d214

August 23, 2023

1 D214 - Data Analytics Graduate Capstone

1.0.1 Task 2: Data Analytics Report

1.0.2 A: Research Question

In the international finance industry, the dynamics of the foreign exchange market (FX) pose a great challenge to firms operating globally and has become an integral factor in the financial decision-making process. The volatility associated with FX rates can have a significant impact on the profitability of a firm. Having a reliable, reasonably accurate forecast of a given currency rate can provide a significant advantage to a firm in terms of planning and decision-making. Moreover, the ability to forecast FX rates may be hold significant value in a company's risk management strategy.

The purpose of this study is to answer the proposed research question: > Is the `ARIMA` time series forecasting model capable of accurately predicting future foreign exchange rates?

The model will be evaluated based on its efficacy in predicting the future rate of several characteristically different currency rates. The `ARIMA` model will be trained on historical time series data and then tested on a holdout sample. The model will then be evaluated based on its ability to accurately predict the future rate of the currencies in the test set.

Multinational corporations often face FX risk as a consequence of operating across multiple countries and dealing with various currencies. Many factors contribute to the composition and extent of a firm's FX risk exposure, but ultimately, it is the volatility and movement of the currencies in a firm's portfolio that most directly impact profitability, competitiveness, and overall financial stability. Considering the multitude of available time series forecasting models, it seems appropriate to assess their predictive accuracy, particularly with respect to FX rates, so leaders are enabled to make data-driven decisions. By accurately predicting future FX rates, firms can optimize hedging strategies, enhance financial planning, and potentially realize significant cost savings. This, in turn, facilitates more strategic decision-making and bolsters the firm's ability to safeguard against unforeseen financial risks.

In this study, the effectiveness of the `ARIMA` model is tested against two specific hypotheses. The hypotheses are as follows:

- Null hypothesis:
  - The mean average percentage error (`MAPE`) of the `ARIMA` model as applied to a 90-day forecast of future foreign exchange rates is greater than 20%

- Alternate Hypothesis:
  - The mean average percentage error (`MAPE`) of the `ARIMA` model as applied to a 90-day forecast of future foreign exchange rates is less than 20%

The null hypothesis predicts a mean average percentage error (`MAPE`) of more than 20% for the 90-day forecast, which would suggest inadequate precision. The alternate hypothesis anticipates a `MAPE` of less than 20%, reflecting a more favorable prediction accuracy. These hypotheses set the stage for the empirical testing that follows, enabling a detailed analysis of the `ARIMA` model's practical applicability in FX rate forecasting.

1.0.3 B: Data Collection

This analysis will utilize a basket of daily FX spot rates for currencies against the US Dollar. As the behavior of FX rates varies somewhat dramatically depending on the pair, it's important to evaluate model accuracy using currencies with significantly different macroeconomics.

The Federal Reserve Economic Data (`FRED`) Daily Exchange Rates datasets were used for the purposes of this analysis. Specifically, the `FRED` website refers to these rates as "H.10". These data are highly reliable, widely used, and readily available records of daily FX spot rates for several currency pairs and dating back many years in most cases so as to provide a sufficient amount of data for training and testing.

The Federal Reserve Bank of St. Louis owns the `FRED` data, which is publicly accessible for research and educational purposes. `FRED` permits the use of this data for academic research so long as the user cites `FRED` and provides a note stating where the data was obtained (as well as any copyright notices that may appear in the data).

In the data-collection process for the following research, a wide range of time series data was collected for six currencies: `GBP`, `CAD`, `CNY`, `JPY`, `INR`, and `ZAR`, all against the US Dollar. This data was obtained for a 40-year period from January 1, 1983, to December 31, 2022, from the `FRED` database.

The data collected included daily foreign exchange rates for the specified currency pairs. The selected range allowed for an extensive historical analysis, with each record containing the date and the closing, mid-point exchange rate value for that day. The data was transformed into a format suitable for time-series analysis, including handling missing values and ensuring consistency across the different currency pairs.

The data collection method made use of a helpful API provided courtesy of the Federal Reserve's website. This programmatic access offered a significant advantage in terms of automation and precision. By utilizing a well-documented API, the process ensured that the collected data was consistent, up-to-date, and aligned with the specific requirements of the research question.

One limitation of this method was the reliance on the external API. This dependency on third-party services could lead to challenges such as unexpected changes in the API's structure, limitations on request frequency, or unavailability of specific series, potentially hindering the data collection process. `FRED` has been providing this API for many years, and it is widely used, so it is unlikely that any significant changes will occur. However, it is important to recognize the potential for such issues and to have a contingency plan in place to mitigate any adverse effects.

Several challenges were encountered during the data collection process, particularly related to error handling and missing data. Robust error handling techniques were implemented to catch any unexpected errors during data retrieval, allowing for informative error messages that assisted in diagnosing issues. As the H10 (daily currency rates) data handles missing values simply by leaving the value as a `'.'` (period), it was necessary to replace these values with a `NaN` value. This was achieved by using the pandas library to replace such values as the data was gathered.

The data-collection process was a critical first step in the research. As such, it was carried out with careful consideration of the advantages and potential pitfalls associated with the chosen methodology. By recognizing and overcoming the inherent challenges, the process successfully laid the groundwork for subsequent stages of the study, ensuring a comprehensive and reliable dataset for foreign exchange rate forecasting.

1.0.4 C: Data Extraction and Preparation

The data-extraction process was implemented using Python's `requests` library to fetch data from the `FRED` database's API. The method employed involved constructing a specific URL and making an HTTP request to retrieve the relevant data series, as demonstrated in the following code snippet:

[ ]:
```
# Import libraries
import requests, numpy as np, pandas as pd, datetime as dt
from plotly.subplots import make_subplots
import plotly.graph_objects as go
from statsmodels.tsa.seasonal import seasonal_decompose
from pmdarima import auto_arima
import concurrent.futures
from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error

# Set constants
API_KEY = 'fbf2a3cac76ec733ee2b8c01ab036950'
URL_BASE = 'https://api.stlouisfed.org/fred/series/observations'
START = pd.Timestamp('1983-01-01').date()
END = pd.Timestamp('2022-12-31').date()
BUSDAYS_IN_RANGE = np.busday_count(START, END)
META_INDEX = ['observation_start', 'observation_end', 'busdays_in_range',␣
↪'actual_start', 'actual_end', 'actual_days', 'nan_count']
SERIES_LIST = ['DEXUSUK', 'DEXCAUS', 'DEXCHUS', 'DEXJPUS', 'DEXINUS', 'DEXSFUS']
CCY_LIST = ['GBP', 'CAD', 'CNY', 'JPY', 'INR', 'ZAR']
```

After importing the necessary packages, a series of constant variables are defined to specify values such as the API key, the base URL, etc. This is where we specify the currency pairs we will retrieve data for, as well as the start and end dates of the time series. We'll refer back to these variables later in the code.

