Python Notes

Giuseppe Giorgio Colabufo

February 13, 2024

Disclaimer

This notebook is created to provide general information and support, but there may be errors or inaccuracies in the content. If you notice any inaccuracies or issues in my notes, please report them. It is always recommended to verify the information provided with reliable sources and consult qualified experts for specific matters. I am not responsible for any consequences arising from the use of the information contained in this notebook. To report any errors or provide feedback, I encourage you to contact me at the specified address below:

[giuseppe.colabufo.2016@polytechnique.org]

Thank you for your understanding and cooperation in helping me improve and correct any errors in the document.

Brief Introduction

These notes are a collection of tricks, features, and code snippets related to the Python programming language that I have gathered over time and found useful in various situations. They represent a set of practical solutions I have discovered and experimented with to tackle specific challenges in my professional or personal journey. I have gathered this information throughout my Python learning journey and found them particularly useful for solving common problems, implementing algorithms, or exploring Python's capabilities. I hope these notes can provide you with practical ideas and solutions to tackle your Python programming challenges.

Within these notes, you will find a variety of tips and ideas, ranging from small tricks that simplify daily tasks to advanced features that can enhance efficiency and user experience with specific tools or technologies. You will also come across code snippets that I have deemed particularly helpful in solving common problems or implementing specific functionalities.

This collection is an (more or less) organic compilation of information that I have updated and expanded over time. I invite you to explore these notes with curiosity and use the proposed solutions as inspiration for your specific needs. Remember, it is always important to adapt and personalize the solutions based on the context in which you apply them.

I hope this information proves useful to you and inspires you on your journey. Enjoy exploring and applying these insights!

Table of Contents

1 Generic Python objects

1.1 Dataframes

1.1.1 Display all dataframe columns

Just use the following option:

```
pd.set_option('display.max_columns', None)
```

Source: Display all dataframe columns in a Jupyter Python Notebook.

1.1.2 Aggregate values in a dataframe by bands

You can aggregate values in a dataframe by bands using the `cut()` function in pandas. The `cut()` function allows you to create categorical bins or bands for continuous data and then perform aggregation operations within those bands.

Here's an example to demonstrate how to aggregate values in a dataframe by bands:

```
import pandas as pd

# Create a sample dataframe
data = {'Value': [10, 25, 15, 30, 35, 20, 5, 12, 18, 22]}
df = pd.DataFrame(data)

# Define the bands using cut() function
bins = [0, 10, 20, 30, 40]
labels = ['0-10', '11-20', '21-30', '31-40']

# Assign each value to a band using cut()
df['Band'] = pd.cut(df['Value'], bins=bins, labels=labels, right=False)

# Aggregate values by band
aggregated_df = df.groupby('Band')['Value'].sum()

print(aggregated_df)
```

Output:

```
Band
0-10      15
11-20     55
21-30     65
31-40     35
Name: Value, dtype: int64
```

In this example, the `cut()` function is used to define the bands based on the 'Value' column. The `bins` parameter specifies the bin edges, and the `labels` parameter provides the labels for each bin.

The `cut()` function assigns each value in the 'Value' column to the corresponding band. The resulting 'Band' column in the dataframe indicates the band to which each value belongs.

Finally, the values in the 'Value' column are aggregated by band using the `groupby()` function, and the `sum()` aggregation function is applied. The resulting `aggregated_df` dataframe shows the sum of values within each band.

You can modify the `bins` and `labels` parameters to define your desired bands. Additionally, you can choose different aggregation functions such as `mean()`, `max()`, or `count()` based on your specific requirements.

1.1.3 Compare two dataframes

Here's a custom function that compares two DataFrames and checks if the values are equal (for strings) or similar (for floats with a tolerance of 1e-6):

```
import pandas as pd
import numpy as np

def compare_dataframes(df1, df2):
    # Check if column labels are equal
    if not df1.columns.equals(df2.columns):
        return False

    # Iterate over columns
    for col in df1.columns:
        # Check data type of the column
        dtype = df1[col].dtype

        if dtype == object:
            # For string columns, check if values are equal
            if not df1[col].equals(df2[col]):
                return False
        elif np.issubdtype(dtype, np.floating):
            # For float columns, check if values are similar within the tolerance
            if not np.allclose(df1[col], df2[col], atol=1e-6):
                return False
        else:
            # For other data types, check if values are equal
            if not df1[col].equals(df2[col]):
                return False

    return True
```

You can use this `compare_dataframes()` function to compare two DataFrames. It returns `True` if the DataFrames are equal or similar based on the comparison rules, and `False` otherwise.

Here's an example of how to use the function:

```
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': ['foo', 'bar', 'baz'], 'C': [0.1, 0.2, 0.3]})
df2 = pd.DataFrame({'A': [1, 2, 4], 'B': ['foo', 'bar', 'qux'], 'C': [0.1, 0.199999, 0.3]})

are_equal = compare_dataframes(df1, df2)
```

```
print("Are the DataFrames equal?", are_equal)
```

Output:

```
Are the DataFrames equal? False
```

In this example, the `compare_dataframes()` function is used to compare the DataFrames `df1` and `df2`, and it returns `False` because the values in the 'A' column are not equal (`3` in `df1` vs `4` in `df2`), and the values in the 'B' column are not equal (`'baz'` in `df1` vs `'qux'` in `df2`). The 'C' column values are considered similar because their difference (`0.1 - 0.199999`) is less than `1e-6`.

1.1.4 Combine dataframes

I have 2 df, both indexed by a ts. df1 has a longer ts. I want to replace values from df2 in df1 if they exists and keep values from df1 otherwise.

You can use the `combine_first()` method in pandas to replace values from `df1` with values from `df2` where they exist, and keep the values from `df1` otherwise. The `combine_first()` method aligns the indexes of the two DataFrames and replaces missing values in the calling DataFrame with non-missing values from the passed DataFrame.

Here's an example of how you can achieve this:

```
import pandas as pd

df1.combine_first(df2)
```

In this example, `df1` is the calling DataFrame and `df2` is the passed DataFrame. The resulting DataFrame will have values from `df2` where they exist and values from `df1` otherwise, using the index alignment between the two DataFrames.

Note that this operation is non-destructive, meaning it returns a new DataFrame with the combined values while leaving `df1` and `df2` unchanged. If you want to modify `df1` in-place, you can use the assignment operator (`=`) to assign the combined DataFrame back to `df1`:

```
df1 = df1.combine_first(df2)
```

To join multiple DataFrames together and keep the existing data from the first DataFrame while adding data for further timestamps from subsequent DataFrames, you can use the `concat()` function in pandas with the `axis=0` parameter.

Here's an example of how you can achieve this:

```
import pandas as pd

# List of DataFrames to join
dfs = [df1, df2, df3, ...]

# Concatenate DataFrames along the rows (axis=0)
combined_df = pd.concat(dfs, axis=0)

# Sort the combined DataFrame by the timestamp index
combined_df = combined_df.sort_index()
```

In this example, `df1`, `df2`, `df3`, … are the DataFrames you want to join. The `concat()` function is used to concatenate the DataFrames along the rows (`axis=0`), which effectively appends the data for further timestamps from subsequent DataFrames. Finally, you can sort the combined DataFrame by the timestamp index using the `sort_index()` function if desired.

