**THE CODE SNIPPET EXPLANATION TO STOCK MARKET PREDICTION(TWITTTER)**

1. FOR THE LIBRARIES : This code snippet imports necessary libraries for data processing and modeling. It also sets up the Python environment for plotting, formatting, and ignoring warnings. Additionally, it sets the start and end dates for a date range. Also use the streamit library to build a web interface that runs the machine learning code from our environment.
2. The load\_data() and the preprocessing(df) method: This code defines two functions.

The first function load\_data() loads the Twitter stock data from a CSV file and returns a Pandas DataFrame. It also includes a text description of the dataset in Markdown format.

The second function preprocessing(df) converts the date column to Pandas datetime format, which is easier to work with when performing time series analysis. Using the @st.cache\_data decorator on load\_data() caches the results of this function so that it only needs to be run once for a given set of input parameters. This can help to improve the performance of the app by reducing redundant data loading.

1. The plotting(df) and resampling(df): This code defines two more functions for visualizing the Twitter stock data.

The plotting(df) function creates a figure with two subplots. The first subplot shows the daily opening, high, low, and closing prices of the stock, while the second subplot shows the daily trading volume.

The resampling(df) function resamples the data to daily, monthly, annual, and quarterly frequency. It then creates a figure with four subplots showing the mean opening, high, low, and closing prices for each respective frequency.

Both functions use the st.pyplot(fig) method to display the resulting figures in the Streamlit app. Note that the @st.cache\_resource decorator is used for resampling(df), as this function doesn't actually change the data itself, but only creates a new visualization based on it. Caching can help improve performance by reducing unnecessary computation if the same data is requested multiple times.

1. Seasonality(df): This code defines a third function called `seasonality(df)` which performs seasonal decomposition of the stock price data and then displays four line charts to visualize the decomposed components: observed, trend, seasonal, and residual.

The function takes as input a DataFrame containing at least two columns - one for the date and another for the closing stock price - and then uses the `pd.date\_range()` function to generate a sequence of business days between two dates. The function then sets the date column as the index of the DataFrame and applies a seasonal decomposition with period value 5 on the closing prices using the `seasonal\_decompose()` function from the statsmodels library. The resulting decomposed components - trend, seasonal, and residual - are then plotted using the `st.line\_chart()` method. Finally, the function splits the DataFrame into training and test sets with an 80/20 split and displays them in Streamlit using the `st.write()` and `st.subheader()` methods. Note that no caching decorator is used in this case since the output depends on the input data and cannot be reused for different input values.

1. This code defines a fourth function called `stationary\_check(series, window=5)` which checks for stationarity of a time series data.

The function takes as input a pandas series (or DataFrame column) and a rolling window size (default is 5). It then computes the rolling mean and standard deviation of the series using the specified window size and plots them along with the original series using matplotlib.

Next, the function applies the Augmented Dickey-Fuller test on the series to determine its stationarity. The results of the test are stored in a pandas Series object and displayed in Streamlit using the `st.write()` and `st.subheader()` methods.

Finally, the function displays the plot using the `st.pyplot()` method. Note that caching is used here since the output depends only on the input series and window size, and can be reused for the same input values. The `@st.cache\_data` decorator indicates that the output should be cached on disk instead of in memory.

1. This code defines a function `fuller\_test(series, window=5)` that performs the Augmented Dickey-Fuller test on a given time series data after applying a Box-Cox transformation to make it more stationary.

The function takes as input a pandas series (or DataFrame column) and a rolling window size (default is 5). It first applies the Box-Cox transformation on the series using the `boxcox()` function from the scipy library, and then converts the resulting transformed series to a pandas Series object.

Next, the function computes the rolling mean and standard deviation of the transformed series using the specified window size and plots them along with the transformed series using matplotlib, similar to the previous function.

Then, the function applies the Augmented Dickey-Fuller test on the transformed series to determine its stationarity. The results of the test are stored in a pandas Series object and displayed in Streamlit using the `st.write()` method.

Finally, the function displays the plot using the `st.pyplot()` method. Again, caching is used here to avoid recomputing the same output for the same input values.

1. This code defines a function `regular\_trans(series, window=5)` that performs regular differencing on a given time series data to make it more stationary.

