template

March 18, 2025

1 Supply and Demand Principles Outside of an Introductory Economics Course: An Investigation of the Energy Information Administration's(EIA) Weekly Petroleum Status Report(WPSR) and Crude Oil Futures

This project aims to analyze the impact of unexpected changes in crude oil supply, as reported by the EIA WPSR, on the price of crude oil. By comparing historical EIA WPSR data and 3rd-party supply estimates alongside minute-resolution crude oil price data, the analysis investigates the market's reaction to these supply "surprises."

1.1 Research Question 1. How might one characterize the effect that the EIA WPSR release have on the Crude Oil Futures Market? Is it statistically significant and over what time interval is this effect the strongest?

Yes, we conclude that the release of the WPSR definitely affects the Crude Oil Futures market based on 2 main findings:

Finding #1: In the minutes before(~30 min) the release of the EIA WPSR, trading volume steadily drops before sharply jumping exactly when the report is released. This suggests that before the release, traders are wary of the price change that will result from the report and thus resulting in less trading activity. Then at the moment of release, the information is absorbed by the market leading to a significant jump in activity as traders seek to capitalize on the subsequent price change. ##### Finding #2: Once the report is released, minute-to-minute price behavior is different in a statistically different way for around 3 minutes. In comparing post-release price activity to relevant "normal" price behavior, it's especially apparent (even visually) that price seems to be much more likely to move when a report is released compared to days when there is no report. Additionally, when conducting a two-sample Kolmogorov-Smirnov test on the minute-to-minute price change distributions after WPSR releases, the p-value stays below the chosen threshould (<0.1) until the end of the 3rd minute after the report is released. Thus it is likely that it takes around 2 minutes before the market fully absorbs the new information of the WPSR release and resumes "regular" trading behavior.

1.2 Research Question 2. Among the plethora of new market-relevant information found in the the EIA WPSR, what portion/s of the data is most relevant to the traders, and thus most relevant to the “irregular” price behavior?

Based on the findings of Research Question #1 we now looked to figure out *how* the WPSR was causing a change in market behavior. In our research we initially hypothesized traders would fall into two main interpretations of the EIA WPSR information, all fundamentally based on simple microeconomic supply and demand principles; an excess supply imbalance would result in price decreases and an excess demand imbalance would result in price increases.

Interpretation #1: Traders are taking the raw week-to-week change in Crude Oil supply as the best reflection of market supply and demand imbalance. This would mean that the price behavior should be somewhat linearly correlated with domestic crude oil supply changes, i.e increasingly positive weekly supply changes lead to similarly significant price decreases and vice versa. ##### Interpretation #2: Traders are taking the difference between EIA’s supply change estimate and 3rd party supply change predictions(Investing.com or American Petroleum Institute) as the best reflection of market supply and demand imbalance. This interpretation builds on the first one, in that it essentially assumes the market price always reflects some consensus on the running supply change and that each release of the WPSR acts as a sort of correction. ##### After looking deeper into the data with visualizations and regression tests we found the relationship between supply-demand imbalances and price movement to be heavily obscured by noise and that the relationship. At this point we also realized the relationship didn’t seem very linearly correlated as we had hoped. Ultimately, we decided to move forward with Interpretation #1 as it seemed to result in the least noise upon visual inspection.

1.3 Research Question 3. Given the noise and possible non-linearity found in the relationship between the WPSR supply change information and the crude oil futures price movement, how can we use machine learning techniques to model this relationship and predict the price behavior based on the information found in the WPSR?

All in all, we compared a total of 6 different types of machine learning models to try and capture possible non-linear patterns in the data. Unfortunately, out of OLS Linear Regression, Kernel Regression(Gaussian Kernel), LOESS Regression, Random Forest Regression, Random Forest Regression + Gradient Boosting(XGBoost), and Ridge Regression we weren’t able to produce any significantly accurate models, with our best model being a Random Forest Regressor which achieved an R-squared value of 0.0171 and a Mean Squared Error of 0.0290.

After careful consideration of the impending deadline and the class this project was meant for, we concluded that the exact mechanisms behind the erratic price behavior upon the EIA’s WPSR release likely lay beyond the scope of our knowledge and expertise. Thus, while it is definitely possible and even likely that the WPSR affects crude oil futures prices, the relationship between the two is likely much more complex than the supply and demand principles that can be found in an introductory Microeconomics textbook.

1.4 Challenge Goals

Multiple Datasets: This project meets the Multiple Datasets challenge goal since it utilizes 3 distinct datasets: - Energy Information Administration's Weekly Petroleum Status Report - American Petroleum Institute's Weekly Statistical Bulletin - West Texas Intermediate Crude Oil Futures Contracts

Additionally, all of our research questions rely on using at least 2 and sometimes multiple datasets for their answer. Lastly, we fulfill the last requirement of this challenge goal since we merge multiple datasets throughout the project such as when we are creating the price windows for EIA WPSR release days. Additionally #### New Library: This project meets the New Library challenge goal since it utilizes a multitude of new libraries not covered in the class such as plotly, scipy, statsmodels, and xgboost. #### Advanced Machine Learning: This project meets the Advanced Machine Learning challenge goal in two ways: - We applied a gradient boosting technique to a Random Forest Regressor model called XGBoost which isn't included in the scikit-learn library. - We also compare 6 different machine learning algorithms on various statistics such as MSE and R^2 with hyperparameter optimization done using GridSearchCV.

1.5 Collaboration and Conduct

Students are expected to follow Washington state law on the Student Conduct Code for the University of Washington. In this course, students must:

- Indicate on your submission any assistance received, including materials distributed in this course.
- Not receive, generate, or otherwise acquire any substantial portion or walkthrough to an assessment.
- Not aid, assist, attempt, or tolerate prohibited academic conduct in others.

Update the following code cell to include your name and list your sources. If you used any kind of computer technology to help prepare your assessment submission, include the queries and/or prompts. Submitted work that is not consistent with sources may be subject to the student conduct process.

```
[116]: your_name = "Evan Sun, Ruifeng Tian, and Aaron Shayne Jacowitz"
sources = [
    "EIA weekly crude oil supply (Google Search) -> https://www.eia.gov/
 ↪petroleum/supply/weekly/pdf/wpsrall.pdf",
    "crude oil futures contract intraday historical data (Google Search) ->␣
 ↪https://www.backtestmarket.com/en/historical-data/commodities/crude-oil",
    "est.localize (Google Search) -> https://stackoverflow.com/questions/
 ↪15641898/python-timezone-localize-not-working",
    "b in datetime python (Google Search) -> https://stackoverflow.com/
 ↪questions/61699115/b-vs-b-in-datetime-module-python-3",
    ".to_datetime errors coerce (Google Search) -> https://pandas.pydata.org/
 ↪docs/reference/api/pandas.to_datetime.html",
    ".apply pandas (Google Search) -> https://pandas.pydata.org/docs/reference/
 ↪api/pandas.DataFrame.apply.html",
    "check for nan values in pandas (LLM Prompt)",
    "install plotly (Google Search) -> https://pypi.org/project/plotly/",
    "what is -m option in python pip (LLM Prompt)",
    "Crude oil inventories EIA (Google Search) -> https://www.investing.com/
 ↪economic-calendar/eia-crude-oil-inventories-75",
    "how to hide a python cell but keep output in jupyter notebook (Google␣
 ↪Search) -> https://jupyterbook.org/en/stable/interactive/hiding.html"
]

assert your_name != "", "your_name cannot be empty"
assert ... not in sources, "sources should not include the placeholder ellipsis"
assert len(sources) >= 6, "must include at least 6 sources, inclusive of␣
 ↪lectures and sections"
```

1.6 Data Setting and Methods

1. Data Setting

The analysis utilizes three primary datasets:

- **EIA WPSR Forecast Data (icom_eia_forecasts):** This dataset comprises historical records of EIA WPSR releases, including:
  - **Release Date and Time:** The date and time of the EIA report release. The time zone of this data is EST
  - **Actual Change in Crude Oil Inventories:** The officially reported change in U.S. commercial crude oil inventories (in millions of barrels).
  - **Market Forecasts:** Pre-release consensus forecasts for the change in crude oil inventories (in millions of barrels), sourced from a third-party provider.
  - **Previous Period's Actual:** The actual inventory change from the preceding reporting period.
  - The dataset is provided in a tabular format with date and time components spread across 'Release Date' and 'Time' columns, with inventory figures often suffixed with 'M' to denote millions.
- **Minute-Resolution WTI Crude Oil Price Data (min_WTI):** This dataset provides high-frequency, minute-by-minute price and volume data for WTI crude oil futures. Key variables include:
  - **Date and Time:** The date and time of each minute bar.
  - **Open, High, Low, Close (OHLC) Prices:** The open, high, low, and closing prices for each minute interval.
  - **Volume:** The trading volume within each minute interval.
  - The dataset is structured with 'Date' and 'Time' columns, requiring combination to create a unified datetime index. The time zone of this data is specified as GMT-6 or CST.
- **API WSB Data (fxstreet_api_forecasts):** This dataset provides historical records of API WSB releases, including:
  - **Date (Reference):** The date on which the entry was released with the week of reference in parentheses.
  - **Actual:** The estimated change in U.S. commerical crude oil inventories as estimated by the American Petroleum Institute.

  - **Deviation:** A supply surprise statistic calculated by a third-party (FXStreet).
  - **Consensus:** An "agreed" upon prediction by Wall Street on the week's supply change.

2. Data Transformations

- **Datetime Standardization:** Both datasets required standardization of their date and time formats to enable accurate time-based merging and analysis. After standardization, the 'Release_Datetime' column in `icom_eia_forecasts` and a new 'Datetime' column in `min_WTI` were set as the index for their respective dataframes to facilitate time-based data retrieval and merging. Timezone considerations were addressed by converting all datetimes to Eastern Standard Time (EST) to ensure consistency.