Note that the order of the DataFrames in the `dfs` list determines the priority of the data. The data from the first DataFrame (`df1`) will be preserved, and if there are overlapping timestamps in subsequent DataFrames, their data will replace the existing data in the combined DataFrame.

Make sure that the timestamps are properly aligned and have the same data type (e.g., pandas `Timestamp` or `datetime.datetime`) in all DataFrames to avoid any issues during concatenation and sorting.

1.1.5 Codify string values as int

If you don't know the specific values of the strings in your DataFrame column, you can use the `factorize` function from the pandas library. It automatically assigns integer codes to the unique values in a column. Here's an example:

```
# Assuming 'df' is your DataFrame and 'col_name' is the column with string values

# Codify the string values as integers
df['col_name'] = pd.factorize(df['col_name'])[0]
```

The `factorize` function assigns a unique integer code to each unique value in the 'col_name' column, regardless of the specific string values. The resulting codified values will be integers starting from 0 and increasing sequentially.

After executing this code, the 'col_name' column will contain the codified integer values instead of the original string values.

Recover the association of codified values:

You can recover the association to decode the codified values. The `factorize` function returns two outputs: the encoded values and the unique labels. You can use the unique labels to map the encoded values back to their original string values. Here's an example:

```
# Assuming 'df' is your DataFrame and 'col_name' is the column with codified values

# Get the unique labels and encoded values
encoded_values, unique_labels = pd.factorize(df['col_name'])

# Create a dictionary to map the encoded values to their corresponding labels
label_mapping = dict(zip(encoded_values, unique_labels))

# Decode the values in the 'col_name' column
df['col_name'] = df['col_name'].map(label_mapping)
```

After executing this code, the 'col_name' column will contain the original string values instead of the codified values. The `label_mapping` dictionary allows you to map the encoded values back to their corresponding labels.

1.1.6 Write a DataFrame to a SQL database

To write a DataFrame to a SQLite database, you can use the `to_sql()` method provided by pandas. Here's an example:

```
import sqlite3

# Establish a connection to the SQLite database
conn = sqlite3.connect('your_database.db')

# Write the DataFrame to the database
df.to_sql('your_table_name', conn, if_exists='replace')

# Close the connection
conn.close()
```

In this code, `df` refers to your DataFrame that you want to write to the database. `'your_database.db'` is the name of your SQLite database file. `'your_table_name'` is the name you want to give to the table where the DataFrame will be stored. The `if_exists` parameter is set to `'replace'` to overwrite the table if it already exists. You can change it to `'append'` if you want to append the DataFrame to an existing table.

After running this code, the DataFrame will be written to the SQLite database specified.

1.1.7 Easy one-hot encoding

To encode categorical variables in a DataFrame column for use in an artificial neural network, you can use one-hot encoding. One-hot encoding converts each unique category into a binary vector, where each element represents the presence or absence of a particular category.

You can use the `get_dummies` function from pandas to perform one-hot encoding. Here's an example:

```
import pandas as pd

# Assuming your DataFrame is called df and the column with categorical
# values is 'category_col'
encoded_df = pd.get_dummies(df, columns=['category_col'])
```

The `get_dummies` function will create new columns in the encoded DataFrame for each unique category in the specified column. The values in these columns will be 1 or 0, indicating the presence or absence of the corresponding category.

After performing one-hot encoding, you can use the encoded DataFrame as input for your artificial neural network.

Here's an example of one-hot encoding a DataFrame column using `get_dummies`:

Input DataFrame (`df`):

id	category_col
1	A

id	category_col
2	B
3	C
4	A
5	B

Output DataFrame (`encoded_df`):

id	category_col_A	category_col_B	category_col_C
1	1	0	0
2	0	1	0
3	0	0	1
4	1	0	0
5	0	1	0

In this example, the original DataFrame has a column named "category_col" with categorical values. After applying one-hot encoding using `get_dummies`, new columns are created for each unique category in the "category_col" column. Each column is binary and represents the presence or absence of a particular category.

Note that the original "category_col" column is replaced by the encoded columns in the output DataFrame.

1.1.8 Dropped rows

To see the rows that were dropped when using `dropna` on a subset of columns, you can use the `index` attribute of the DataFrame before and after dropping the rows. By comparing the two index sets, you can identify the rows that were dropped. Here's an example:

```
import pandas as pd

# Sample DataFrame
data = {
    'A': [1, 2, None, 4, 5],
    'B': [None, 8, 9, 10, 11],
    'C': [15, 16, 17, 18, None]
}
df = pd.DataFrame(data)

# Save the original index
original_index = df.index

# Drop rows with missing values in columns 'A' and 'B'
subset_cols = ['A', 'B']
df.dropna(subset=subset_cols, inplace=True)
```

```
# Compare the original index with the current index to find the dropped rows
dropped_rows = original_index.difference(df.index)

# Print the rows that were dropped
print(df.loc[dropped_rows])
```

In this example, we create a DataFrame with missing values in columns 'A', 'B', and 'C'. We then drop the rows with missing values in columns 'A' and 'B' using the `dropna` method. Finally, we compare the original index with the current index to find the dropped rows and print them.

1.1.9 Cyclic forward fill NaN

You can forward fill NaN values in a DataFrame in a cyclic way using a custom function. In the function `cyclic_forward_fill_all`, we loop through all columns of the DataFrame and apply the cyclic forward fill to each column separately. This way, it works for all columns in your DataFrame. Adjust the value of `k` as needed.

```
# Define a function to cyclically forward fill NaN values for all columns
def cyclic_forward_fill_all(df, k):
    for column in df.columns:
        series = df[column]
        n = len(series)
        for i in range(n):
            if pd.isna(series[i]):
                series[i] = series[i - k] if i >= k else series[i]
```

Here's an example of how to use it:

```
import pandas as pd
import numpy as np

# Create a sample DataFrame
data = {'A': [1, 2, 3, np.nan, np.nan, 6, 7, 8, 9],
        'B': [np.nan, 20, np.nan, 40, 50, np.nan, 70, np.nan, 90]}
df = pd.DataFrame(data)

# Apply cyclic_forward_fill_all to the DataFrame with k=2
k = 2
cyclic_forward_fill_all(df, k)

print(df)
```

The provided approach iterates through each column and each element, which can be relatively slow for large DataFrames. A more efficient way to cyclically forward fill NaN values in a DataFrame is to use vectorized operations with NumPy. Here's an optimized version of the function:

```
# Define a function to cyclically forward fill NaN values for all columns
def cyclic_forward_fill_all(df, k):
    values = df.values
    for i in range(k, len(df)):
        mask = np.isnan(values[i])
```

```
        if mask.any():
            values[i, mask] = values[i - k, mask]
```

This optimized version uses NumPy to handle the filling operation efficiently and avoids explicit loops over rows and columns. It should be significantly faster for larger DataFrames.

