_data`, making sure to set the index column to the first column of the CSV file (`Date`) and set `parse_dates=True`.

In [3]:
```
xlv_data = pd.read_csv('xlv_data.csv', index_col='Date', parse_dates=True)
print(xlv_data)
```

```
                  Open        High         Low       Close   Adj Close  \
Date
2004-01-02   30.200001   30.440001   30.120001   30.219999   21.567184
2004-01-05   30.400000   30.500000   30.139999   30.360001   21.667091
2004-01-06   30.469999   30.480000   30.309999   30.450001   21.731337
2004-01-07   30.450001   30.639999   30.309999   30.639999   21.866926
2004-01-08   30.700001   30.700001   30.320000   30.510000   21.774158
...                ...         ...         ...         ...         ...
2024-03-22  145.850006  146.220001  145.259995  145.440002  145.440002
2024-03-25  145.710007  145.860001  145.009995  145.240005  145.240005
2024-03-26  145.529999  145.940002  145.139999  145.770004  145.770004
2024-03-27  147.009995  147.710007  146.619995  147.710007  147.710007
2024-03-28  147.919998  148.229996  147.679993  147.729996  147.729996

             Volume
Date
2004-01-02   628700
2004-01-05   191500
2004-01-06   289300
2004-01-07   262300
2004-01-08   214300
...             ...
2024-03-22  5537200
2024-03-25  5253000
2024-03-26  6942400
2024-03-27  8797400
2024-03-28  8090200

[5094 rows x 6 columns]
```

Use the `info()` and `describe()` methods to get an overview of how many rows of data there are in `xlv_data`, what columns are present and what their data types are, and what some basic statistics (mean, std, quartiles, min/max values) of the columns look like.

In [4]:
```
xlv_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 5094 entries, 2004-01-02 to 2024-03-28
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Open       5094 non-null   float64
 1   High       5094 non-null   float64
 2   Low        5094 non-null   float64
 3   Close      5094 non-null   float64
 4   Adj Close  5094 non-null   float64
 5   Volume     5094 non-null   int64
dtypes: float64(5), int64(1)
memory usage: 278.6 KB
```

In [5]:
```
xlv_data.describe()
```

Out [5]:

	Open	High	Low	Close	Adj Close	Vc
count	5094.000000	5094.000000	5094.000000	5094.000000	5094.000000	5.09400(
mean	65.342311	65.730397	64.924197	65.349097	58.242299	7.22895
std	36.695351	36.915853	36.477869	36.712468	37.932219	5.44580
min	22.010000	22.290001	21.629999	21.879999	16.812475	5.87000(
25%	31.990000	32.132501	31.812500	31.990000	24.508568	3.79055(
50%	57.100000	57.400000	56.680000	57.010000	48.387001	6.58285(
75%	90.657503	91.077497	89.927500	90.557499	82.941315	9.55955(
max	147.919998	148.270004	147.679993	147.860001	147.729996	6.64702(

How many `NaN` rows are there in `xlv_data` ?

In [6]:
```
answer = xlv_data.isnull().sum()
answer
```

Out [6]:
```
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64
```

Take a look at the final five rows of `xlv_data` .

In [7]:
```
xlv_data.tail()
```

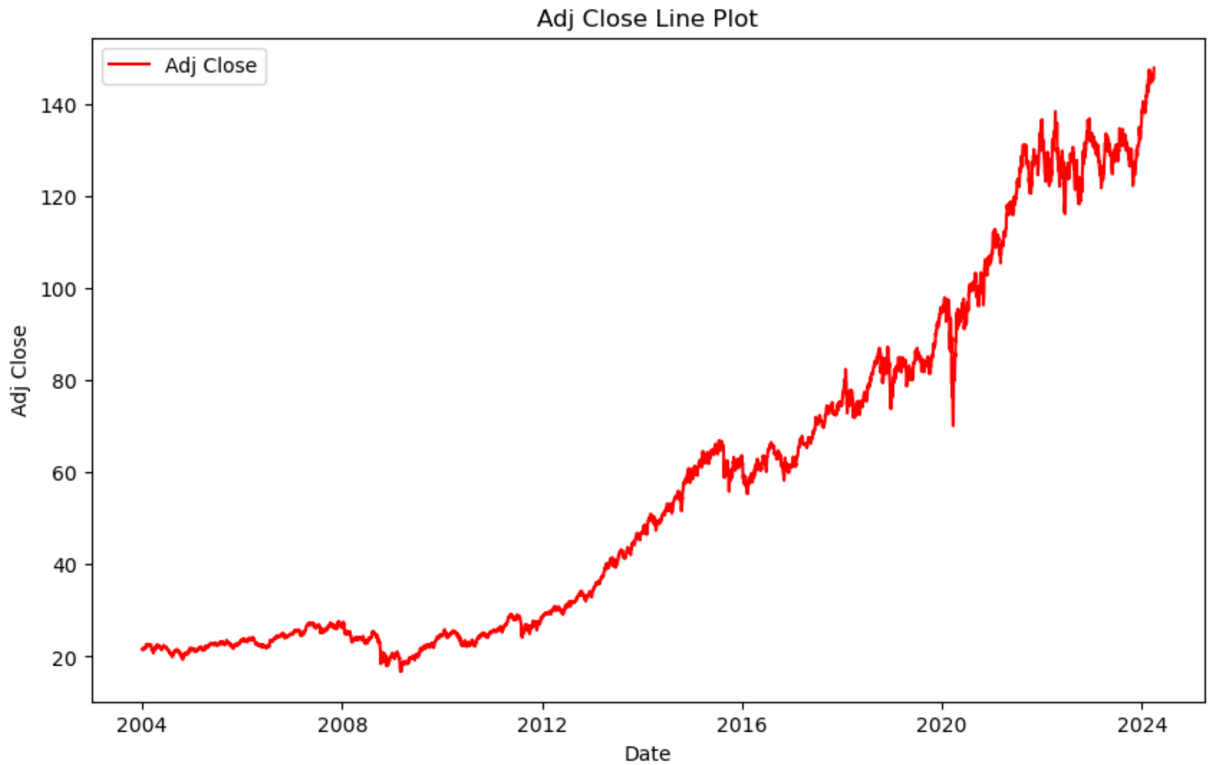
Out [7]:

	Open	High	Low	Close	Adj Close	Volume
Date						
2024-03-22	145.850006	146.220001	145.259995	145.440002	145.440002	5537200
2024-03-25	145.710007	145.860001	145.009995	145.240005	145.240005	5253000
2024-03-26	145.529999	145.940002	145.139999	145.770004	145.770004	6942400
2024-03-27	147.009995	147.710007	146.619995	147.710007	147.710007	8797400
2024-03-28	147.919998	148.229996	147.679993	147.729996	147.729996	8090200

Raw OHLC data is not suitable for training models. The absolute price level of a security is boundless in theory and not particularly menaningful. In the next section, you are going to engineer useful features from all of these columns. For now, as a visual sanity check, plot `Adj Close` as a line plot.

In [8]:
```
plt.figure(figsize=(10, 6))
plt.plot(xlv_data.index, xlv_data['Adj Close'], label='Adj Close', color='re
```

```
plt.title('Adj Close Line Plot')
plt.xlabel('Date')
plt.ylabel('Adj Close')
plt.legend()
plt.show()
```

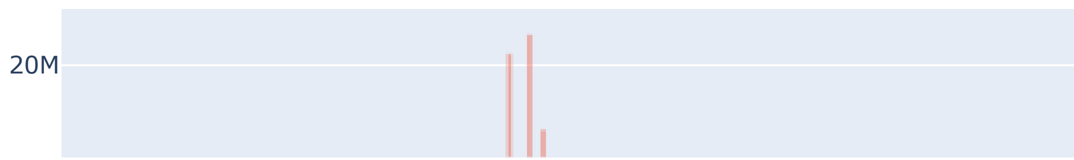


Bonus: The cell below plots the combined candlestick + volume chart for the last 15 months of data using Plotly.

In [9]:
```
data_since_2023 = xlv_data["2023-01-01":]

figure = make_subplots(specs=[[{"secondary_y": True}]])
figure.add_traces(
    go.Candlestick(
        x=data_since_2023.index,
        open=data_since_2023.Open,
        high=data_since_2023.High,
        low=data_since_2023.Low,
        close=data_since_2023.Close,
    ),
    secondary_ys=[True],
)
figure.add_traces(
    go.Bar(x=data_since_2023.index, y=data_since_2023.Volume, opacity=0.5),
    secondary_ys=[False],
)

figure.update_layout(
    title="XLV Candlestick Chart Since 2023",
    xaxis_title="Date",
    yaxis_title="Volume",
    yaxis2_title="Price",
```

```
    showlegend=False,
)
figure.update_yaxes(fixedrange=False)
figure.layout.yaxis2.showgrid = False
figure.show()
```

