Capstone Project Report

Crude Oil Price Forecasting

Abhishek Krovvidi
Kavyasri Jadala
Madhumathi Ponnusamy
Pranay Kumar Chakilam
Shashank Reddy Manda

MGMT 5900- Big Data

Dr. Alok R. Chaturvedi

Mitchell E. Daniels, Jr. School of Business, Purdue University

December 16,2023.

Overview of Data Sources

Our dataset is gathered daily and incorporates a weighted average of volumes, spanning from 2011 to the present. This extensive timeline reflects the dataset's complexity. We've developed a machine learning model to predict crude oil prices over the next seven days, utilizing both streaming and batch data on Google Cloud Platform (GCP).

Key Data Sources and Indicators:

a. *Commodity Prices:*

- Our primary source was the St. Louis Fed's FRED API.
- Focus on oil market representation, especially the West Texas Intermediate (WTI) price at Cushing, Oklahoma (DCOILWTICO).

b. *Debt Market Indicators:*

- The dataset includes a spectrum of bond market indicators, particularly LIBOR rates at varying maturities (overnight, 1-month, 3-month, and 12-month).
- To broadly represent consumer and corporate markets, we incorporated indices for high yield returns and prime corporate debt returns.

c. *Energy-Related Series:*

- To gauge energy sector trends, we collected data on natural gas and energy sector volatility, again sourced from the St. Louis Fed's FRED API.
- Key metrics include the Henry Hub Natural Gas Spot Price (MHHNGSP)[1] and the CBOE Energy Sector ETF Volatility Index (VXXLECLS)[2]

d. Traditional Currencies:

- Our currency analysis involved archived Federal Government database records, focusing on the exchange rates of the US Dollar against key global currencies and their historical patterns.
- This included the Chinese Yuan (DEXCHUS)[3], Japanese Yen (DEXJPUS)[4], Euro (DEXUSEU)[5], Mexican Peso (DEXMXUS)[6], and Australian Dollar (DEXUSAL) [7].

1. Data pre-processing

a. Generating in Calendar Attributes for Merge

To account for time-related variables, we've developed a function named `**generate_calendar `.** This function is essential for augmenting our dataset with calendar attributes, a critical step in refining our data analysis. By integrating these attributes, such as month or weekday, we enhance our machine learning model's capability to interpret and utilize temporal patterns effectively. This approach is particularly beneficial when combined with the extensive data procured from the St. Louis Fed's FRED API, allowing for a more nuanced and comprehensive understanding of the underlying trends and patterns.

i. Code

```
import numpy as np
import pandas as pd
from pandas.tseries.offsets import YearEnd
from pandas.tseries.holiday import USFederalHolidayCalendar

def generate_calendar(year, drop_index=False):
    '''
    Simple function to generate a calendar containing
    US holidays, weekdays and holiday weeks.
    '''
    start_date = pd.to_datetime('1/1/'+str(year))
    end_date = start_date + YearEnd()
    DAT = pd.date_range(str(start_date), str(end_date), freq='D')
    MO = [d.strftime('%B') for d in DAT]
    holidays = USFederalHolidayCalendar().holidays(start=start_date, end=end_date)

    cal_df = pd.DataFrame({'date':DAT, 'month':MO})
    cal_df['year'] = [format(d, '%Y') for d in DAT]
    cal_df['weekday'] = [format(d, '%A') for d in DAT]
    cal_df['is_weekday'] = cal_df.weekday.isin(['Monday','Tuesday','Wednesday','Thursday','Friday'])
    cal_df['is_weekday'] = cal_df['is_weekday'].astype(int)
    cal_df['is_holiday'] = cal_df['date'].isin(holidays)
    cal_df['is_holiday'] = cal_df['is_holiday'].astype(int)
    cal_df['is_holiday_week'] = cal_df.is_holiday.rolling(window=7, center=True, min_periods=1).sum()
    cal_df['is_holiday_week'] = cal_df['is_holiday_week'].astype(int)

    if not drop_index:
        cal_df.set_index('date', inplace=True)

    return cal_df

def make_calendars(year_list, drop_index):
    cal_list = [generate_calendar(year, drop_index=drop_index) for year in year_list]
    cal_df = pd.concat(cal_list)
    return cal_df

year_list = [str(int(i)) for i in np.arange(2011, 2019)]
cal_df = make_calendars(year_list, drop_index=False)
cal_df.head()
```

ii. Output

Date	Month	Year	Weekday	Is Weekday	Is Holiday	Is Holiday Week
01/01/23	January	2023	Sunday	0	0	0
03/01/23	January	2023	Monday	1	0	0
04/01/23	January	2023	Tuesday	1	0	0
05/01/23	January	2023	Wednesday	1	0	0
06/01/23	January	2023	Thursday	1	0	0

b. Integrating Calendar Data and Handling Missing Values

Incorporating calendar details into our dataset, which starts from 2011 and involves an outer join, inevitably results in missing values. This phenomenon is common in financial datasets, notably in indicators like the Dow Jones Industrial Average (DJIA), where non-trading days like weekends and holidays lead to NaN (Not a Number) values. Additionally, some metrics are not recorded on a daily basis, further contributing to these gaps.

To manage these missing values, we utilize the fillna function from the Pandas library in a sequential two-step process. The first step involves applying the 'backfill' (bfill) method, which fills a missing value with the next valid observation. Subsequently, we use the 'forward fill' (ffill) method, where each NaN is replaced by the most recent non-null value preceding it.

This methodology is selected for its straightforwardness, allowing us to avoid more complex data imputation techniques. While this approach may not be the most scientifically rigorous, it provides a practical solution for handling missing data in our financial time series analysis.

```
econ_df = econ_df.join(cal_df, how='outer')
econ_df = econ_df.fillna(method='bfill')
econ_df = econ_df.fillna(method='ffill')
```

c. Eliminating Records Beyond Current Date

To ensure the dataset only includes records up to the present day, we utilize Python's `datetime` module to filter out entries dated beyond the current date. This process is important to eliminate any future dates that might have been inadvertently added by the calendar function, as these dates are not applicable to our current analysis.

Here's the revised code snippet:

```
from datetime import datetime as dt
import pandas as pd

# Filtering out records in the dataset that are dated after today's date
current_date = dt.now()
current_date_filter = pd.to_datetime(econ_df.index.values) <= current_date
econ_df = econ_df[current_date_filter]
```

In this code, `dt.now()` is used to get the current date and time. The dataframe `**econ_df**` is then filtered to retain only those records where the index (assumed to be dates) is on or before the current date. This ensures the analysis is conducted only on relevant, past or current data.

d. Generating One-Hot Encoded Features

In preparation for neural network modeling, the categorical columns 'month', 'year', and 'weekday' in our dataset are transformed into one-hot encoded vectors. This conversion, accomplished using Pandas' **pd.get_dummies** function, changes these columns into a machine learning-friendly format.

```
# One-hot encoding of specific columns for neural network compatibility
econ_df = pd.get_dummies(econ_df, columns=['month', 'year', 'weekday'], drop_first=True)
```

e. Transforming Column Names to Lowercase

For consistency and accessibility in data management, we convert all column names in the **econ_df** DataFrame to lowercase. This standardization is a fundamental aspect of data preprocessing, ensuring uniformity in referencing columns.

```
# Standardizing column names to lowercase for uniformity
econ_df.columns = [column_name.lower() for column_name in econ_df.columns]
print(econ_df.columns.tolist())
```

The resulting columns will include financial indicators, one-hot encoded time attributes, and additional features like 'is_weekday', 'is_holiday', and 'is_holiday_week'.