- **Supply Surprise Calculation:** To quantify the unexpected component of EIA releases, a 'supply_surprise' column was calculated in the `icom_eia_forecasts` dataframe using the `calculate_supply_surprise(df)` function. This function computes the difference between the 'Actual' and 'Forecast' inventory change values. It also handles the 'M' suffix in the inventory figures, converting them to numeric values (in millions of barrels) before calculating the difference.

- **Price Window Extraction:** To analyze the intraday price reaction around EIA releases, the `get_price_windows(eia_release_times, price_data, window_minutes_before=60, window_minutes_after=60)` function was utilized. This function extracts minute-resolution price data from `min_WTI` for a specified window period (e.g., 60 minutes before and 60 minutes after) each EIA report release time. The function iterates through each release time in `icom_eia_forecasts`, retrieves the corresponding price window from `min_WTI` based on the datetime index, and concatenates these windows into a new dataframe (`price_window_60min`) for event study analysis.

3. Methods

To address the research questions regarding the market impact of EIA WPSR supply surprises, the following analytical methods are employed:

- **Event Study Methodology:** An event study approach is used to examine the short-term impact of EIA WPSR releases on WTI crude oil prices. The 'event' is defined as the EIA WPSR report release. Price changes are measured over various time intervals *relative to* the release time (e.g., 1 minute after, 2 minutes after, etc)

- **Percentage Price Change Calculation:** To standardize price reaction comparisons across different release events and price levels, percentage price changes are calculated. For each EIA release, the percentage price change is computed as: ((Price at Time t+ Δt - Price at Time t) / Price at Time t) * 100%, where Time t is the release time and Δt represents the time interval (e.g., 1 minute, 5 minutes).

- **Visualizations:** Interactive visualizations are generated to explore the data and results:
  - **Histograms:** Histograms are created to visualize the distribution of supply surprises and the distributions of percentage price changes for each time interval. This helps assess the shape and characteristics of these distributions, including normality.
  - **Scatter Plots:** Scatter plots are used to examine the average trading volume around EIA release times, illustrating intraday volume patterns and potential volume spikes associated with releases.

- **Kolmogorov-Smirnov Test:** To formally test for the EIA WPSR associated price change distributions, a two-sample Kolmogorov-Smirnov (KS) test is applied. This non-parametric test compares the empirical cumulative distribution function of the post-release price changes on days of release to post-release price changes on days without a report, providing a statistical measure of the difference in market behavior and a p-value to assess the significance of any deviations from normality.

```
[98]: def standardize_eia_datetime(row):
          """Standardizes datetime for EIA forecast"""
          release_date = pd.to_datetime(row['Release Date'], format='%d-%b-%y',␣
       ↪errors='coerce').date()
          release_time = pd.to_datetime(row['Time'], format='%H:%M', errors='coerce').
       ↪time()

          if pd.isna(release_date) or pd.isna(release_time):
              print('nan')
              return pd.NaT

          combined_datetime = pd.Timestamp.combine(release_date, release_time)
          return combined_datetime

      test_cases = [
          {
              'input': {'Release Date': '01-Jan-23', 'Time': '09:00'},
              'expected': pd.Timestamp('2023-01-01 09:00:00')
          },
          {
              'input': {'Release Date': '15-Dec-22', 'Time': '14:30'},
              'expected': pd.Timestamp('2022-12-15 14:30:00')
          },
          {
              'input': {'Release Date': '31-Dec-22', 'Time': '23:59'},
              'expected': pd.Timestamp('2022-12-31 23:59:00')
          },
          {
              'input': {'Release Date': '01-Apr-23', 'Time': '00:00'},
              'expected': pd.Timestamp('2023-04-01 00:00:00')
          }
      ]

      def test_standardize_eia_datetime():
          for i, test_case in enumerate(test_cases):
              input_row = test_case['input']
              expected_output = test_case['expected']

              try:
                  actual_output = standardize_eia_datetime(input_row)
```

```
            assert actual_output == expected_output, f"Test case {i+1} failed:␣
↪expected {expected_output}, got {actual_output}"
        except AssertionError as e:
            print(e)
        except Exception as e:
            print(f"Test case {i+1} failed with an unexpected error: {e}")

test_standardize_eia_datetime()


def standardize_wti_datetime(row):
    """Standardizes datetime for min_WTI dataframe."""
    date_str = row['Date']
    time_str = row['Time']

    combined_datetime = pd.to_datetime(date_str + ' ' + time_str, format='%d/%m/
↪%Y %H:%M:%S', errors='coerce')

    if pd.isna(combined_datetime):
        print('nan')
        return pd.NaT
    return combined_datetime

test_cases = [
    {
        'input': {'Date': '01/01/2023', 'Time': '09:00:00'},
        'expected': pd.Timestamp('2023-01-01 09:00:00')
    },
    {
        'input': {'Date': '15/12/2022', 'Time': '14:30:00'},
        'expected': pd.Timestamp('2022-12-15 14:30:00')
    },
    {
        'input': {'Date': '31/12/2022', 'Time': '23:59:59'},
        'expected': pd.Timestamp('2022-12-31 23:59:59')
    },
    {
        'input': {'Date': '01/04/2023', 'Time': '00:00:00'},
        'expected': pd.Timestamp('2023-04-01 00:00:00')
    }
]


def test_standardize_wti_datetime():
    for i, test_case in enumerate(test_cases):
        input_row = test_case['input']
        expected_output = test_case['expected']
        actual_output = standardize_wti_datetime(input_row)
        try:
            assert actual_output == expected_output, f"Test case {i+1} failed:␣
↪expected {expected_output}, got {actual_output}"
        except AssertionError as e:
            print(e)

test_standardize_wti_datetime()

def calculate_supply_surprise(df):
    """
    Calculates the 'supply_surprise' column as the difference between 'Actual'␣
↪and 'Forecast'.
    Handles 'M' suffix and converts to numeric.

    Args:
        df (pd.DataFrame): icom_eia_forecasts DataFrame.

    Returns:
        pd.Series: Supply surprise values.
    """
    actual = df['Actual'].str.replace('M', '', regex=False).astype(float)
    forecast = df['Forecast'].str.replace('M', '', regex=False).astype(float)
    supply_surprise = actual - forecast
    return supply_surprise

test_data = {
    'Actual': ['10M', '20M', '15.5M', '10'],
    'Forecast': ['8M', '22M', '14M', '5']
}

df_test = pd.DataFrame(test_data)

expected_results = [2.0, -2.0, 1.5, 5.0, pd.NA, pd.NA]

def test_calculate_supply_surprise():
    result = calculate_supply_surprise(df_test)

    for i, (actual, expected) in enumerate(zip(result, expected_results)):
        if pd.isna(expected):
            assert pd.isna(actual), f"Test case {i+1} failed: expected NA, got␣
↪{actual}"
        else:
            try:
                assert actual == expected, f"Test case {i+1} failed: expected␣
↪{expected}, got {actual}"
```

```
            except AssertionError as e:
                print(e)

test_calculate_supply_surprise()


def get_price_windows(eia_release_times, price_data, window_minutes_before=60,␣
 ↪window_minutes_after=60):
    """
    Extracts price data windows around EIA report release times.

    Args:
        eia_release_times (pd.DatetimeIndex): Index of icom_eia_forecasts␣
 ↪(release datetimes).
        price_data (pd.DataFrame): min_res_OIH dataframe with Datetime index.
        window_minutes_before (int): Minutes to include before release time.
        window_minutes_after (int): Minutes to include after release time.

    Returns:
        pd.DataFrame: A DataFrame containing price data for all events, within␣
 ↪the specified windows.
                      Returns an empty DataFrame if no data is found within any␣
 ↪window.
    """
    price_windows_list = []

    for release_time in eia_release_times:
        start_time = release_time - pd.Timedelta(minutes=window_minutes_before)

        end_time = release_time + pd.Timedelta(minutes=window_minutes_after)

        window_data = price_data.loc[start_time:end_time].copy()


        if not window_data.empty:
            window_data['Release_Datetime'] = release_time
            price_windows_list.append(window_data)

    if price_windows_list:
        price_windows_df = pd.concat(price_windows_list)
        return price_windows_df
    else:
        return pd.DataFrame()


eia_release_times = pd.DatetimeIndex([
    '2023-01-01 09:00:00',
    '2023-01-02 09:00:00',
    '2023-01-03 09:00:00'
])

price_data = pd.DataFrame({
    'Price': [100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111],
}, index=pd.DatetimeIndex([
    '2023-01-01 08:30:00',
    '2023-01-01 09:00:00',
    '2023-01-01 09:30:00',
    '2023-01-01 10:00:00',
    '2023-01-02 08:30:00',
    '2023-01-02 09:00:00',
    '2023-01-02 09:30:00',
    '2023-01-02 10:00:00',
    '2023-01-03 08:30:00',
    '2023-01-03 09:00:00',
    '2023-01-03 09:30:00',
    '2023-01-03 10:00:00'
]))

expected_results = pd.DataFrame({
    'Price': [100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111],
    'Release_Datetime': [
        '2023-01-01 09:00:00', '2023-01-01 09:00:00', '2023-01-01 09:00:00',␣
 ↪'2023-01-01 09:00:00',
        '2023-01-02 09:00:00', '2023-01-02 09:00:00', '2023-01-02 09:00:00',␣
 ↪'2023-01-02 09:00:00',
        '2023-01-03 09:00:00', '2023-01-03 09:00:00', '2023-01-03 09:00:00',␣
 ↪'2023-01-03 09:00:00'
    ]
}, index=pd.DatetimeIndex([
    '2023-01-01 08:30:00', '2023-01-01 09:00:00', '2023-01-01 09:30:00',␣
 ↪'2023-01-01 10:00:00',
    '2023-01-02 08:30:00', '2023-01-02 09:00:00', '2023-01-02 09:30:00',␣
 ↪'2023-01-02 10:00:00',
    '2023-01-03 08:30:00', '2023-01-03 09:00:00', '2023-01-03 09:30:00',␣
 ↪'2023-01-03 10:00:00'
]))

expected_results['Release_Datetime'] = pd.
 ↪to_datetime(expected_results['Release_Datetime'])

def test_get_price_windows():
    result = get_price_windows(eia_release_times, price_data,␣
 ↪window_minutes_before=60, window_minutes_after=60)
```

```
    try:
        pd.testing.assert_frame_equal(result, expected_results, check_like=True)
    except AssertionError as e:
        print("Test failed:")
        print(e)

test_get_price_windows()


def time_difference(time1, time2):
    """
    Calculates the time difference in seconds between two datetime.time objects.

    Args:
        time1: The first datetime.time object.
        time2: The second datetime.time object.

    Returns:
        The time difference in minutes as a float.
    """
    dummy_date = datetime.date(1, 1, 1)
    datetime1 = datetime.datetime.combine(dummy_date, time1)
    datetime2 = datetime.datetime.combine(dummy_date, time2)

    time_delta = datetime2 - datetime1

    return time_delta.total_seconds() / 60

def test_time_difference():
    time1 = datetime.time(12, 0, 0)
    time2 = datetime.time(12, 0, 0)
    assert abs(time_difference(time1, time2) - 0.0) < 1e-6, "Test case 1 failed"

    time1 = datetime.time(12, 0, 0)
    time2 = datetime.time(12, 1, 0)
    assert abs(time_difference(time1, time2) - 1.0) < 1e-6, "Test case 2 failed"

    time1 = datetime.time(12, 0, 0)
    time2 = datetime.time(13, 0, 0)
    assert abs(time_difference(time1, time2) - 60.0) < 1e-6, "Test case 3␣
 ↪failed"

    time1 = datetime.time(13, 0, 0)
    time2 = datetime.time(12, 0, 0)
    assert abs(time_difference(time1, time2) + 60.0) < 1e-6, "Test case 6␣
 ↪failed"
```

```
test_time_difference()




def format_fxstreet_date(date_str):
    """
    Extracts the date part from the fxstreet date string and formats it as day/
    ↪month/year.

    Args:
        date_str (str): The original date string from the 'Date' column.

    Returns:
        str: Formatted date string in day/month/year format (e.g., "04/03/
    ↪2025").
             Returns original string if formatting fails.
    """
    try:
        date_part = date_str.split('(')[0].strip()
        datetime_obj = pd.to_datetime(date_part, format='%m/%d/%Y')
        formatted_date_str = datetime_obj.strftime('%d/%m/%Y')
        return formatted_date_str
    except Exception as e:
        return date_str

test_cases = [
    ("04/03/2025 (Wednesday)", "03/04/2025"),
    ("12/31/2023 (Sunday)", "31/12/2023"),
    ("01/01/2024 (Tuesday)", "01/01/2024"),
    ("04/03/2025", "03/04/2025"),
    ("13/13/2023 (Invalid)", "13/13/2023 (Invalid)"),
]

def test_format_fxstreet_date():
    for input_date, expected_output in test_cases:
        actual_output = format_fxstreet_date(input_date)
        assert actual_output == expected_output, f"Failed on input:␣
    ↪{input_date}"


test_format_fxstreet_date()


def find_nearest_date_api(row, fxstreet_api_forecasts_dates):
    """
    Finds the Actual_API value from fxstreet_api_forecasts corresponding to the␣
 ↪nearest date
    to the DateString in the given row.

    Args:
        row (pd.Series): A row from merged_price_window_0to2min DataFrame.
        fxstreet_api_forecasts_dates (pd.Series): The Date_Formatted column␣
 ↪from fxstreet_api_forecasts.

    Returns:
        float: The Actual_API value corresponding to the nearest date, or NaN␣
 ↪if no match.
    """
    date_to_match = row['DateString']

    time_diffs = abs(fxstreet_api_forecasts_dates - date_to_match)

    nearest_date_index = time_diffs.idxmin()

    return fxstreet_api_forecasts['Actual_API'].iloc[nearest_date_index]

mock_row = pd.Series({'DateString': '03/04/2025'})

fxstreet_api_forecasts = pd.DataFrame({
    'Date_Formatted': [
        '03/04/2025',
        '03/05/2025',
        '03/03/2025',
        '03/10/2025'
    ],
    'Actual_API': [65.0, 66.0, 64.0, 67.0]
})

fxstreet_api_forecasts['Date_Formatted'] = pd.
 ↪to_datetime(fxstreet_api_forecasts['Date_Formatted'], format='%d/%m/%Y')

fxstreet_api_forecasts_dates = fxstreet_api_forecasts['Date_Formatted']

test_cases = [
    (pd.Series({'DateString': '03/04/2025'}), 65.0),
    (pd.Series({'DateString': '03/04/2025 12:00:00'}), 65.0),
    (pd.Series({'DateString': '03/06/2025'}), 66.0),
    (pd.Series({'DateString': '03/02/2025'}), 64.0),
    (pd.Series({'DateString': '03/07/2025'}), 66.0),
]
```

```
def test_find_nearest_date_api():
    for i, (mock_row, expected_output) in enumerate(test_cases):
        if 'DateString' in mock_row:
            try:
                mock_row['DateString'] = pd.to_datetime(mock_row['DateString'],␣
 ↪format='%d/%m/%Y %H:%M:%S')
            except ValueError:
                mock_row['DateString'] = pd.to_datetime(mock_row['DateString'],␣
 ↪format='%d/%m/%Y')

        actual_output = find_nearest_date_api(mock_row,␣
 ↪fxstreet_api_forecasts_dates)

        if expected_output is None:
            assert pd.isna(actual_output), f"Failed on input:␣
 ↪{mock_row['DateString']}"
        else:
            assert actual_output == expected_output, f"Failed on input:␣
 ↪{mock_row['DateString']}"

test_find_nearest_date_api()
```