A further modification would consist in add a random error in filling the df.

```
def cyclic_forward_fill_with_error(df, k, error_std):
    values = df.values
    for i in range(k, len(df)):
        mask = np.isnan(values[i])
        if mask.any():
            values[i, mask] = values[i - k, mask]
            # Add random error only to previous NaN data
            values[i, mask] += np.random.normal(0, error_std, mask.sum())
```

1.2 Dictionaries

Visualize a dictionary To visualize a dictionary, you can use the `print` function to display its key-value pairs. Here's an example:

```
my_dict = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}

# Print the dictionary
print(my_dict)
```

Output:

```
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
```

Alternatively, if you want a more formatted and readable output, you can use the `pprint` module in Python's standard library. Here's an example:

```
import pprint

my_dict = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}

# Pretty-print the dictionary
pprint.pprint(my_dict)
```

Output:

```
{'key1': 'value1',
 'key2': 'value2',
 'key3': 'value3'}
```

The `pprint` module provides more control over the formatting of the dictionary, especially if the dictionary has nested structures or long values.

1.3 Plots

1.3.1 How many curves are plotted in a figure with pyplot?

The `pyplot` module provides a function called `get_lines()` which returns a list of all the Line2D objects representing the plotted curves in the current figure. You can then use the `len()` function to get the count of curves. Here's an example:

```
import matplotlib.pyplot as plt

# Plotting curves
plt.plot([1, 2, 3, 4], [5, 6, 7, 8])
plt.plot([1, 2, 3, 4], [9, 10, 11, 12])
plt.scatter([1, 2, 3, 4], [13, 14, 15, 16])

# Get the number of curves
num_curves = len(plt.gca().get_lines())

print("Number of curves plotted:", num_curves)
```

Output:

```
Number of curves plotted: 3
```

In this example, we plot two lines and one scatter plot in the current figure using `plt.plot()` and `plt.scatter()`. The `plt.gca().get_lines()` function returns a list of Line2D objects representing the plotted curves. By applying `len()` to this list, we get the count of curves, which is printed as the output.

Note that `plt.gca()` returns the current

1.3.2 Filter lines with no points

You can filter lines with no points from the list of Line2D objects returned by `plt.gca().get_lines()`. Each Line2D object has a property called `get_xydata()` which returns the coordinates of the line. You can check the length of the coordinates to determine if a line has no points.

Here's an example that demonstrates how to filter lines with no points:

```
import matplotlib.pyplot as plt

# Plotting curves
plt.plot([1, 2, 3, 4], [5, 6, 7, 8])
plt.plot([1, 2, 3, 4], [])  # Empty line with no points
plt.plot([], [])  # Empty line with no points
plt.scatter([1, 2, 3, 4], [13, 14, 15, 16])

# Get the lines
lines = plt.gca().get_lines()

# Filter lines with no points
```

```
filtered_lines = [line for line in lines if len(line.get_xydata()) > 0]

print("Number of lines with points:", len(filtered_lines))
```

Output:

```
Number of lines with points: 2
```

In this example, we plot three lines, one of which is empty (no points). After getting the lines using `plt.gca().get_lines()`, we filter the lines by checking the length of their `get_xydata()` coordinates. Lines with no points will have an empty array of coordinates, so we exclude them from the filtered list.

The resulting `filtered_lines` list will only contain the lines that have at least one point. The length of this list represents the number of lines with points.

1.3.3 Number of points in a scatter plot

To get the number of points in a scatter plot, you can access the data used to create the scatter plot and check its length. If you are using a library like Matplotlib, you can use the `get_offsets()` method to get the coordinates of the points and then use the `len()` function to get the number of points.

Here's an example using Matplotlib:

```
import matplotlib.pyplot as plt

# Create a scatter plot
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]
plt.scatter(x, y)

# Get the number of points
num_points = len(plt.gca().collections[0].get_offsets())

# Print the number of points
print(f"Number of points: {num_points}")
```

In this example, we first create a scatter plot using the `scatter()` function from Matplotlib. We then access the current Axes object using `plt.gca()` and retrieve the scatter plot's collections using `collections[0]`. Finally, we use the `get_offsets()` method to obtain the coordinates of the points and calculate the number of points using `len()`.

1.3.4 Bar plots

You can create a bar plot of your DataFrame using the `plot()` function in Pandas. Here's an example:

```
import pandas as pd
import matplotlib.pyplot as plt

# Sample DataFrame
```

```
data = {
    'Category': ['A', 'B', 'C', 'D'],
    'Value': [10, 20, 15, 30]
}

df = pd.DataFrame(data)

# Create a bar plot
df.plot(kind='bar', x='Category', y='Value', legend=False)
plt.xlabel('Category')
plt.ylabel('Value')
plt.title('Bar Plot of Categories and Values')
plt.show()
```

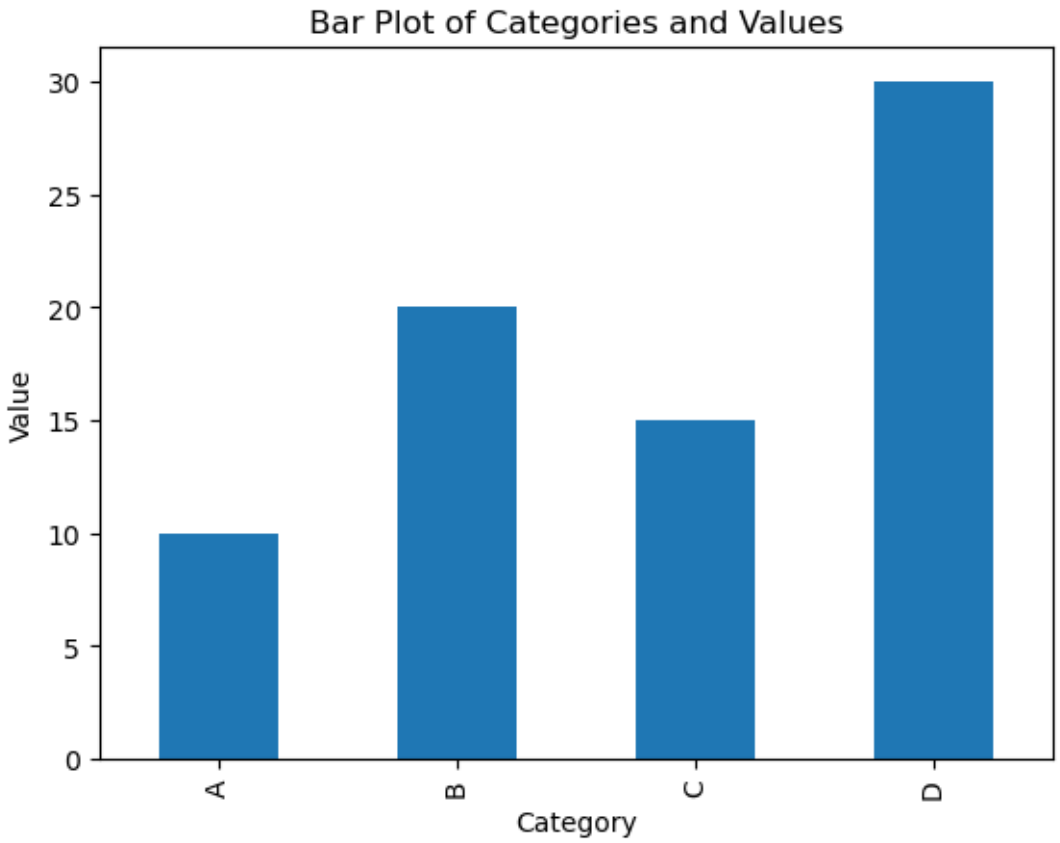
In this example, the DataFrame contains two columns: 'Category' and 'Value'. The `plot()` function is used to create a bar plot with 'Category' on the x-axis and 'Value' on the y-axis. Adjust the labeling and styling as needed to customize the appearance of the plot.