The resulting column names will be:
['sp500', 'nasdaqcom', 'djia', 'ru2000pr', 'bogmbasew', 'dexjpus', 'dexuseu', 'dexchus', 'dexusal', 'vixcls', 'usdontd156n', 'usd1mtd156n', 'usd3mtd156n', 'usd12md156n', 'bamlhyh0a0hym2triv', 'bamlcc0a1aaatriv', 'goldamgbd228nlbm', 'dcoilwtico', 'mhhngsp', 'vxxlecls', 'is_weekday', 'is_holiday', 'is_holiday_week', 'month_august', 'month_december', 'month_february', 'month_january', 'month_july', 'month_june', 'month_march', 'month_may', 'month_november', 'month_october', 'month_september', 'year_2012', 'year_2013', 'year_2014', 'year_2015', 'year_2016', 'year_2017', 'year_2018', 'weekday_monday', 'weekday_saturday', 'weekday_sunday', 'weekday_thursday', 'weekday_tuesday', 'weekday_wednesday'].

2. Feature Engineering

a. Enhancing Data Signal and Reducing Noise through Feature Engineering:

Our strategy for refining the data involves reshaping and processing it to highlight meaningful trends over time while minimizing noise.

i. Data Transformation:

The econ_df DataFrame is melted into a long format with three columns: 'date', 'variable', and 'value'. And its output as shown in the figure:

```
econ_df_melt = econ_df.copy()
econ_df_melt.reset_index(inplace=True)
econ_df_melt.rename(columns={'index': 'date'}, inplace=True)
econ_df_melt = econ_df_melt.melt(id_vars=['date'])
print(econ_df_melt.head())
```

date	variable	value
2023-01-01	sp500	1271.87
2023-01-02	sp500	1271.87
2023-01-03	sp500	1271.87
2023-01-04	sp500	1270.20
2023-01-05	sp500	1276.56

ii. Signal Processing:

We apply a split-apply-combine method. The data is grouped by 'variable', then we calculate the percent change and a rolling window mean of this percent change.

```
onehot_cols = [...]  # List of binary columns
window = 30  # Rolling window size
smooth_df = pd.DataFrame()

for variable, group_df in econ_df_melt.groupby('variable'):
    if variable not in onehot_cols:
        group_df['pct_change'] = group_df['value'].pct_change()
        colname = 'rolling_' + str(window) + '_mean'
        group_df[colname] = group_df['pct_change'].rolling(window=window).mean()
    else:
        group_df[colname] = group_df['value']
    smooth_df = smooth_df.append(group_df)

print(smooth_df.head())
```

This process yields a dataset where raw values are replaced with rolling window percent changes, making it more sensitive to market trends.The transformed DataFrame smooth_df now contains columns that better reflect the underlying trends and are more suitable for analytical and modeling purposes.

b. Implementing the Split-Apply-Combine Process for Feature Calculation

To enhance the analytical value of our econ_df_melt dataset, we employ a split-apply-combine strategy. This approach entails segmenting the dataset based on distinct 'variable' values, performing specific calculations on each subset, and then amalgamating the results. Importantly, the binary columns listed in onehot_cols are excluded from this transformation as they do not necessitate similar modifications.

List of Binary Columns (onehot_cols):

```
onehot_cols = [
    'is_weekday', 'is_holiday', 'is_holiday_week',
    'month_august', 'month_december', 'month_february',
    'month_january', 'month_july', 'month_june', 'month_march',
    'month_may', 'month_november', 'month_october', 'month_september',
    'year_2011', 'year_2012', 'year_2013', 'year_2014',
    'year_2015', 'year_2016', 'year_2017', 'year_2018',
    'weekday_monday', 'weekday_saturday', 'weekday_sunday',
    'weekday_thursday', 'weekday_tuesday', 'weekday_wednesday'
]
```

Applying the Transformation with a Rolling Window:

```python
window = 30  # Rolling window size
smooth_df = pd.DataFrame()

# Implementing split-apply-combine based on the 'variable' column
for variable, group_df in econ_df_melt.groupby('variable'):
    if variable not in onehot_cols:
        group_df['pct_change'] = group_df['value'].pct_change()  # Calculate percent change
        colname = 'rolling_' + str(window) + '_mean'
        group_df[colname] = group_df['pct_change'].rolling(window=window).mean()  # Compute rolling mean
    else:
        group_df[colname] = group_df['value']  # Retain binary columns as-is

    smooth_df = smooth_df.append(group_df)  # Reassemble the processed data into smooth_df

print(smooth_df.head())
```

In this process, we iterate over each group defined by unique 'variable' values. For non-binary columns, we calculate the percentage change and then compute its rolling window mean. The binary columns are simply carried over without alteration. The resulting dataset, smooth_df, now includes refined features that better represent trends and patterns over time, thereby increasing its utility for modeling and analysis.

Sample Output:

Index	Date	pct_change	rolling_30_mean	value	variable
44927	2023-01-01	NaN	NaN	479.31	bamlc0a1aaatriv
44928	2023-01-02	0.00000	NaN	479.31	bamlc0a1aaatriv
44929	2023-01-03	0.00000	NaN	479.31	bamlc0a1aaatriv
44930	2023-01-04	-0.000542	NaN	479.05	bamlc0a1aaatriv
44931	2023-01-05	-0.007348	NaN	475.53	bamlc0a1aaatriv

3. Data Visualization and Restoration to Original Structure

a. Data Visualization Approach

The data visualization process involves generating time-series plots for each continuous feature in the dataset. Alongside each time series plot, an inverted histogram is created on the right side to analyze the distribution of percent changes over a specific window period. This comprehensive approach allows for a thorough examination of both the temporal trends and distributional characteristics of the data.