1.7 Results

1.8 Research Question 2. Among the plethora of new market-relevant information found in the the EIA WPSR, what portion/s of the data is most relevant to the traders, and thus most relevant to the "irregular" price behavior?

Based on the findings of Research Question #1 we now looked to figure out *how* the WPSR was causing a change in market behavior. In our research we initially hypothesized traders would fall into two main interpretations of the EIA WPSR information, all fundamentally based on simple microeconomic supply and demand principles; an excess supply imbalance would result in price decreases and an excess demand imbalance would result in price increases.

Interpretation #1: Traders are taking the raw week-to-week change in Crude Oil supply as the best reflection of market supply and demand imbalance. This would mean that the price behavior should be somewhat linearly correlated with domestic crude oil supply changes, i.e increasingly positive weekly supply changes lead to similarly significant price decreases and vice versa. ##### Interpretation #2: Traders are taking the difference between EIA's supply change estimate and 3rd party supply change predictions(Investing.com or American Petroleum Institute) as the best reflection of market supply and demand imbalance. This interpretation builds on the first one, in that it essentially assumes the market price always reflects some consensus on the running supply change and that each release of the WPSR acts as a sort of correction. ##### After looking deeper into the data with visualizations and regression tests we found the relationship between supply-demand imbalances and price movement to be heavily obscured by noise and that the relationship. At this point we also realized the relationship didn't seem very linearly correlated as we had hoped. Ultimately, we decided to move forward with Interpretation #1 as it seemed to result in the least noise upon visual inspection.

1.9 Research Question 3. Given the noise and possible non-linearity found in the relationship between the WPSR supply change information and the crude oil futures price movement, how can we use machine learning techniques to model this relationship and predict the price behavior based on the information found in the WPSR?

All in all, we compared a total of 6 different types of machine learning models to try and capture possible non-linear patterns in the data. Unfortunately, out of OLS Linear Regression, Kernel Regression(Gaussian Kernel), LOESS Regression, Random Forest Regression, Random Forest Regression + Gradient Boosting(XGBoost), and Ridge Regression we weren't able to produce any significantly accurate models, with our best model being a Random Forest Regressor which achieved an R-squared value of 0.0171 and a Mean Squared Error of 0.0290.

After careful consideration of the impending deadline and the class this project was meant for, we concluded that the exact mechanisms behind the erratic price behavior upon the EIA's WPSR release likely lay beyond the scope of our knowledge and expertise. Thus, while it is definitely possible and even likely that the WPSR affects crude oil futures prices, the relationship between the two is likely much more complex than the supply and demand principles that can be found in an introductory Microeconomics textbook.

1.9.1 Research Question 1. How might one characterize the effect that the EIA WPSR release have on the Crude Oil Futures Market? Is it statistically significant and over what time interval is this effect the strongest?

Our first research question sought to understand how the EIA WPSR release affects the Crude Oil Futures Market, specifically addressing its statistical significance and the duration of its strongest impact. We definitively conclude that the WPSR release has a significant impact on the market, supported by two key findings:

Finding #1: Trading Volume Dynamics Around WPSR Release

We observed a distinct pattern in trading volume in the minutes surrounding the WPSR release. As illustrated in the figure below, trading volume steadily decreases in the approximately 30 minutes leading up to the release. However, precisely at the moment of the WPSR release, there is a sharp and substantial jump in trading volume.

```
[118]: fig_volume_scatter_release
```

This behavior suggests that traders become hesitant to engage in trading activity as the report release approaches, anticipating potential price volatility. Upon the report's release, the market rapidly absorbs the information, leading to a surge in trading activity as participants react to the news and attempt to capitalize on the anticipated price adjustments. This volume spike is a clear indicator of the WPSR's immediate market impact.

Finding #2: Statistically Significant Change in Minute-to-Minute Price Behavior Post-Release

Beyond volume, we examined the minute-to-minute price behavior before and after WPSR releases to look for any statistically significant changes. [**Embed Visualization: Two histograms indicating how prices move around the time of EIA WPSR release on days with a release and days without a release (fig_heatmap_release and fig_heatmap_non_release)**] visually highlights the difference in price change distributions between release days and normal days. It becomes apparent that price fluctuations are noticeably more frequent and potentially larger in magnitude immediately following a WPSR release compared to days without a release.