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt

# Sample DataFrame
data = {
    'Category': ['A', 'B', 'C', 'D'],
    'Value': [10, 20, 15, 30]
}

df = pd.DataFrame(data)

# Create a bar plot
df.plot(kind='bar', x='Category', y='Value', legend=False)
plt.xlabel('Category')
plt.ylabel('Value')
plt.title('Bar Plot of Categories and Values')
plt.show()
```



1.3.5 Interactive zoom in plots

N.B. I only tried the Python code from the following example (and similar) in Jupyter.

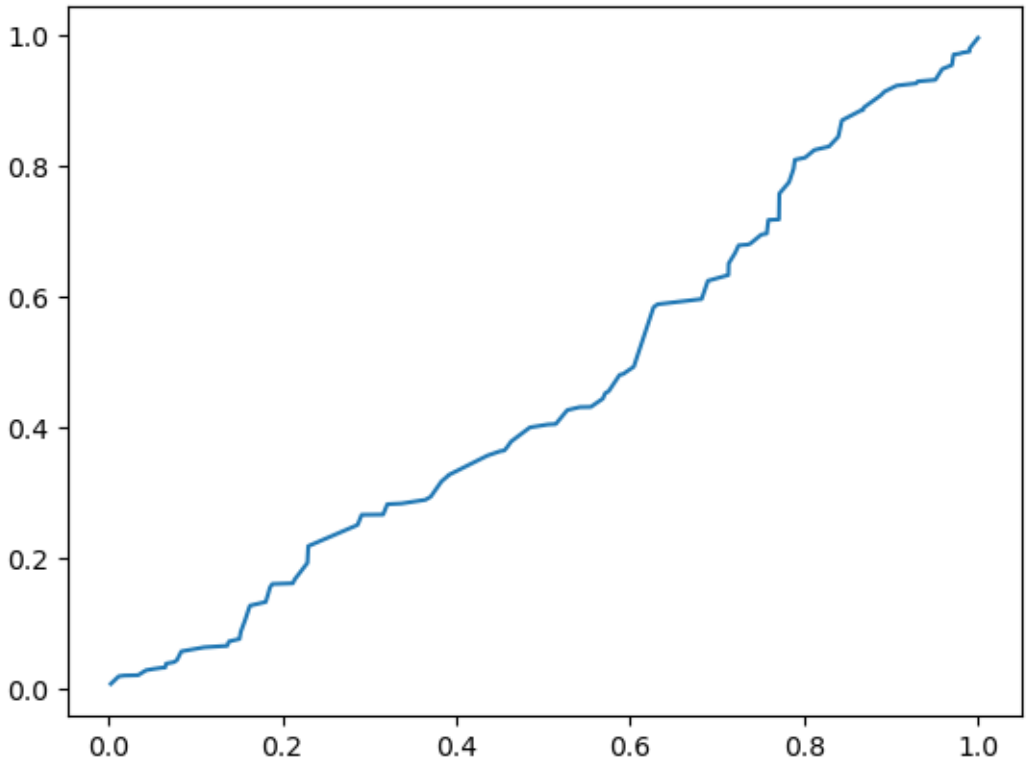
It is possible to zoom when inline plots are activated, through `mpld3` package. First, install it with `!pip install mpld3`, then add to the notebook:

[2]:
```
%matplotlib inline
import mpld3
mpld3.enable_notebook()
```

Here's an example:

[3]:
```
import matplotlib.pyplot as plt
import numpy as np
df = np.random.rand(100, 100)
df.sort()
plt.plot(df[0], df[1])
```

[3]:
```
[<matplotlib.lines.Line2D at 0x1950b361ca0>]
```



Source: StackOverflow ipython notebook –pylab inline: zooming of a plot.

1.3.6 More interactive zoom in plots

N.B. I only tried the Python code from the following example (and similar) in Jupyter.

The `plotly` library, which provides more advanced interactive features including zooming and panning. Here's an example of how you can achieve this using `plotly`:

```
import numpy as np
import plotly.express as px

# Generate some sample data
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Create a plotly figure
fig = px.line(x=x, y=y)

# Update the layout to enable zooming and panning
fig.update_layout(
    xaxis=dict(
        rangeselector=dict(
            buttons=list([
```

```
                dict(count=1, label="1d", step="day", stepmode="backward"),
                dict(count=7, label="1w", step="day", stepmode="backward"),
                dict(count=1, label="1m", step="month", stepmode="backward"),
                dict(step="all")
            ])
        ),
        rangeslider=dict(visible=True),
        type="date"
    )
)

# Display the interactive plot
fig.show()
```

In this example, we use `plotly.express` to create a line plot, and then we use the `update_layout` method to configure the x-axis with a range selector and rangeslider for zooming and panning interactions.

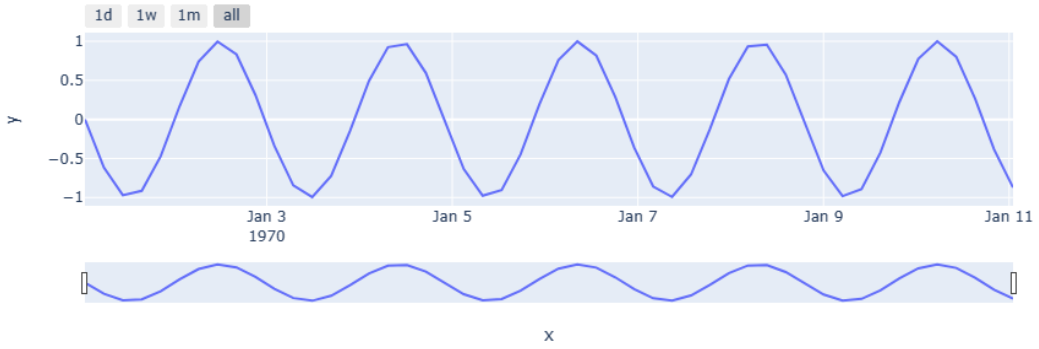
[4]:
```
import numpy as np
import plotly.express as px

# Generate some sample data
x = np.linspace(0, 100*60*60*24*100)
y = np.sin(x)

# Create a plotly figure
fig = px.line(x=x, y=y)

# Update the layout to enable zooming and panning
fig.update_layout(
    xaxis=dict(
        rangeselector=dict(
            buttons=list([
                dict(count=1, label="1d", step="day", stepmode="backward"),
                dict(count=7, label="1w", step="day", stepmode="backward"),
                dict(count=1, label="1m", step="month", stepmode="backward"),
                dict(step="all")
            ])
        ),
        rangeslider=dict(visible=True),
        type="date"
    )
)