XLV Candlestick Chart Since 2023



1.2. Data for The Volatility Index `VIX`

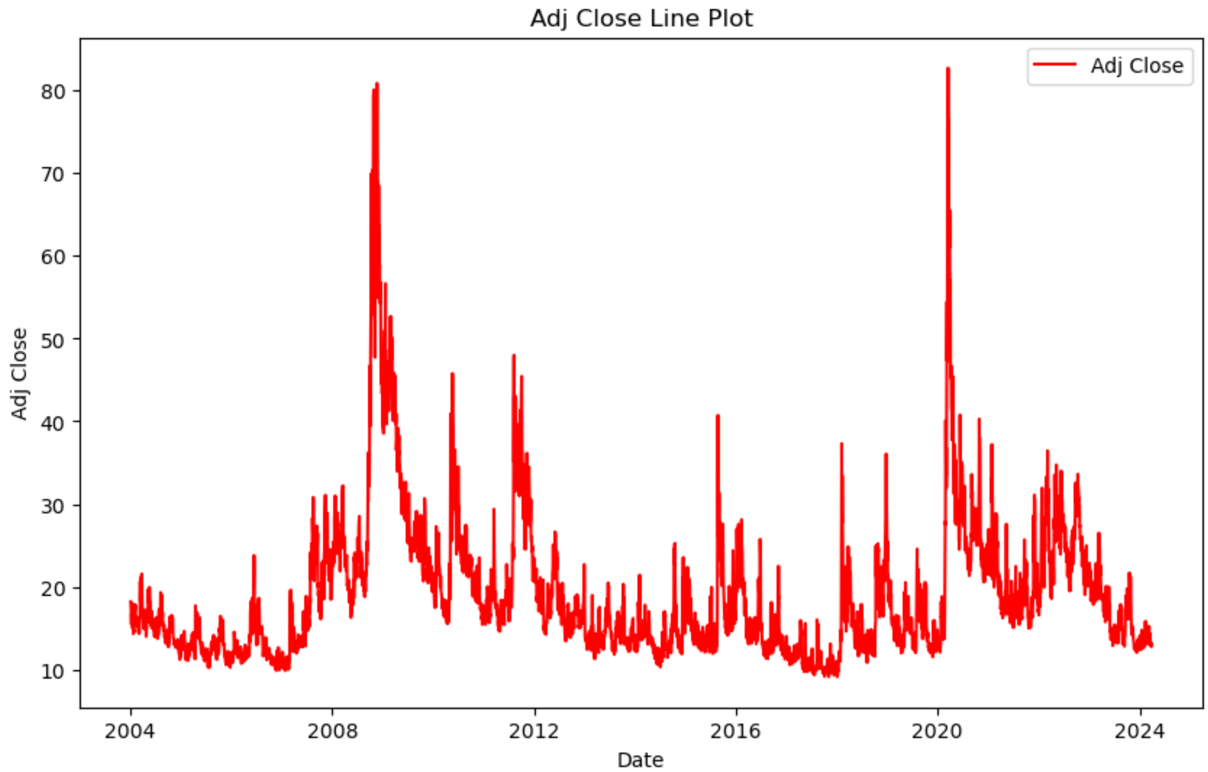
As before, we have downloaded daily data for the volatility index (INDEXCBOE: VIX) over the same time period using `yfinance` and provided it to you in a CSV file named `vix_data.csv`. Load the data into a variable named `vix_data`. Make sure to set the index and parse the dates correctly.

In [10]:
```
vix_data = pd.read_csv('vix_data.csv', index_col='Date', parse_dates=True)
```

Plot a line chart of the `Adj Close` value of the VIX using your method of choice (e.g. `plotly` or `matplotlib`).

In [11]:
```
plt.figure(figsize=(10, 6))
plt.plot(vix_data.index, vix_data['Adj Close'], label='Adj Close', color='re
plt.title('Adj Close Line Plot')
```

```
plt.xlabel('Date')
plt.ylabel('Adj Close')
plt.legend()
plt.show()
```



1.3. Google Trends Data

The **monthly** evolution of search interest in the term "recession" in the U.S. over the period of interest (Jan. 2003 - Mar. 2024) from the Google Trends website has been provided to you as a CSV file. We will load this data using Pandas into a DataFrame named `google_trends_data`, set the index column of the DataFrame to the "`Month`" column from the CSV and have Pandas try and parse these dates automatically.

> Note: The "Month" column in the CSV is in "YYYY-MM" format.

In [12]:
```
google_trends_data = pd.read_csv('GoogleTrendsData.csv')
google_trends_data['Month'] = pd.to_datetime(google_trends_data['Month'], fo
google_trends_data.set_index('Month', inplace=True)
```

As noted above, the CSV lists **monthly** search trends data and the `Month` column is in YYYY-MM format. How has Pandas interpreted and parsed these dates into specific dates? Take a look at `google_trends_data`'s index.

In [13]:
```
google_trends_data.index
```

Out[13]:
```
DatetimeIndex(['2004-01-01', '2004-02-01', '2004-03-01', '2004-04-01',
               '2004-05-01', '2004-06-01', '2004-07-01', '2004-08-01',
               '2004-09-01', '2004-10-01',
               ...
               '2023-06-01', '2023-07-01', '2023-08-01', '2023-09-01',
               '2023-10-01', '2023-11-01', '2023-12-01', '2024-01-01',
               '2024-02-01', '2024-03-01'],
              dtype='datetime64[ns]', name='Month', length=243, freq=None)
```

We would have liked to assign the data points to the last day of the respective months, as this data would have been available at the *end* of each period. Shift the index column of `google_trends_data` to do this.

> Hint: You can use `pd.offsets.MonthEnd()` from Pandas.

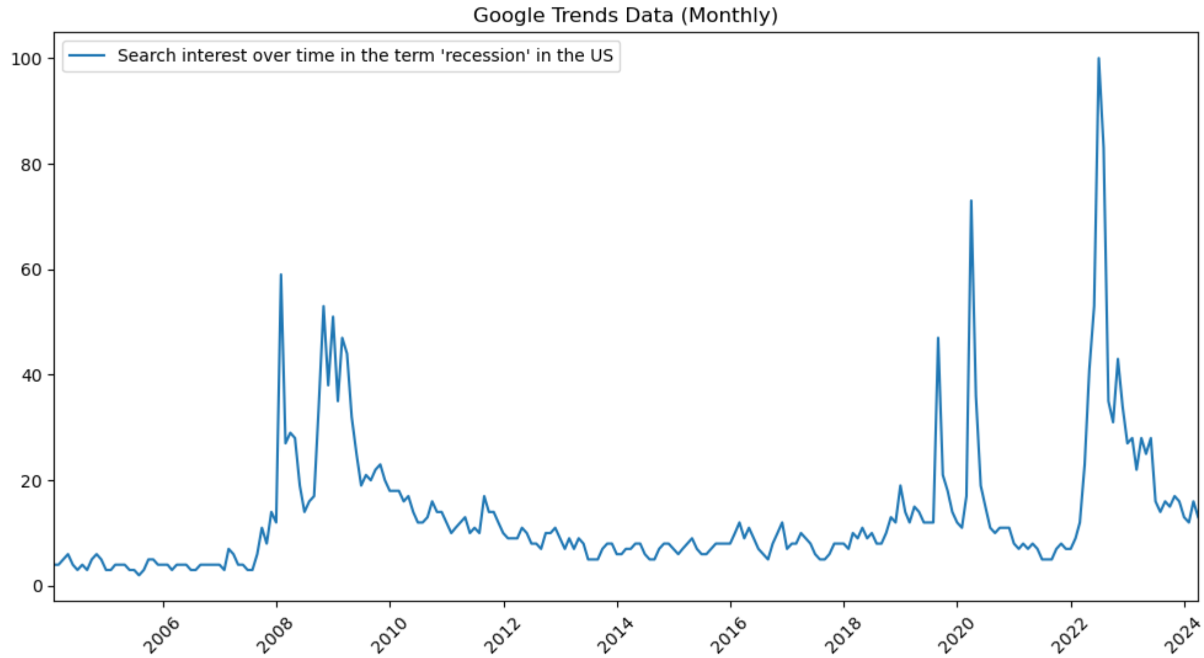
In [14]:
```
google_trends_data.index = google_trends_data.index + pd.offsets.MonthEnd(0)
google_trends_data.index
```

Out[14]:
```
DatetimeIndex(['2004-01-31', '2004-02-29', '2004-03-31', '2004-04-30',
               '2004-05-31', '2004-06-30', '2004-07-31', '2004-08-31',
               '2004-09-30', '2004-10-31',
               ...
               '2023-06-30', '2023-07-31', '2023-08-31', '2023-09-30',
               '2023-10-31', '2023-11-30', '2023-12-31', '2024-01-31',
               '2024-02-29', '2024-03-31'],
              dtype='datetime64[ns]', name='Month', length=243, freq=None)
```

Run the cell below to visualize this data as a line plot.

> **Note from Google:** "Numbers represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. A score of 0 means there was not enough data for this term."