The next steps include constructing the URL and making the HTTP request. The `requests` library is used in conjunction with some custom functions I created to handle the request and response. This was done to provide a structure for potential repeated requests and to handle any errors that may occur.

[ ]:
```
# Create function to get raw json return from FRED database
```

```
def get_series_json(series_id, start, end, api_key=API_KEY, file_type='json',␣
↪url_base=URL_BASE):
    url = f'{url_base}?
↪series_id={series_id}&observation_start={start}&observation_end={end}'
    url += f'&api_key={api_key}&file_type={file_type}'
    try:
        resp = requests.get(url)
        resp.raise_for_status()  # Raise exception if invalid response
        return resp
    except Exception as e:
        errmsg = resp.json()['error_message'].replace('series', f'series␣
↪{series_id}')
        print(f'Error: {resp.status_code}\n{errmsg}')
        return None
```

This approach allowed for automation, enabling easy and consistent extraction of data and ensuring accurate and up-to-date information. Additionally, it allowed for other steps in the data retrieval, transformation, and cleaning process to be more automated and streamlined.

Once the data was extracted, the next phase involved its transformation and preparation. This process was carried out using the `pandas` and `numpy` libraries, allowing for reasonably seamless transformation and cleaning. Once again, a handful of custom funtions were written to handle the specific characteristics of the `FRED` daily currency rate time series data. The extracted data was transformed into a `pandas` DataFrame, and any missing values were handled through forward and backward filling, as illustrated by the following code:

[ ]:
```
# Create function to transform valid json response from FRED into a dataframe
def transform_series_json(resp, series_id):
    resp = resp.json()
    obs = pd.DataFrame(resp.pop('observations'))[['date', 'value']]
    obs['date'] = pd.to_datetime(obs['date'])
    obs.set_index('date', inplace=True)
    meta = pd.DataFrame({
        series_id: {'observation_start': resp['observation_start'],
            'observation_end': resp['observation_end'],
            'busdays_in_range': BUSDAYS_IN_RANGE,
            'actual_days': resp['count'],
            'actual_start': obs.index.min().date(),
            'actual_end': obs.index.max().date(),
            'nan_count': obs[obs.value == '.'].count().value}})
    meta = meta.reindex(META_INDEX)
    obs.loc[obs.value == '.'] = np.nan
    obs.columns = [series_id]
    obs[series_id] = obs[series_id].astype(float, errors='raise')
    return obs, meta

# Create function to fill missing values in FRED series datafame
def fill_series_na(df):
```

```python
    df.fillna(method='ffill', inplace=True)  # Fill missing values with last␣
↪observation
    df.fillna(method='bfill', inplace=True)  # Then, fill with next observation
    return df

# Create a function to get a time series from FRED and return a clean dataframe
def get_series(series_id, start, end, api_key=API_KEY, file_type='json',␣
↪fill_na=None):
    fill_na = True if fill_na is None else fill_na  # Default
    try:
        resp = get_series_json(series_id=series_id, start=start, end=end,␣
↪api_key=api_key, file_type=file_type)
        df, meta = transform_series_json(resp, series_id=series_id)
        df = fill_series_na(df) if fill_na else df
    except Exception as e:
        print(f'Error retrieving {series_id}.\n{e}')
        return None
    return df, meta

# Convenience function to get multiple series at once
def get_multiple_series(series_list, start, end, fill_na=None):
    fill_na = True if fill_na is None else fill_na  # Default
    df_list = []
    meta_list = []
    for series in series_list:
        df, meta = get_series(series_id=series, start=start, end=end)
        df_list.append(df)
        meta_list.append(meta)
    dfs = pd.concat(df_list, axis=1)
    metas = pd.concat(meta_list, axis=1)

    print(f'\nDownloaded {len(df_list)} / {len(series_list)} series')
    print(f'\nMeta Info on downloaded series: \n{metas.to_markdown()}')
    print(f'\nCombined series dataframe: \n{dfs.set_index(dfs.index.date).
↪head().to_markdown()}')
    return dfs, metas
```

The choice of these libraries offered flexibility and efficiency, enhancing performance and speeding up the data preparation process. They were instrumental in moving from raw data to a clean and structured format suitable for analysis. Nonetheless, this process required careful attention to the unique characteristics of the data, adding a layer of complexity to the preparation phase.

As was mentioned in section B, the `FRED` daily currency rate data imputes missing values as a period (`'.'`) in their raw data. This presented a minor challenge in terms of handling these values, as it is a somewhat unconventionional way to represent `null` values. However, this was fairly easily overcome by using the `pandas` library's `fillna()` method to replace these values with `NaN` values. This was done so that the `pandas` library could then handle the missing values using forward and backward filling with the next and previous values non-null values, respectively.

The combination of Python libraries such as `requests`, `pandas`, and `numpy` was integral to the data extraction and preparation processes, offering the distinct advantage of a blend of automation, accuracy, adaptability, and efficiency. These tools facilitated a smooth transition from raw data to a format conducive to analysis, notwithstanding the need to manage the complexity of the data and dependencies on external services. The methodology adopted in this analysis endeavored to embody a practical approach to handling large-scale data, underlining the efficiency and flexibility of the `Python` programming language.

One disadvantage of this approach was the additional time required to write custom functions to handle the data. The above code was not strictly necessary for the data extraction and preparation process. However, it appears to have been an ostensibly useful and efficient approach, particularly with regard to the potential for future use. In performing analysis such as the analysis in this study, it can certainly be beneficial to invest the time and effort when a new data pipeline is being established so as to pave the way for future projects if such is the case.

[ ]:
```
dfs, metas = get_multiple_series(series_list=SERIES_LIST, start=START, end=END,
    fill_na=True)
```

```
Downloaded 6 / 6 series

Meta Info on downloaded series:
|                   | DEXUSUK    | DEXCAUS    | DEXCHUS    | DEXJPUS    |
DEXINUS    | DEXSFUS    |
|:------------------|:-----------|:-----------|:-----------|:-----------|:------
-----|:-----------|
| observation_start | 1983-01-01 | 1983-01-01 | 1983-01-01 | 1983-01-01 |
1983-01-01 | 1983-01-01 |
| observation_end   | 2022-12-31 | 2022-12-31 | 2022-12-31 | 2022-12-31 |
2022-12-31 | 2022-12-31 |
| busdays_in_range  | 10435      | 10435      | 10435      | 10435      | 10435
| 10435      |
| actual_start      | 1983-01-03 | 1983-01-03 | 1983-01-03 | 1983-01-03 |
1983-01-03 | 1983-01-03 |
| actual_end        | 2022-12-30 | 2022-12-30 | 2022-12-30 | 2022-12-30 |
2022-12-30 | 2022-12-30 |
| actual_days       | 10435      | 10435      | 10435      | 10435      | 10435
| 10435      |
| nan_count         | 395        | 395        | 456        | 395        | 403
| 404        |

Combined series dataframe:
|            |   DEXUSUK |   DEXCAUS |   DEXCHUS |   DEXJPUS |   DEXINUS |
DEXSFUS |
|:-----------|----------:|----------:|----------:|----------:|----------:|------
----:|
| 1983-01-03 |    1.6235 |    1.23   |    1.9275 |     232   |      9.62 |
1.0695 |
| 1983-01-04 |    1.621  |    1.2298 |    1.914  |     229.8 |      9.64 |
```

```
1.0667 |
| 1983-01-05 |     1.621  |     1.2297 |     1.914  |      229.1 |       9.64 |
1.0684 |
| 1983-01-06 |     1.6065 |     1.2313 |     1.9044 |      229.8 |       9.7  |
1.0712 |
| 1983-01-07 |     1.61   |     1.2267 |     1.9044 |      229.1 |       9.73 |
1.0712 |
```