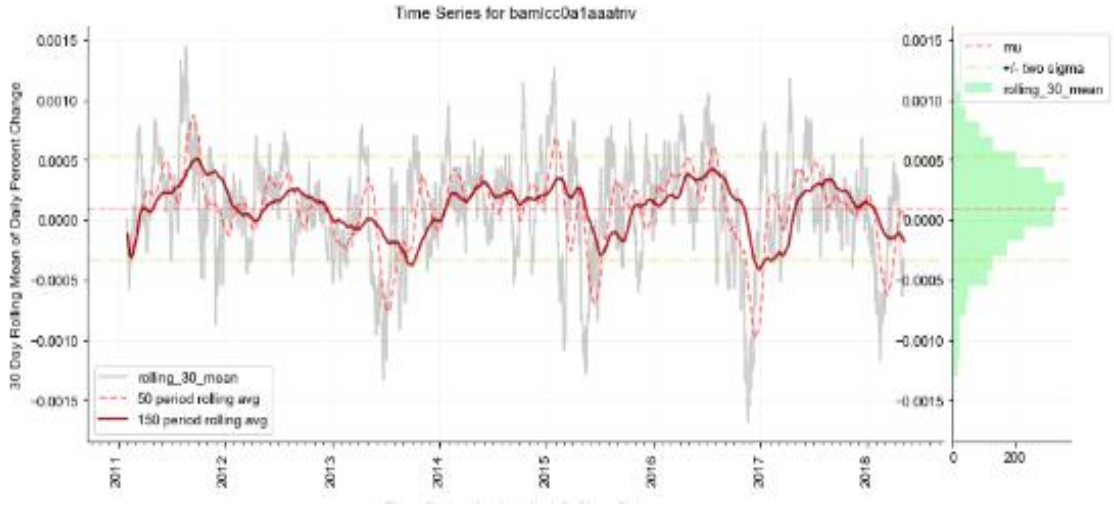
b. Python Code for Visualization

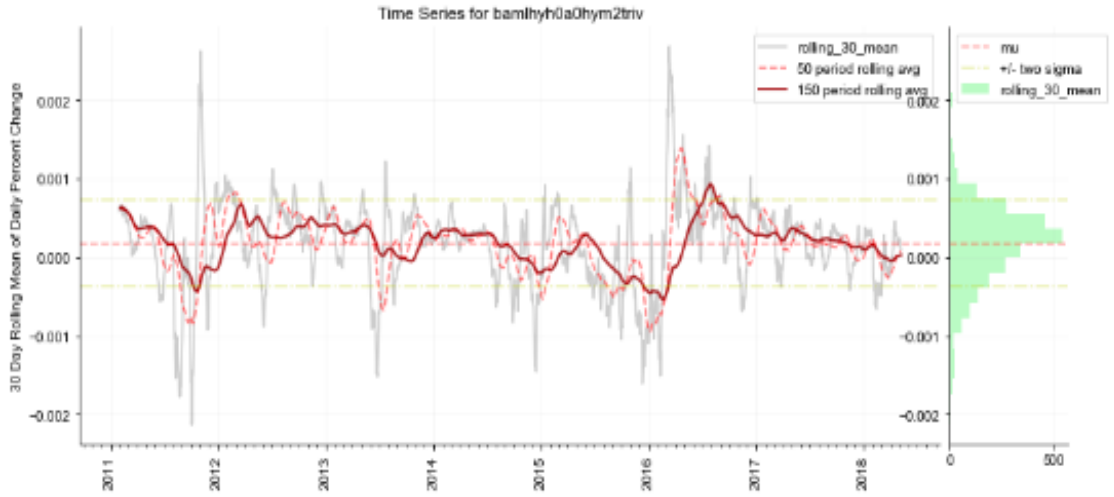
The following Python code uses matplotlib and seaborn for visualization and mpl_toolkits for additional plot features. The function visualize_data_with_histograms is designed to plot a time series alongside its rolling average, incorporating an adjacent histogram to assess the distribution of values.

```
import matplotlib.pyplot as plt
from matplotlib.dates import YearLocator, MonthLocator, DateFormatter
from mpl_toolkits.axes_grid1 import make_axes_locatable
import seaborn as sns
%matplotlib inline

# Setting locators and formatter for dates
yearly_locator = YearLocator()
monthly_locator = MonthLocator()
yearly_formatter = DateFormatter('%Y')

def visualize_data_with_histograms(dataframe, date_column, series_column,
                                   group_column, prepend_title='{}', labels=None,
                                   rotate_x=0, pad_x_label=60, rolling_window=15,
                                   columns_to_exclude=[]):
    """
    This function plots a time series and its rolling average, including a histogram
    to the right side of the time series plot.
    Parameters:
    dataframe: DataFrame containing the time series data
    date_column: Column name for datetime data
    series_column: Column name for the time series data
    group_column: Column name for grouping the plots
    labels: Dictionary containing labels for x and y axes
    prepend_title: String to prepend to the title of each subplot
    rotate_x: Degrees to rotate x-axis labels
    pad_x_label: Padding for the x-axis label
    rolling_window: Window size for the rolling average
    columns_to_exclude: List of column names to exclude from plotting
    """
```

```
    unique_groups = dataframe[group_column].unique()
    num_rows = len(unique_groups) - len(columns_to_exclude)
    plot_size = (13, 6 * num_rows)
    fig, axis_set = plt.subplots(num_rows, 1, figsize=plot_size)
    prepend_title_hist = 'Histogram of ' + prepend_title
    counter = 0

    for idx, group_value in enumerate(unique_groups):
        subset_df = dataframe[dataframe[group_column] == group_value]
        if group_value not in columns_to_exclude:
            current_axis = axis_set[counter]
            current_axis.plot(subset_df[date_column], subset_df[series_column], alpha=0.2, color='black')
            # Plot rolling averages with different windows
            for multiple in [1, 3]:
                roll_window = rolling_window * multiple
                subset_df[f'rolling_{roll_window}_avg'] = subset_df[series_column].rolling(window=roll_window, min_periods=min(5, roll_window)).mean()
                current_axis.plot(subset_df[date_column], subset_df[f'rolling_{roll_window}_avg'], label=f'{roll_window} period rolling avg')

            # Statistical lines
            mean = subset_df[series_column].mean()
            std_dev = subset_df[series_column].std()
            for sigma_multiplier in [-1, 0, 1]:
                current_axis.axhline(mean + sigma_multiplier*std_dev, linestyle='--', color='red' if sigma_multiplier == 0 else 'yellow', alpha=0.3)

            current_axis.set_title(prepend_title.format(group_value))
            current_axis.legend(loc='best')
            current_axis.set_ylim(mean - 3*std_dev, mean + 3*std_dev)

            if labels:
                current_axis.set_xlabel(labels['x_label'])
                current_axis.set_ylabel(labels['y_label'])

            current_axis.xaxis.set_minor_locator(monthly_locator)
            current_axis.grid(alpha=0.1)
```

```
            if rotate_x != 0:
                plt.setp(current_axis.get_xticklabels(), rotation=rotate_x)

            # Histogram on the right
            divider = make_axes_locatable(current_axis)
            right_axis = divider.append_axes('right', 1.2, pad=0.1, sharey=current_axis)
            right_axis.grid
```





c. Data Reshaping Process

After performing the analytical steps on our dataset, it's essential to reshape the data back to its original wide format. This format is particularly useful for visualizations and analyses that require the dataset's initial structure. To achieve this, we employ a pivot operation on the transformed dataframe.

Here is the Python code snippet for pivoting the melted **dataframe (smooth_df)** back to its original wide format. This operation incorporates the newly calculated features into the dataset. The code concludes with displaying the first few records of the reshaped dataframe.