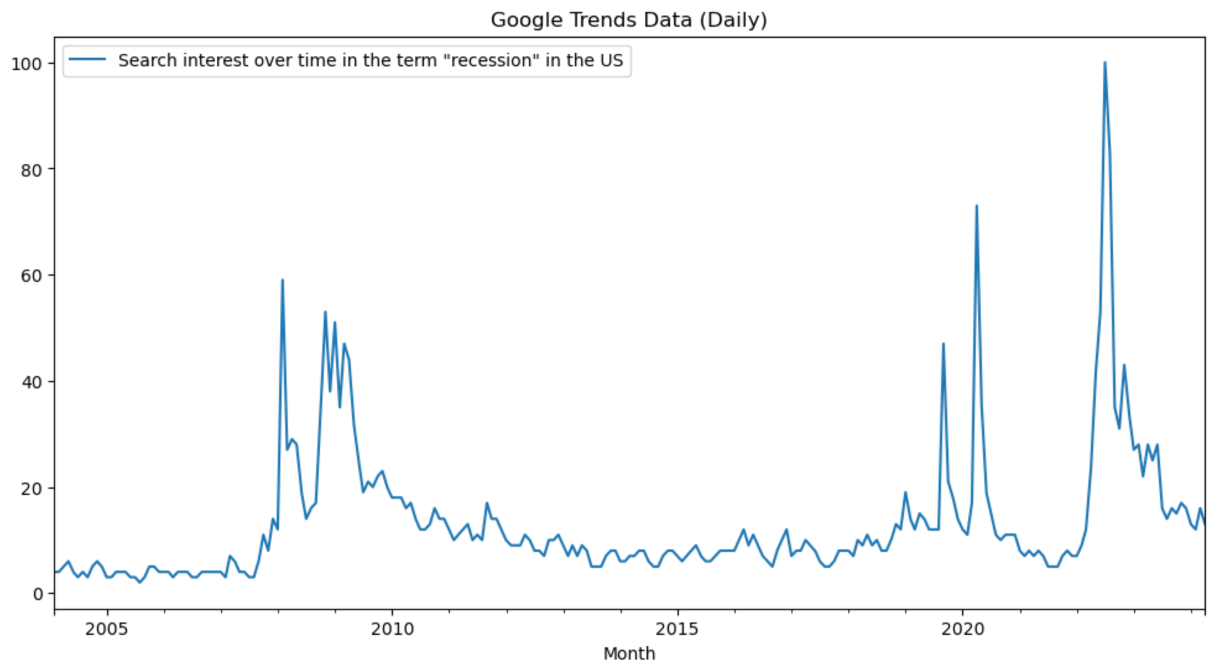
In [15]:
```
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(google_trends_data)
date_fmt = mdates.DateFormatter("%Y-%m")
plt.xlim(google_trends_data.index[0], google_trends_data.index[-1])
plt.xticks(rotation=45)
plt.title("Google Trends Data (Monthly)")
plt.legend(["Search interest over time in the term 'recession' in the US"])
plt.show()
```



But not every month-end is a trading day. Also, what value should the model train on for all the days in between month-ends? Below, we have provided you with code to convert the monthly data to daily and interpolate the end-of-month values to get all the in-between values. You will be using this new `google_trends_daily` data going forward.

In [16]:
```
google_trends_daily = google_trends_data.resample('D').asfreq().interpolate(
```

In [17]:
```
# The shape of the chart should not have changed
google_trends_daily.plot.line(title="Google Trends Data (Daily)", figsize=(1
    labels=['Search interest over time in the term "recession" in the US']
 );
```



2. Feature Engineering and Analysis

In this section, you will create a new DataFrame called `data` which will house all of the features as well as the prediction target. Then you will analyze the features and look for potentially problematic features.

Start by running the cell below to create `data` as an empty DataFrame with just an index that matches `XLV` 's.

In [18]:
```
data = pd.DataFrame(index=xlv_data.index)
data.head()
```

Out[18]:

Date
2004-01-02
2004-01-05
2004-01-06
2004-01-07
2004-01-08

2.1. Feature Engineering

2.1.1. Month and Weekday

Add the `month` and `weekday` columns to `data` as categorical features (integer labels) from its index.

In [19]:
```
data['Month'] = data.index.month
data['Weekday'] = data.index.weekday
data.tail()
```

Out[19]:

Date	Month	Weekday
2024-03-22	3	4
2024-03-25	3	0
2024-03-26	3	1
2024-03-27	3	2
2024-03-28	3	3

You do not want to train a model using these columns as they are, because the numbers themselves and the inherent "order" of months and weekdays do not really have any significance, but the model may interpret them as meaningful. You could either (a) use one-hot encoding to turn each category to a separate binary feature, or (b) treat them as cyclical features. The choice is somewhat arbitrary and depends on how important a "feature" you believe the cyclicality to be.

Below, you will:

- Treat `month` as a cyclical feature, creating two features (`month_sin` and `month_cos`). (👉See: Trigonometric features)
- One-hot-encode `weekday` and create five additional features of type `int32` (one for each business day) with the `weekday` prefix. (👉See: `pandas.get_dummies()`)
- Make sure the original `month` and `weekday` columns are no longer present in `data` . (`drop()` them if necessary.)

In [20]:
```
# Treat `month` as a "cyclical" feature with a period of 12 months.
data["month_sin"] = np.sin(2 * np.pi * data['Month'] / 12)
data["month_cos"] = np.cos(2 * np.pi * data['Month'] / 12)

# Drop the original `month` column.
data.drop('Month', axis=1, inplace=True)

# Treat `weekday` as a "categorical" feature and one-hot-encode it.
weekdays = pd.get_dummies(data['Weekday'], prefix='weekday', drop_first=Fals
data = pd.concat([data, weekdays], axis=1)
data.drop('Weekday', axis=1, inplace=True)
data.head()
```

Out[20]:

Date	month_sin	month_cos	weekday_0	weekday_1	weekday_2	weekday_3
2004-01-02	0.5	0.866025	False	False	False	False
2004-01-05	0.5	0.866025	True	False	False	False
2004-01-06	0.5	0.866025	False	True	False	False
2004-01-07	0.5	0.866025	False	False	True	False
2004-01-08	0.5	0.866025	False	False	False	True

2.1.2. Historical Returns

Next, add features for historical returns of the XLV ETF from its `Adj Close` column. For each date, calculate rolling **simple** returns over the past 1, 5, 10 and 20 days. Create 4 columns in `data` named `ret_#d_hist` where `#` is the lookback period. The list `hist_ret_lookbacks` is provided if you wish to use it.

In [21]:
```
# Create features for 1-day, 5-day, 10-day and 20-day historical returns
hist_ret_lookbacks = [1, 5, 10, 20]
```

```
for i in hist_ret_lookbacks:
    column_name = f'ret_{i}d_hist'
    data[column_name] = xlv_data['Adj Close'].pct_change(periods=i)
data.head()
```

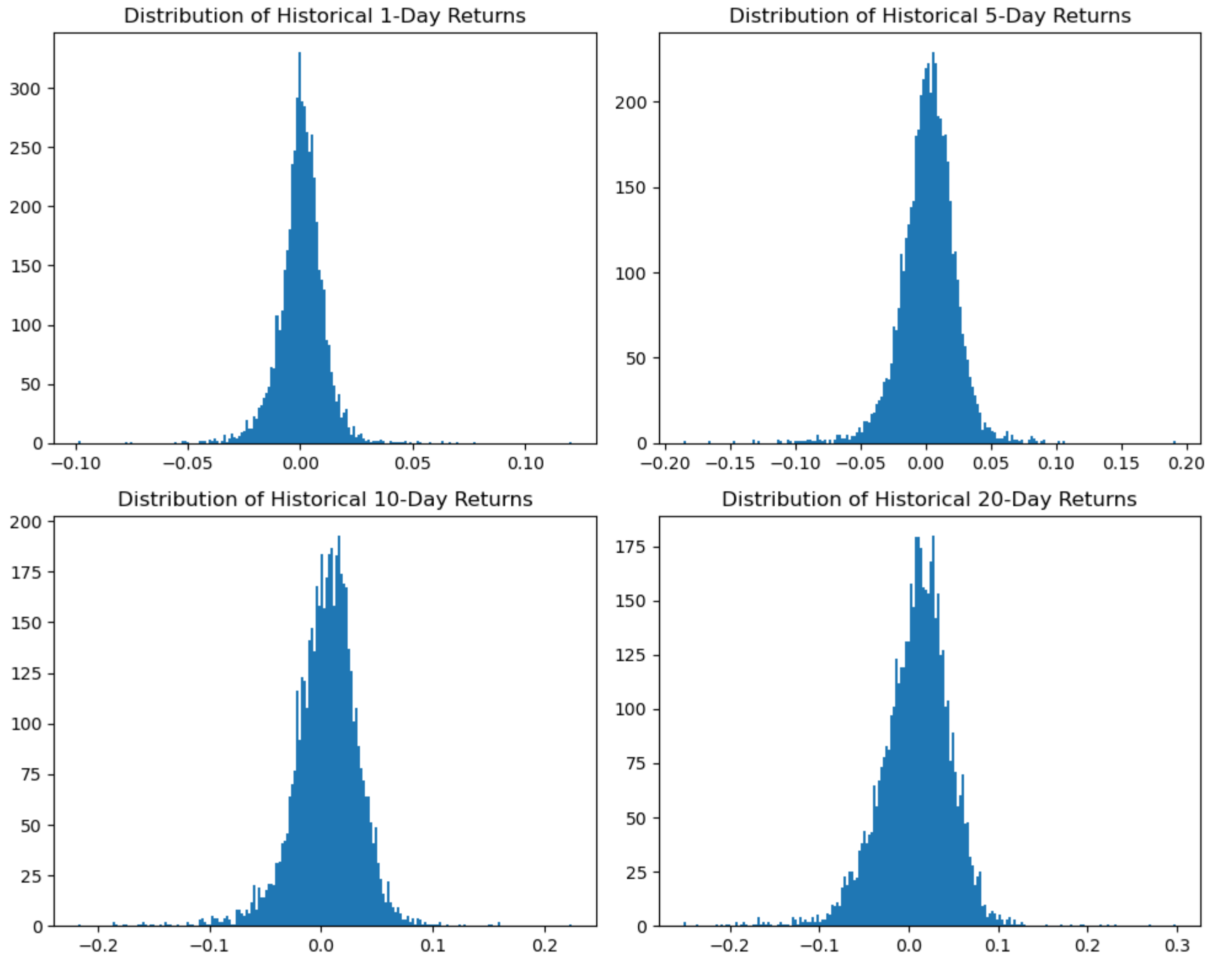
Out[21]:

	month_sin	month_cos	weekday_0	weekday_1	weekday_2	weekday_3
Date						
2004-01-02	0.5	0.866025	False	False	False	False
2004-01-05	0.5	0.866025	True	False	False	False
2004-01-06	0.5	0.866025	False	True	False	False
2004-01-07	0.5	0.866025	False	False	True	False
2004-01-08	0.5	0.866025	False	False	False	True

The cell below plots the histograms of the returns you just calculated. They should look normally distributed around zero.