# Display the interactive plot
fig.show()
```



1.4 Lists

1.4.1 Remove duplicates

To remove duplicates from a Python list while preserving the order of the original list, you can use a variety of approaches. Here are a few methods:

1. Using a Loop and a New List (Order Preserved):

```
original_list = [3, 1, 2, 2, 4, 3, 5, 4]
unique_list = []
for item in original_list:
    if item not in unique_list:
        unique_list.append(item)
```

2. Using a Set (Order Not Preserved):

```
original_list = [3, 1, 2, 2, 4, 3, 5, 4]
unique_list = list(set(original_list))
```

3. Using List Comprehension (Order Preserved):

```
original_list = [3, 1, 2, 2, 4, 3, 5, 4]
unique_list = []
[unique_list.append(x) for x in original_list if x not in unique_list]
```

4. Using the `collections.OrderedDict` (Order Preserved):

```
from collections import OrderedDict
original_list = [3, 1, 2, 2, 4, 3, 5, 4]
unique_list = list(OrderedDict.fromkeys(original_list))
```

Each of these methods can be used to remove duplicates from a list, and the choice of method depends on whether you need to preserve the order of the original list or not. If order preservation is essential, options 1 and 3 are preferred.

Here's how you can use these methods to remove duplicates from your list. Replace `original_list` with your list of values:

```python
# Method 1: Loop and New List (Order Preserved)
unique_list = []
for item in original_list:
    if item not in unique_list:
        unique_list.append(item)

# Method 2: Set (Order Not Preserved)
unique_list = list(set(original_list))

# Method 3: List Comprehension (Order Preserved)
unique_list = []
[unique_list.append(x) for x in original_list if x not in unique_list]

# Method 4: Ordered Dictionary (Order Preserved)
from collections import OrderedDict
unique_list = list(OrderedDict.fromkeys(original_list))
```

You can choose the method that best fits your specific requirements.

1.5 Miscellanea

1.5.1 Tuples from string

To transform a string representation of a tuple into an actual tuple, you can use the `ast.literal_eval()` function from the `ast` module. Here's an example:

```python
import ast

string_tuple = "('temperature_2m', 'max')"
tuple_value = ast.literal_eval(string_tuple)

print(tuple_value)
```

Output:

```
('temperature_2m', 'max')
```

The `ast.literal_eval()` function safely evaluates the given string as a Python expression and returns the corresponding object. In this case, it converts the string representation of the tuple into an actual tuple object.

1.5.2 Print and log

In this approach, we define a custom `log` function that takes care of both printing messages to the console and saving them to a file. This should avoid recursion and attribute errors.

```python
# Define the file where you want to save the output
output_file = 'output.txt'

# Define a function to log messages to both console and a file
def log(message):
    with open(output_file, 'a') as file:
```

```
        print(message)
        file.write(message + '\n')

# Now, use the log function to print your messages
log("This will be printed to the console and saved in 'output.txt.'")
```

You can use this `log` function throughout your script to manage the output as needed.

1.6 External resources

Follows a list of interesting blog posts and papers. Most of them are from Re-thought.

- Add new columns in a dataframe in pandas

2 Errors and subtleties to pay attention to

2.1 DateOffset for months

Be sure to write `months` as parameter and not `month` (note the final *s*). Otherwise it will do something unexpected withouth raising any exception.

Correct code:

```
import pandas as pd

# Create a date
date = pd.to_datetime('2023-07-19')

# Add a month to the date using pd.DateOffset
one_month_offset = pd.DateOffset(months=1)
new_date = date + one_month_offset

print("Original Date:", date)
print("New Date (After Adding One Month):", new_date)
```

2.2 Outer join

Inexplicably the following instruction won't work (i.e. it will produce a `merged_df` different from the one we expect - different number of rows):

```
merged_df = df1.join(df2, on=['latitude', 'longitude', 'ts'], how='outer')
```

A possible workaround was to extend df1 and df2 and then join them together with

```
merged_df = df1.join(df2, on=['latitude', 'longitude', 'ts'], how='left')
```

Note: in the example my `df1` was indexed by `('latitude', 'longitude', 'ts', 'code')` while my `df2` was indexed by `('latitude', 'longitude', 'ts')`. Apparently this shouldn't cause bizarre behaviour as it does.

3 Data augmentation

Warning on data augmentation for numeric data

Data augmentation techniques are commonly used in machine learning tasks, particularly in image or text data where transformations like rotations, translations, or flipping can be applied. However, for numeric data in a DataFrame, traditional data augmentation techniques may not be applicable. Numeric data typically involves continuous variables, and applying random transformations may not make sense or could introduce unintended bias.

Instead of traditional data augmentation, you can consider other techniques for working with numeric data, such as:

1. Scaling and Normalization: You can apply scaling techniques like min-max scaling or standardization to normalize the numeric features within a specific range or with zero mean and unit variance.
2. Feature Engineering: You can create additional features by performing mathematical operations on existing columns or by deriving statistical features like mean, median, or percentiles.
3. Sampling Techniques: You can use sampling techniques like oversampling or undersampling to balance imbalanced datasets.
4. Outlier Detection and Handling: Identify and handle outliers in your numeric data using techniques such as Z-score, percentile-based methods, or using machine learning algorithms.
5. Dimensionality Reduction: Apply dimensionality reduction techniques like Principal Component Analysis (PCA) or t-SNE to reduce the number of dimensions in your numeric data while preserving important information.

These techniques are more suited for numeric data and can help enhance your analysis or modeling tasks.

3.1 Data augmentation techniques

If you need to generate additional data to increase the size of your dataset, you can consider the following techniques for data augmentation with numeric data:

1. Random Noise: Add random noise to your numeric data by introducing small random variations to the existing values. This can help create variations in the data while preserving the underlying patterns.
2. Data Smoothing: Apply smoothing techniques such as moving averages or rolling averages to smooth out the data and create additional data points.
3. Interpolation: Use interpolation methods such as linear interpolation or spline interpolation to fill in missing values or generate new data points between existing data points.
4. Bootstrap Sampling: Apply bootstrap sampling to generate new samples by randomly selecting data points from your existing dataset with replacement. This can help create variations and increase the dataset size.
5. Synthetic Data Generation: Use techniques like generative models or simulation to generate synthetic data that closely resembles the original data distribution. This can be particularly useful when you have prior knowledge about the underlying data distribution.

It's important to note that data augmentation for numeric data may require careful consideration of the specific characteristics and properties of your data. It's recommended to experiment with different augmentation techniques and evaluate the impact on your specific task or analysis. Additionally, make sure to validate the augmented data to ensure it reflects the desired properties and maintains the integrity of the original data.

3.1.1 Random noise

Here's an example of applying random noise to a DataFrame for data augmentation:

```
import pandas as pd
import numpy as np

# Create a sample DataFrame
data = {'A': [10, 20, 30, 40, 50],
        'B': [1.5, 2.5, 3.5, 4.5, 5.5]}
df = pd.DataFrame(data)