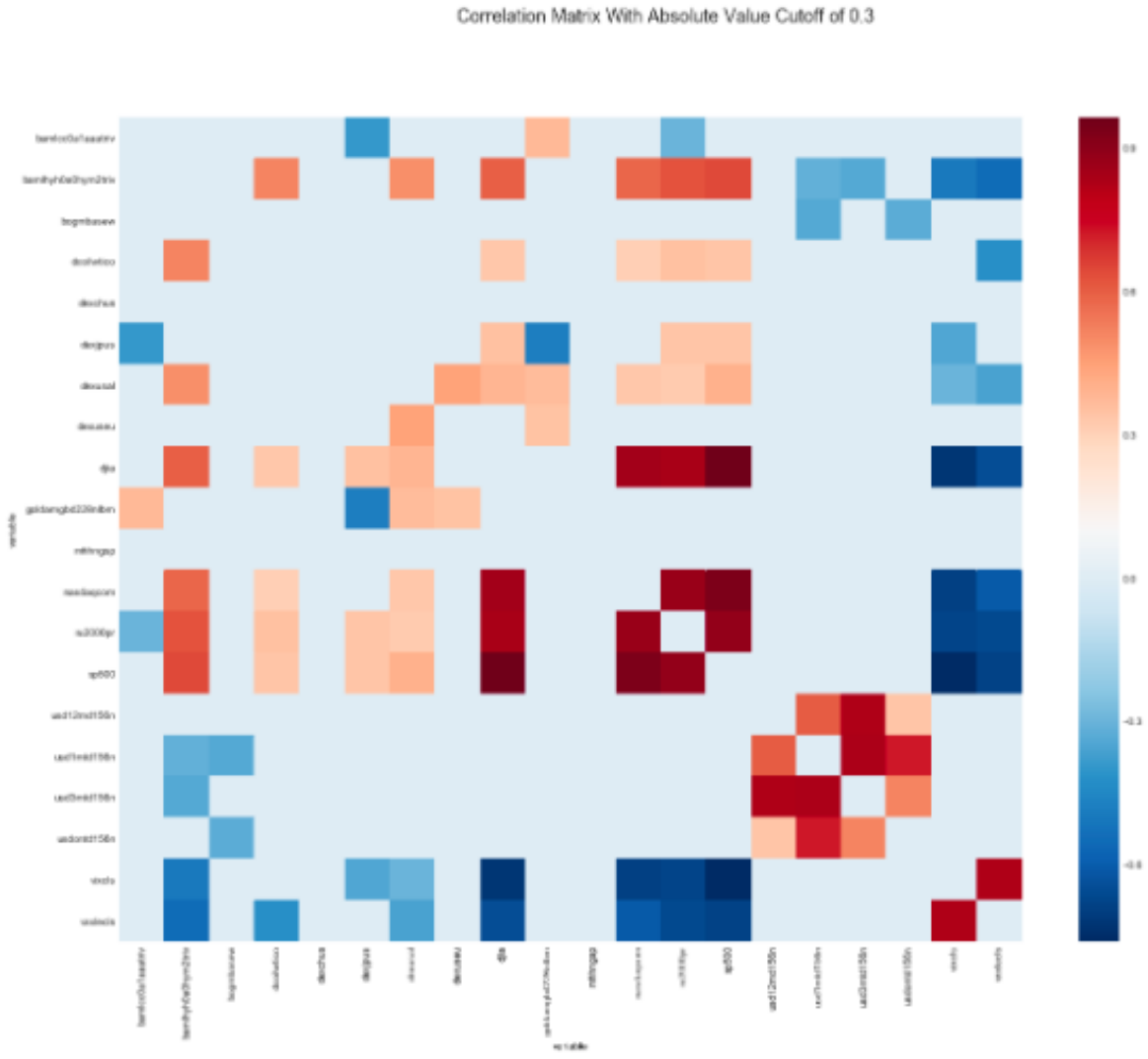
```
# Pivot the melted dataframe to restore original wide format with new features
reshaped_df = smooth_df.pivot(index='date', columns='variable', values='rolling_30_mean')
reshaped_df.dropna(inplace=True)

# Display the first few records of the reshaped dataframe
print(reshaped_df.head())
```

Date	bamlcc0a1a	bamlhyh0a(	bogmbasew	dcoilwtico	dexchus	dexjpus	dexusal	dexuseu	djia	goldamgbd	vixcls		
31/01/23	0.000007	0.000589	0.001434	-0.000108	0.000063	0.000181	-0.00072	0.000867	0.000634	-0.001722	0.00471		
01/02/23	-0.000074	0.00062	0.001434	-0.000639	0.000023	-0.000018	-0.000233	0.001056	0.001049	-0.001622	0.001467		
02/02/23	-0.000175	0.000663	0.001434	-0.00055	-0.000168	0.000121	-0.00043	0.001022	0.001054	-0.001634	0.000843		
03/02/23	-0.000219	0.000622	0.001434	0.000117	-0.000254	-0.000181	0.000348	0.000882	0.001052	-0.001746	0.000103		
vxxlecls	weekday_m	weekday_sa	weekday_su	weekday_th	weekday_tu	weekday_w	year_2012	year_2013	year_2014	year_2015	year_2016	year_2017	year_2018
0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0

Executing this code snippet will reorganize the data into its original format, with columns representing each variable and rows indexed by date. The inclusion of new features, like rolling

averages, enriches the dataset for subsequent visualizations and analyses. The final step, reshaped_df.head(), provides a quick view of the first few entries in this restructured dataset.

d. Visualize: Heatmap



To gain insights into the relationships among various financial indicators, we engage in two key analytical approaches:

i. Scatterplot Matrix for Selected Columns: This visualization technique involves creating scatterplot matrices for a subset of columns. These matrices enable us to visually explore the relationships between different financial indicators, making it easier to identify patterns or correlations that might not be immediately apparent.

ii. Data Preparation for Crude Oil Price Prediction: We also prepare the dataset specifically for predicting crude oil prices. This involves integrating actual crude oil prices into our smoothed

dataset. Following this, we split the data into subsets for model training and evaluation, ensuring that the model is well-trained and its performance accurately assessed.

e. Correlation Heatmap Analysis

To further our understanding of the interconnections between financial indicators, we utilize the plot_correlation_heatmap function. This function serves a crucial role in our analysis:

- Pearson Correlation Coefficient Calculation: It calculates the Pearson correlation coefficient between each pair of variables in our dataset. This coefficient is a statistical measure that expresses the extent of a linear relationship between two variables.
- Heatmap Visualization: The calculated correlations are then visualized using a heatmap. This graphical representation provides an immediate and intuitive visual understanding of the strength and direction of relationships between pairs of variables.
- Filtering with a Threshold Parameter: The heatmap includes a threshold parameter, allowing us to filter and display only those correlations that exceed a specific absolute value. This feature is particularly useful in highlighting the most significant and relevant relationships, aiding us in focusing our analysis on the most impactful connections among the financial indicators.

Through these methods, we are equipped to uncover intricate relationships within the financial data, enhancing our overall analysis and predictive modeling for crude oil prices and other related financial metrics.

4. Visualizing Model Training

a. Data Modeling for Predicting Crude Oil Price

The objective is to develop a predictive model for crude oil prices. The process involves several key steps to prepare, refine, and utilize the data effectively.

b. Preparing Data for the Prediction Model

i. Mapping Target Values:

We integrate the actual crude oil prices (labeled 'dcoilwtico') into our smoothed dataset (smooth_df).A dictionary is created from the original dataset (econ_df) containing crude oil prices and their corresponding dates.These values are then mapped onto smooth_df using dates as the index. This alignment is crucial for correlating our features with the correct target variable.

ii. Shifting Target Values:

The target variable 'dcoilwtico' is shifted backward by a predetermined window period.This shift aims to align the prediction with future values based on present and past data, ensuring that the features reflect the economic conditions relevant to the forecast period.

iii. Cleaning Up Data:

We employ dropna to eliminate rows with NaN values, which are a result of the shifting process.This cleanup is vital to maintain data integrity, ensuring that the dataset is devoid of missing values that could adversely affect the modeling process.

iv. Data Export

The processed dataset is saved as a CSV file. This step is a standard practice in data processing workflows to maintain data integrity and provide checkpoints.

c. Splitting and Scaling Data

i. Data Splitting:

The dataset is divided into training and testing sets using train_test_split from sklearn.model_selection.A specific proportion of the data, defined by train_size, is utilized for model training, while the remainder is set aside for testing.

ii. Feature Scaling:

StandardScaler from sklearn.preprocessing is applied to normalize the feature set.Feature scaling is crucial in many machine learning algorithms, particularly for optimizing the performance and convergence of algorithms like stochastic gradient descent.

d. *Final Test Set Split*:

The test set is further split into a validation set and a final test set.The validation set aids in fine-tuning the model during training and preventing overfitting.The final test set is reserved for an unbiased evaluation of the model's performance post-training.

e. Visualizing Model Training

i. Plot Training and Validation Loss:

The visualize_model_loss function plots the training and validation loss across each epoch.This visualization is key to identifying and addressing issues such as overfitting or underfitting, and it provides insights into the model's learning and convergence trends.These comprehensive steps form the foundation for building a robust and accurate predictive model for crude oil prices, ensuring a thorough understanding and application of machine learning techniques.