In [22]:

```
hist_ret_lookbacks = [1, 5, 10, 20] # In case it was deleted from the previo

fig, axs = plt.subplots(2, 2, figsize=(10, 8))


def plot_hist_returns(ax, data, col, title):
    ax.hist(data[col], bins=200)
    ax.set_title(title)


for i, n_days in enumerate(hist_ret_lookbacks):
    plot_hist_returns(
        axs[i // 2, i % 2], data, f"ret_{n_days}d_hist", f"Distribution of H
    )

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



2.1.3. Trade Volumes

As trading volumes span several orders of magnitude, take the natural logarithm of `Volume` and use it as a feature instead. This helps emphasize variations in its lower range. Use `np.log()` and call this new feature `log_volume` .

Note: For tree-based models such as Decision Trees and Random Forests, scaling is not necessary. But feature scaling becomes critically important if you use other model types (e.g. distance-based models).

In [23]:
```
data["log_volume"] = np.log(xlv_data['Volume'])
data.head()
```

Out[23]:

Date	month_sin	month_cos	weekday_0	weekday_1	weekday_2	weekday_3
2004-01-02	0.5	0.866025	False	False	False	False
2004-01-05	0.5	0.866025	True	False	False	False
2004-01-06	0.5	0.866025	False	True	False	False
2004-01-07	0.5	0.866025	False	False	True	False
2004-01-08	0.5	0.866025	False	False	False	True

2.1.4. Technical Indicators

Add a feature named `ibs` which is calculated as (Close - Low) / (High - Low). This measure, a number between zero and one and sometimes referred to as the "Internal Bar Strength", denotes how "strong" the closing price is relative to the high and low prices within the same period.

> **Note:** Make sure to use `Close` (not `Adj Close`).

In [24]:

```
# Engineer the technical indicator "Internal Bar Strength" (IBS) from XLV's
data["ibs"] = ((xlv_data['Close'] - xlv_data['Low'])/(xlv_data['High'] - xlv
data.head()
```

Out[24]:

Date	month_sin	month_cos	weekday_0	weekday_1	weekday_2	weekday_3
2004-01-02	0.5	0.866025	False	False	False	False
2004-01-05	0.5	0.866025	True	False	False	False
2004-01-06	0.5	0.866025	False	True	False	False
2004-01-07	0.5	0.866025	False	False	True	False
2004-01-08	0.5	0.866025	False	False	False	True

Run the cell below to add a few more technical indicators, including Bollinger Band features and indicators, as well as the Relative Strength Index (RSI).

In [25]:

```
# Get some more technical indicators using the `ta` library

indicator_bb = BollingerBands(close=xlv_data["Close"], window=20, window_dev
indicator_rsi = RSIIndicator(close=xlv_data["Close"], window=14)

# Add Bollinger Bands features
data["bb_bbm"] = indicator_bb.bollinger_mavg()
data["bb_bbh"] = indicator_bb.bollinger_hband()
data["bb_bbl"] = indicator_bb.bollinger_lband()
```

```
# Add Bollinger Band high and low indicators
data["bb_bbhi"] = indicator_bb.bollinger_hband_indicator()
data["bb_bbli"] = indicator_bb.bollinger_lband_indicator()

# Add Width Size and Percentage Bollinger Bands
data["bb_bbw"] = indicator_bb.bollinger_wband()
data["bb_bbp"] = indicator_bb.bollinger_pband()

# Add RSI
data["rsi"] = indicator_rsi.rsi()
```

2.1.5. The Target of Prediction

Add the column `tgt_is_pos_ret_5d_fut` as type `int` to `data`, denoting whether forward-looking 5-day returns on each day are positive (a value of `1`) or negative (a value of `0`).

> **Note:** Again, as before, calculte **simple** returns from the `Adj Close` column of `xlv_data`.

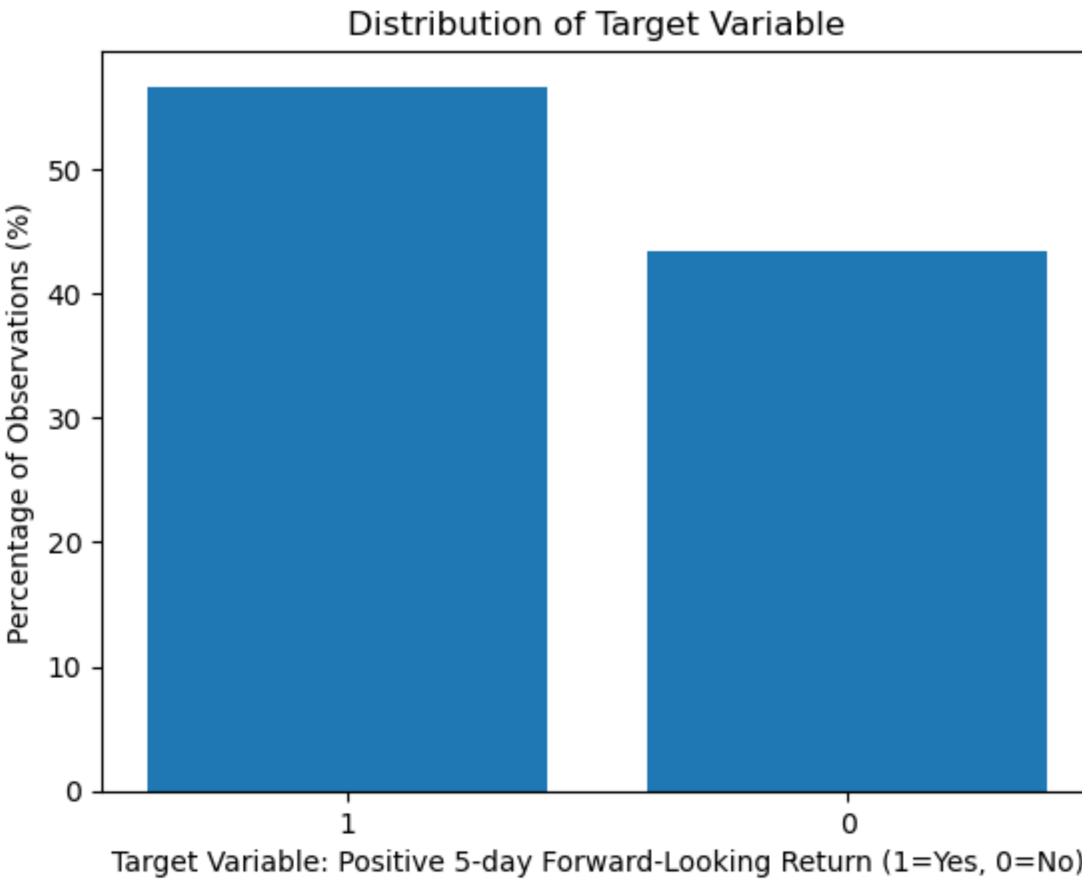
In [26]:
```
# Create the prediction target: an integer indicating whether future 5-day r
data['ret_5d_fut'] = (xlv_data['Adj Close'].shift(-5)/xlv_data['Adj Close'])
data['tgt_is_pos_ret_5d_fut'] = (data['ret_5d_fut'] > 0).astype(int)
```

Run the cells below to get an idea of how balanced the distribution of the target variable is throughout the data.

In [27]:
```
target_col_name = "tgt_is_pos_ret_5d_fut"
# Inspect the distribution of the target variable
target_value_counts = data[target_col_name].value_counts()
target_value_counts / len(data)
```

Out[27]:
```
tgt_is_pos_ret_5d_fut
1    0.566156
0    0.433844
Name: count, dtype: float64
```

In [28]:
```
target_value_percentages = target_value_counts / len(data) * 100

