Appendix – Interactions with LLM

Query: for a df, how do I change the column ordering such that the first column (0th index) is retained as is and the remaining ones are reversed?

Response:

You can change the column ordering in a DataFrame such that the first column (0th index) is retained as is and the remaining columns are reversed using the following approach:

```python
# Assuming df is your DataFrame

columns = df.columns.tolist()  # Get the list of column names

new_columns = columns[:1] + columns[1:][::-1]  # Retain the first column and reverse the remaining columns

df = df[new_columns]  # Reassign the DataFrame with the new column ordering


# Display the DataFrame with the new column ordering

print(df)
```

In this code:

- `columns` is a list containing the names of all columns in the DataFrame.

- `new_columns` is created by concatenating the first column (`columns[:1]`) with the remaining columns reversed (`columns[1:][::-1]`).

- Finally, the DataFrame `df` is reassigned with the new column ordering using `df[new_columns]`.

Query: Is there a method in Python that counts the number of business days between 2 dates?

Response: Yes, you can use the `numpy.busday_count()` function from NumPy to count the number of business days between two dates. Here's how you can do it:

```python
import numpy as np


# Define the start and end dates
start_date = '2024-01-01'
end_date = '2024-01-10'


# Count the number of business days between the two dates
num_business_days = np.busday_count(start_date, end_date)


print("Number of business days between", start_date, "and", end_date, ":", num_business_days)
```

In this example:

- We import NumPy as `np`.
- We define the start and end dates as strings in the format 'YYYY-MM-DD'.
- We use `np.busday_count(start_date, end_date)` to count the number of business days between the two dates.
- The function excludes both the start and end dates from the count, considering only the intermediate business days.
- The result is stored in the variable `num_business_days` and printed.

Query: #visualize with torch.no_grad(): predicted = model(X_test.to(device)).to('cpu').numpy() temp = y_test-predicted plt.plot(y_test, label = 'Actual Close') plt.plot(predicted, label = 'Predicted Close') plt.xlabel('Day') plt.ylabel('Close') plt.legend() plt.show() - **how can I convert this data into mse?**

Response: To convert the data into Mean Squared Error (MSE), you need to calculate the MSE between the `y_test` (actual values) and the `predicted` values. Here's how you can do it:

```python
from sklearn.metrics import mean_squared_error

# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(y_test, predicted)

print("Mean Squared Error (MSE):", mse)
```

In this code:

- `mean_squared_error` function from `sklearn.metrics` is used to calculate the MSE between `y_test` (actual values) and `predicted` values.

- The calculated MSE value is printed. This value represents the overall difference between the actual and predicted values, with lower MSE indicating better model performance.

Query: How to get RMSE in pytorch?

Response: To calculate RMSE (Root Mean Squared Error) in PyTorch, you can follow these steps:

1. Compute the squared differences between the predicted values and the actual values.

2. Calculate the mean of the squared differences.

3. Take the square root of the mean squared error to obtain RMSE.

Here's how you can do it in PyTorch:

```python
import torch

# Example of predicted values (output from the model)
predicted_values = torch.tensor([2.5, 3.7, 4.2, 5.1, 6.0])

# Example of ground truth values
ground_truth_values = torch.tensor([3.0, 4.0, 4.5, 5.0, 5.5])

# Compute squared differences
squared_diff = (predicted_values - ground_truth_values)**2

# Calculate mean squared error
mse = torch.mean(squared_diff)

# Calculate RMSE
rmse = torch.sqrt(mse)

print("Root Mean Squared Error (RMSE):", rmse.item())
```

In this code:

- `predicted_values` is a tensor containing the predicted values outputted by the model.
- `ground_truth_values` is a tensor containing the ground truth values.

- We compute the squared differences between predicted and ground truth values using `(predicted_values - ground_truth_values)**2`.

- Then, we calculate the mean squared error (MSE) using `torch.mean()`.

- Finally, we take the square root of MSE to obtain RMSE using `torch.sqrt()`.

Query: X_test.describe() - how to convert this tensor object into a pandas dataframe and view summary statistics?

Response: To convert a PyTorch tensor object into a pandas DataFrame and view summary statistics, you can follow these steps:

1. Convert the PyTorch tensor to a NumPy array using the `.numpy()` method.
2. Create a pandas DataFrame from the NumPy array.
3. Use the `.describe()` method on the DataFrame to view summary statistics.