As an example what the data retrieval process is doing "under the hood" so to speak, the below will show the `GBP/USD` currency pair data for the 2022 year in its raw form and each stage of the data preparation process:

[ ]:
```
# Retrieve raw data from FRED
example = get_series_json(series_id='DEXUSUK', start='2022-01-01',␣
 ↪end='2022-12-31')
print(pd.DataFrame(example.json()).head().to_markdown(index=False))
```

```
| realtime_start   | realtime_end   | observation_start   | observation_end   |
units    |   output_type | file_type    | order_by         | sort_order   |
count |    offset |    limit | observations
|
|:-----------------|:---------------|:--------------------|:------------------
|:--------|--------------:|:------------|:-----------------|:-------------|-----
---:|---------:|--------:|:----------------------------------------------------
---------------------------------------------------|
| 2023-08-23       | 2023-08-23     | 2022-01-01          | 2022-12-31        |
lin     |             1 | json        | observation_date | asc          |
260 |         0 |   100000 | {'realtime_start': '2023-08-23', 'realtime_end':
'2023-08-23', 'date': '2022-01-03', 'value': '1.3469'} |
| 2023-08-23       | 2023-08-23     | 2022-01-01          | 2022-12-31        |
lin     |             1 | json        | observation_date | asc          |
260 |         0 |   100000 | {'realtime_start': '2023-08-23', 'realtime_end':
'2023-08-23', 'date': '2022-01-04', 'value': '1.3544'} |
| 2023-08-23       | 2023-08-23     | 2022-01-01          | 2022-12-31        |
lin     |             1 | json        | observation_date | asc          |
260 |         0 |   100000 | {'realtime_start': '2023-08-23', 'realtime_end':
'2023-08-23', 'date': '2022-01-05', 'value': '1.3573'} |
| 2023-08-23       | 2023-08-23     | 2022-01-01          | 2022-12-31        |
lin     |             1 | json        | observation_date | asc          |
260 |         0 |   100000 | {'realtime_start': '2023-08-23', 'realtime_end':
'2023-08-23', 'date': '2022-01-06', 'value': '1.3539'} |
| 2023-08-23       | 2023-08-23     | 2022-01-01          | 2022-12-31        |
lin     |             1 | json        | observation_date | asc          |
260 |         0 |   100000 | {'realtime_start': '2023-08-23', 'realtime_end':
'2023-08-23', 'date': '2022-01-07', 'value': '1.3583'} |
```

[ ]:
```
# Show raw data missing value example
obs = pd.DataFrame(example.json().pop('observations'))[['date', 'value']]
```

```python
print(obs.head(15).to_markdown(index=False))
```

```
| date       | value   |
|:-----------|:--------|
| 2022-01-03 | 1.3469  |
| 2022-01-04 | 1.3544  |
| 2022-01-05 | 1.3573  |
| 2022-01-06 | 1.3539  |
| 2022-01-07 | 1.3583  |
| 2022-01-10 | 1.3567  |
| 2022-01-11 | 1.3622  |
| 2022-01-12 | 1.3698  |
| 2022-01-13 | 1.3724  |
| 2022-01-14 | 1.367   |
| 2022-01-17 | .       |
| 2022-01-18 | 1.3588  |
| 2022-01-19 | 1.3625  |
| 2022-01-20 | 1.3642  |
| 2022-01-21 | 1.3562  |
```

[ ]:
```python
# Next step in data cleaning is to transform the raw json into a dataframe and metadata
example, meta = transform_series_json(example, series_id='DEXUSUK')
print(example.head(15).to_markdown(index=False))
print(meta.to_markdown())
```

```
|   DEXUSUK |
|----------:|
|    1.3469 |
|    1.3544 |
|    1.3573 |
|    1.3539 |
|    1.3583 |
|    1.3567 |
|    1.3622 |
|    1.3698 |
|    1.3724 |
|    1.367  |
|  nan      |
|    1.3588 |
|    1.3625 |
|    1.3642 |
|    1.3562 |
|                   | DEXUSUK    |
|:------------------|:-----------|
| observation_start | 2022-01-01 |
| observation_end   | 2022-12-31 |
| busdays_in_range  | 10435      |
```

```
| actual_start       | 2022-01-03 |
| actual_end         | 2022-12-30 |
| actual_days        | 260        |
| nan_count          | 10         |
```

[ ]:
```python
# Fill the NaN values in the dataframe
example = fill_series_na(example)
print(example.head(15).to_markdown(index=False))
```

```
|   DEXUSUK |
|----------:|
|    1.3469 |
|    1.3544 |
|    1.3573 |
|    1.3539 |
|    1.3583 |
|    1.3567 |
|    1.3622 |
|    1.3698 |
|    1.3724 |
|    1.367  |
|    1.367  |
|    1.3588 |
|    1.3625 |
|    1.3642 |
|    1.3562 |
```

The `get_multiple_series()` function combines all of these steps into a single function, allowing for easy retrieval, cleaning, and transforming of several currency pairs at once. This function is used to retrieve the data for all six currency pairs, as seen above in the initial example. Now that the data has been retrieved and prepared, it is ready for analysis.