plt.bar(target_value_percentages.index.astype(str), target_value_percentages
plt.xlabel("Target Variable: Positive 5-day Forward-Looking Return (1=Yes, 0
plt.ylabel("Percentage of Observations (%)")
plt.title("Distribution of Target Variable")
plt.show()
```



Does the data look relatively balanced or grossly unbalanced in the distribution of the target variable? Why is this important?

In [29]:
```
answer = "The data is close to 50% so it is balanced. If the target variable
```

2.1.6. Stitching Everything Together

You will now add the `vix_data` and `google_trends_daily` as features to `data`. You will also rename the column corresponding to the VIX feature. Run the cell below to do so.

In [30]:
```
# Join with the Google Trends data and VIX data
data = data.join(google_trends_daily, how="left")
data = data.join(vix_data["Adj Close"], how="left")
data.rename(columns={"Adj Close": "vix"}, inplace=True)
```

2.2. Further Data Preprocessing and Cleaning

While engineering new features, some `NaN` values were created. You now need to clean the combined DataFrame. Inspect `data` to see how many `NaN` values there are per column.

In [31]:
```
data.isnull().sum()
```

Out[31]:
```
month_sin                  0
month_cos                  0
weekday_0                  0
weekday_1                  0
weekday_2                  0
weekday_3                  0
weekday_4                  0
ret_1d_hist                1
ret_5d_hist                5
ret_10d_hist              10
ret_20d_hist              20
log_volume                 0
ibs                        0
bb_bbm                    19
bb_bbh                    19
bb_bbl                    19
bb_bbhi                    0
bb_bbli                    0
bb_bbw                    19
bb_bbp                    19
rsi                       13
ret_5d_fut                 5
tgt_is_pos_ret_5d_fut      0
recession_search_trend    20
vix                        0
dtype: int64
```

Some features, such as historical returns, RSI, Bollinger Bands and BB indicators cannot be calculated for the first `n` days due to their "rolling" nature. In general, missing values can sometimes be imputed with reasonable estimates. But here you will simply drop the rows containing them. The largest `n` is `20`, corresponding to the calculation of 20-day historical returns. Drop the first 20 rows of `data`.

In [32]:
```
data = data.drop(data.index[:20])
data.shape
```

Out[32]:
```
(5074, 25)
```

Are there any more missing values?

In [33]:
```
data.isnull().sum()
```

Out[33]:
```
month_sin                  0
month_cos                  0
weekday_0                  0
weekday_1                  0
weekday_2                  0
weekday_3                  0
weekday_4                  0
ret_1d_hist                0
ret_5d_hist                0
ret_10d_hist               0
ret_20d_hist               0
log_volume                 0
ibs                        0
bb_bbm                     0
bb_bbh                     0
bb_bbl                     0
bb_bbhi                    0
bb_bbli                    0
bb_bbw                     0
bb_bbp                     0
rsi                        0
ret_5d_fut                 5
tgt_is_pos_ret_5d_fut      0
recession_search_trend     0
vix                        0
dtype: int64
```

Even if there aren't, you remember that when you calculated the target variable (`tgt_is_pos_ret_5d_fut`) based on forward-looking 5-day rolling returns, you could not have known future returns for the last five days of `data` ! Therefore the last 5 rows of data should be dropped.

In [34]:
```
data = data.drop(data.index[-5:])
```

Let us take a final look at the types and statistical characteristics of the set of features and targets.

In [35]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 5069 entries, 2004-02-02 to 2024-03-21
Data columns (total 25 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   month_sin               5069 non-null   float64
 1   month_cos               5069 non-null   float64
 2   weekday_0               5069 non-null   bool   
 3   weekday_1               5069 non-null   bool   
 4   weekday_2               5069 non-null   bool   
 5   weekday_3               5069 non-null   bool   
 6   weekday_4               5069 non-null   bool   
 7   ret_1d_hist             5069 non-null   float64
 8   ret_5d_hist             5069 non-null   float64
 9   ret_10d_hist            5069 non-null   float64
 10  ret_20d_hist            5069 non-null   float64
 11  log_volume              5069 non-null   float64
 12  ibs                     5069 non-null   float64
 13  bb_bbm                  5069 non-null   float64
 14  bb_bbh                  5069 non-null   float64
 15  bb_bbl                  5069 non-null   float64
 16  bb_bbhi                 5069 non-null   float64
 17  bb_bbli                 5069 non-null   float64
 18  bb_bbw                  5069 non-null   float64
 19  bb_bbp                  5069 non-null   float64
 20  rsi                     5069 non-null   float64
 21  ret_5d_fut              5069 non-null   float64
 22  tgt_is_pos_ret_5d_fut   5069 non-null   int64  
 23  recession_search_trend  5069 non-null   float64
 24  vix                     5069 non-null   float64
dtypes: bool(5), float64(19), int64(1)
memory usage: 856.4 KB
```

In [36]:
```
data.describe()
```

Out[36]:

	month_sin	month_cos	ret_1d_hist	ret_5d_hist	ret_10d_hist	ret_2
count	5.069000e+03	5.069000e+03	5069.000000	5069.000000	5069.000000	5069.
mean	-2.838770e-03	-7.572043e-03	0.000427	0.002097	0.004185	0.
std	7.097124e-01	7.045852e-01	0.010493	0.021729	0.029813	0.
min	-1.000000e+00	-1.000000e+00	-0.098610	-0.185835	-0.217250	-0
25%	-8.660254e-01	-8.660254e-01	-0.004458	-0.009204	-0.011609	-0
50%	-2.449294e-16	-1.836970e-16	0.000633	0.002931	0.005917	C
75%	8.660254e-01	5.000000e-01	0.005891	0.014858	0.021953	0.
max	1.000000e+00	1.000000e+00	0.120547	0.192308	0.223935	C

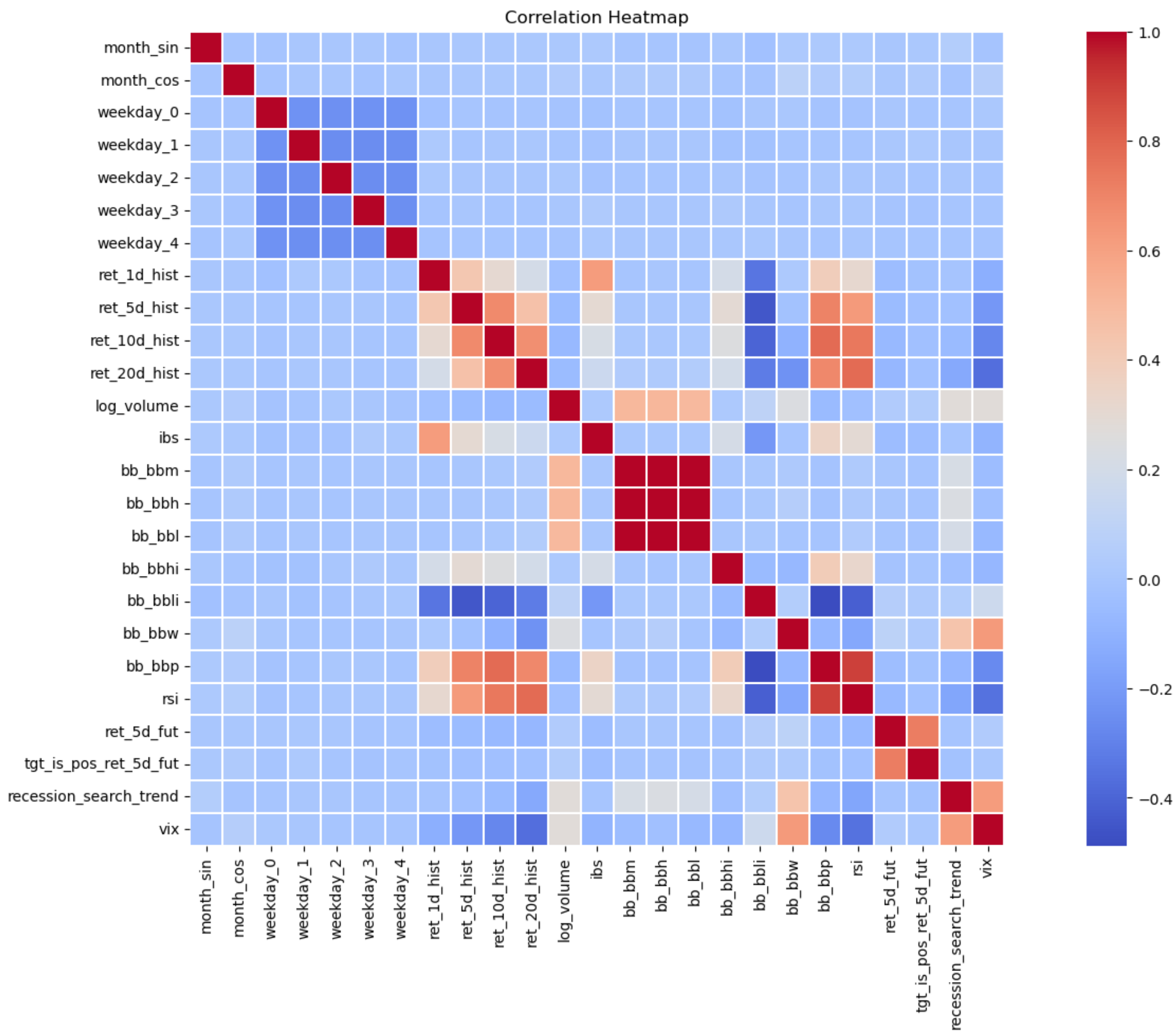
2.3. Correlation Analysis

Correlation analysis can be a rough and early form of feature importance analysis.

Features that are highly correlated (in either direction) with each other but not with the target variable, are a sign of multicollinearity problems, which means they may not contribute much additional information in predicting the target. In fact, depending on the algorithm used, multicollinearity may result in stability and reliability issues. Checking the correlation matrix can be helpful in identifying such features.

Plot the heatmap of the correlation matrix of features/target and identify a cluster of 3 features that are almost certainly collinear. (Hint: `bb_bbm` is one of them.) You can pass the correlation matrix directly to Seaborn's `heatmap()` method.