Here's the code:

```python
import pandas as pd

# Assuming X_test is your PyTorch tensor
# Convert PyTorch tensor to NumPy array
X_test_np = X_test.numpy()

# Create pandas DataFrame from NumPy array
X_test_df = pd.DataFrame(X_test_np)

# View summary statistics
summary_stats = X_test_df.describe()
```

```
print(summary_stats)
```

This code will give you summary statistics (count, mean, std, min, 25%, 50%, 75%, and max) for each column in your DataFrame `X_test_df`. Adjust the code accordingly based on your specific tensor object and its dimensions.

Query: how to make the sentiment_data dataframe have only data of the columns 'date', 'pos','neg'

Response: To select specific columns from a DataFrame `sentiment_data` and create a new DataFrame containing only those columns ('date', 'pos', 'neg'), you can use the indexing operator `[]`. Here's how you can do it:

```python
# Assuming sentiment_data is your DataFrame
selected_columns = ['date', 'pos', 'neg']
sentiment_data_subset = sentiment_data[selected_columns]
```

This code will create a new DataFrame `sentiment_data_subset` containing only the columns 'date', 'pos', and 'neg' from the original DataFrame `sentiment_data`. Adjust the column names and DataFrame variable names according to your specific DataFrame and requirements.

Query: how can I get only unique date values and for these unique date values, average out the positive and negative values?

Response: To achieve this, you can use the `groupby()` function along with the `agg()` function in pandas to group the data by the 'date' column, and then calculate the average of the 'pos' and 'neg' values for each unique date. Here's how you can do it:

```python
```

Group by 'date' column and calculate the average of 'pos' and 'neg' for each date

average_sentiment = sentiment_data.groupby('date').agg({'pos': 'mean', 'neg': 'mean'})

Reset index to make 'date' a regular column

average_sentiment.reset_index(inplace=True)

Print the resulting DataFrame

print(average_sentiment)
```

This code will create a new DataFrame `average_sentiment` where each row corresponds to a unique date, and the 'pos' and 'neg' columns contain the average positive and negative values, respectively, for that date. Adjust the column names and DataFrame variable names according to your specific DataFrame and requirements.

---

**Query:** How can I do the following in Python? There are 2 pandas dataframes: processed_stock_df and transformed_sentiment_data. I want join these 2 dataframes when the date fields of both are equal. However, I want to retain all entries of processed_stock_df and if the date field isn't available in transformed_sentiment_data for a row, match its other columns as NaN or NA

**Response:** You can achieve this using the `merge()` function in pandas with the `how='left'` argument to retain all entries from the left DataFrame (`processed_stock_df`) and match the corresponding rows from the right DataFrame (`transformed_sentiment_data`) based on the 'date' column. Here's how you can do it:

```python

# Assuming 'date' column is present in both DataFrames

# Perform a left join on 'processed_stock_df' and 'transformed_sentiment_data'

merged_df = processed_stock_df.merge(transformed_sentiment_data, on='date', how='left')

# Print the resulting merged DataFrame

print(merged_df)

```

This code will perform a left join on the 'date' column, retaining all entries from `processed_stock_df` and matching rows from `transformed_sentiment_data` based on the 'date' column. If the 'date' is not available in `transformed_sentiment_data` for a row in `processed_stock_df`, the corresponding columns from `transformed_sentiment_data` will be filled with NaN values. Adjust the column names and DataFrame variable names according to your specific DataFrames and requirements.

Query: How to drop rows based on dates less than '13-01-2021' and dates greater than '08-31-2021' for the dataframe merged_df and column 'Date'?

Response: To drop rows based on dates less than '13-01-2021' and greater than '08-31-2021' in the 'Date' column of the dataframe 'merged_df', you can use the following code:

```python
import pandas as pd

# Assuming 'Date' column is in datetime format
merged_df['Date'] = pd.to_datetime(merged_df['Date'])

# Filter rows based on date criteria
filtered_df = merged_df[(merged_df['Date'] >= '2021-01-13') & (merged_df['Date'] <= '2021-08-31')]

# Alternatively, if 'Date' column is not in datetime format
# merged_df['Date'] = pd.to_datetime(merged_df['Date'], format='%d-%m-%Y')

# filtered_df = merged_df[(merged_df['Date'] >= '13-01-2021') & (merged_df['Date'] <= '31-08-2021')]
```

This code will create a new dataframe `filtered_df` that contains rows where the 'Date' column falls within the specified date range.

Query: In pandas, how to get the dataframe rows having NaN values?

Response: To get the rows with NaN values in a pandas DataFrame, you can use the `isna()` or `isnull()` method along with boolean indexing. Here's how you can do it:

```python
import pandas as pd

# Assuming df is your DataFrame
# Use boolean indexing to select rows with NaN values
rows_with_nan = df[df.isna().any(axis=1)]

# Display the rows with NaN values
print(rows_with_nan)
```

This code will create a new DataFrame `rows_with_nan` containing only the rows from the original DataFrame `df` that have at least one NaN value. Adjust `df` to your actual DataFrame variable name. You can also specify specific columns by indexing them if you're only interested in checking NaN values in certain columns.

Query: I've got a numpy array of dimensions (160,7). The 7 represents 7 features it has. I wish to add 2 more features from 2 columns of a pandas dataframe. How can I do this?
```

PFB the details:

Pandas dataframe name and columns - merged_df['pos'] and merged_df['neg']

Name of the numpy array – X

Response: You can achieve this by first extracting the values from the two columns of the pandas DataFrame and then concatenating them as new columns to your numpy array. Here's how you can do it:

```python
import numpy as np

# Assuming merged_df is your pandas DataFrame and X is your numpy array
# Extract values from 'pos' and 'neg' columns of merged_df
pos_values = merged_df['pos'].values
neg_values = merged_df['neg'].values

# Concatenate pos_values and neg_values as new columns to X
# First, reshape pos_values and neg_values to match the dimensions of X
pos_values_reshaped = pos_values.reshape(-1, 1)  # Reshape to (n, 1)
neg_values_reshaped = neg_values.reshape(-1, 1)  # Reshape to (n, 1)

# Concatenate pos_values_reshaped and neg_values_reshaped horizontally to X
X_with_extra_features = np.hstack((X, pos_values_reshaped, neg_values_reshaped))

# Now X_with_extra_features contains the original features of X plus the 'pos' and 'neg' columns
```

In this code:

- `pos_values` and `neg_values` are extracted from the 'pos' and 'neg' columns of the pandas DataFrame `merged_df`.

- The extracted values are reshaped to have a single column, matching the number of rows in `X`.

- Finally, the reshaped 'pos' and 'neg' values are concatenated horizontally (along axis 1) to the original numpy array `X`, resulting in `X_with_extra_features` which contains all the original features of `X` plus the 'pos' and 'neg' columns.

Query: There are 2 columns - pos and neg in the dataframe merged_df. I want to replace the NaN in the columns with the mean of those columns. How to do that?

Response: You can replace NaN values in the 'pos' and 'neg' columns with the mean of each respective column using the `fillna()` method:

```python
merged_df['pos'].fillna(merged_df['pos'].mean(), inplace=True)
merged_df['neg'].fillna(merged_df['neg'].mean(), inplace=True)
```

This code will replace NaN values in the 'pos' and 'neg' columns with the mean of each column, respectively, and update the DataFrame in place.

Query: How can I reduce the dimension of X from torch.Size([9, 1]) to torch.Size([7,1])?

Response: To reduce the dimension of a tensor `X` from `torch.Size([9, 1])` to `torch.Size([7, 1])`, you can use tensor slicing or indexing. Here's how you can achieve it:

```python
import torch
```

```
# Example tensor X with size [9, 1]
X = torch.randn(9, 1)

# Slice the tensor to select the first 7 rows
X_reduced = X[:7]

# Check the size of the reduced tensor
print(X_reduced.size())  # Output: torch.Size([7, 1])
```

In this code:

- `X[:7]` selects the first 7 rows of the tensor `X`.
- The resulting tensor `X_reduced` will have a size of `[7, 1]`, effectively reducing the dimension from `[9, 1]` to `[7, 1]`.

Query: plt.plot(pred, label = 'Predicted Closing Price') plt.xlabel('Day') plt.ylabel('Closing Price') plt.legend() plt.show() - for this code, can I set the x-values to the column stock_price_processed['Date']?

Response: Yes, you can set the x-values to the 'Date' column from your dataframe `stock_price_processed`. Here's how you can modify the code:

```python
plt.plot(stock_price_processed['Date'], y, label='Actual Closing Price')
plt.plot(stock_price_processed['Date'], pred, label='Predicted Closing Price')
plt.xlabel('Date')
plt.ylabel('Closing Price')
```

plt.legend()

plt.show()

```

This will plot the actual and predicted closing prices against the corresponding dates from the 'Date' column in your dataframe. Make sure that the 'Date' column is in the correct datetime format for proper plotting.

---

**Query:** plt.plot( dates,y, label = 'Actual Closing Price') plt.plot(dates,pred, label = 'Predicted Closing Price') - what if I want to plot dates on the x-axis but it's a dataframe?

**Response:** If `dates` is a DataFrame containing datetime values, you can plot it on the x-axis by accessing the datetime values as a series. Here's how you can do it:

```python

import matplotlib.pyplot as plt

# Assuming 'dates' is a DataFrame with datetime values as the index

# You can access the index as a series using dates.index

dates_series = dates.index

# Plot the data with dates on the x-axis

plt.plot(dates_series, y, label='Actual Closing Price')

plt.plot(dates_series, pred, label='Predicted Closing Price')

# Add labels and legend

plt.xlabel('Date')

```
plt.ylabel('Closing Price')
plt.legend()

# Show the plot
plt.show()
```

This way, you can plot the datetime values from the DataFrame `dates` on the x-axis of your plot.

Query: Is there any other way to do it? It has only one column and those are the dates. Can that be converted to a series object?

Response: Yes, if `dates` is a DataFrame with datetime values as the index and only one column, you can directly use the index as a series object. Here's how you can do it:

```python
import matplotlib.pyplot as plt

# Plot the data with dates on the x-axis
plt.plot(dates.index, y, label='Actual Closing Price')
plt.plot(dates.index, pred, label='Predicted Closing Price')

# Add labels and legend
plt.xlabel('Date')
plt.ylabel('Closing Price')
plt.legend()
```

```
# Show the plot

plt.show()
```

By using `dates.index`, you are directly accessing the datetime index of the DataFrame as a series object, which can be used as the x-axis values in the plot.

Query: No, its got a regular index.

Response: If `dates` DataFrame has a regular index and the dates are stored as values in a column, you can plot them on the x-axis by accessing that column. Here's how you can do it:

```python
import matplotlib.pyplot as plt


# Plot the data with dates on the x-axis

plt.plot(dates['Date'], y, label='Actual Closing Price')

plt.plot(dates['Date'], pred, label='Predicted Closing Price')


# Add labels and legend

plt.xlabel('Date')

plt.ylabel('Closing Price')

plt.legend()


# Show the plot

plt.show()
```

Here, `'Date'` is the name of the column containing the dates in the `dates` DataFrame. Replace `'Date'` with the actual column name if it's different. This will plot the data with dates on the x-axis using the values from the specified column.

Query: Can I get the first index with date > 2021-05-31 in the dataframe dates_df?

Response: Yes, you can use boolean indexing to filter the DataFrame `dates_df` based on the condition `date > '2021-05-31'` and then use `.idxmin()` to get the index label of the first occurrence. Here's how you can do it:

```python
first_index_after_date = dates_df[dates_df['date'] > '2021-05-31'].index.min()

print(first_index_after_date)
```

Replace `'date'` with the actual name of the date column in your DataFrame. This will give you the index label of the first row where the date is greater than '2021-05-31'.

Query: the date labels are displaying data for around 60 days, so how should they spaced in the graph for aesthetic purposes and how to accomplish that?

Response: To improve the aesthetics of the graph and ensure that the date labels are evenly spaced, you can adjust the frequency of the date labels on the x-axis. You can achieve this by setting the locator and formatter for the x-axis using `matplotlib.dates` module.

Here's how you can do it:

```python
import matplotlib.pyplot as plt

import matplotlib.dates as mdates
```

```
# Assuming 'dates' is a list or array containing datetime objects
# Convert it to a list of date strings for plotting
date_strings = [date.strftime('%Y-%m-%d') for date in dates]

# Plotting the graph
plt.plot(date_strings, y, label='Actual Closing Price')
plt.plot(date_strings, pred, label='Predicted Closing Price')

# Rotate the x-axis labels for better readability
plt.xticks(rotation=45)

# Set the locator and formatter for the x-axis
plt.gca().xaxis.set_major_locator(mdates.AutoDateLocator())
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))

# Add labels and legend
plt.xlabel('Date')
plt.ylabel('Closing Price')
plt.legend()

# Display the plot
plt.show()
```

This code snippet rotates the x-axis labels for better readability, sets the locator to automatically determine the date intervals, and formats the date labels as `%Y-%m-%d` (Year-Month-Day). Adjust the rotation angle and date format as needed to fit your specific requirements.