To quantify this observation, we conducted a two-sample Kolmogorov-Smirnov (KS) test on the distributions of minute-to-minute price changes after WPSR releases, comparing them to "normal" price change distributions (days without releases). [**Embed Visualization: Line plot showing how the p-value of a two-sample Kolmogorov-Smirnov test comparing post-release price distributions on days with and without release changes with each minute (line_plot_p_value_KS_test)**] demonstrates the p-values obtained from these tests over the minutes following the release. The p-value remains below our chosen significance threshold of 0.1 for the first three minutes after the report release. This indicates that the price change distributions in these minutes are statistically significantly different from normal price behavior.

The p-value rising above 0.1 after the third minute suggests that the market, on average, takes approximately 2 minutes to fully incorporate the information from the WPSR and revert to its typical trading patterns. This statistically significant deviation in price behavior for around 3 minutes post-release further reinforces the EIA WPSR's impactful, albeit short-lived, influence on the crude oil futures market.

1.9.2 1.9 Research Question 2: Identifying Relevant WPSR Data Portions for "Irregular" Price Behavior

Building upon the evidence of the WPSR's market impact from Research Question 1, our second research question aimed to pinpoint which specific data components within the WPSR are most pertinent to traders and contribute most significantly to the observed "irregular" price behavior. We initially hypothesized that traders primarily interpret WPSR information through the lens of basic supply and demand principles. Our core assumption was that market reactions would stem from perceived supply-demand imbalances signaled by the WPSR, with excess supply leading to price decreases and excess demand driving price increases.

We formulated two primary interpretations of how traders might utilize WPSR data to assess these imbalances:

Interpretation #1: Reaction to Raw Week-to-Week Crude Oil Supply Changes

This interpretation posits that traders focus on the direct week-over-week change in domestic crude oil supply as reported by the EIA. Under this view, market price reactions should exhibit a linear correlation with these supply changes. Specifically, increasingly positive weekly supply changes (indicating increased supply) should correspond to proportionally significant price decreases, and conversely, negative supply changes (indicating decreased supply) should lead to price increases.

Interpretation #2: Reaction to Supply Surprise - Deviation from Expectations

The second interpretation builds upon the first, suggesting that traders consider the "supply surprise" as the key driver of market reactions. This "surprise" is defined as the difference between the EIA's reported supply change and pre-release supply change predictions from third-party sources (such as Investing.com or the American Petroleum Institute - API). This perspective assumes that the market already incorporates some consensus expectation of the supply change prior to the WPSR release. The WPSR, then, acts as a correction mechanism, with the market reacting primarily to the deviation from these pre-existing expectations.

To investigate these interpretations, we explored the relationship between supply-demand imbalances (as defined by both interpretations) and subsequent price movements through visualizations and regression analysis. [**Embed Visualization: Scatter Plot of Supply Surprise (Investing.com) vs. Percentage Price Change (0-2 min) (fig_scatter_surprise_vs_pricechange_Investingcom)**] and [**Embed Visualization: Scatter Plot of Supply Change (EIA) vs. Percentage Price Change (0-2 min) (fig_scatter_surprise_vs_pricechange_EIA)**] visually represent these relationships.

Our analysis revealed that the relationships between these hypothesized supply-demand imbalance metrics and price movements are considerably obscured by noise. The expected linear correlation was not strongly evident in the data. While visually inspecting the scatter plots, we observed slightly less noise associated with Interpretation #1 (raw supply change) compared to Interpretation #2 (supply surprise). Consequently, we proceeded with Interpretation #1 for further analysis, acknowledging the inherent noise and potential non-linearity in the relationship.

1.9.3 1.10 Research Question 3: Machine Learning for Modeling the WPSR-Price Relationship

Given the observed noise and the possibility of non-linear relationships between WPSR supply change information and crude oil futures price movements, our third research question explored the application of machine learning techniques to model and potentially predict price behavior based on WPSR data. We aimed to determine if machine learning could capture any underlying patterns that were not apparent through linear models and basic statistical analysis.

We evaluated a total of six different machine learning models, selected for their ability to capture potentially non-linear patterns in the data. These models included:

- **OLS Linear Regression** (as a baseline linear model)
- **Kernel Regression (Gaussian Kernel)**
- **LOESS Regression** (Locally Estimated Scatterplot Smoothing)
- **Random Forest Regression**
- **Random Forest Regression with Gradient Boosting (XGBoost)**
- **Ridge Regression** (Regularized Linear Regression)

[**Embed Visualization: Kernel Regression: Actual vs. Predicted Values (kernel_reg_plot)**], [**Embed Visualization: LOESS Regression: Percentage Change vs Actual (loess_plot)**], [**Embed Visualization: Random Forest Regression: Actual vs. Predicted Values (rf_actual_pred_fig)**], [**Embed Visualization: XGBoost Regression: Actual vs. Predicted Values (xgb_actual_pred_fig)**], [**Embed Visualization: Ridge Regression: Actual vs. Predicted Values (prediction_plot_ridge_reg)**] visually compare the performance of some of these models. Additionally, [**Embed Visualization: Random Forest Feature Importance (rf_feature_fig)**] and [**Embed Visualization: XGBoost Feature**

Importance (xgb_feature_fig)] illustrate the feature importance as determined by the Random Forest and XGBoost models.

Unfortunately, despite testing these diverse models and tuning hyperparameters using GridSearchCV, we were unable to develop any models with significant predictive accuracy. Our best performing model was a Random Forest Regressor, which achieved an R-squared value of only 0.0171 and a Mean Squared Error of 0.0290 on the test set. The low R-squared indicates that the models explain a very small proportion of the variance in the price changes, and the MSE, while providing a measure of error, is still relatively high considering the scale of price changes.

[**Report Model Performance Metrics (MSE and R-squared) for all 6 models in a table here**]

After careful consideration of the project's scope, the course context, and the limited success of our machine learning efforts, we conclude that the precise mechanisms driving the erratic price behavior following EIA WPSR releases likely involve complexities beyond our current understanding and analytical toolkit. While our analysis confirms that the WPSR release significantly affects crude oil futures prices, the relationship between specific WPSR supply information and price movement appears to be far more intricate and less directly driven by simple supply and demand principles as presented in introductory microeconomics. Further research employing more advanced econometric techniques, incorporating broader market factors, and potentially exploring higher-frequency data and more nuanced interpretations of market expectations could be necessary to develop more robust predictive models for WPSR-driven price behavior.

[99]:
```
import pandas as pd
import datetime
import pytz
import plotly.graph_objects as go
from plotly.offline import iplot
from plotly.subplots import make_subplots
from scipy import stats
import numpy as np
import matplotlib.pyplot as plt
import plotly.express as px
import statsmodels.api as sm
from sklearn.linear_model import Ridge
from sklearn.kernel_ridge import KernelRidge
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.ensemble import RandomForestRegressor
from xgboost import XGBRegressor
```

[ ]:
```
##########################################################################################
# FOR THE GRADER: You will notice that the outputs of project.ipynb and project.
 ↪html look wildly #
# different. We've talked to the prof about this already, the ipynb notebook is␣
 ↪just there to    #
# confirm that the code runs without error on a smaller dataset, while the .
 ↪html file is output  #
```

```
# produced by running the notebook locally on the full dataset.                 ␣
↪                   #
##########################################################################################

'''
Also if you are reading the .html project you may be wondering why none of the␣
↪cells are
producing output. I have added a %%capture to the top of cells that produce a␣
↪plot, preventing
the cells from outputting. I did this so I could embed the plots in markdown␣
↪cells which I think
is much easier than having to scroll through code while trying to read the␣
↪research results. If
you would rather have the plots show up next to the code, just comment out the␣
↪%%capture line.
'''

pd.set_option('display.max_rows', 400)
pd.set_option('display.max_columns', 20)

icom_eia_forecasts = pd.read_csv('data/InvestingcomEIA.csv')
fxstreet_api_forecasts = pd.read_csv('data/FXStreetAPI.csv')
min_WTI = pd.read_csv(
    'data/cl-1m.csv',
    sep=';',
    header=None,
    names=['Date', 'Time', 'Open', 'High', 'Low', 'Close', 'Volume']
)
min_WTI = min_WTI[1225975:]
```

[101]:
```
#4:50
icom_eia_forecasts['Release_Datetime'] = icom_eia_forecasts.
↪apply(standardize_eia_datetime, axis=1)

eastern_tz = pytz.timezone('US/Eastern')
icom_eia_forecasts['Release_Datetime_EST'] =␣
↪icom_eia_forecasts['Release_Datetime'].dt.tz_localize(eastern_tz,␣
↪ambiguous='infer', nonexistent='shift_forward')

chicago_tz = pytz.timezone('America/Chicago')
icom_eia_forecasts['Release_Datetime_CST'] =␣
↪icom_eia_forecasts['Release_Datetime_EST'].dt.tz_convert(chicago_tz)

min_WTI['Datetime'] = min_WTI.apply(standardize_wti_datetime, axis=1)
```

```
min_WTI['Datetime_CST'] = min_WTI['Datetime'].dt.tz_localize(chicago_tz,␣
↪ambiguous='infer', nonexistent='shift_forward')

icom_eia_forecasts = icom_eia_forecasts.set_index('Release_Datetime_CST').
↪sort_index()
min_WTI = min_WTI.set_index('Datetime_CST').sort_index()
```

[102]:
```
%%capture
# Histogram of EIA WPSR Crude Oil Supply Surprise
# fig_hist_EIA_supply_surprise

icom_eia_forecasts['supply_surprise'] =␣
↪calculate_supply_surprise(icom_eia_forecasts)

supply_surprise_data = icom_eia_forecasts['supply_surprise']

fig_hist_EIA_supply_surprise = go.Figure(data=[go.
↪Histogram(x=supply_surprise_data, nbinsx=100)])

fig_hist_EIA_supply_surprise.update_layout(
    title='Histogram of EIA WPSR Crude Oil Supply Surprise',
    xaxis_title='Supply Surprise (Actual - Forecast, Million Barrels)',
    yaxis_title='Frequency (Number of Releases)',
    bargap=0.1
)

fig_hist_EIA_supply_surprise.show()
```