In [37]:
```
corr_mat = data.corr()
plt.figure(figsize=(20, 10))
sns.heatmap(corr_mat, annot=False, fmt='.2f', cmap='coolwarm', cbar=True, sq
plt.title("Correlation Heatmap")
plt.show()
```



In such scenarios, we usually eliminate all but one of the collinear features. Keep `bb_bbm` and drop the other two features that are highly linearly related to it.

In [38]:
```
#drop bbw and bbh
```

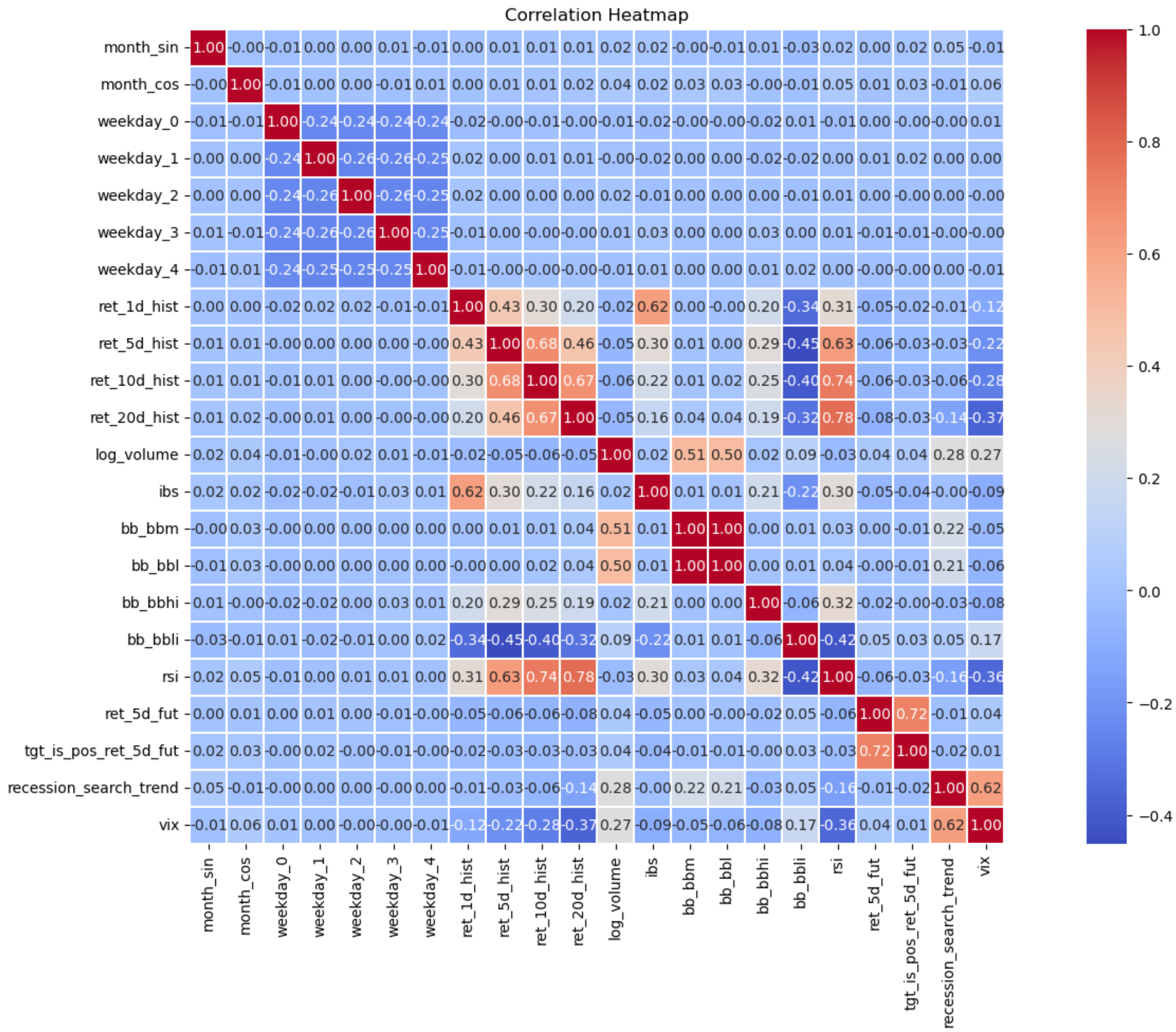
```
data = data.drop(['bb_bbw', 'bb_bbh'], axis=1)
```

There is also one feature that is very highly correlated with `rsi` (which makes intuitive sense, as it, too, is a measure of relative strength). Find it an eliminate it, leaving `rsi` intact.

In [39]:
```
data = data.drop(['bb_bbp'], axis=1)
```

Plot the heatmap of the new, reduced correlation matrix.

In [40]:
```
corr_mat = data.corr()
plt.figure(figsize=(20, 10))
sns.heatmap(corr_mat, annot=True, fmt='.2f', cmap='coolwarm', cbar=True, squ
plt.title("Correlation Heatmap")
plt.show()
```



Features that are highly correlated (negatively or positively) **with the target variable** are likely more important. Which two (2) independent variables (features) are correlated more than 4% (**in either direction**) with the boolean target variable denoting whether 5-day future returns are positive?

In [41]:
```
answer = "The bolliger band lower bound is highly correlated to 5, 10, 20 da
```

3. The Training-Validation-Testing Split

In this section, you will split the `data` set into two sets: the training and validation set, and the testing set. You will then come up with a baseline score so that you have a reference point for evaluating your model's performance.

Note: Technically, since you are not going to use classical statistics-based time-series prediction methods (such as ARIMA), you can shuffle the data before splitting it. But for ease of interpretability and backtesting, you may as well keep the data in its original order. This is fine as long as the distributions of features and the target variable do not significantly shift over time. - And that is an important assumption related to drift analysis, which was covered in the course, but we will not get to in this project.

3.1. The Split

It is time to split the data, temporally, into the training + validation and testing sets. You will train and optimize (i.e. cross-validate) your model on the first 80% of the data, and use the remaining 20% for the test set (i.e. to evaluate the performance of your model). Use the `train_test_split()` method from scikit-learn's `model_selection` module to perform the split.

> **Note:** Please make sure to set `shuffle=False` and `random_state=RANDOM_STATE`.

In [42]:
```
from sklearn.model_selection import train_test_split
X = data.drop(columns=['tgt_is_pos_ret_5d_fut'])
y = data['tgt_is_pos_ret_5d_fut']
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=
X_train_val.shape, X_test.shape, y_train_val.shape, y_test.shape
```

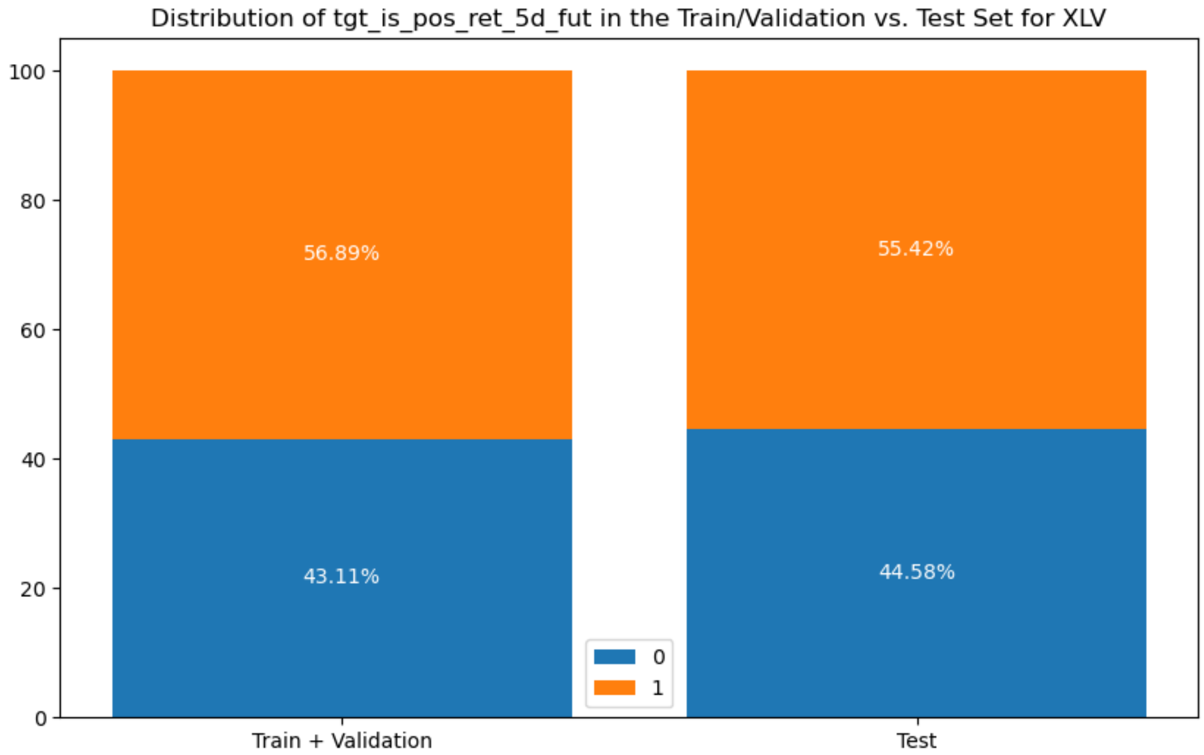
Out[42]:
```
((4055, 21), (1014, 21), (4055,), (1014,))
```

3.2. Baseline Model and Score

Earlier, you inspected the distribution of the target variable across the entire data set. Run the cell below to analyze at the distribution of the target variable in each split.