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
%matplotlib inline

# Define visualization columns
visualization_columns = [
    'bamlcc0a1aaatriv', 'bamlhyh0a0hym2triv', 'bogmbasew', 'dcoilwtico',
    'dexchus', 'dexjpus', 'dexusal', 'dexuseu', 'djia', 'goldamgbd228nlbm',
    'mhhngsp', 'nasdaqcom', 'ru2000pr', 'sp500', 'usd12md156n', 'usd1mtd156n',
    'usd3mtd156n', 'usdontd156n', 'vixcls', 'vxxlecls'
]
```

```
# Function to plot correlation heatmap
def plot_correlation_heatmap(data_frame, threshold=None, plot_title=''):
    corr_matrix = data_frame.corr(method='pearson')
    np.fill_diagonal(corr_matrix.values, 0)
    if threshold is not None:
        corr_matrix = corr_matrix[corr_matrix.abs() > threshold].fillna(0)
    plt.figure(figsize=(20, 15))
    sns.heatmap(corr_matrix, cmap='coolwarm', annot=False)
    plt.title(plot_title, fontsize=18)
    plt.show()
    return corr_matrix
```

```
# Set cutoff for correlation and visualize heatmap
cutoff_threshold = 0.3
plot_correlation_heatmap(smooth_df[visualization_columns], cutoff_threshold, 'Filtered Correlation Heatmap')

# Prepare the data for predicting crude oil prices
target_column = 'dcoilwtico'
smooth_df[target_column] = smooth_df.index.map(econ_df[target_column].to_dict())
smooth_df[target_column] = smooth_df[target_column].shift(periods=-window)
smooth_df.dropna(inplace=True)

# Export the processed data
filename = 'processed_data.csv'
smooth_df.to_csv(filename)
data['processed_data'] = smooth_df

# Data splitting and scaling
def prepare_data_for_modeling(data_frame, features, target, scale_columns=None, size=0.7):
    X = data_frame[features]
    y = data_frame[target]
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=size)
    scaler = StandardScaler()
    if scale_columns:
        X_train[scale_columns] = scaler.fit_transform(X_train[scale_columns])
        X_test[scale_columns] = scaler.transform(X_test[scale_columns])
    return X_train, y_train, X_test, y_test

# Identify columns to normalize and split the data
normalize_columns = [col for col in smooth_df if col not in onehot_cols + [target_column]]
feature_columns = [col for col in smooth_df if col != target_column]
X_train, y_train, X_test, y_test = prepare_data_for_modeling(smooth_df, feature_columns, target_column, normalize_columns)

# Further split for final evaluation
test_cutoff = len(X_test) // 2
X_validation, y_validation = X_test[:test_cutoff], y_test[:test_cutoff]
X_final_test, y_final_test = X_test[test_cutoff:], y_test[test_cutoff:]
```

```
# Define a function to plot model loss
def visualize_model_loss(history, plot_width=11):
    training_loss = history['loss']
    validation_loss = history['val_loss']
    epochs = range(1, len(training_loss) + 1)
    sns.set_style("white")
    plt.figure(figsize=(plot_width, 6))
    plt.plot(epochs, training_loss, 'g--', label='Training Loss')
    plt.plot(epochs, validation_loss, 'b-.', label='Validation Loss')
    plt.title('Training & Validation Loss Over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

5. Conducting a Grid Search for Neural Network Hyperparameter Optimization

a. Defining the Model Generation Function

We initiate our neural network hyperparameter optimization by defining a function create_nn_model. This function facilitates building a feedforward neural network with dense layers and enables experimentation with various hyperparameters, including the number of neurons per layer, the number of hidden layers, regularization strength, dropout rate, and more. While default settings like ReLU activation and mean squared error loss are pre-set, the function offers the flexibility to explore other activation functions, optimizers, and loss functions.

b. Setting Up the Hyperparameter Space

We establish a hyperparameter space, including lists or ranges of values for neurons (neurons_range), the number of dense layers (layers_range), dropout rates (dropout_options), and regularization strengths (reg_values). These parameters form the basis of our grid search to identify the optimal combination for our predictive model.

c. Performing the Grid Search

Through nested loops, we traverse each hyperparameter combination within our defined space. For each combination, we:

i. Build a new model with create_nn_model, utilizing the current hyperparameters.
ii. Implement an EarlyStopping callback with a patience setting of 1, halting training if there's no improvement in validation loss.
iii. Train the model on training data, monitoring the validation loss.
iv. Evaluate the model's performance on training, validation, and test datasets, computing the R-squared value for each set.
v. Saving and Analyzing Results

We record each model's performance and configuration in a dictionary results_summary, facilitating a post-analysis to discern the best models based on test R-squared values.

```python
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.regularizers import l1_l2
from keras.callbacks import EarlyStopping
from sklearn.metrics import r2_score

# Function to create neural network model
def create_nn_model(neurons, layers, reg_rate, dropout_rate):
    model = Sequential()
    model.add(Dense(neurons, activation='relu', input_shape=(X_train.shape[1],),
                    kernel_regularizer=l1_l2(l1=reg_rate, l2=reg_rate)))
    for _ in range(layers):
        model.add(Dense(neurons, activation='relu', kernel_regularizer=l1_l2(l1=reg_rate, l2=reg_rate)))
        model.add(Dropout(dropout_rate))
    model.add(Dense(1))
    model.compile(optimizer='rmsprop', loss='mse')
    return model

# Hyperparameter ranges
dropout_options = [0.0, 0.1, 0.2, 0.3]
neurons_range = [16, 22, 28, 34]
layers_range = [1, 2]
reg_values = [0.005, 0.001, 0.0005]

results_summary = {}

# Grid search implementation
for neurons in neurons_range:
    for layers in layers_range:
        for dropout_rate in dropout_options:
            for reg_rate in reg_values:
                model = create_nn_model(neurons, layers, reg_rate, dropout_rate)
                early_stopping = EarlyStopping(monitor='val_loss', patience=1)
                history = model.fit(X_train, y_train, epochs=1000, batch_size=X_train.shape[0] // 4,
                                    verbose=0, validation_data=(X_val, y_val), callbacks=[early_stopping])
                model_id = f"{neurons}_{layers}_{dropout_rate}_{reg_rate}"
                results_summary[model_id] = evaluate_model(model, X_train, y_train, X_val, y_val, X_test, y_test)
```

```python
# Function to evaluate the model
def evaluate_model(model, X_train, y_train, X_val, y_val, X_test, y_test):
    # Make predictions
    train_pred = model.predict(X_train)
    val_pred = model.predict(X_val)
    test_pred = model.predict(X_test)
    # Calculate R-squared for each set
    train_r2 = r2_score(y_train, train_pred)
    val_r2 = r2_score(y_val, val_pred)
    test_r2 = r2_score(y_test, test_pred)
    # Continue the function to evaluate the model
    return {
        'train_r2': train_r2,
        'val_r2': val_r2,
        'test_r2': test_r2,
        'history': history.history,
        'neurons': neurons,
        'layers': layers,
        'dropout': dropout_rate,
        'regularizer': reg_rate
    }
# Iterate over the results to print the performance
for model_id, metrics in results_summary.items():
    print(f"Model ID: {model_id}")
    print(f"Train R-squared: {metrics['train_r2']:.4f}")
    print(f"Validation R-squared: {metrics['val_r2']:.4f}")
    print(f"Test R-squared: {metrics['test_r2']:.4f}")
    print("-" * 80)

# Optionally, plot the loss curves for the best performing models
def plot_training_curves(history):
    plt.figure(figsize=(10, 5))
    plt.plot(history['loss'], label='Training Loss')
    plt.plot(history['val_loss'], label='Validation Loss')
    plt.title('Training vs Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

# Identify the best model based on validation R-squared
best_model_id = max(results_summary, key=lambda id: results_summary[id]['val_r2'])
best_model_metrics = results_summary[best_model_id]
print(f"Best Model ID: {best_model_id}")
print(f"Best Model Validation R-squared: {best_model_metrics['val_r2']:.4f}")

# Plotting the training and validation loss curves for the best model
plot_training_curves(best_model_metrics['history'])
```