[103]:
```
%%capture
# Histogram of One-Minute Percentage Price Changes in Crude Price
# fig_hist_1min_wti
# 1:41

percentage_price_changes_1min_wti = []
previous_close_price = None

for index, row in min_WTI.iterrows():
    current_close_price = row['Close']
    if previous_close_price is not None and previous_close_price != 0:
        percentage_change = ((current_close_price - previous_close_price) /␣
↪previous_close_price) * 100.0
        percentage_price_changes_1min_wti.append(percentage_change)
    else:
        percentage_price_changes_1min_wti.append(float('nan'))
    previous_close_price = current_close_price
```

```
percentage_price_changes_1min_wti_series = pd.
 ↪Series(percentage_price_changes_1min_wti, index=min_WTI.index)
min_WTI['Percent_Change'] = percentage_price_changes_1min_wti_series
percentage_price_changes_1min_wti_series =␣
 ↪percentage_price_changes_1min_wti_series.dropna()

fig_hist_1min_wti = go.Figure(data=[go.
 ↪Histogram(x=percentage_price_changes_1min_wti, nbinsx=1600)])

fig_hist_1min_wti.update_layout(
    title='Histogram of One-Minute Percentage Price Changes in Crude Price',
    xaxis_title='One-Minute Percentage Price Change (%)',
    yaxis_title='Frequency (Number of Minutes)',
    xaxis_range=[-3, 3]
)

fig_hist_1min_wti.show()
```

[104]:
```
%%capture
# Scatter plot of average trading volume one hour before and after the EIA WPSR␣
 ↪release time
# fig_volume_scatter_release

price_window_60min = get_price_windows(icom_eia_forecasts.index, min_WTI,␣
 ↪window_minutes_before=60, window_minutes_after=60)
price_window_60min.sort_index(inplace=True)

price_window_60min['Time_to_Release_Minutes'] = (price_window_60min.index -␣
 ↪price_window_60min['Release_Datetime']).dt.total_seconds() / 60
average_volume_by_time = price_window_60min.
 ↪groupby('Time_to_Release_Minutes')['Volume'].mean().reset_index()

fig_volume_scatter_release = go.Figure(data=[go.Scatter(
    x=average_volume_by_time['Time_to_Release_Minutes'],
    y=average_volume_by_time['Volume'],
    mode='markers',
    marker=dict(size=8),
    text=average_volume_by_time['Volume'],
    hovertemplate="Time to Release: %{x:.0f} minutes<br>Average Volume: %{y:.
 ↪0f}<extra></extra>"
)])

fig_volume_scatter_release.update_layout(
    title='Average Trading Volume Around EIA WPSR Release',
    xaxis_title='Minutes Relative to EIA Report Release',
    yaxis_title='Average Volume',
    xaxis=dict(
        tickvals=[-60, -45, -30, -15, 0, 15, 30, 45, 60],
        ticktext=['-60', '-45', '-30', '-15', 'Release', '+15', '+30', '+45',␣
↪'+60']
    ),
    hovermode="closest"
)

fig_volume_scatter_release.show()
```

[105]:
```
%%capture
# Two histograms indicating how prices move around the time of EIA WPSR release␣
↪on days with a
# release and days without a release
# fig_heatmap_release
# fig_heatmap_non_release


price_window_60min['Time_Delta_Minutes'] = (price_window_60min.index -␣
↪price_window_60min['Release_Datetime']).dt.total_seconds() / 60

time_intervals_release = sorted(price_window_60min['Time_Delta_Minutes'].
↪unique())
price_change_bins = np.linspace(-1, 1, num=41)
price_change_labels = [f'{bin_val:.2f}%' for bin_val in price_change_bins]

price_window_60min['Price_Change_Bin'] = pd.
↪cut(price_window_60min['Percent_Change'], bins=price_change_bins,␣
↪labels=price_change_labels[:-1], include_lowest=True)


heatmap_data_release = price_window_60min.groupby(['Price_Change_Bin',␣
↪'Time_Delta_Minutes'], observed=False).size().unstack(fill_value=0)

heatmap_data_release = heatmap_data_release.
↪reindex(columns=time_intervals_release, fill_value=0)
heatmap_data_release = heatmap_data_release.reindex(index=price_change_labels[:
↪-1], fill_value=0)

fig_heatmap_release = go.Figure(data=go.Heatmap(
    z=heatmap_data_release.values,
    x=heatmap_data_release.columns,
    y=heatmap_data_release.index,
    colorscale='Viridis',
    colorbar=dict(title='Frequency')
))
```

```
fig_heatmap_release.update_layout(
    title='Frequency of Percentage Price Changes Around EIA WPSR Release',
    xaxis_title='Time Relative to Release (Minutes)',
    yaxis_title='Percentage Price Change Bins',
    yaxis=dict(autorange="reversed"),
    xaxis=dict(tickvals=time_intervals_release[::5], ticktext=[int(x) for x in␣
  ↪time_intervals_release[::5]])
)


eia_release_dates = icom_eia_forecasts.reset_index()['Release_Datetime_CST'].dt.
  ↪date
min_wti_dates = min_WTI.reset_index()['Datetime_CST'].dt.date
mask_non_eia_days = ~min_wti_dates.isin(eia_release_dates)

min_WTI_no_eia_days = min_WTI.reset_index()[mask_non_eia_days].copy()

wti_with_eia_releases = pd.merge_asof(
    left=min_WTI_no_eia_days.reset_index(),
    right=icom_eia_forecasts.reset_index(),
    left_on='Datetime_CST',
    right_on='Release_Datetime_CST',
    direction='nearest',
    tolerance=pd.Timedelta('2D')
)

wti_with_eia_releases = wti_with_eia_releases.set_index('Datetime_CST')

wti_with_eia_releases.dropna(subset=['Release_Datetime_CST'], inplace=True)

time_diff = [time_difference(t1, t2) for t1, t2 in zip(wti_with_eia_releases.
  ↪reset_index()['Release_Datetime_CST'].dt.time, wti_with_eia_releases.
  ↪reset_index()['Datetime_CST'].dt.time)]

time_diff_series = pd.Series(time_diff)
wti_with_eia_releases.reset_index(inplace=True)
wti_with_eia_releases['Time_till_Release'] = (time_diff_series)
wti_with_eia_releases.set_index('Datetime_CST', inplace=True)

non_release_price_windows_60min = wti_with_eia_releases.copy().
  ↪loc[wti_with_eia_releases['Time_till_Release'].abs() <= 60.0]
non_release_price_windows_60min.reset_index(inplace=True)
non_release_price_windows_60min.drop(columns=['Datetime_CST', 'Open', 'High',␣
  ↪'Low', 'Close', 'Release_Datetime_CST', 'Release_Datetime_EST'],␣
  ↪inplace=True)
```

```
time_intervals_non_release =␣
 ↪sorted(non_release_price_windows_60min['Time_till_Release'].unique())

non_release_price_windows_60min['Price_Change_Bin'] = pd.
 ↪cut(non_release_price_windows_60min['Percent_Change'],␣
 ↪bins=price_change_bins, labels=price_change_labels[:-1], include_lowest=True)


heatmap_data_non_release = non_release_price_windows_60min.
 ↪groupby(['Price_Change_Bin', 'Time_till_Release'], observed=False).size().
 ↪unstack(fill_value=0)

heatmap_data_non_release = heatmap_data_non_release.
 ↪reindex(columns=time_intervals_non_release, fill_value=0)
heatmap_data_non_release = heatmap_data_non_release.
 ↪reindex(index=price_change_labels[:-1], fill_value=0)

heatmap_non_release_values = heatmap_data_non_release.values

min_val = np.min(heatmap_non_release_values)
max_val = np.max(heatmap_non_release_values)

if max_val > min_val:
    heatmap_non_release_values_scaled_linear = (heatmap_non_release_values -␣
 ↪min_val) / (max_val - min_val) * 100.0
else:
    heatmap_non_release_values_scaled_linear = np.
 ↪zeros_like(heatmap_non_release_values)

fig_heatmap_non_release = go.Figure(data=go.Heatmap(
    z=heatmap_non_release_values_scaled_linear,
    x=heatmap_data_non_release.columns,
    y=heatmap_data_non_release.index,
    colorscale='Viridis',
    colorbar=dict(title='Frequency')
))

fig_heatmap_non_release.update_layout(
    title='Frequency of Percentage Price Changes Around EIA WPSR Release On␣
 ↪Non-Release Days',
    xaxis_title='Time Relative to Release (Minutes)',
    yaxis_title='Percentage Price Change Bins',
    xaxis=dict(tickvals=time_intervals_non_release[::5], ticktext=[int(x) for x␣
 ↪in time_intervals_non_release[::5]])
)
```

```
fig_heatmap_release.show()
fig_heatmap_non_release.show()
```