# Define the noise level
noise_level = 0.1  # Adjust this value to control the magnitude of noise

# Add random noise to the DataFrame
noise = np.random.normal(scale=noise_level, size=df.shape)
df_augmented = df + noise

# Print the original and augmented DataFrames
print("Original DataFrame:")
print(df)
print("\nAugmented DataFrame:")
print(df_augmented)
```

In this example, random noise is generated using NumPy's `random.normal()` function with a specified noise level (`noise_level`). The noise is added to each element of the DataFrame `df` using element-wise addition, resulting in the augmented DataFrame `df_augmented`. The magnitude of the noise is controlled by adjusting the `noise_level` value.

By applying random noise, the augmented DataFrame introduces small random variations to the original data, creating additional data points with slight variations. This can help increase the size of the dataset and add variability to the existing data.

3.1.2 Bootstrap sampling

Bootstrap sampling is a technique used for data augmentation where samples are drawn from the original dataset with replacement to create new synthetic samples. Here's an example of how you can perform bootstrap sampling on a DataFrame using pandas:

```
import pandas as pd
import numpy as np

# Create a sample DataFrame
```

```
data = {'Column1': [1, 2, 3, 4, 5]}
df = pd.DataFrame(data)

# Perform bootstrap sampling
n_samples = 10  # Number of synthetic samples to generate
sampled_df = df.sample(n=n_samples, replace=True)

# Print the sampled DataFrame
print(sampled_df)
```

In this example, the DataFrame `df` contains a single column `'Column1'` with numeric values. By using the `sample` method in pandas with the `replace=True` parameter, bootstrap sampling is performed to generate `n_samples` (in this case, 10) synthetic samples. Each synthetic sample is drawn from the original DataFrame `df` with replacement. The resulting `sampled_df` DataFrame contains the synthetic samples generated through bootstrap sampling.

3.1.3 Interpolation

Interpolation is a common technique used for data augmentation, especially for time series or continuous data. Here's an example of how you can perform linear interpolation on a DataFrame using pandas:

```
import pandas as pd
import numpy as np

# Create a sample DataFrame with missing values
data = {'Time': [1, 2, 4, 7],
        'Value': [10, np.nan, 30, 40]}
df = pd.DataFrame(data)

# Perform linear interpolation
interpolated_df = df.interpolate()

# Print the interpolated DataFrame
print(interpolated_df)
```

In this example, the DataFrame `df` has two columns: `'Time'` and `'Value'`. The `'Value'` column contains missing values represented as `np.nan`. By using the `interpolate` method in pandas, linear interpolation is performed to fill in the missing values. The resulting `interpolated_df` DataFrame contains the original data with missing values replaced by interpolated values based on the surrounding data points.

Note that there are different interpolation methods available in pandas, such as linear, quadratic, cubic, etc. You can specify the desired interpolation method by passing the `method` parameter to the `interpolate` method.

4 Forecasting models

4.1 General considerations

Sample size as optimisation problem

- Bigger samples are better.
- Sample size is often determined by pragmatic considerations.
- Sample size should be seen as one consideration in an optimisation problem where the cost in time, money, effort, and so on of obtaining additional participants is weighed against the benefits of having additional participants.

A Rough Rule of Thumb

In terms of very rough rules of thumb within the typical context of observational psychological studies involving things like ability tests, attitude scales, personality measures, and so forth, I sometimes think of:

- n=100 as adequate
- n=200 as good
- n=400+ as great

These rules of thumb are grounded in the 95% confidence intervals associated with correlations at these respective levels and the degree of precision that I'd like to theoretically understand the relations of interest. However, it is only a heuristic.

Sources: https://stats.stackexchange.com/questions/10079/rules-of-thumb-for-minimum-sample-size-for-multiple-regression

4.2 Regression

4.2.1 Fitting different models

Here's an example of fitting different models using scikit-learn for a regression problem:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df[features], df[target],
                                                    test_size=0.2, random_state=42)

# Linear Regression
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)
linear_preds = linear_model.predict(X_test)
linear_mse = mean_squared_error(y_test, linear_preds)
print("Linear Regression MSE:", linear_mse)

# Decision Tree Regressor
```

```
tree_model = DecisionTreeRegressor()
tree_model.fit(X_train, y_train)
tree_preds = tree_model.predict(X_test)
tree_mse = mean_squared_error(y_test, tree_preds)
print("Decision Tree MSE:", tree_mse)

# Random Forest Regressor
forest_model = RandomForestRegressor()
forest_model.fit(X_train, y_train)
forest_preds = forest_model.predict(X_test)
forest_mse = mean_squared_error(y_test, forest_preds)
print("Random Forest MSE:", forest_mse)
```

In this example, we're fitting three different models: Linear Regression, Decision Tree Regressor, and Random Forest Regressor. For each model, we calculate the mean squared error (MSE) between the predicted values (`y_pred`) and the actual target values (`y_test`). The model with the lowest MSE is considered the best performer for this particular evaluation metric.

Note that you'll need to replace `df[features]` and `df[target]` with your actual feature columns and target variable from your DataFrame. Additionally, make sure to import the necessary modules and preprocess your data as required before fitting the models.

Cross-validation Here's an example using cross-validation with different models:

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor

# Assuming 'X' is your feature data and 'y' is your target variable

# Create a list of different models to try
models = [
    LinearRegression(),
    DecisionTreeRegressor(),
    RandomForestRegressor()
]

# Perform cross-validation for each model
for model in models:
    scores = cross_val_score(model, X, y, cv=5, scoring='r2')
    mean_score = scores.mean()
    print(f"{type(model).__name__}: R2 Score = {mean_score}")
```

In this example, we create a list of three different models: `LinearRegression`, `DecisionTreeRegressor`, and `RandomForestRegressor`. We then iterate over each model, perform 5-fold cross-validation using the `cross_val_score` function, and calculate the mean R2 score as the evaluation metric. The code outputs the model's name along with the mean R2 score.

You can customize the models, evaluation metric, and cross-validation settings based on your

specific requirements.

4.3 Tuning

4.3.1 GridSearchCV

Here's an example of how you can use `GridSearchCV` with `RandomForestRegressor`, `ExtraTreesRegressor`, and `GradientBoostingRegressor`:

```
from sklearn.ensemble import RandomForestRegressor, ExtraTreesRegressor,
                             GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import load_boston

# Load the Boston housing dataset
boston = load_boston()
X = boston.data
y = boston.target

# Define the parameter grids for each regressor
param_grid_rf = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10]
}

param_grid_et = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10]
}

param_grid_gb = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7]
}

# Create the GridSearchCV objects for each regressor
grid_search_rf = GridSearchCV(RandomForestRegressor(), param_grid_rf, cv=5,
                              scoring='neg_mean_squared_error')
grid_search_et = GridSearchCV(ExtraTreesRegressor(), param_grid_et, cv=5,
                              scoring='neg_mean_squared_error')
grid_search_gb = GridSearchCV(GradientBoostingRegressor(), param_grid_gb, cv=5,
                              scoring='neg_mean_squared_error')

# Fit the models
grid_search_rf.fit(X, y)
grid_search_et.fit(X, y)
```

```
grid_search_gb.fit(X, y)