1.0.5 D: Analysis

As the data has been prepared, it is now ready for some exploratory analysis prior to the modeling phase. The first step is to visualize the data to get a sense of the trends and patterns in the data. We will be using the `plotly` library to create interactive plots that allow for easy exploration of the data. The following code snippet shows the `plotly` code used to create the interactive plot:

[ ]:
```python
# Keep a copy of the original dataframe and create rates dataframe, align, and␣
↪set column names
# to easier to read currency codes
rates = dfs.copy()
rates.columns = CCY_LIST
rates.index.freq = 'B'
aligned = dfs.copy()
aligned.iloc[:, 1:] = aligned.iloc[:, 1:].rdiv(1)
aligned.head()
```

[ ]:

```
            DEXUSUK   DEXCAUS   DEXCHUS   DEXJPUS   DEXINUS   DEXSFUS
date
1983-01-03   1.6235  0.813008  0.518807  0.004310  0.103950  0.935016
1983-01-04   1.6210  0.813140  0.522466  0.004352  0.103734  0.937471
1983-01-05   1.6210  0.813206  0.522466  0.004365  0.103734  0.935979
1983-01-06   1.6065  0.812150  0.525100  0.004352  0.103093  0.933532
1983-01-07   1.6100  0.815195  0.525100  0.004365  0.102775  0.933532
```

[ ]:

```
layout = {'title': '<b>Currency 40-Year Daily Rates</b><br><sup><i>(1983-2022)</
 ↪i></sup>',
          'width': 1800,
          'height': 1000,
          'template': 'seaborn',
          'hovermode': 'x unified'}

def plot_all_rates(df, layout, x_title, y_title, ht, yshared=False):
    fig = make_subplots(rows=2, cols=3, shared_xaxes=True, vertical_spacing=0.
 ↪05, horizontal_spacing=0.02,
                        subplot_titles=([ccy for ccy in df]),␣
 ↪shared_yaxes=yshared, x_title=x_title, y_title=y_title)
    for i, ccy in enumerate(df):
        trace = go.Scatter(x=df.index, y=df[ccy], mode='lines', name=ccy,␣
 ↪hovertemplate=ht)
        if i // 3 < 1:
            fig.add_trace(trace, row=1, col=i+1)
        else:
            fig.add_trace(trace, row=2, col=i-2)
    fig.update_layout(layout)
    return fig

x_title = 'Date 1983-2022'
y_title = 'Daily Rate Against US Dollar<br><sup><i>Except in the case of GBP␣
 ↪which is reverse</i></sup>'
rate_fig = plot_all_rates(df=rates, layout=layout, x_title=x_title,␣
 ↪y_title=y_title, ht='%{y:,.4f}')
rate_fig.show()
```

One perhaps subtle additional transformation performed in the above example is the inversion of one of the currency pairs (`GBP/USD`). This was done to make the visualization more intuitive, as the market quote convention of `GBP/USD` which many FX data sources follow, just as what we retrieved from the `FRED` API, is counter to the rest of the rates which are quoted as `USD/{CCY}`. This is a minor detail, but it is important to note that the data was transformed in this way to align the directionality of the charts. Put simply, what we are seeing in the above chart is the daily rate of exchange for each currency against the `USD` where a higher value indicates a stronger `USD` currency (weakening foreign currency) and a lower value indicates a stronger foreign currency (weakening `USD`).

```python
def plot_all_hist(df, layout, x_title, y_title, ht, yshared=False):
    fig = make_subplots(rows=2, cols=3, shared_xaxes=False, vertical_spacing=0.
↪05, horizontal_spacing=0.02,
                        subplot_titles=([ccy for ccy in df]),␣
↪shared_yaxes=yshared, x_title=x_title, y_title=y_title)
    for i, ccy in enumerate(df):
        trace = go.Histogram(x=rates[ccy], name=ccy, nbinsx=25)
        if i // 3 < 1:
            fig.add_trace(trace, row=1, col=i+1)
        else:
            fig.add_trace(trace, row=2, col=i-2)
    fig.update_layout(layout)
    return fig

layout['title'] = '<b>Currency 40-Year Daily Rates Histogram</
↪b><br><sup><i>(1983-2022)</i></sup>'
x_title = 'Date 1983-2022'
y_title = 'Daily Rate Against US Dollar<br><sup><i>Except in the case of GBP␣
↪which is reverse</i></sup>'
hist_fig = plot_all_hist(df=rates, layout=layout, x_title=x_title,␣
↪y_title=y_title, ht='%{y}')
hist_fig.show()
```

This view of the data provides a useful starting point for the analysis, allowing for a visual inspection of the data and the identification of any trends or patterns. The interactive nature of the plot allows for interaction with the data and details on specific data points, enhancing the exploratory analysis process. The histogram view of the data provides an insight into the distribution of the data.

```python
aligned_diff = aligned.pct_change().dropna().add(1).cumprod()
aligned_diff.columns = CCY_LIST
layout['title'] = '<b>Currency 40-Year Cumulative Percentage Change</
↪b><br><sup><i>(1983-2022)</i></sup>'
x_title = 'Date 1983-2022'
y_title = 'Cumulative Foreign Currency Rate Percentage Change VS US Dollar'
diff_fig = plot_all_rates(df=aligned_diff, layout=layout, x_title=x_title,␣
↪y_title=y_title, ht='%{y:,.2%}', yshared=True)
diff_fig.show()
```

Alternatively, as we have standardized the data to be measured in cumulative percentage change against the USD, we can also view the data in a single plot, as shown below:

```python
# Plot all currencies in aligned_diff a single plotly chart
fig = go.Figure()
for ccy in aligned_diff:
    fig.add_trace(go.Scatter(x=aligned_diff.index, y=aligned_diff[ccy],␣
↪mode='lines', name=ccy, hovertemplate='%{y:,.0%}',
                            ))
```

```
layout['width'] = 1800
layout['height'] = 1200
layout['xaxis_title'] = 'Date 1983-2022'
layout['yaxis_title'] = 'Cumulative Foreign Currency Rate Percentage Change VS␣
↪US Dollar'
fig.update_layout(layout)
fig.show()
```

This view transforms the data into a percentage change format, allowing for a comparison of the cumulative percentage change in the rates over the 40-year period. This view provides a useful perspective on the data, highlighting the relative performance of each currency pair over this long period. While there have been moderate fluctuations in the `GBP`, `CAD`, and `JPY` rates, the `INR`, `CNY`, and `ZAR` rates have experienced significant changes over the 40-year period. Namely, dramatic weakening against the US Dollar over time.

As a further step in the exploratory analysis, we can decompose the time series data into its constituent components. This allows us to peel back the layers of the data and gain a deeper understanding of the underlying trends and patterns, should any exist. The following code snippet shows the **`plot_seasonal()`** and **`multi_plot_seasonal()`** functions used to create the seasonal decomposition plots. Becuase the time series data is daily, we will be analyzing seasonality at the monthly level. This is done by defining the **`resample`** parameter as `M` (monthly) in the **`plot_seasonal()`** function (the **`multi_plot_seasonal()`** function passes this parameter to the **`plot_seasonal()`** function as a convenience). The reason for resampling the data at the monthly level is so that we can see the trends and/or patterns more clearly whereas the daily data becomes obscured by the noise of the daily fluctuations.