6. Evaluate the Model

The evaluate_model function is responsible for calculating the R-squared metric for the training, validation, and test predictions. This allows us to assess the model's performance comprehensively. It returns a dictionary containing these metrics as well as the training history.

a. Print Model Performances

In this section of the script, we iterate over the results_summary dictionary and print out the R-squared values for each model. This provides a quick overview of how well each model has performed.

b. Plot Training Curves

For visual examination of the training process, the plot_training_curves function is available. It creates plots showing the loss evolution across epochs for both the training and validation datasets. If the validation loss starts to increase while the training loss is still decreasing, it may indicate overfitting.

c. Select the Best Model

The best model is determined by identifying the model ID with the highest validation R-squared value within the results_summary dictionary. This step is crucial for understanding which combination of hyperparameters delivers the most promising results on unseen data.

d. Visualize the Best Model's Performance

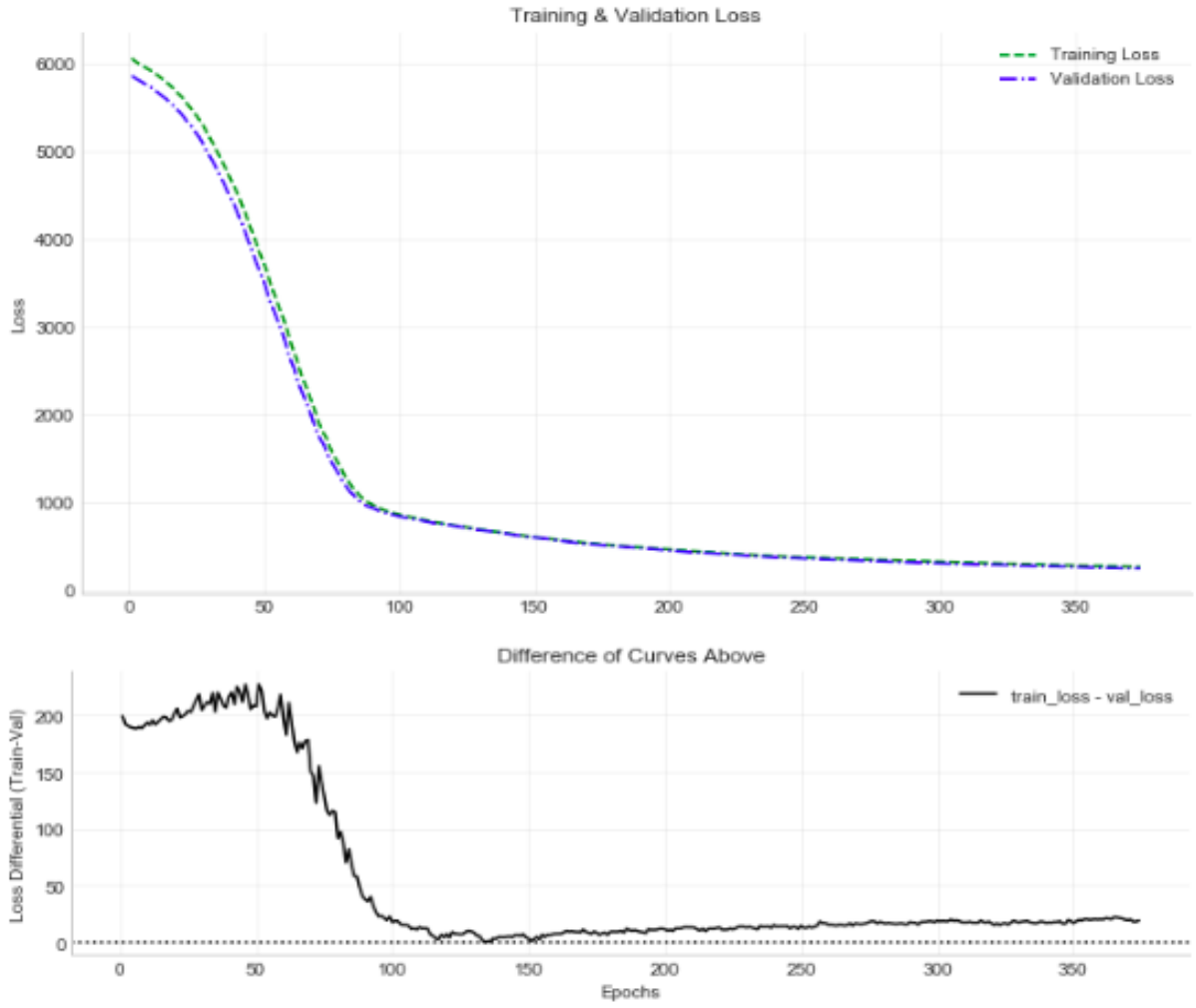
Lastly, we plot the training and validation loss curves of the best model. This visual assessment aids in confirming the model's capacity to learn from the training data effectively, without overfitting to it.

```
i. Model # 1
```

Fully Connected Model w/ Dropout & Regularization

- Regularizer Rate: 0.0050000
- Dropout Rate: 0.000
- Number Dense Layers: 1
- Neurons per Layer: 16

R-squared on training data = 0.8849
R-squared on validation data = 0.8662
R-squared on testing data = 0.8526