# Get the best parameters and best score for each regressor
print("Random Forest Best Parameters:", grid_search_rf.best_params_)
print("Random Forest Best Score:", grid_search_rf.best_score_)
print("Random Forest Best Estimator:", grid_search_rf.best_estimator_)
print()
print("Extra Trees Best Parameters:", grid_search_et.best_params_)
print("Extra Trees Best Score:", grid_search_et.best_score_)
print("Extra Trees Best Estimator:", grid_search_et.best_estimator_)
print()
print("Gradient Boosting Best Parameters:", grid_search_gb.best_params_)
print("Gradient Boosting Best Score:", grid_search_gb.best_score_)
print("Gradient Boosting Best Estimator:", grid_search_gb.best_estimator_)
```

This code performs a grid search with cross-validation to find the best hyperparameters for each regressor (`RandomForestRegressor`, `ExtraTreesRegressor`, and `GradientBoostingRegressor`). It uses the Boston housing dataset as an example, but you can replace `X` and `y` with your own dataset.

4.4 General examples of forecasting models

Forecasting model for a quantity named FG. I want to use GradientBoostingRegressor.

Using GradientBoostingRegressor from the sklearn package is a good choice for building a forecasting model. To avoid shuffling data before splitting into a test and validation set, you can use a time-based splitting approach. Here's a suggested workflow for building your forecasting model:

1. Load your data: Make sure your data is indexed over time, with the target variable FG and any relevant features.
2. Split the data: Instead of shuffling the data, split it into a training set and a holdout set (validation/test set) based on a specific time cutoff. For example, you can use the first 80% of the data for training and the remaining 20% for validation. Ensure that the training data comes before the validation data in terms of time.
3. Preprocess the data: Perform any necessary data preprocessing steps such as handling missing values, scaling numerical features, and encoding categorical variables. Remember to apply these preprocessing steps separately to the training and validation sets, using only the training set to fit any transformations (e.g., mean and standard deviation for scaling).
4. Define and train the model: Import the GradientBoostingRegressor from sklearn. Define your model by setting the appropriate hyperparameters, such as the number of estimators, maximum depth, learning rate, etc. Fit the model using the training data.
5. Evaluate the model: Once the model is trained, use it to make predictions on the validation set. Calculate the evaluation metrics of interest (e.g., mean squared error, mean absolute error) to assess the model's performance. You can also visualize the predicted values against the actual values to gain further insights.
6. Model tuning: If the initial performance is not satisfactory, you can try tuning the hyperparameters of the GradientBoostingRegressor model. This can be done through techniques like

grid search or random search, optimizing for the evaluation metrics you care about.

7. Final model evaluation: Once you are satisfied with the model's performance on the validation set, you can assess its generalization on unseen data by using the holdout set (test set) that was set aside earlier. Repeat the preprocessing steps on the holdout set, followed by making predictions using the trained model. Evaluate the model's performance on the test set using the same evaluation metrics as before.

By following this workflow, you can build a forecasting model using GradientBoostingRegressor that takes into account the temporal nature of your data and avoid shuffling the data during the splitting process.

Here's an example implementation of the workflow described earlier using GradientBoostingRegressor from the sklearn package:

```
import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.model_selection import TimeSeriesSplit
from sklearn.preprocessing import StandardScaler

# Step 1: Load your data
data = pd.read_csv('your_data.csv')
# Assuming your data has columns 'FG' as the target variable and other relevant features

# Step 2: Split the data
train_ratio = 0.8  # 80% for training, 20% for validation
train_size = int(len(data) * train_ratio)
train_data = data.iloc[:train_size]
val_data = data.iloc[train_size:]

# Step 3: Preprocess the data
# Separate target variable and features
X_train = train_data.drop('FG', axis=1)
y_train = train_data['FG']
X_val = val_data.drop('FG', axis=1)
y_val = val_data['FG']

# Scale numerical features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Step 4: Define and train the model
model = GradientBoostingRegressor(n_estimators=100, max_depth=3, learning_rate=0.1)

# Step 5: Evaluate the model
tscv = TimeSeriesSplit(n_splits=3)
for train_index, val_index in tscv.split(X_train_scaled):
    X_train_cv, X_val_cv = X_train_scaled[train_index], X_train_scaled[val_index]
```

```
    y_train_cv, y_val_cv = y_train.iloc[train_index], y_train.iloc[val_index]
    model.fit(X_train_cv, y_train_cv)
    y_pred_train = model.predict(X_train_cv)
    y_pred_val = model.predict(X_val_cv)

    # Calculate evaluation metrics
    mse_train = mean_squared_error(y_train_cv, y_pred_train)
    mae_train = mean_absolute_error(y_train_cv, y_pred_train)
    mse_val = mean_squared_error(y_val_cv, y_pred_val)
    mae_val = mean_absolute_error(y_val_cv, y_pred_val)

    print(f"Training MSE: {mse_train:.2f}")
    print(f"Training MAE: {mae_train:.2f}")
    print(f"Validation MSE: {mse_val:.2f}")
    print(f"Validation MAE: {mae_val:.2f}")

# Step 6: Model tuning (optional)
# If the initial performance is not satisfactory, you can tune the model's hyperparameters

# Step 7: Final model evaluation
# Once you are satisfied with the model's performance on the validation set,
# evaluate it on the test set
model.fit(X_train_scaled, y_train)
y_pred_test = model.predict(X_val_scaled)

mse_test = mean_squared_error(y_val, y_pred_test)
mae_test = mean_absolute_error(y_val, y_pred_test)

print(f"Test MSE: {mse_test:.2f}")
print(f"Test MAE: {mae_test:.2f}")
```

Please note that this is a basic implementation, and you may need to modify it to suit your specific dataset and requirements. Additionally, the code assumes that you have your data stored in a CSV file named 'your_data.csv'. Make sure to replace it with the actual file path or modify the code accordingly if your data is in a different format.

A TimeSeriesSplit with 3 splits is used to perform cross-validation on the training data (Step 5). The model is trained and evaluated on each split separately. Finally, the model is trained on the entire training set, and its performance is evaluated on the test set (Step 7).

4.4.1 Test multiple models simultaneously

Here's an adapted version of the code that allows you to test multiple models simultaneously on the same data:

```
import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error
```

```
from sklearn.model_selection import TimeSeriesSplit
from sklearn.preprocessing import StandardScaler

# Step 1: Load your data
data = pd.read_csv('your_data.csv')
# Assuming your data has columns 'FG' as the target variable and other relevant features

# Step 2: Split the data
train_ratio = 0.8  # 80% for training, 20% for validation
train_size = int(len(data) * train_ratio)
train_data = data.iloc[:train_size]
val_data = data.iloc[train_size:]

# Step 3: Preprocess the data
# Separate target variable and features
X_train = train_data.drop('FG', axis=1)
y_train = train_data['FG']
X_val = val_data.drop('FG', axis=1)
y_val = val_data['FG']

# Scale numerical features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Step 4: Define and train the models
models = {
    'GradientBoosting': GradientBoostingRegressor(n_estimators=100, max_depth=3,
                                                  learning_rate=0.1),
    'LinearRegression': LinearRegression()
}