The function takes as input a pandas series (or DataFrame column) and a rolling window size (default is 5). It first applies first-order differencing to the series using the `diff()` method from pandas. The differenced series obtained is stored in `series\_diff`.

Next, the function defines an inner function `plot\_rolling\_stats(series, window)` that plots the rolling mean and standard deviation of the differenced series, similar to the previous function, but for differencing.

Then, the function applies the Augmented Dickey-Fuller test on the differenced series to determine its stationarity. The results of the test are stored in a pandas Series object and displayed in Streamlit using the `st.write()` method.

Finally, the function displays the plot using the `st.pyplot()` method and adds a subheader titled "Regular Data" above the output of ADF test results with `st.subheader()`.

1. This code defines a function `autocorrelation(series, window=5)` that performs autocorrelation analysis on a given time series data.

The function takes as input a pandas series (or DataFrame column) and a rolling window size (default is 5). It first creates a new matplotlib figure object with four subplots by calling `plt.figure()` and `fig.add\_subplot()`.

The top left plot shows the original time series along with its rolling mean and standard deviation using `ax\_1.plot()` method.

The top right plot shows the autocorrelation function (ACF) of the time series up to 40 lags using `plot\_acf()` method from `statsmodels.graphics.tsaplots` module, also adjusting the y-axis for readability.

The bottom left plot shows the partial autocorrelation function (PACF) of the time series up to 40 lags, using `plot\_pacf()` method with `method='ols'` option and also setting the y-axis limit for readability.

The bottom right plot shows the rolling mean and standard deviation of the differenced time series obtained by applying first-order differentiation using `diff()` method from pandas. Finally, the function displays the entire figure using the `st.write()` method, decorated with `@st.cache\_data` decorator to cache the results generated by this function so that it doesn't have to be recalculated every time the user interacts with the app.

1. This function `fit\_arima(df)` makes use of the `auto\_arima()` function from the `pmdarima` module for Automatic ARIMA modeling, which automatically finds the best hyperparameters for an SARIMA model by performing a grid search over different combinations of values.
2. The function takes in a pandas DataFrame `df` containing the time series data to be modeled. It first splits this data into training and testing sets by selecting the first 80% of the data as training set and the remaining 20% as test set using Pandas' slicing notation.

It then creates an exogenous variable `exogenous` by taking only `'Open'` and `'Close'` columns from the training set except first 5 rows (because the differencing is done on the first order). This exogenous variable can be used later to improve the forecast accuracy by including external factors that influence the target variable.

Next, it fits an ARIMA model with automatic selection of hyperparameters using `auto\_arima()` method on the Close column of the training dataset after removing the first row.

The `auto\_arima()` method takes many optional parameters for fine-tuning the search functionality. In this case, it is called with maximum orders of 10, maximum differences of 2, seasonal factor of 5, trend 'ct', information criterion "oob" (out-of-bag) and out-of-sample size of 10% of the length of the training data.

Finally, the function returns the fitted ARIMA model object. The `@st.cache\_data` decorator is used here to cache the results generated by this function so that it does not have to be re-calculated every time the user interacts with the app.

1. The function `predict\_arima(model, df)` takes in two arguments: a fitted ARIMA model object `model`, and a pandas DataFrame `df` containing the time series data to be used for prediction.

This function first splits the data into training and test sets using Pandas' slicing notation in the same way as the `fit\_arima()` function. It then extracts the exogenous variables for the test set by selecting only the `'Open'` and `'Close'` columns from the test set (excluding the first 5 rows) and assigning them to the `exogenous` variable.

Next, it uses the `model.predict()` method to generate predictions for the length of the test set and returns those predictions.

The predict method takes two important parameters: the number of periods to forecast (`n\_periods`) and the exogenous variable values (`exogenous`). By providing the exogenous variables as inputs, the method allows us to generate more accurate predictions by taking into account external factors that could impact the target variable.

Finally, the function returns the predicted values as an array of the same length as the test set.

1. The `plot\_residuals(df)` function takes in a pandas DataFrame `df` containing the residuals of a time series model. This function plots the residuals and their autocorrelation function (ACF) using matplotlib.

First, the function retrieves the residuals from the DataFrame by calling the `.resid()` method. This method should only be called on a fitted time series model (like an ARIMA or SARIMAX model), as it returns the difference between the actual and predicted values for each time step. Next, the function creates a figure object with two subplots side by side using `fig.add\_subplot(121)` and `fig.add\_subplot(122)`.