ii. Model # 2

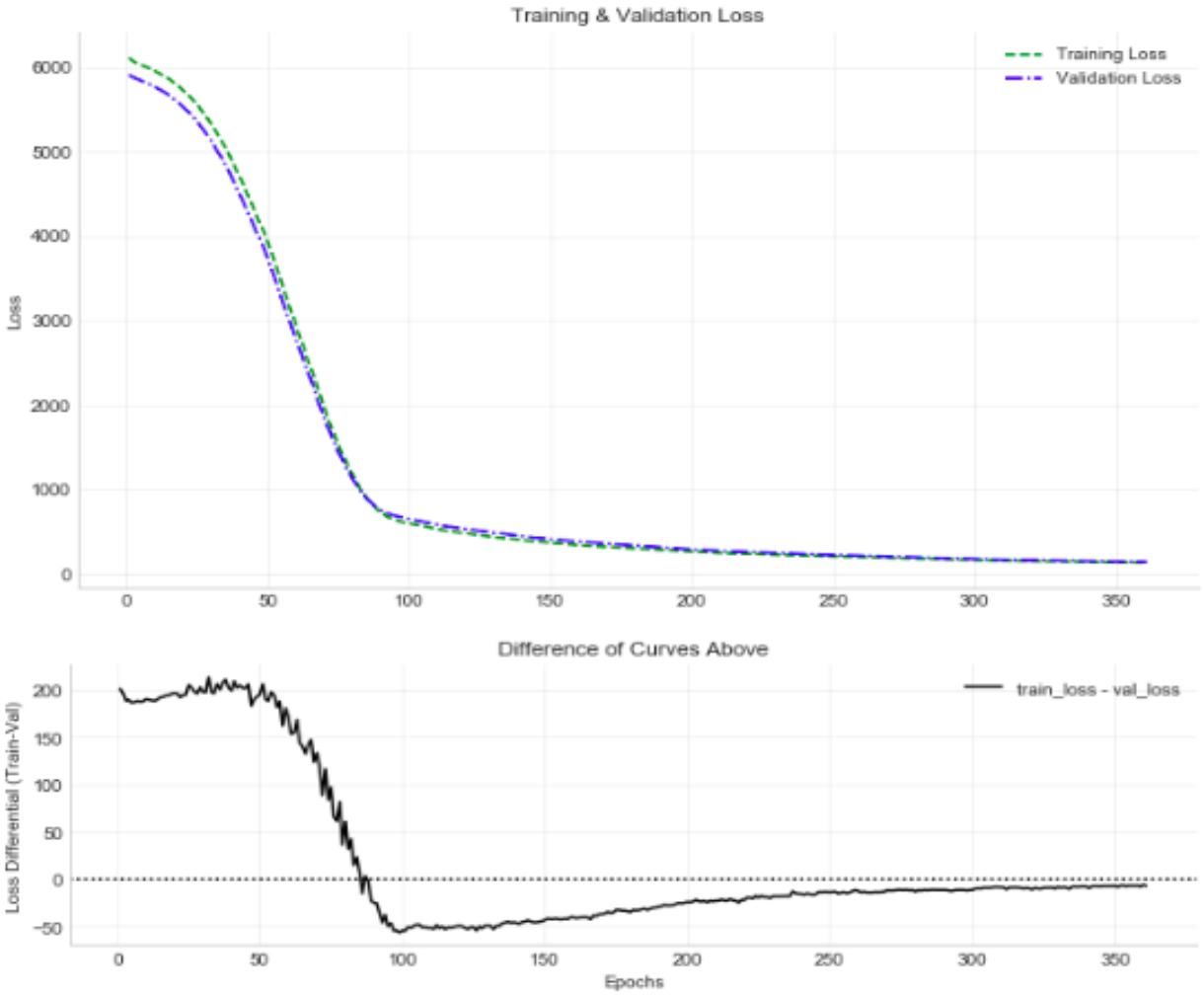
Fully Connected Model w/ Dropout & Regularization

- Regularizer Rate: 0.0010000
- Dropout Rate: 0.000
- Number Dense Layers: 1
- Neurons per Layer: 16

R-squared on training data = 0.8797

R-squared on validation data = 0.8545

R-squared on testing data = 0.8506

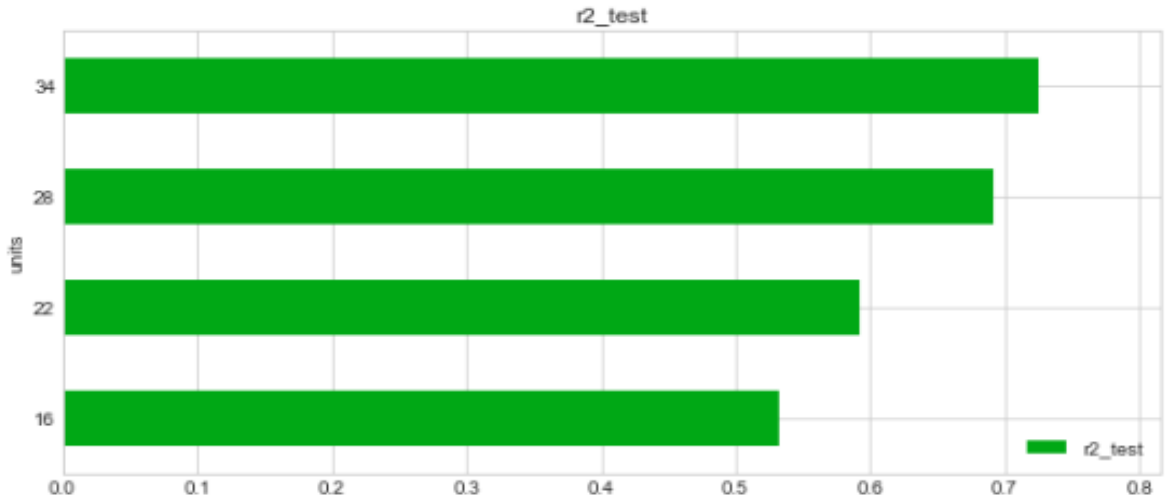


7. Plot the Train, Test and Validation r2_score

In this section, we focus on visualizing the R-squared values obtained from our model evaluations. We begin by converting the R-squared values to floating-point numbers for better precision during subsequent calculations.

We then group the results by the number of neurons per layer and plot the average R-squared values for the training, validation, and test sets. To facilitate comparison, we use a horizontal bar chart. This visualization helps us understand how the number of units impacts the model's performance across different datasets, providing insights into the model's sensitivity to this hyperparameter.

```
# Visualization and Analysis of Model Performance

# Visualizing R-squared Values

# To ensure precision, convert the R-squared values to floating-point numbers
results_df[r2_cols] = results_df[r2_cols].astype(np.float32)

# Group the results by the number of units and plot their average R-squared values
units_performance = results_df.groupby(['neurons'])[r2_cols].mean()

# Create a horizontal bar chart to compare the model's performance based on the number of units
units_performance.plot(kind='barh', subplots=True, figsize=(10, 15), color=['#1f77b4', '#ff7f0e', '#2ca02c'])
plt.tight_layout()
plt.show()
```



8. Selecting the Optimum Model

We focus on selecting the best-performing model based on the highest R-squared value achieved on the test set. We utilize the idxmax method to identify the index of this optimal model within our results.Once the best model is identified, we save it to a file (optimal_model.h5) for future use. This ensures that the effort put into training the best model does not need to be repeated.Additionally, we visually assess the model's performance by plotting actual vs. predicted values, providing an intuitive understanding of how well the model generalizes to unseen data.