The figure below is broken into the four component parts of the time series data returned by **`seasonal_decompose()`** (from the **`statsmodels.tsa`** library): **`observed`**, **`trend`**, **`seasonal`**, and **`residual`**. The **`observed`** data is simply the raw data which we've already inspected above, the **`trend`** component uses a moving average to smooth the data, the **`seasonal`** component attempts to isolate any seasonality that can be observed in the data at the specified frequency (in this case, monthly as mentioned above), and finally the **`residual`** component is the difference between the **`observed`** and **`trend`** components and is also referred to as the "noise" in the data.

[ ]:
```
def plot_seasonal(series, resample=None):
    if resample is not None:
        series = series.resample(resample).mean()
    decomp = seasonal_decompose(series)
    decomp_fig = make_subplots(rows=4, cols=1, shared_xaxes=True)
    decomp_fig.add_trace(go.Scatter(x=decomp.observed.index, y=decomp.observed.
↪values, name='Observed'), row=1, col=1)
    decomp_fig.add_trace(go.Scatter(x=decomp.trend.index, y=decomp.trend.
↪values, name='Trend'), row=2, col=1)
    decomp_fig.add_trace(go.Scatter(x=decomp.seasonal.index, y=decomp.seasonal.
↪values, name='Seasonal'), row=3, col=1)
    decomp_fig.add_trace(go.Scatter(x=decomp.resid.index, y=decomp.resid.
↪values, name='Residuals'), row=4, col=1)
```

```
    ynames = ['<b>Observed</b>', '<b>Trend</b>', '<b>Seasonal</b>', '<b>Residuals</b>']
    for i, name in enumerate(ynames):
        decomp_fig.update_yaxes(title_text=name, row=i+1)
    return decomp_fig

def multi_plot_seasonal(df, resample=None):
    if resample is not None:
        df = df.resample(resample).mean()
    ncols = df.shape[1]
    decomp_fig = make_subplots(rows=4, cols=ncols, subplot_titles=[f'<b>{rate}</b>' for rate in rates.columns], shared_xaxes=True,
                               vertical_spacing=0.01, horizontal_spacing=0.03)
    for col in df:
        decomp = seasonal_decompose(df[col])
        figcol = df.columns.get_loc(col) + 1
        decomp_fig.add_trace(go.Scatter(x=decomp.observed.index, y=decomp.observed.values, name=f'{col} - Observed'), row=1, col=figcol)
        decomp_fig.add_trace(go.Scatter(x=decomp.trend.index, y=decomp.trend.values, name=f'{col} - Trend'), row=2, col=figcol)
        decomp_fig.add_trace(go.Scatter(x=decomp.seasonal.index, y=decomp.seasonal.values, name=f'{col} - Seasonal'), row=3, col=figcol)
        decomp_fig.add_trace(go.Scatter(x=decomp.resid.index, y=decomp.resid.values, name=f'{col} - Residuals'), row=4, col=figcol)
    ynames = ['<b>Observed</b>', '<b>Trend</b>', '<b>Seasonal</b>', '<b>Residuals</b>']
    for i, name in enumerate(ynames):
        decomp_fig.update_yaxes(title_text=name, row=i+1, col=1)
        decomp_fig.update_yaxes(tickformat='.1f')
    return decomp_fig

decomp_multiplot = multi_plot_seasonal(rates, resample='M')
layout['title'] = '<b>Currency 40-Year Monthly Seasonal Decomposition</b>'
layout['showlegend'] = False
layout['width'] = 2000
layout['margin'] = {'t': 100, 'b': 50, 'l': 30, 'r': 30}
del layout['yaxis_title'], layout['xaxis_title']
decomp_multiplot.update_layout(layout)
decomp_multiplot.show()
```

Now that we have a better understanding of the data, we can dive into the modeling phase. The first step is to split the data into training and testing sets. The training set will be used to train the `ARIMA` model, and the testing set will be used to evaluate the model's performance over the last 90 days of the time horizon. The following code snippet shows the `train_test_split()` function used to split the data into training and testing sets:

[ ]:
```python
# Build train/test split
def train_test_split(df, days=90):
    end = df.index[-1]
    start = end - dt.timedelta(days=days)
    end = df.index[df.index.get_indexer([start], method='nearest')][0]
    start = df.index[df.index.get_indexer([start], method='nearest') + 1][0]
    train = df.loc[:end].copy()
    test = df.loc[start:].copy()
    return train, test

trains, tests = train_test_split(rates, days=90)
```

Next, we will iteratively train the `ARIMA` model on the training set for each currency pair. The model will then be used to predict the future rate of each currency pair for the next 90 days. The following code snippet shows the `auto_arima()` function from the `pmdarima` package used to train the `ARIMA` models on the training sets utilizing the `ProcessPoolExecutor()` function to parallelize the process in an effort to speed up with training. The model selection process utilizes the `Akaike Information Criteria (AIC)` as a loss function which the algorithm seeks to minimize as it attempts to derive the optimal `ARIMA` model parameters. The optimal model and results of the model training are then returned from the function and stored in a dictionary for use in the next step:

[ ]:
```python
arimafits = {}
with concurrent.futures.ProcessPoolExecutor(max_workers=8) as executor:
    future_to_arima = {executor.submit(auto_arima, trains[ccy]): ccy for ccy in␣
 ↪trains}
    for future in concurrent.futures.as_completed(future_to_arima):
        ccy = future_to_arima[future]
        print(f'\n{"-"*100}\n{"="*40}  Training {ccy}.... ␣
 ↪{"="*40}\n{"-"*100}\n')
        try:
            arimafits[ccy] = future.result()
        except Exception as e:
            print(f'{ccy} generated an exception: {e}')
        else:
            print(f'{ccy} ARIMA Summary:\n{arimafits[ccy].summary().
 ↪as_text()}\n{"-"*100}\n\n')
```

```
--------------------------------------------------------------------------------
--------------------
========================================  Training CAD…
========================================
--------------------------------------------------------------------------------
--------------------

CAD ARIMA Summary:
                               SARIMAX Results
```

```
==============================================================================
Dep. Variable:                      y   No. Observations:                10370
Model:               SARIMAX(0, 1, 0)   Log Likelihood               39169.407
Date:                Wed, 23 Aug 2023   AIC                         -78336.814
Time:                        02:18:00   BIC                         -78329.567
Sample:                    01-03-1983   HQIC                        -78334.366
                         - 09-30-2022
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
sigma2      3.064e-05    1.9e-07    160.983      0.000    3.03e-05     3.1e-05
================================================================================
===
Ljung-Box (L1) (Q):                   1.27   Jarque-Bera (JB):
27663.45
Prob(Q):                              0.26   Prob(JB):
0.00
Heteroskedasticity (H):               3.08   Skew:
-0.14
Prob(H) (two-sided):                  0.00   Kurtosis:
11.00
================================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
--------------------------------------------------------------------------------
--------------------



--------------------------------------------------------------------------------
--------------------
========================================  Training GBP…
========================================
--------------------------------------------------------------------------------
--------------------

GBP ARIMA Summary:
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                10370
Model:               SARIMAX(1, 1, 0)   Log Likelihood               33588.699
Date:                Wed, 23 Aug 2023   AIC                         -67173.399
Time:                        02:18:06   BIC                         -67158.906
Sample:                    01-03-1983   HQIC                        -67168.502
```

```
                       - 09-30-2022
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.0470      0.007      6.353      0.000       0.033       0.062
sigma2      8.992e-05   6.37e-07    141.111      0.000    8.87e-05    9.12e-05
================================================================================
===
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):
14345.12
Prob(Q):                              0.98   Prob(JB):
0.00
Heteroskedasticity (H):               0.55   Skew:
-0.36
Prob(H) (two-sided):                  0.00   Kurtosis:
8.72
================================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
--------------------------------------------------------------------------------
--------------------



--------------------------------------------------------------------------------
--------------------
========================================  Training JPY…
========================================
--------------------------------------------------------------------------------
--------------------

JPY ARIMA Summary:
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                10370
Model:               SARIMAX(5, 2, 0)   Log Likelihood              -13109.761
Date:                Wed, 23 Aug 2023   AIC                          26231.523
Time:                        02:18:09   BIC                          26275.002
Sample:                    01-03-1983   HQIC                         26246.214
                         - 09-30-2022
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
```

```
ar.L1         -0.8176      0.007   -121.503      0.000      -0.831      -0.804
ar.L2         -0.6388      0.009    -74.309      0.000      -0.656      -0.622
ar.L3         -0.4709      0.009    -53.226      0.000      -0.488      -0.454
ar.L4         -0.3310      0.008    -39.414      0.000      -0.347      -0.315
ar.L5         -0.1620      0.007    -24.363      0.000      -0.175      -0.149
sigma2         0.7341      0.006    132.034      0.000       0.723       0.745
================================================================================
===
Ljung-Box (L1) (Q):                   4.85   Jarque-Bera (JB):
11106.61
Prob(Q):                              0.03   Prob(JB):
0.00
Heteroskedasticity (H):               0.39   Skew:
-0.32
Prob(H) (two-sided):                  0.00   Kurtosis:
8.03
================================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
--------------------------------------------------------------------------------
--------------------




--------------------------------------------------------------------------------
--------------------
========================================  Training INR…
========================================
--------------------------------------------------------------------------------
--------------------

INR ARIMA Summary:
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                10370
Model:               SARIMAX(0, 1, 2)   Log Likelihood                2128.235
Date:                Wed, 23 Aug 2023   AIC                          -4248.471
Time:                        02:18:16   BIC                          -4219.484
Sample:                    01-03-1983   HQIC                         -4238.677
                         - 09-30-2022
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
intercept      0.0069      0.002      3.691      0.000       0.003       0.011
```

```
ma.L1         -0.0340      0.004     -8.654      0.000      -0.042      -0.026
ma.L2         -0.0433      0.004    -11.179      0.000      -0.051      -0.036
sigma2         0.0388      0.000    279.192      0.000       0.039       0.039
================================================================================
===
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):
433444.40
Prob(Q):                              0.98   Prob(JB):
0.00
Heteroskedasticity (H):               5.25   Skew:
1.47
Prob(H) (two-sided):                  0.00   Kurtosis:
34.54
================================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
--------------------------------------------------------------------------------
--------------------




--------------------------------------------------------------------------------
--------------------
========================================  Training CNY…
========================================
--------------------------------------------------------------------------------
--------------------

CNY ARIMA Summary:
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                10370
Model:               SARIMAX(5, 2, 0)   Log Likelihood               20091.968
Date:                Wed, 23 Aug 2023   AIC                         -40171.936
Time:                        02:18:23   BIC                         -40128.457
Sample:                    01-03-1983   HQIC                        -40157.245
                         - 09-30-2022
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.8374      0.001  -1049.220      0.000      -0.839      -0.836
ar.L2         -0.6693      0.001   -674.185      0.000      -0.671      -0.667
ar.L3         -0.5026      0.001   -479.929      0.000      -0.505      -0.501
ar.L4         -0.3372      0.001   -341.567      0.000      -0.339      -0.335
```

```
ar.L5         -0.1656      0.001   -211.005      0.000      -0.167      -0.164
sigma2         0.0012   3.45e-07   3520.138      0.000       0.001       0.001
================================================================================
===
Ljung-Box (L1) (Q):                   5.83   Jarque-Bera (JB):
9867544007.96
Prob(Q):                              0.02   Prob(JB):
0.00
Heteroskedasticity (H):               0.06   Skew:
57.36
Prob(H) (two-sided):                  0.00   Kurtosis:
4780.90
================================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
--------------------------------------------------------------------------------
--------------------




--------------------------------------------------------------------------------
--------------------
========================================  Training ZAR…
========================================
--------------------------------------------------------------------------------
--------------------

ZAR ARIMA Summary:
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                10370
Model:               SARIMAX(2, 1, 2)   Log Likelihood               10526.312
Date:                Wed, 23 Aug 2023   AIC                         -21040.624
Time:                        02:18:52   BIC                         -20997.145
Sample:                    01-03-1983   HQIC                        -21025.933
                         - 09-30-2022
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
intercept      0.0057      0.003      1.883      0.060      -0.000       0.012
ar.L1         -1.5432      0.010   -150.065      0.000      -1.563      -1.523
ar.L2         -0.9436      0.010    -92.275      0.000      -0.964      -0.924
ma.L1          1.5534      0.009    178.245      0.000       1.536       1.571
ma.L2          0.9619      0.009    112.305      0.000       0.945       0.979
```

```
sigma2         0.0077   3.88e-05    197.994      0.000       0.008       0.008
================================================================================
===
Ljung-Box (L1) (Q):                   0.16   Jarque-Bera (JB):
90947.26
Prob(Q):                              0.69   Prob(JB):
0.00
Heteroskedasticity (H):              27.58   Skew:
0.34
Prob(H) (two-sided):                  0.00   Kurtosis:
17.49
================================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
--------------------------------------------------------------------------------
--------------------
```

The `auto_arima()` optimizer did a lot of the heavy lifting in terms of finding the optimal model parameters for each currency pair. This is clearly a significant advantage in the modeling process, as it allows for a much more efficient and effective approach to model selection. Now that the models have been trained and optimized, we will proceed to the forecasting phase. The following code snippet shows the custom `plot_all_forecasts()` function used to iterate through the trained models, generate forecasts for the test set, and finally plot the results in a single figure for easy comparison:

[ ]:
```python
layout = {'title': 'Currency Rate 90-Day Forecast',
          'width': 2000,
          'height': 1200,
          'template': 'seaborn',
          'hovermode': 'x unified'}
hovertemp = '%{y:,.4f}'

def plot_all_forecasts(trains, arimafits, layout, x_title, y_title, historical,␣
 ↪sma, sma_df):
    fcfig = make_subplots(rows=2, cols=3, shared_xaxes=True, vertical_spacing=0.
 ↪05, horizontal_spacing=0.02,
                          subplot_titles=([ccy for ccy in trains]),␣
 ↪shared_yaxes=False, x_title=x_title, y_title=y_title)
    for i, ccy in enumerate(trains):
        train = trains[ccy]
        test = tests[ccy]
        year = str(trains.index[-1].year)
```

```
        fc = arimafits[ccy].arima_res_.get_prediction(start=train.index[-1],␣
↪end=test.index[-1]).summary_frame()
        if i // 3 < 1:
            row = 1
            col = i+1
        else:
            row = 2
            col = i-2
        fcfig.add_trace(go.Scatter(name='Forecast', x=fc.index, y=fc['mean'],␣
↪mode='lines', line=dict(color='#e66830'), showlegend=False,␣
↪hovertemplate=hovertemp), row=row, col=col)
        fcfig.add_trace(go.Scatter(name='Upper CI', x=fc.index,␣
↪y=fc['mean_ci_upper'], line=dict(width=0), mode='lines', showlegend=False,␣
↪hovertemplate=hovertemp), row=row, col=col)
        fcfig.add_trace(go.Scatter(name='Lower CI', x=fc.index,␣
↪y=fc['mean_ci_lower'], marker=dict(color="#444"), line=dict(width=0),␣
↪mode='lines',  fillcolor='rgba(66, 107, 133, 0.3)',
                                 fill='tonexty', showlegend=False,␣
↪hovertemplate=hovertemp), row=row, col=col)
        fcfig.add_trace(go.Scatter(name='Actual', x=test.index, y=test,␣
↪mode='lines', line=dict(color='#00b2c9'), hovertemplate=hovertemp, ␣
↪showlegend=False), row=row, col=col)
        if historical:
            fcfig.add_trace(go.Scatter(name='Historical', x=train.loc[year].
↪index, y=train.loc[year], mode='lines', line=dict(color='#200040'),␣
↪showlegend=False, hovertemplate=hovertemp), row=row, col=col)
        if sma:
            fcfig.add_trace(go.Scatter(name='SMA', x=sma_df.index,␣
↪y=sma_df[ccy], mode='lines', line=dict(color='#d3de00'),␣
↪hovertemplate=hovertemp, showlegend=False), row=row, col=col)

    fcfig.update_layout(layout)
    return fcfig

fcfig = plot_all_forecasts(trains, arimafits, layout, x_title='Date range for␣
↪2022 year',
                           y_title='Daily Rate Against US␣
↪Dollar<br><sup><i>with forecast, actual, and upper/lower confidence bounds␣
↪for last 90-days</i></sup>',
                           historical=True, sma=False, sma_df=None)
fcfig.show()
```

The time horizon for the chart above spans all of the 2022 year. Only the last 90 calendar days of the time horizon are the predictions from the test dataset. The rest of the data is the training set data, providing a recent historical context for the test set data. This training portion is represented in the dark blue line leading up to the predictions. The lighter blue line represents the actual test set data. Representing the forecasted values output from the ARIMA models, the orange line plots out the predictions, and finally the grey shaded area represents the 95% confidence interval bounding the forecasted values. The confidence interval is a measure of the uncertainty in the forecasted values and therefore provides a range of probability.

Not surprisingly, the results of the `ARIMA` model predictions are quite interesting. The model appears to have performed quite well for a few of the currencies, but not so well for others. Most actuals appear to at least fall within the 95% confidence interval, but there are a few exceptions. The `GBP` rate for example appears to have a few predicted values that fall outside of the confidence interval. This is not necessarily a bad thing, as the confidence interval is a measure of probability. However, for the `INR` rate, the model certainly appears to have performed quite well, picked up on the trend in the data, and generally predicted the future values reasonably well.

The next step is to evaluate the performance of the model predictions. This will be done by calculating the mean average percentage error (`MAPE`) for each currency pair. The `MAPE` is a measure of the accuracy of the model predictions and is calculated as the average of the absolute percentage error (`APE`) for each prediction. The efficacy of the `ARIMA` model will be evaluated based on the `MAPE` of the predictions for the test set as visualized in the chart above.

[ ]:
```python
# Create function to evaluate forecasts using MSE, RMSE, NRMSE, and MAPE
def eval_forecasts(preds, tests, arima):
    eval_results = {}
    metric = ['MSE', 'RMSE', 'Mean y', 'NRMSE', 'MAPE']
    for ccy in preds:
        if arima:
            pred = preds[ccy].arima_res_.get_prediction(start=tests.index[0],␣
 ↪end=tests.index[-1])._predicted_mean
        else:
            pred = preds[ccy]
        act = tests[ccy]
        mse = mean_squared_error(act, pred)
        rmse = mean_squared_error(act, pred, squared=False)  # squared=False␣
 ↪actually returns RMSE as default is MSE (squared=True)
        mape = mean_absolute_percentage_error(act, pred)  # Mean Absolute␣
 ↪Percentage Error
        results = [mse, rmse, act.mean(), rmse / act.mean(), mape]
        eval_results[ccy] = results
    return pd.DataFrame(eval_results, index=metric).T.sort_values('MAPE',␣
 ↪ascending=False)

# Set number formats for evaluation dataframe, evaluate, and display results
md_formats = [',.4f', ',.4f', ',.4f', ',.4f', ',.2%', '.2%']
arima_eval = eval_forecasts(preds=arimafits, tests=tests, arima=True)
print(f'ARIMA Evaluation:\n{arima_eval.to_markdown(floatfmt=md_formats)}')
```

```
ARIMA Evaluation:
|     |      MSE |     RMSE |    Mean y |    NRMSE |    MAPE |
|:----|---------:|---------:|----------:|---------:|--------:|
| JPY | 240.7251 | 15.5153 | 141.2715 |   10.98% |   8.81% |
| GBP |   0.0053 |   0.0729 |   1.1752 |    6.21% |   5.14% |
```

```
| ZAR |   0.5203 |  0.7213 |  17.5994 |   4.10% |  3.47% |
| CNY |   0.0605 |  0.2460 |   7.1119 |   3.46% |  3.19% |
| CAD |   0.0005 |  0.0223 |   1.3574 |   1.65% |  1.38% |
| INR |   0.6124 |  0.7826 |  82.1385 |   0.95% |  0.82% |
```