In [43]:
```
train_val_pct = y_train_val.value_counts(normalize=True) * 100
test_pct = y_test.value_counts(normalize=True) * 100

categories = ["Train + Validation", "Test"]
zero_counts = [train_val_pct[0], test_pct[0]]
one_counts = [train_val_pct[1], test_pct[1]]
```

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

ax.bar(categories, zero_counts, label="0")
ax.bar(categories, one_counts, bottom=zero_counts, label="1")

# Add text annotations
for i, (zero, one) in enumerate(zip(zero_counts, one_counts)):
    ax.text(i, zero / 2, f"{zero:.2f}%", ha="center", va="center", color="wh
    ax.text(
        i,
        zero + one / 2,
        f"{one:.2f}%",
        ha="center",
        va="center",
        color="white",
    )
ax.set_title(f"Distribution of {target_col_name} in the Train/Validation vs.
ax.legend()

plt.show()
```



If you were to devise a simple model that naively always predicted the majority class, what would the accuracy score of your model be on the training+validation set? How about on the testing set? Consider the latter your baseline score, i.e. a reference score to compare your more sophisticated model's performace to.

In [44]:
```
majority_class_train = y_train_val.mode()[0]
majority_class_test = y_test.mode()[0]
baseline_accuracy_train_score = (y_train_val == majority_class_train).mean()
baseline_accuracy_test_score = (y_test == majority_class_test).mean()
print(baseline_accuracy_train_score, baseline_accuracy_test_score)
```

```
0.5689272503082614 0.5542406311637081
```

4. Model Training and Tuning

In this section, you will train a `RandomForestClassifier`, a robust, versatile ensemble learning method that uses "bagging" (also known as "bootstrap aggregating") to train multiple Decision Trees. The technical details of the model are beyond the scope of this course, but you may read more about it here.

Run the cell below which defines a function that allows you to plot learning curves annotated with a hyperparameter named `max_depth` which you pass to it.

In [45]:
```
def plot_learning_curves(train_sizes, train_scores, test_scores, max_depth,
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    axs.fill_between(
        train_sizes,
        train_scores_mean - train_scores_std,
        train_scores_mean + train_scores_std,
        alpha=0.1,
        color="b",
    )
    axs.fill_between(
        train_sizes,
        test_scores_mean - test_scores_std,
        test_scores_mean + test_scores_std,
        alpha=0.1,
        color="r",
    )
    axs.plot(
        train_sizes,
        train_scores_mean,
        "o-",
        color="b",
        label="Average Score on Training Sets",
    )
    axs.plot(
        train_sizes,
        test_scores_mean,
        "o-",
        color="r",
        label="Average Score on Test Sets",
    )
    axs.set_xlabel("Training examples")
    axs.set_ylabel("Score")
    axs.set_title(f"Learning Curves (max_depth={max_depth})")
    axs.legend(loc="center left")
    axs.grid(True)
```

Below is the first iteration of your model. It uses the default values for most of its hyperparameters. We have only specified one hyperparameter, `max_depth=10`.

In [46]:
```
max_depth = 10
model = RandomForestClassifier(max_depth=max_depth, random_state=RANDOM_SEED
model.fit(X_train_val, y_train_val)
```

Out[46]:
```
▾ RandomForestClassifier
RandomForestClassifier(max_depth=10, n_jobs=-1, random_state=42)
```

Use the `learning_curve()` method from scikit-learn's `model_selection` module to cross-validate your model, with `accuracy` as the `scoring` metric. Use 10%, 20%, 30%,... , and 100% of the training+validatin data, with 5-fold cross-validation.

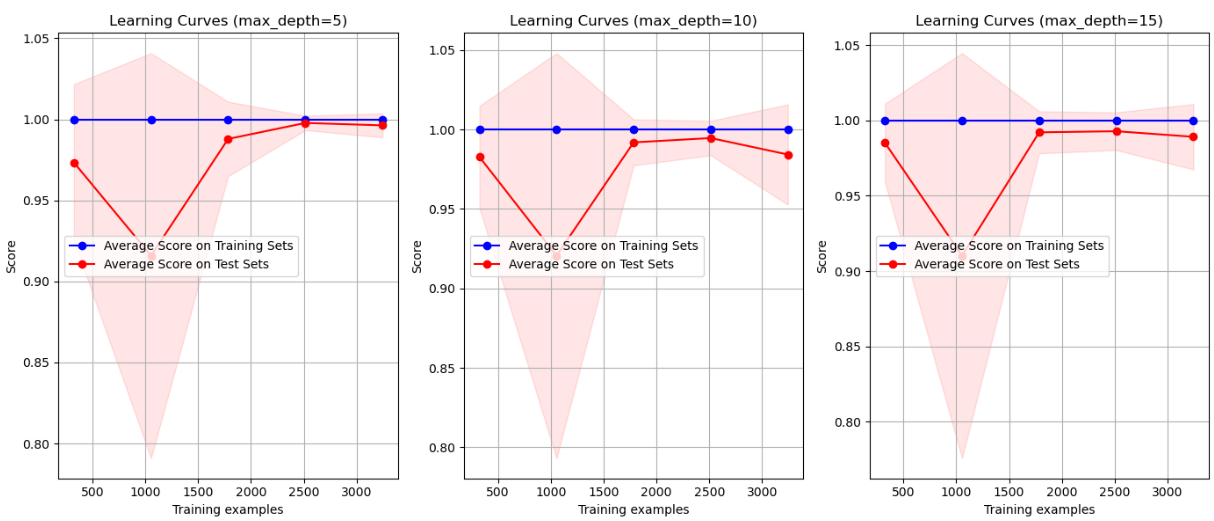
In [47]:
```
from sklearn.model_selection import learning_curve
from sklearn.model_selection import cross_val_score

accuracy = cross_val_score(model, X_train_val, y_train_val, cv=5, n_jobs=-1,

train_sizes, train_scores, test_scores = learning_curve(model, X_train_val,
```

Inspect the learning curves.

In [48]:
```
figure = plt.figure(figsize=(10, 6))
axs = figure.gca()

plot_learning_curves(train_sizes, train_scores, test_scores, max_depth, axs)

plt.show()
```

Wondering what effect different values of the `max_depth` hyperparameter have, you decide to experiment with lower (`10`) and higher (`20`) values of it to see how the plots change. Run the cell below to help you answer the questions that follow it.

In [49]:
```
fig, axs = plt.subplots(1, 3, figsize=(14, 6))
max_depth_range = [5, 10, 15]
for i, max_depth in enumerate(max_depth_range):
    model = RandomForestClassifier(max_depth=max_depth, random_state=RANDOM_
    train_sizes, train_scores, test_scores = learning_curve(
        model, X_train_val, y_train_val, train_sizes=train_sizes, cv=5, scor
    )
    plot_learning_curves(train_sizes, train_scores, test_scores, max_depth, 

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



With a value of `max_depth=15` , does your model overfit or underfit?

In [50]:
```
answer = "The model overfits"
```

With a value of `max_depth=15` , is your performance metric (accuracy score) more likely to improve with more training data or with higher model complexity?

In [51]:
```
answer = "our performance mentric improves with more training samples not ma
```

Random Forest Classifiers have several other hyperparameters, such as `min_samples_split` (default=2), `min_samples_leaf` (default=1) and `n_estimators` (default=100). So far, you have been tuning your model manually. But with all the possible combinations of hyperparameters, this is not tractable.

Use grid search cross-validation (the `GridSearchCV` class from scikit-learn's `model_selection` module) to find the optimal combination of hyperparameters from the search space specified below:

- `max_depth` = 2, 3, 4 or 5
- `min_samples_leaf` = 1, 2, 3 or 4
- `n_estimators` = 50, 75, 100, 125, or 150

As before, use 5-fold cross-validation and accuracy as the `scoring` metric. Name your tuning model `search` .

> Note: Setting `n_jobs=-1` will allow Python to take advantage of parallel computing on your computer to speed up the training.

In [52]:
```
from sklearn.model_selection import GridSearchCV

grid = {
    'max_depth': [2, 3, 4, 5],
    'min_samples_leaf': [1, 2, 3, 4],
    'n_estimators': [50, 75, 100, 125, 150]
}

search = GridSearchCV(model, param_grid=grid, n_jobs=-1, cv=5, scoring='accu

search.fit(X_train_val, y_train_val)
```

Out[52]:
```
▸ GridSearchCV  i ?

▸ best_estimator_:
  RandomForestClassifier

  ▸ RandomForestClassifier  ?
```

Run the cell below to see the top 5 best performing hyperparameter combinations.

In [53]:
```
pd.DataFrame(search.cv_results_).sort_values("rank_test_score").head()
```

Out[53]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth
70	0.106819	0.012411	0.037033	0.021689	5
65	0.132063	0.024044	0.036271	0.010956	5
74	0.432072	0.056825	0.053397	0.017871	5
79	0.258821	0.080189	0.034365	0.007092	5
75	0.120758	0.024387	0.041842	0.020654	5

Looking at the results of GridSearchCV, which hyperparameters yield the highest mean test score?