[106]:
```
%%capture
# Line plot showing how the p-value of a two-sample Kolmogorov-Smirnov␣
 ↪statistical test comparing
# post-release price distributions on days with and without release changes␣
 ↪with each minute.
#
# line_plot_p_value_KS_test = create_ks_test_p_value_plot(ks_results_df)

price_change_bins_labels = heatmap_data_non_release.index
price_change_bins_midpoints = []
for label in price_change_bins_labels:
    lower_bound_str = label.split('%')[0]
    midpoint = float(lower_bound_str) / 100.0
    price_change_bins_midpoints.append(midpoint)
price_change_bins_midpoints = np.array(price_change_bins_midpoints)

minutes_to_test =pd.Series([float(m) for m in range(0, 16)])

ks_results = []

for minute in minutes_to_test:
    if minute not in heatmap_data_non_release.columns or minute not in␣
 ↪heatmap_data_release.columns:
        print(f"Minute {minute} not found in both heatmaps. Skipping.")
        continue

    sample_non_release = []
    frequencies_non_release = heatmap_data_non_release[minute]
    for i, freq in enumerate(frequencies_non_release):
        sample_non_release.extend([price_change_bins_midpoints[i]] * freq)
    sample_non_release = np.array(sample_non_release)

    sample_release = []
    frequencies_release = heatmap_data_release[minute]
    for i, freq in enumerate(frequencies_release):
        sample_release.extend([price_change_bins_midpoints[i]] * freq)
    sample_release = np.array(sample_release)

    if sample_non_release.size > 0 and sample_release.size > 0:
        ks_statistic, p_value = stats.ks_2samp(sample_non_release,␣
 ↪sample_release)
        ks_results.append({
            'Minute': float(minute),
            'KS Statistic': ks_statistic,
            'P-value': p_value
        })
    else:
        ks_results.append({
            'Minute': float(minute),
            'KS Statistic': np.nan,
            'P-value': np.nan,
            'Warning': 'One or both samples are empty, KS test not performed.'
        })

ks_results_df = pd.DataFrame(ks_results)

def create_ks_test_p_value_plot(ks_results_df):
    """
    Generates a line plot of KS test P-values for price change distributions.

    Args:
        ks_results_df: DataFrame containing 'Minute' and 'P-value' columns.

    Returns:
        matplotlib.figure.Figure: The generated plot figure.
    """
    line_plot_p_value_KS_test = plt.figure(figsize=(10, 6))
    plt.plot(ks_results_df['Minute'], ks_results_df['P-value'], marker='o',␣
 ↪linestyle='-')
    plt.axhline(0.1, color='r', linestyle='--', label='Significance Level (0.
 ↪1)')
    plt.title('KS Test P-values for Price Change Distributions (Release vs.␣
 ↪Non-Release Days)')
    plt.xlabel('Minutes Relative to Release')
    plt.ylabel('P-value')
    plt.xticks(ks_results_df['Minute'])
    plt.legend()
    plt.grid(True)
    return line_plot_p_value_KS_test

line_plot_p_value_KS_test = create_ks_test_p_value_plot(ks_results_df)
```

[107]:
```
%%capture
# Scatter Plot of Supply Surprise (Investing.com) vs. Percentage Price Change␣
 ↪(0-2 min)
# fig_scatter_surprise_vs_pricechange_Investingcom
# surprise_vs_pricechange_Investingcom_r_2



price_window_0to2min = price_window_60min[
    (price_window_60min['Time_Delta_Minutes'] >= 0.0) &␣
  ↪(price_window_60min['Time_Delta_Minutes'] <= 2.0)
]

merged_price_window_0to2min = pd.merge(
    price_window_0to2min.reset_index(),
    icom_eia_forecasts.reset_index()[['Release_Datetime_CST',␣
  ↪'supply_surprise', 'Actual', 'Forecast', 'Previous']],
    left_on='Release_Datetime',
    right_on='Release_Datetime_CST',
    how='left'
)

merged_price_window_0to2min = merged_price_window_0to2min.
  ↪drop(columns=['Release_Datetime_CST'])

merged_price_window_0to2min['Date'] = str(merged_price_window_0to2min['Date'])

date_string_series = merged_price_window_0to2min['Datetime_CST'].dt.
  ↪strftime('%d/%m/%Y')
merged_price_window_0to2min['Date'] = date_string_series

fxstreet_api_forecasts['Date_Formatted'] = fxstreet_api_forecasts['Date'].
  ↪apply(format_fxstreet_date)
fxstreet_api_forecasts['Date_Formatted'] = pd.
  ↪to_datetime(fxstreet_api_forecasts['Date_Formatted'], format='%d/%m/%Y').dt.
  ↪date

fxstreet_api_forecasts['Actual_API'] = fxstreet_api_forecasts['Actual']
fxstreet_api_forecasts['Actual_API'] = fxstreet_api_forecasts['Actual_API'].
  ↪astype('float')


merged_price_window_0to2min['DateString'] = pd.
  ↪to_datetime(merged_price_window_0to2min['Date'], format='%d/%m/%Y')
fxstreet_api_forecasts['Date_Formatted'] = pd.
  ↪to_datetime(fxstreet_api_forecasts['Date_Formatted'], format='%d/%m/%Y')

merged_price_window_0to2min_api = merged_price_window_0to2min.copy()
merged_price_window_0to2min_api['Actual_API'] = merged_price_window_0to2min_api.
  ↪apply(
    find_nearest_date_api,
    axis=1,
    args=(fxstreet_api_forecasts['Date_Formatted'],)
)
```

```
merged_price_window_0to2min_api = merged_price_window_0to2min_api[678:]

grouped_by_date = merged_price_window_0to2min.groupby('Date')

date_percentage_changes = []

for date, group_df in grouped_by_date:
    earliest_minute_row = group_df.sort_values(by='Datetime_CST').iloc[0]
    earliest_close_price = earliest_minute_row['Close']

    latest_minute_row = group_df.sort_values(by='Datetime_CST').iloc[-1]
    latest_close_price = latest_minute_row['Close']

    if earliest_close_price != 0:
        percentage_change = ((latest_close_price - earliest_close_price) /␣
 ↪earliest_close_price) * 100.0
    else:
        percentage_change = float('nan')

    date_percentage_changes.append({
        'Date': date,
        'Start_Datetime': earliest_minute_row['Datetime_CST'],
        'End_Datetime': latest_minute_row['Datetime_CST'],
        'Start_Price': earliest_close_price,
        'End_Price': latest_close_price,
        'Percentage_Change_0to2min': percentage_change,
        'supply_surprise': earliest_minute_row['supply_surprise'],
        'Actual' : float(str(earliest_minute_row['Actual']).replace('M', '')),
        'Forecast' : float(str(earliest_minute_row['Forecast']).replace('M',␣
 ↪'')),
        'Previous' : float(str(earliest_minute_row['Previous']).replace('M',␣
 ↪''))
    })

date_percentage_change_df = pd.DataFrame(date_percentage_changes) # icom


date_percentage_change_df['Start_Datetime'] = pd.
 ↪to_datetime(date_percentage_change_df['Start_Datetime'])

min_datetime = date_percentage_change_df['Start_Datetime'].min()
max_datetime = date_percentage_change_df['Start_Datetime'].max()

date_percentage_change_df['Time_Proportion'] =␣
 ↪(date_percentage_change_df['Start_Datetime'] - min_datetime) / (max_datetime␣
 ↪- min_datetime)
```

```
if (max_datetime - min_datetime) == pd.Timedelta(0):
    date_percentage_change_df['Time_Proportion'] = 0.5

not_nan_mask = date_percentage_change_df['supply_surprise'].notna()

date_percentage_change_df = date_percentage_change_df[not_nan_mask]

nan_mask = date_percentage_change_df['supply_surprise'].isna()

nan_count = nan_mask.sum()

nan_rows = date_percentage_change_df[nan_mask]

X = date_percentage_change_df['supply_surprise']
y = date_percentage_change_df['Percentage_Change_0to2min']

X = sm.add_constant(X)

model = sm.OLS(y, X)
results = model.fit()

surprise_vs_pricechange_Investingcom_r_2 = results.rsquared

print("\n--- Coefficients ---")
print(results.params)
print("\n--- R-squared ---")
print(f"R-squared: {results.rsquared}")

fig_scatter_surprise_vs_pricechange_Investingcom = px.scatter(
    date_percentage_change_df,
    x='supply_surprise',
    y='Percentage_Change_0to2min',
    color='Time_Proportion',
    hover_data=['Date', 'Start_Datetime', 'End_Datetime', 'Start_Price',␣
↪'End_Price'],
    title='Scatter Plot of Supply Surprise (Investing.com) vs. Percentage Price␣
↪Change (0-2 min)',
    labels={
        'supply_surprise': 'EIA WPSR Supply Surprise (Millions of Barrels)',
        'Percentage_Change_0to2min': 'Percentage Price Change (0-2 min)',
        'Time_Proportion': 'Time Progression'
    },
    color_continuous_scale=px.colors.sequential.Viridis
)

fig_scatter_surprise_vs_pricechange_Investingcom.update_layout(
    xaxis_title='EIA WPSR Supply Surprise (Millions of Barrels)',
    yaxis_title='Percentage Price Change (0-2 min)'
)

fig_scatter_surprise_vs_pricechange_Investingcom.show()
```

[108]:
```
%%capture
# Scatter Plot of Supply Surprise (API) vs. Percentage Price Change (0-2 min)
# fig_scatter_surprise_vs_pricechange_api
# surprise_vs_pricechange_API_r_2

grouped_by_date_api = merged_price_window_0to2min_api.groupby('Date')

date_percentage_changes_api = []

for date, group_df in grouped_by_date_api:
    earliest_minute_row = group_df.sort_values(by='Datetime_CST').iloc[0]
    earliest_close_price = earliest_minute_row['Close']

    latest_minute_row = group_df.sort_values(by='Datetime_CST').iloc[-1]
    latest_close_price = latest_minute_row['Close']

    if earliest_close_price != 0:
        percentage_change = ((latest_close_price - earliest_close_price) /␣
 ↪earliest_close_price) * 100.0
    else:
        percentage_change = float('nan')

    date_percentage_changes_api.append({
        'Date': date,
        'Start_Datetime': earliest_minute_row['Datetime_CST'],
        'End_Datetime': latest_minute_row['Datetime_CST'],
        'Start_Price': earliest_close_price,
        'End_Price': latest_close_price,
        'Percentage_Change_0to2min': percentage_change,
        'supply_surprise': float(str(earliest_minute_row['Actual']).
 ↪replace('M', '')) - earliest_minute_row['Actual_API'],
        'Actual_API' : earliest_minute_row['Actual_API'],
        'Actual_EIA' : earliest_minute_row['Actual']
    })

date_percentage_change_df_api = pd.DataFrame(date_percentage_changes_api) # api


date_percentage_change_df_api_sorted = date_percentage_change_df_api.
 ↪sort_values(by='supply_surprise')
```

```
date_percentage_change_df_api_sorted['Start_Datetime'] = pd.
↪to_datetime(date_percentage_change_df_api_sorted['Start_Datetime'])

min_datetime = date_percentage_change_df_api_sorted['Start_Datetime'].min()
max_datetime = date_percentage_change_df_api_sorted['Start_Datetime'].max()

date_percentage_change_df_api_sorted['Time_Proportion'] =␣
↪(date_percentage_change_df_api_sorted['Start_Datetime'] - min_datetime) /␣
↪(max_datetime - min_datetime)


X = date_percentage_change_df_api_sorted['supply_surprise']
y = date_percentage_change_df_api_sorted['Percentage_Change_0to2min']

X = sm.add_constant(X)

model = sm.OLS(y, X)
results = model.fit()

surprise_vs_pricechange_API_r_2 = results.rsquared

print("\n--- Coefficients ---")
print(results.params)
print("\n--- R-squared ---")
print(f"R-squared: {results.rsquared}")

fig_scatter_surprise_vs_pricechange_api = px.scatter(
    date_percentage_change_df_api_sorted,
    x='supply_surprise',
    y='Percentage_Change_0to2min',
    color='Time_Proportion',
    hover_data=['Date', 'Start_Datetime', 'End_Datetime', 'Start_Price',␣
↪'End_Price'],
    title='Scatter Plot of Supply Surprise (API) vs. Percentage Price Change␣
↪(0-2 min)',
    labels={
        'supply_surprise': 'EIA WPSR Supply Surprise (Millions of Barrels)',
        'Percentage_Change_0to2min': 'Percentage Price Change (0-2 min)',
        'Time_Proportion': 'Time Progression'
    },
    color_continuous_scale=px.colors.sequential.Viridis
)

fig_scatter_surprise_vs_pricechange_api.update_layout(
    xaxis_title='EIA WPSR Supply Surprise (Millions of Barrels)',
    yaxis_title='Percentage Price Change (0-2 min)'
)
```

```
fig_scatter_surprise_vs_pricechange_api.show()
```