In the first subplot (`ax\_1`), the residuals are plotted against their indices using `ax\_1.plot(residuals)`. The title is set to "Residuals of Returns" with `ax\_1.set\_title("Residuals of Returns", size=24)`.

In the second subplot (`ax\_2`), the ACF of the residuals is plotted using the `plot\_acf()` function from the statsmodels library. The `lags` parameter is set to 40 to show the ACF up to 40 lag values, while `zero=False` is used to remove the line at zero lag. The y-axis limit is then set to -0.5 and 0.5 using `ax\_2.set\_ylim(-0.5, 0.5)` to emphasize significant lags. Finally, the function displays the plot object using `st.pyplot(fig)` from Streamlit.

1. The main() is where our program is executing all the functions I created that handles each task in chunks. This is the main function for a Streamlit app that performs various data preprocessing, exploratory data analysis (EDA), modeling and prediction on Twitter stock market price data.

The function starts by displaying the title of the app using `st.title("Twitter Stock Market Price Prediction")`.

Next, it loads the data using the `load\_data()` function, preprocesses it using `preprocessing()`, and displays the resulting DataFrame using `st.write(df)`.

Then it displays some basic information about the dataset such as shape, number of features, column names, and data types using `st.subheader()` and `st.write()`.

It then proceeds to perform EDA by displaying the first few rows of the data, summary statistics, and some visualizations like line chart, area chart, and bar chart using `st.line\_chart()`, `st.area\_chart()`, and `st.bar\_chart()`.

The function also includes other functionalities like plotting figure and axes, resampling, seasonality trends, stationary check, regular transform, autocorrelation, modeling with ARIMA algorithm, residual analysis, and prediction. Each of these functionalities has its own corresponding helper function.

Finally, the function displays some insights about the data and visualization of predicted data using the output of the ARIMA model.

Overall, this function is the backbone of the Streamlit app and provides useful insights and interactions for end-users.

1. The Generate\_Prediction : This is a Python function named `generate\_predictions` that takes in three arguments: `train`, `test`, and `exogenous`. ‘train` represents the training data, which should be a Pandas DataFrame that contains historical data for a particular time series. `test` is also a Pandas DataFrame that contains the test data or the data for which we want to generate predictions. `exogenous` is a string variable that represents the exogenous variable that we will use as input to the SARIMAX model. The function then fits a SARIMAX (Seasonal Autoregressive Integrated Moving Average with Exogenous Regressors) model to the training data using the `SARIMAX` class from the Statsmodels library. It specifies an order of `(1, 1, 0)` and a seasonal\_order of `(1, 0, 0, 12)`. After fitting the model, it generates predictions for the test data by calling the `predict()` method on the fitted model and passing in `len(test)` steps for which predictions are needed, along with the specified `exogenous` variable from the `test` dataframe.

Finally, the function converts the predicted values and actual test data to Pandas Series, respectively, assigns them to `preds\_series` and `actual\_series`, and returns them as a tuple `(actual\_series, preds\_series)`.

1. The run() in the main() : This is a Python function named `run` that uses the `generate\_predictions` function defined earlier and generates predictions for stock price data.

The first step of this function is to load a CSV file named TWITTER.csv containing stock price data into a Pandas dataframe called `stock\_data`. The `parse\_dates` parameter is used to ensure that the 'Date' column is correctly recognized as a date format.

Next, it subsets the dataset into two parts: training data and test data. The training data consists all rows with dates before Jan 1, 2020, while the test data is made up of rows between Jan 1, 2020 and Jun 22, 2022.

Then it creates a sidebar with user input parameters using the `st.sidebar` function from Streamlit library. This allows users to select an exogenous variable ('Volume', 'High', or 'Low') and set start/end dates for both training and test sets.

After this, it subsets the stock data based on the selected user input dates.

Finally, it calls the `generate\_predictions` function with the specified inputs and stores the resulting actual and predicted values in variables `actual` and `preds`, respectively. It then displays the actual stock prices in a header followed by a plot of actual versus predicted values using matplotlib and streamlit's `plt.subplots` and `st.pyplot` functions.

Lastly it runs the `run` function.