In [54]:
```
best_max_depth = search.best_params_['max_depth']
best_min_samples_leaf = search.best_params_['min_samples_leaf']
best_n_estimators = search.best_params_['n_estimators']
print(best_max_depth, best_min_samples_leaf, best_n_estimators)
```

```
5 3 50
```

Looking more closely at the DataFrame of top 5 results, varying which hyperparameter did not seem to have any effect, at least in the top-ranking score?

In [55]:
```
answer = "all top 5 results have max_depth parameter set to 5 so seems like
```

5. Model Evaluation and Interpretation

In this section, you will evaluate the performance metrics of the best model you found in the previous section and analyze feature importance in relation to model performance.

5.1. Evaluation (Performance Metrics)

It is finally time to train your model on the entire training + validation set with the optimal set of hyperparameters you just found, and evaluate its performance on the test set.

Train (`fit()`) a `RandomForestClassifier` on the training data with the optimal combination of hyperparameters you found in the previous section. Name it 'clf'.

> **Note:** Remember to set `random_state=RANDOM_SEED` for consistency of results, and set `n_jobs=-1` to automatically speed up the run.

In [56]:
```
clf = RandomForestClassifier(max_depth=5, min_samples_leaf=3, n_estimators=5
clf.fit(X_train_val, y_train_val)
```

Out[56]:
```
▾                    RandomForestClassifier                         i ?

RandomForestClassifier(max_depth=5, min_samples_leaf=3, n_estimator
s=50,
                         n_jobs=-1, random_state=42)
```

Store your trained model's predictions on the **testing** set in a variable named `y_test_pred`.

In [57]:
```
y_test_pred = clf.predict(X_test)
```

Complete the Python dictionary in the code cell below to evaluate your model and answer the questions that follow.

In [58]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f

evaluation = {
    "accuracy": accuracy_score(y_test, y_test_pred),
    "precision": precision_score(y_test, y_test_pred),
    "recall": recall_score(y_test, y_test_pred),
    "f1": f1_score(y_test, y_test_pred)
}
display(evaluation)
```

```
{'accuracy': 0.9990138067061144,
 'precision': 1.0,
 'recall': 0.998220640569395,
 'f1': 0.9991095280498664}
```

Explain, in words and citing the actual numbers from the evaluation report above, what the **precision** and **recall** scores mean.

In [59]:
```
answer = "we get a precision of 1.0 and recall of 99%. Precision=TruePositiv
```

Run the cell below to get a more detailed report.

In [60]:
```
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       452
           1       1.00      1.00      1.00       562

    accuracy                           1.00      1014
   macro avg       1.00      1.00      1.00      1014
weighted avg       1.00      1.00      1.00      1014
```

How many True Negatives, False Negatives, False Positives and True Positives did the model predict on the test set? Find out using the `confusion_matrix()` method from scikit-learn's `metrics` module.

In [61]:
```
cm = confusion_matrix(y_test, y_test_pred)
cm
```

Out[61]:
```
array([[452,   0],
       [  1, 561]])
```

Answer the question from earlier.

> Note: Feel free to rename the variables. We will not reference them later.

In [62]:
```
num_TrueNeg = cm[0, 0]
num_FalseNeg = cm[1, 0]
num_FalsePos = cm[0, 1]
num_TruePos = cm[1, 1]
```

Is the model overfitting or underfitting? Did it manage to capture the variance on the training set but fail to generalize to the testing set? Take a look at the `classification_report()` and `confusion_matrix()` on the **training** data.

In [63]:
```
y_train_val_pred = clf.predict(X_train_val)
print(classification_report(y_train_val, y_train_val_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      1748
           1       1.00      1.00      1.00      2307

    accuracy                           1.00      4055
   macro avg       1.00      1.00      1.00      4055
weighted avg       1.00      1.00      1.00      4055
```

In [64]:
```
confusion_matrix(y_train_val, y_train_val_pred)
```

Out[64]:
```
array([[1748,    0],
       [   0, 2307]])
```

How does your model's performance compare to the baseline in terms of `accuracy` ?

In [65]:
```
answer = "Our model's baseline test accuracy has gone up from 55% to 99%"
```

How do the `precision` and `recall` of your model compare to those of the baseline model?

In [66]:
```
answer = "We did not calculate precision and recall on the baseline model."
```

5.2. Revisiting Feature Importance

You decide to see if there are any features that are not contributing significantly to the performance of the model. Use the `feature_importances_` property of your classifier.

In [67]:
```
feats_imp = clf.feature_importances_
sorted_indices = np.argsort(feats_imp)
```

Create a new training set named `X_train_val_reduced` and a new testing set named `X_test_reduced` by eliminating any features from the old train/test sets that had a feature importance of less than `0.5%` .

In [68]:
```
# Drop features that have an importance of 0.05% or less...
feats_to_drop = []
for idx in sorted_indices:
    if np.abs(feats_imp[idx]) < 0.005:
        feats_to_drop.append(idx)
X_train_val_reduced = np.delete(X_train_val, feats_to_drop, axis=1)
X_test_reduced = np.delete(X_test, feats_to_drop, axis=1)
```

Re-do your grid search cross-validation with the same grid of hyperparameters as before but with the **reduced** feature set.

In [69]:
```
model = RandomForestClassifier(random_state=RANDOM_SEED, n_jobs=-1)

search = GridSearchCV(model, param_grid=grid, n_jobs=-1, cv=5, scoring='accu

search.fit(X_train_val_reduced, y_train_val)
```

Out[69]:
```
▸ GridSearchCV
i ?
▸ best_estimator_:
RandomForestClassifier
▸ RandomForestClassifier ?
```

In [70]:
```
pd.DataFrame(search.cv_results_).sort_values("rank_test_score").head()
```

Out[70]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth
0	0.096968	0.003509	0.031745	0.014506	2
41	0.196854	0.030897	0.042096	0.013851	4
42	0.263496	0.055707	0.046031	0.016895	4
43	0.322338	0.051254	0.043069	0.006255	4
44	0.462734	0.059806	0.050756	0.005981	4

In [71]:
```
search.best_params_
```

Out[71]:
```
{'max_depth': 2, 'min_samples_leaf': 1, 'n_estimators': 50}
```

Train a new classifier on the reduced feature set with the best hyperparameters combination from the new grid search and then inspect its accuracy on the test set (with the **reduced** feature set).

In [72]:
```
clf = RandomForestClassifier(max_depth=2, min_samples_leaf=1, n_estimators=5
clf.fit(X_train_val_reduced, y_train_val)
```

Out[72]:
```
▾ RandomForestClassifier                                   i ?

RandomForestClassifier(max_depth=2, n_estimators=50, n_jobs=-1, ran
dom_state=42)
```

In [73]:
```
y_test_pred = clf.predict(X_test_reduced)
evaluation = {
    "accuracy": accuracy_score(y_test, y_test_pred),
    "precision": precision_score(y_test, y_test_pred),
    "recall": recall_score(y_test, y_test_pred),
    "f1": f1_score(y_test, y_test_pred)
}
display(evaluation)
```

```
{'accuracy': 1.0, 'precision': 1.0, 'recall': 1.0, 'f1': 1.0}
```

In [74]:
```
print(classification_report(y_test, y_test_pred, zero_division=0))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       452
           1       1.00      1.00      1.00       562

    accuracy                           1.00      1014
   macro avg       1.00      1.00      1.00      1014
weighted avg       1.00      1.00      1.00      1014
```

In [75]:
```
confusion_matrix(y_test, y_test_pred)
```

Out[75]:
```
array([[452,   0],
       [  0, 562]])
```

How does the accuracy compare to your last trained model?

In [76]:
```
answer = "Accuracy slightly increases, goes up to 100% from 99% now."
```

How does the accuracy compare to the baseline?

In [77]:
```
answer = "The baseline accuracy was 55%. With reduced feature set we gte a 1
```

Take a look at the classification report and confusion matrices on the **training data** with the **reduced feature set** as well:

In [78]:
```
y_train_val_pred = clf.predict(X_train_val_reduced)
print(classification_report(y_train_val, y_train_val_pred, zero_division=0))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      1748
           1       1.00      1.00      1.00      2307

    accuracy                           1.00      4055
   macro avg       1.00      1.00      1.00      4055
weighted avg       1.00      1.00      1.00      4055
```

In [79]:
```
confusion_matrix(y_train_val, y_train_val_pred)
```

Out[79]:
```
array([[1748,    0],
       [   0, 2307]])
```

What would your next course of action be? In particular, share your thoughts on the following:

- Further optimization of this model
- Pursuing a different trading strategy or market (instruments) altogether
- Anything else?

In [80]:
```
answer = "I would take our model and see its performance in forward testing/
```

What do you think of the fact that we used interpolated **monthly** Google Trends data to try and predict short-term (5-day) price movements?

In [81]:
```
answer = "yeah going monthly to daily is upsampling the data via linear inte
```

Conclusion

These results highlight the challenges in consistently training AI/ML models that outperform naive baseline scores in financial markets due to factors such as non-stationary data, low signal-to-noise ratio, high market efficiency, and a competitive and adversarial trading environment. It would be necessary to gather much more data (and higher quality data) than we have in this project, and to engineer much more complex features and models to eke out even a slight gain in performance. It is therefore essential to use your domain knowledge, have realistic expectations, and constantly monitor your modeling assumptions and metrics. We hope that this project enables you to do so by giving you the tools, techniques and ideas to keep in mind.