Query: I've got 2 method calls that both display graphs: visualize_actual_predicted(X_train, y_train, 'Stock Price Prediction with Historic and Sentiment Data (Training)') visualize_sentiment_data(training_sentiment_data['Date'], training_sentiment_data['pos'], training_sentiment_data['neg'], title = 'Reddit Sentiment Data (Training Period)') - **can I have these two plotted next to each other in a jupyter notebook?**

Response: Yes, you can plot these two graphs next to each other in a Jupyter Notebook. You can use Matplotlib, a popular plotting library in Python, to create subplots and display the graphs side by side. Here's how you can do it:

```python
import matplotlib.pyplot as plt


# Assuming visualize_actual_predicted and visualize_sentiment_data functions create plots

# Replace these function calls with actual code to generate the plots if needed


# Create a figure and two subplots

fig, axs = plt.subplots(1, 2, figsize=(15, 5))


# Plot the first graph on the left subplot

plt.sca(axs[0])

visualize_actual_predicted(X_train, y_train, 'Stock Price Prediction with Historic and Sentiment Data (Training)')


# Plot the second graph on the right subplot

plt.sca(axs[1])

visualize_sentiment_data(training_sentiment_data['Date'], training_sentiment_data['pos'], training_sentiment_data['neg'], title='Reddit Sentiment Data (Training Period)')
```

Adjust layout to prevent overlap

plt.tight_layout()

Show the plots

plt.show()

```

This code will create a figure with two subplots side by side, and then it will plot the graphs generated by the `visualize_actual_predicted` and `visualize_sentiment_data` functions in each subplot accordingly. Adjust the figure size (`figsize`) as needed to ensure the plots fit well within the notebook.

---

**Query:** And what should I do with the ax values inside the functions?

**Response:** If you're modifying the functions `visualize_actual_predicted` and `visualize_sentiment_data` to accept the `ax` parameter for specifying the axes (subplots) explicitly, you'll need to adjust the implementation of these functions accordingly. Here's how you can modify them:

```python
def visualize_actual_predicted(X_train, y_train, title, ax=None):
    # Your code to visualize actual vs predicted data
    if ax is None:
        plt.figure()  # Create a new figure if ax is not provided
        ax = plt.gca()  # Get the current axes

    # Plot actual vs predicted data on the specified axes
    ax.plot(X_train, y_train, label='Actual')
    ax.plot(X_train, predicted_values, label='Predicted')

```
    ax.set_title(title)
    ax.set_xlabel('X-axis label')
    ax.set_ylabel('Y-axis label')
    ax.legend()

def visualize_sentiment_data(date_data, pos_data, neg_data, title, ax=None):
    # Your code to visualize sentiment data
    if ax is None:
        plt.figure() # Create a new figure if ax is not provided
        ax = plt.gca() # Get the current axes

    # Plot sentiment data on the specified axes
    ax.plot(date_data, pos_data, label='Positive Sentiment')
    ax.plot(date_data, neg_data, label='Negative Sentiment')
    ax.set_title(title)
    ax.set_xlabel('Date')
    ax.set_ylabel('Sentiment Score')
    ax.legend()

Assuming you have other necessary code for data processing and visualization
```

In these modified functions:

- We've added a condition to check if `ax` is `None`. If `ax` is not provided (i.e., it's `None`), we create a new figure and get the current axes using `plt.figure()` and `plt.gca()` respectively.

- We then plot the data on the specified axes (`ax`) rather than creating new subplots internally.

With these modifications, the functions will work both independently (if `ax` is not provided) and within the context of plotting side by side in a Jupyter Notebook (if `ax` is explicitly provided).

Query: ax.xticks(rotation=45) - how to convert this for an axis object?

Response: To set the rotation of x-axis labels for an axis object (`ax`), you can use the `tick_params()` method. Here's how you can convert `ax.xticks(rotation=45)` for an axis object:

```python
ax.tick_params(axis='x', rotation=45)
```

This line of code sets the rotation of the x-axis labels to 45 degrees for the specified axis (`'x'`) using the `tick_params()` method.

Query: I have a list of lists, having 3 columns each. How can I convert this into a pandas dataframe and further print this table in an aesthetic manner?

Response: You can convert your list of lists into a pandas DataFrame using the `pd.DataFrame()` constructor. Then, you can use the `display()` function to print the DataFrame in a more aesthetic manner. Here's how you can do it:

```python
import pandas as pd

# Assuming your list of lists is named 'data'
data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Convert list of lists to pandas DataFrame
```

```
df = pd.DataFrame(data, columns=['Column1', 'Column2', 'Column3'])


# Print the DataFrame
display(df)
```

This will display the DataFrame in a tabular format with gridlines, making it easier to read. If you're using Jupyter Notebook or another environment that supports rich display, you'll see the DataFrame rendered nicely. If you're working in a plain Python script or terminal, you'll still get a tabular output, but without the styling and gridlines.