[109]:
```
%%capture
# Scatter Plot of Supply Change (EIA) vs. Percentage Price Change (0-2 min)
# fig_scatter_surprise_vs_pricechange_EIA
# surprise_vs_pricechange_EIA_r_2

date_percentage_change_df_sorted = date_percentage_change_df.
  ↪sort_values(by='Actual')

X = date_percentage_change_df_sorted['Actual']
y = date_percentage_change_df_sorted['Percentage_Change_0to2min']

X = sm.add_constant(X)
model = sm.OLS(y, X)
results = model.fit()

surprise_vs_pricechange_EIA_r_2 = results.rsquared

print("\n--- Coefficients ---")
print(results.params)
print("\n--- R-squared ---")
print(f"R-squared: {results.rsquared}")


fig_scatter_surprise_vs_pricechange_EIA = px.scatter(
    date_percentage_change_df_sorted,
    x='Actual',
    y='Percentage_Change_0to2min',
    color='Time_Proportion',
    hover_data=['Date', 'Start_Datetime', 'End_Datetime', 'Start_Price',␣
  ↪'End_Price'],
    title='Scatter Plot of Supply Change (EIA) vs. Percentage Price Change (0-2␣
  ↪min)',
    labels={
        'Actual': 'EIA WPSR Supply Surprise (Millions of Barrels)',
        'Percentage_Change_0to2min': 'Percentage Price Change (0-2 min)',
        'Time_Proportion': 'Time Progression'
    },
    color_continuous_scale=px.colors.sequential.Viridis
)

fig_scatter_surprise_vs_pricechange_EIA.update_layout(
    xaxis_title='EIA WPSR Supply Change (Millions of Barrels)',
    yaxis_title='Percentage Price Change (0-2 min)'
)
```

```
fig_scatter_surprise_vs_pricechange_EIA.show()
```

[110]:
```
%%capture
# Kernel Regression: Actual vs. Predicted Values
# kernel_reg_plot
# mse_Kernel_Regression
# r2_Kernel_Regression

X = date_percentage_change_df_sorted[['Actual']]
y = date_percentage_change_df_sorted['Percentage_Change_0to2min']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1,␣
 ↪random_state=42)


param_grid = {
    "alpha": np.logspace(-3, 3, 7),
    "gamma": np.logspace(-3, 3, 7)
}

kernel_reg = KernelRidge(kernel='rbf')

grid_search = GridSearchCV(kernel_reg, param_grid, cv=5,␣
 ↪scoring='neg_mean_squared_error', n_jobs=-1)

grid_search.fit(X_train, y_train)

best_kernel_reg = grid_search.best_estimator_

y_pred_test = best_kernel_reg.predict(X_test)
mse_Kernel_Regression = mean_squared_error(y_test, y_pred_test)
r2_Kernel_Regression = r2_score(y_test, y_pred_test)

print(f"Best Kernel Regression model (Gaussian Kernel) with hyperparameters:␣
 ↪{grid_search.best_params_}")
print(f"Mean Squared Error on Test Set: {mse_Kernel_Regression:.4f}")
print(f"R-squared on Test Set: {r2_Kernel_Regression:.4f}")

def create_kernel_regression_plot(X_test, y_test, y_pred_test):
    """
    Generates a scatter plot comparing actual vs. predicted values from a␣
 ↪Kernel Regression model.

    Args:
        X_test (pd.DataFrame): DataFrame containing the test features,␣
 ↪including 'Actual' column.
```

```
        y_test (pd.Series): Series of actual target values.
        y_pred_test (np.ndarray): Array of predicted target values from the␣
↪Kernel Regression model.

    Returns:
        matplotlib.figure.Figure: The matplotlib Figure object containing the␣
↪plot.
                                   This can be stored and displayed later.
    """
    fig, ax = plt.subplots(figsize=(10, 6))

    ax.scatter(X_test['Actual'], y_test, color='blue', label='Actual Values',␣
↪alpha=0.7)
    ax.scatter(X_test['Actual'], y_pred_test, color='red', label='Kernel␣
↪Regression Predictions', alpha=0.7)

    ax.set_xlabel('Actual')
    ax.set_ylabel('Percentage_Change_0to2min')
    ax.set_title('Kernel Regression: Actual vs. Predicted Values')
    ax.legend()
    ax.grid(True)

    return fig

kernel_reg_plot = create_kernel_regression_plot(X_test, y_test, y_pred_test)
```

[111]:
```
%%capture
# LOESS Regression: Percentage Change vs Actual
# loess_plot
# mse_loess
# r2_loess

y = date_percentage_change_df_sorted['Percentage_Change_0to2min']
X = date_percentage_change_df_sorted['Actual']

lowess = sm.nonparametric.lowess(y, X, frac=0.3)

x_fitted = lowess[:, 0]
y_fitted = lowess[:, 1]

date_percentage_change_df_sorted['LOESS_Fitted_Percentage_Change'] = np.
↪interp(X, x_fitted, y_fitted)

mse_loess = mean_squared_error(y,␣
↪date_percentage_change_df_sorted['LOESS_Fitted_Percentage_Change'])
r2_loess = r2_score(y,␣
↪date_percentage_change_df_sorted['LOESS_Fitted_Percentage_Change'])
```

```python
print(f"Mean Squared Error (MSE) of the LOESS model: {mse_loess:.6f}")
print(f"R-squared (R2) of the LOESS model: {r2_loess:.6f}")

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

ax.scatter(X, y, label='Observed Data', alpha=0.6)
ax.plot(x_fitted, y_fitted, color='red', label='LOESS Fit')
ax.set_xlabel('Actual')
ax.set_ylabel('Percentage_Change_0to2min')
ax.set_title('LOESS Regression: Percentage Change vs Actual')
ax.legend()
ax.grid(True)

loess_plot = fig
```