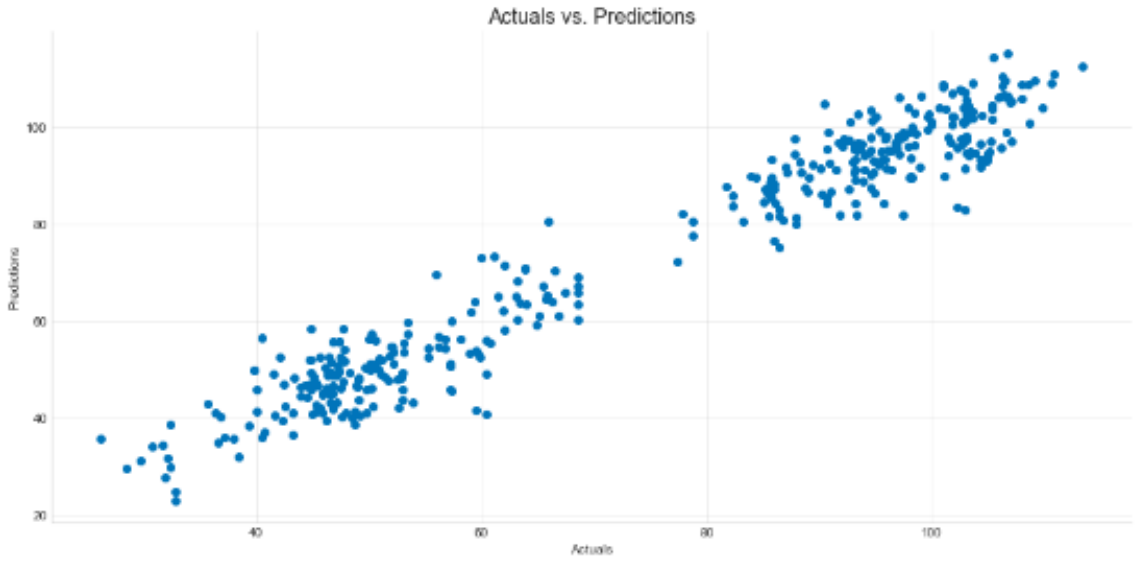
```
# Selecting the Optimum Model

# To identify the model with the highest test R-squared value, we use the idxmax method
optimal_model_index = results_df['r2_test'].idxmax()

# Save the best-performing model to a file for future use, avoiding the need for retraining
optimal_model = models_archive[optimal_model_index]['model']
optimal_model.save('optimal_model.h5')

# Visualize model performance by plotting actual vs. predicted values
predictions = optimal_model.predict(X_test)
plt.figure(figsize=(15, 7))
plt.scatter(y_test, predictions, edgecolors='k', alpha=0.7)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Comparison of Actual and Predicted Values', fontsize=16)
sns.despine()
plt.show()
```

units	34
layers	2
dropout	0
regularizer	0.005
r2_train	0.956788
r2_val	0.938761
r2_test	0.945236

9. Examining the Best Model

We take a closer look at the best model we've selected. We start by plotting its training history to gain insights into its performance over epochs and save this plot as 'optimal_model_training_history.png'. Next, we print a summary of the best model to understand its architectural structure and the number of parameters it comprises. Lastly, we display specific information about the best model's hyperparameters and performance metrics from the results_df dataframe, helping us understand the configuration and results associated with the chosen model

```
# Examining the Best Model

# We delve deeper into the architecture and performance of the best model

# Plot the training history of the best model
optimal_model_history = models_archive[optimal_model_index]['history']
plot_training_history(optimal_model_history)
plt.savefig('optimal_model_training_history.png')

# Print a summary of the best model to understand its structure
print("Best Model Summary:")
optimal_model.summary()

# Display the best model's hyperparameters and performance metrics
optimal_model_info = results_df.loc[optimal_model_index]
print("Best Model Info:")
print(optimal_model_info)
```



Upon examination, we have identified the best-performing model configuration, which consists of 22 units per layer, two layers (excluding the input layer), no dropout, and an l1/l2 regularizes set at 0.005. However, it is noteworthy that the R-squared score for all datasets is exceptionally high. This level of performance should raise suspicions and prompt a deeper investigation to ensure the model's reliability and generalization capability.

10. Prediction Generation and Evaluation

a. Data Preprocessing Function

We define a function called prepare_data_for_prediction, which takes a DataFrame along with the names of feature columns, target columns, and any columns that require normalization. This function standardizes the feature data, aligning it with the preprocessing steps expected by our model.

```
def prepare_data_for_prediction(data, feature_names, target_name, columns_to normalize=None):
    # Extract features and target
    features = data[feature_names]
    target = data[target_name]

    # Initialize standard scaler
    scaler = StandardScaler()

    # Normalize features if specified
    if columns_to_normalize:
        features[columns_to_normalize] = scaler.fit_transform(features[columns_to_normalize])
    else:
        features = pd.DataFrame(scaler.fit_transform(features), columns=features.columns)

    # Return the prepared features and target
    return features, target
```

b. Prepare the data for prediction

```
# Prepare the data for prediction
prepared_X, prepared_Y = prepare_data_for_prediction(smoothed_dataframe, feature_columns, target_column, columns_to_normalize)
```

c. Visualization of Predictions

We generate predictions for the entire dataset using our optimal model and visualize these predictions alongside the actual target values. Additionally, we plot the prediction errors to assess where the model's predictions diverge from reality.

```
# Generate predictions
predictions_array = optimal_model.predict(prepared_X.to_numpy()).flatten()
predictions_series = pd.Series(predictions_array, index=pd.to_datetime(prepared_X.index))

# Create a DataFrame to compare predictions with actuals
comparison_df = pd.DataFrame({'predicted': predictions_series, 'actual': prepared_Y}).dropna()

# Plotting the predictions and actuals
plt.figure(figsize=(15, 7))
plt.plot(comparison_df.index, comparison_df['predicted'], label='Predicted', linestyle=':', color='green', alpha=0.9)
plt.plot(comparison_df.index, comparison_df['actual'], label='Actual', color='black', alpha=0.7, linewidth=0.5)
plt.xlabel('Date')
plt.ylabel('Crude Oil Price')
plt.title('Comparison of Predicted and Actual Crude Oil Prices', fontsize=16)
plt.legend()
plt.grid(visible=True, which='both', linestyle='--', linewidth=0.5)
sns.despine()
plt.show()

# Plotting the prediction errors
plt.figure(figsize=(15, 3))
prediction_errors = comparison_df['predicted'] - comparison_df['actual']
plt.plot(comparison_df.index, prediction_errors, label='Prediction Error', linestyle='--', color='red', alpha=0.5)
plt.xlabel('Date')
plt.ylabel('Prediction Error')
plt.title('Prediction Errors Over Time', fontsize=16)
plt.legend()
plt.grid(visible=True, which='both', linestyle='--', linewidth=0.5)
sns.despine()
plt.show()
```

In this section, we preprocess the data for prediction, generate predictions using our best model, and visualize the model's performance by comparing the predicted values with actual values. We also plot the prediction errors to gain insights into where the model's predictions deviate from the actual data.

	pred	dcoilwtico
2018-04-04	58.768600	68.56
2018-04-05	57.743031	68.56
2018-04-06	60.051060	68.56
2018-04-07	60.604538	68.56
2018-04-08	61.832554	68.56

d. Calculate R-score

This code calculates and prints the R-squared value for the entire dataset, providing a measure of the model's overall performance. R-squared on entire dataset: 0.9540

```
# Calculating and printing the R-squared value for the final predictions
final_r2_score = r2_score(comparison_df['actual'], comparison_df['predicted'])
print(f"R-squared for the entire dataset: {final_r2_score:.4f}")
```

Conclusion:

This Project Report on "Crude Oil Price Forecasting" is a comprehensive study that integrates various data sources such as commodity prices, debt market indicators, energy-related series, and traditional currencies to develop a machine learning model for predicting crude oil prices. The report details extensive data preprocessing techniques, including handling missing values, generating one-hot encoded features, and transforming column names. Advanced feature engineering strategies are employed to enhance data signal and reduce noise, followed by sophisticated data visualization techniques to analyze trends and relationships.

The project culminates in building a predictive model using neural network hyperparameter optimization, evaluating the model's performance using R-squared metrics, and selecting the best model based on test set performance. The final model's architecture and performance metrics are scrutinized, and its reliability and generalization capability are critically evaluated. The report concludes with the generation and evaluation of predictions, showcasing the effectiveness of the model in forecasting crude oil prices and underscoring the importance of thorough data processing and model evaluation in predictive analytics