[ ]:

```python
# Set moving average window to 90 days and calculate SMA
ma_window = 90
sma = rates.rolling(ma_window).mean()
sma = sma.loc[tests.index[0]:]

fcfig = plot_all_forecasts(trains, arimafits, layout, x_title=f'Forecast Range {tests.index[0].date()} - {tests.index[-1].date()}',
                           y_title='Daily Rate Against US Dollar<br><sup><i>with forecast, Simple Moving Average actual, and upper/lower confidence bounds for last 90-days</i></sup>',
                           historical=False, sma=True, sma_df=sma)
layout['title'] = 'Currency Rate 90-Day Forecast with 90-Day SMA'
fcfig.show()

# Evaluate SMA and ARIMA forecasts
sma_eval = eval_forecasts(sma, tests, arima=False)
print(f'ARIMA Evaluation: \n{arima_eval.to_markdown(floatfmt=md_formats)}\n')
print(f'SMA Evaluation: \n{sma_eval.to_markdown(floatfmt=md_formats)}\n')

combined_eval = sma_eval[['MAPE']].merge(arima_eval[['MAPE']], left_index=True, right_index=True, suffixes=('_SMA', '_ARIMA'))
combined_eval['Better Model'] = combined_eval.apply(lambda x: 'SMA' if x['MAPE_SMA'] < x['MAPE_ARIMA'] else 'ARIMA', axis=1)
print(f'Combined Evaluation: \n{combined_eval.to_markdown(floatfmt=".2%")}\n')
```

```
ARIMA Evaluation:
|     |      MSE |    RMSE |   Mean y |   NRMSE |   MAPE |
|:----|---------:|--------:|---------:|--------:|-------:|
| JPY | 240.7251 | 15.5153 | 141.2715 |  10.98% |  8.81% |
| GBP |   0.0053 |  0.0729 |   1.1752 |   6.21% |  5.14% |
| ZAR |   0.5203 |  0.7213 |  17.5994 |   4.10% |  3.47% |
| CNY |   0.0605 |  0.2460 |   7.1119 |   3.46% |  3.19% |
| CAD |   0.0005 |  0.0223 |   1.3574 |   1.65% |  1.38% |
| INR |   0.6124 |  0.7826 |  82.1385 |   0.95% |  0.82% |

SMA Evaluation:
|     |     MSE |   RMSE |   Mean y |   NRMSE |   MAPE |
|:----|--------:|-------:|---------:|--------:|-------:|
| JPY | 45.0085 | 6.7088 | 141.2715 |   4.75% |  4.26% |
| GBP |  0.0022 | 0.0466 |   1.1752 |   3.97% |  3.59% |
| ZAR |  0.5119 | 0.7155 |  17.5994 |   4.07% |  3.21% |
| CNY |  0.0524 | 0.2289 |   7.1119 |   3.22% |  2.79% |
| CAD |  0.0014 | 0.0375 |   1.3574 |   2.76% |  2.24% |
```

```
| INR |  3.0060 | 1.7338 |  82.1385 |   2.11% |  1.89% |

Combined Evaluation:
|     |   MAPE_SMA |   MAPE_ARIMA | Better Model   |
|:----|-----------:|-------------:|:---------------|
| JPY |      4.26% |        8.81% | SMA            |
| GBP |      3.59% |        5.14% | SMA            |
| ZAR |      3.21% |        3.47% | SMA            |
| CNY |      2.79% |        3.19% | SMA            |
| CAD |      2.24% |        1.38% | ARIMA          |
| INR |      1.89% |        0.82% | ARIMA          |
```

1.0.6 E: Data Summary and Implications

The objective of this study was to evaluate the efficacy of the `ARIMA` time series forecasting model in accurately predicting future foreign exchange rates. The model was trained on historical time series data for six characteristically different currencies and then tested on a holdout sample. The model was then evaluated based on its ability to accurately predict the future rate of the currencies in the test set using the mean average percentage error (`MAPE`) as a measure of accuracy. The null hypothesis assumed a `MAPE` exceeding 20%, while the alternate hypothesis rejects the null if the `MAPE` remains below 20%. The results of the analysis showed that each of the six currencies had a `MAPE` significantly below the 20% threshold, suggesting the potential for practical applicability of the `ARIMA` model in FX rate forecasting. As a result, the null hypothesis was rejected in favor of the alternate hypothesis.

The model seems to have performed particulaly well for the `INR` and `CAD` rates, with `MAPE` values of 0.82% and 1.38%, respectively. The `JPY` predictions performed the worst, with a `MAPE` of 8.81%. A comparison with a Simple Moving Average (`SMA`) model further reveals some nuances. While the `ARIMA` model won out over the `SMA` for `INR` and `CAD` currencies, the latter was more accurate for the remaining currencies. This comparative analysis highlights the multifaceted nature of foreign exchange rate prediction and the need to recognize that no single model will universally fit all scenarios.

One limitation of this analysis is the nature of FX rates and their potential for structurally breaking with past behavior. Economic fluctuations, underlying process changes, sudden political events, and unforeseen global occurrences can have significant impacts on currency exchange rates, making predictions inherently uncertain and hindering the utility of the `ARIMA` model. For example, it's glaringly obvious that China's currency (here identified as `CNY`) experienced a huge disruption in the rate around 1994. This was the year that China devalued its currency by 33% overnight. This is a prime example of a structural break in the data that would be difficult for any model to predict. This study did not incorporate such external factors, possibly affecting the precision of predictions.

Based on these results, a combined approach using both `ARIMA` and `SMA` might be beneficial, choosing the model that best fits each particular currency's characteristics. Firms may be able to leverage the models to adapt their forecasting approach to the specific currency pair, thereby enhancing the accuracy of predictions. This could facilitate more effective financial planning and decision-making, potentially leading to significant cost savings and improved competitiveness.

With regard to future research, it's possible focusing on integrating external factors such as economic indicators, political events, or global incidents into the model could improve its predictive accuracy. Indeed, the data for such factors is readily available (most of which is likely accessible through `FRED`) and could be incorporated into the model to provide a more holistic view of the factors influencing global currency movements.

Furthermore, this analysis can be expanded by exploring other time-series models, such as Exponential Smoothing State Space Models (`ETS`) or neural network structures such as Long Short-Term Memory networks (`LSTM`). Comparing these with `ARIMA` and `SMA` can provide a broader perspective on their suitability for different currency pairs and perhaps even identify a more effective model for FX rate forecasting.

In conclusion, this study has provided valuable insights into the prediction of foreign exchange rates using the `ARIMA` model, revealing areas of strength as well as areas where it might not be the best choice. The insights offer pathways for more effective currency forecasting, potentially enabling firms to implement hedging strategies, enhance financial planning, and realize cost savings. This, in turn, can enable more strategic decision-making and bolsters a firm's ability to mitigate against unpredictable financial risks.