[112]:
```python
%%capture
# rf_feature_fig
# xgb_feature_fig
# rf_actual_pred_fig
# xgb_actual_pred_fig
# rf_mse
# rf_r2
# xgb_mse
# xgb_r2

X = date_percentage_change_df_sorted[['Actual', 'Forecast', 'Start_Price']]
y = date_percentage_change_df_sorted['Percentage_Change_0to2min']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)


param_grid_rf = {
    'n_estimators': [40, 50, 60],
    'max_depth': [1, 5],
    'min_samples_split': [2, 3, 4, 5],
    'min_samples_leaf': [4, 5, 6, 7]
}

param_grid_xgb = {
    'n_estimators': [10, 15, 20, 25],
    'max_depth': [1, 3, 5, 7, 9, 11],
    'learning_rate': [0.0001, 0.001, 0.005],
    'subsample': [0.5, 0.6, 0.7],
    'colsample_bytree': [0.7, 0.8, 0.9]
}
```

```
grid_search_rf = GridSearchCV(estimator=RandomForestRegressor(random_state=42),
                           param_grid=param_grid_rf,
                           scoring='neg_mean_squared_error',
                           cv=3,
                           n_jobs=-1)
grid_search_rf.fit(X_train, y_train)
best_rf_model = grid_search_rf.best_estimator_

grid_search_xgb = GridSearchCV(estimator=XGBRegressor(objective='reg:
 ↪squarederror', random_state=42),
                            param_grid=param_grid_xgb,
                            scoring='neg_mean_squared_error',
                            cv=3,
                            n_jobs=-1)
grid_search_xgb.fit(X_train, y_train)
best_xgb_model = grid_search_xgb.best_estimator_


rf_predictions = best_rf_model.predict(X_test)
xgb_predictions = best_xgb_model.predict(X_test)

rf_mse = mean_squared_error(y_test, rf_predictions)
rf_r2 = r2_score(y_test, rf_predictions)

xgb_mse = mean_squared_error(y_test, xgb_predictions)
xgb_r2 = r2_score(y_test, xgb_predictions)

print("Best Random Forest Regressor Performance (after GridSearchCV):")
print(f"  Mean Squared Error: {rf_mse:.4f}")
print(f"  R-squared: {rf_r2:.4f}")
print(f"  Best parameters: {grid_search_rf.best_params_}")

print("\nBest XGBoost Regressor Performance (after GridSearchCV):")
print(f"  Mean Squared Error: {xgb_mse:.4f}")
print(f"  R-squared: {xgb_r2:.4f}")
print(f"  Best parameters: {grid_search_xgb.best_params_}")


def plot_feature_importance(best_rf_model, best_xgb_model, X):
    """
    Generates and returns two separate matplotlib figure objects for feature␣
 ↪importance plots
    for Random Forest and XGBoost models.

    Args:
        best_rf_model: Trained Random Forest model object.
        best_xgb_model: Trained XGBoost model object.
        X: DataFrame used for training, needed for column names.

    Returns:
        tuple: A tuple containing two matplotlib.figure.Figure objects:
               (rf_feature_importance_fig, xgb_feature_importance_fig)
    """
    rf_feature_importance_fig = plt.figure(figsize=(10, 5))
    plt.subplot(1, 1, 1)
    rf_feature_importance = pd.Series(best_rf_model.feature_importances_,␣
 ↪index=X.columns).sort_values(ascending=False)
    rf_feature_importance.plot(kind='bar')
    plt.title('Best Random Forest - Feature Importance')
    plt.ylabel('Importance Score')
    plt.tight_layout()

    xgb_feature_importance_fig = plt.figure(figsize=(10, 5))
    plt.subplot(1, 1, 1)
    xgb_feature_importance = pd.Series(best_xgb_model.feature_importances_,␣
 ↪index=X.columns).sort_values(ascending=False)
    xgb_feature_importance.plot(kind='bar')
    plt.title('Best XGBoost - Feature Importance')
    plt.ylabel('Importance Score')
    plt.tight_layout()

    return rf_feature_importance_fig, xgb_feature_importance_fig


def plot_actual_vs_predicted(y_test, rf_predictions, xgb_predictions):
    """
    Generates and returns two separate matplotlib figure objects for Actual vs.␣
 ↪Predicted plots
    for Random Forest and XGBoost models.

    Args:
        y_test: Actual target values.
        rf_predictions: Predictions from the Best Random Forest model.
        xgb_predictions: Predictions from the Best XGBoost model.

    Returns:
        tuple: A tuple containing two matplotlib.figure.Figure objects:
               (rf_actual_vs_predicted_fig, xgb_actual_vs_predicted_fig)
    """
    rf_actual_vs_predicted_fig = plt.figure(figsize=(6, 6))
    plt.subplot(1, 1, 1)
    plt.scatter(y_test, rf_predictions)
    plt.xlabel('Actual Percentage Change')
    plt.ylabel('Predicted Percentage Change (Best Random Forest)')
    plt.title('Best Random Forest: Actual vs. Predicted')
    plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],␣
 ↪color='red')
    plt.tight_layout()

    xgb_actual_vs_predicted_fig = plt.figure(figsize=(6, 6))
    plt.subplot(1, 1, 1)
    plt.scatter(y_test, xgb_predictions)
    plt.xlabel('Actual Percentage Change')
    plt.ylabel('Predicted Percentage Change (Best XGBoost)')
    plt.title('Best XGBoost: Actual vs. Predicted')
    plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],␣
 ↪color='red')
    plt.tight_layout()

    return rf_actual_vs_predicted_fig, xgb_actual_vs_predicted_fig

rf_feature_fig, xgb_feature_fig = plot_feature_importance(best_rf_model,␣
 ↪best_xgb_model, X)
rf_actual_pred_fig, xgb_actual_pred_fig = plot_actual_vs_predicted(y_test,␣
 ↪rf_predictions, xgb_predictions)
```

[113]:
```
%%capture
# mse_ridge
# r2_ridge
# prediction_plot_ridge_reg
# mse_baseline
# r2_baseline

def ridge_regression_for_price_change(dataframe):
    """
    Fits a Ridge regression model and calculates performance metrics.
    """
    try:
        X = dataframe[['Actual', 'Start_Price']]
        y = dataframe['Percentage_Change_0to2min']
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
 ↪2, random_state=42)

        mean_percentage_change_train = y_train.mean()
        y_baseline_pred_test = np.full(y_test.shape,␣
 ↪mean_percentage_change_train)

        mse_baseline_test = mean_squared_error(y_test, y_baseline_pred_test)
        r2_baseline_test = r2_score(y_test, y_baseline_pred_test)
        ridge_model = Ridge(alpha=1.0)
        ridge_model.fit(X_train, y_train)
        y_pred_test_ridge = ridge_model.predict(X_test)
        mse_ridge_test = mean_squared_error(y_test, y_pred_test_ridge)
        r2_ridge_test = r2_score(y_test, y_pred_test_ridge)

        coefficients = pd.DataFrame({'Feature': ['Intercept'] + list(X.columns),
                                     'Coefficient': [ridge_model.intercept_] +␣
 ↪list(ridge_model.coef_)})

        plt.figure(figsize=(8, 6))
        plt.scatter(y_test, y_pred_test_ridge, alpha=0.7, label='Ridge␣
 ↪Predictions')
        plt.scatter(y_test, y_baseline_pred_test, alpha=0.7, marker='x',␣
 ↪color='green', label='Baseline Predictions (Mean)')
        plt.plot(y_test, y_test, color='red', linestyle='--', linewidth=1,␣
 ↪label='Perfect Prediction')
        plt.xlabel('Actual Percentage Change (Test Set)')
        plt.ylabel('Predicted Percentage Change (Test Set)')
        plt.title('Ridge vs. Baseline (Mean) Prediction: Actual vs Predicted␣
 ↪Price Change')
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        predictions_plot = plt


        return ridge_model, mse_ridge_test, r2_ridge_test, coefficients,␣
 ↪predictions_plot, mse_baseline_test, r2_baseline_test

    except Exception as e:
        print(f"An error occurred during Ridge regression fitting: {e}")
        return None, None, None, None, None, None, None

ridge_model, mse_ridge, r2_ridge, coefficients_df, prediction_plot_ridge_reg,␣
 ↪mse_baseline, r2_baseline =␣
 ↪ridge_regression_for_price_change(date_percentage_change_df_sorted.copy())

if ridge_model:
    print("\n--- Model Comparison Results ---")
    print("\nRidge Regression Model:")
    print(f"  Test Mean Squared Error: {mse_ridge:.4f}")
    print(f"  Test R-squared: {r2_ridge:.4f}")

    print("\nBaseline Model (Mean Prediction):")
    print(f"  Test Mean Squared Error: {mse_baseline:.4f}")
    print(f"  Test R-squared: {r2_baseline:.4f}")

    print("\nPrediction Plot:")
    prediction_plot_ridge_reg.show()
```

1.10 Implications and Limitations

Replace this text with your analysis. Who might benefit from your analysis and who might be excluded or otherwise harmed by it? What about the data setting might have impacted your results? Explain at least 3 limitations of your analysis and how others should or shouldn't be advised to use your conclusions. You may remove the code cell below if you don't need it.

1.10.1 1.11 Implications and Limitations

Implications:

- **Market Timing and Trading Strategies:** Our findings regarding the short-term market reaction (approximately 2-3 minutes) post-WPSR release could be of interest to high-frequency traders seeking to capitalize on the initial market response. However, given the low predictability indicated by our machine learning models, relying solely on WPSR data for profitable trading strategies is highly risky.
- **Understanding Market Efficiency:** The rapid market absorption of WPSR information (within minutes) suggests a relatively efficient market, at least in terms of incorporating publicly available information. However, the noise and low predictability also indicate that the market reaction is not entirely deterministic or easily modeled using simple supply and demand factors and basic machine learning.
- **EIA WPSR Report Significance:** Our project reinforces the importance of the EIA WPSR as a market-moving event. Even if the exact price reaction is hard to predict, the report undoubtedly triggers significant trading activity and short-term price volatility in the crude oil futures market.

Limitations:

1. **Model Simplicity and Feature Engineering:** Our machine learning models were relatively basic and relied on a limited set of features (primarily supply change and prior price). More sophisticated models incorporating a wider range of market indicators (e.g., volatility, interest rates, global economic news, sentiment analysis), and more advanced feature engineering (e.g., lagged variables, technical indicators) might yield improved results.
2. **Data Noise and Non-Linearity:** The inherent noise in financial market data and the potential for non-linear and complex relationships between WPSR data and price movements pose a significant challenge. Simple linear models and even basic non-linear models may be insufficient to capture the underlying dynamics.
3. **Limited Scope of WPSR Data Interpretation:** Our project focused on two simplified interpretations of how traders might react to WPSR data. In reality, market participants likely employ far more sophisticated and diverse strategies, considering multiple aspects of the WPSR report, incorporating proprietary information, and reacting to market sentiment and momentum in addition to fundamental supply-demand factors.
4. **Focus on Aggregate Supply Change:** We primarily focused on the aggregate crude oil supply change. The WPSR contains a wealth of other information (e.g., gasoline and distillate

inventories, refinery utilization, production data). Investigating the market's reaction to these other components might reveal additional insights and potentially stronger predictive signals.

5. **Time Horizon:** Our analysis focused on the immediate (minute-to-minute and up to 2-minute) price reactions. The WPSR might have longer-term impacts on crude oil prices that our short-term analysis did not capture.

Recommendations and Cautions:

- **Further Research:** Future research should explore more advanced modeling techniques, incorporate a broader range of market variables, and delve deeper into the nuances of market expectations and WPSR data interpretation.
- **Trading Cautions:** Based on our findings, it is **not advisable** to use the simple models developed in this project for real-world trading decisions. The low predictive accuracy and high level of noise in the market make any trading strategy based solely on these models extremely risky.
- **Context-Dependent Interpretation:** The market's reaction to WPSR data is likely highly context-dependent, influenced by prevailing market conditions, global events, and overall economic sentiment. Any interpretation of WPSR data and its potential market impact should consider these broader contextual factors.
- **Beyond Simple Supply and Demand:** Relying solely on introductory microeconomic supply and demand principles to understand crude oil market reactions to complex reports like the WPSR is likely insufficient. A more nuanced and multi-faceted approach is required to capture the intricacies of market behavior in this context.