# Step 5: Evaluate the models
tscv = TimeSeriesSplit(n_splits=3)
for model_name, model in models.items():
    print(f"Evaluating {model_name}...")
    for train_index, val_index in tscv.split(X_train_scaled):
        X_train_cv, X_val_cv = X_train_scaled[train_index], X_train_scaled[val_index]
        y_train_cv, y_val_cv = y_train.iloc[train_index], y_train.iloc[val_index]
        model.fit(X_train_cv, y_train_cv)
        y_pred_train = model.predict(X_train_cv)
        y_pred_val = model.predict(X_val_cv)

        # Calculate evaluation metrics
        mse_train = mean_squared_error(y_train_cv, y_pred_train)
        mae_train = mean_absolute_error(y_train_cv, y_pred_train)
        mse_val = mean_squared_error(y_val_cv, y_pred_val)
        mae_val = mean_absolute_error(y_val_cv, y_pred_val)
```

```
        print(f"Training MSE: {mse_train:.2f}")
        print(f"Training MAE: {mae_train:.2f}")
        print(f"Validation MSE: {mse_val:.2f}")
        print(f"Validation MAE: {mae_val:.2f}")

    # Step 7: Final model evaluation
    # Once you are satisfied with the model's performance on the validation set,
    # evaluate it on the test set
    model.fit(X_train_scaled, y_train)
    y_pred_test = model.predict(X_val_scaled)

    mse_test = mean_squared_error(y_val, y_pred_test)
    mae_test = mean_absolute_error(y_val, y_pred_test)

    print(f"Test MSE: {mse_test:.2f}")
    print(f"Test MAE: {mae_test:.2f}")
    print("----------------------------")
```

In this adapted version, a dictionary named `models` is created to store multiple models that you want to test. Each model is associated with a name as the key. The code then iterates over each model, fitting and evaluating it using the TimeSeriesSplit cross-validation. Finally, the model performance is evaluated on the test set.

Feel free to add or modify the models in

the `models` dictionary according to your requirements.

4.4.2 Repeat test for a time range

To repeat the forecast for different days within a pandas `date_range`, you can modify the code as follows:

```
import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.model_selection import TimeSeriesSplit
from sklearn.preprocessing import StandardScaler

# Step 1: Load your data
data = pd.read_csv('your_data.csv')
# Assuming your data has columns 'FG' as the target variable and other relevant features

# Step 2: Split the data
train_ratio = 0.8  # 80% for training, 20% for validation
train_size = int(len(data) * train_ratio)
train_data = data.iloc[:train_size]
val_data = data.iloc[train_size:]

# Step 3: Preprocess the data
```

```
# Separate target variable and features
X_train = train_data.drop('FG', axis=1)
y_train = train_data['FG']
X_val = val_data.drop('FG', axis=1)
y_val = val_data['FG']

# Scale numerical features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Step 4: Define and train the model
model = GradientBoostingRegressor(n_estimators=100, max_depth=3, learning_rate=0.1)
model.fit(X_train_scaled, y_train)

# Step 5: Evaluate the model
tscv = TimeSeriesSplit(n_splits=3)
for train_index, val_index in tscv.split(X_train_scaled):
    X_train_cv, X_val_cv = X_train_scaled[train_index], X_train_scaled[val_index]
    y_train_cv, y_val_cv = y_train.iloc[train_index], y_train.iloc[val_index]
    model.fit(X_train_cv, y_train_cv)
    y_pred_train = model.predict(X_train_cv)
    y_pred_val = model.predict(X_val_cv)

    # Calculate evaluation metrics
    mse_train = mean_squared_error(y_train_cv, y_pred_train)
    mae_train = mean_absolute_error(y_train_cv, y_pred_train)
    mse_val = mean_squared_error(y_val_cv, y_pred_val)
    mae_val = mean_absolute_error(y_val_cv, y_pred_val)

    print(f"Training MSE: {mse_train:.2f}")
    print(f"Training MAE: {mae_train:.2f}")
    print(f"Validation MSE: {mse_val:.2f}")
    print(f"Validation MAE: {mae_val:.2f}")

# Step 6: Model tuning (optional)
# If the initial performance is not satisfactory, you can tune the model's hyperparameters

# Step 7: Final model evaluation
# Once you are satisfied with the model's performance on the validation set,
# evaluate it on the test set
model.fit(X_train_scaled, y_train)
y_pred_test = model.predict(X_val_scaled)

mse_test = mean_squared_error(y_val, y_pred_test)
mae_test = mean_absolute_error(y_val, y_pred_test)

print(f"Test MSE: {mse_test:.2f}")
```

```
print(f"Test MAE: {mae_test:.2f}")

# Step 8: Repeat forecast for different days in a pandas date_range
forecast_dates = pd.date_range(start='2023-06-17', end='2023-06-30', freq='D')
for date in forecast_dates:
    # Preprocess the features for the forecast date
    # ... (preprocess the features for the forecast date accordingly)

    # Make predictions for the forecast date
    # ... (apply necessary transformations to the forecast features, e.g., scaling

)
    forecast_features = scaler.transform(forecast_features)
    # Assuming you have preprocessed the forecast features

    forecast = model.predict(forecast_features)
    print(f"Forecast for {date}: {forecast}")
```

In Step 8, you define a `forecast_dates` range using `pd.date_range` with the desired start and end dates and the desired frequency (e.g., daily, with `'D'`). Then, for each date in the `forecast_dates` range, you preprocess the features specific to that forecast date and make predictions using the trained model. The forecasted values are printed for each forecast date. Make sure to adjust the preprocessing and transformation steps for the forecast features to suit your specific requirements.

4.4.3 More models and data range

Apologies for the confusion. If you have multiple models and want to repeat the forecast for different days within a pandas `date_range`, you can modify the code as follows:

```
import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.model_selection import TimeSeriesSplit
from sklearn.preprocessing import StandardScaler

# Step 1: Load your data
data = pd.read_csv('your_data.csv')
# Assuming your data has columns 'FG' as the target variable and other relevant features

# Step 2: Split the data
train_ratio = 0.8  # 80% for training, 20% for validation
train_size = int(len(data) * train_ratio)
train_data = data.iloc[:train_size]
val_data = data.iloc[train_size:]

# Step 3: Preprocess the data
# Separate target variable and features
X_train = train_data.drop('FG', axis=1)
```

```
y_train = train_data['FG']
X_val = val_data.drop('FG', axis=1)
y_val = val_data['FG']

# Scale numerical features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Step 4: Define the models
models = {
    'GradientBoosting': GradientBoostingRegressor(n_estimators=100, max_depth=3,
                                                  learning_rate=0.1),
    'LinearRegression': LinearRegression()
}

# Step 5: Train the models
for model_name, model in models.items():
    model.fit(X_train_scaled, y_train)

# Step 6: Evaluate the models
tscv = TimeSeriesSplit(n_splits=3)
for model_name, model in models.items():
    print(f"Evaluating {model_name}...")
    for train_index, val_index in tscv.split(X_train_scaled):
        X_train_cv, X_val_cv = X_train_scaled[train_index], X_train_scaled[val_index]
        y_train_cv, y_val_cv = y_train.iloc[train_index], y_train.iloc[val_index]
        model.fit(X_train_cv, y_train_cv)
        y_pred_train = model.predict(X_train_cv)
        y_pred_val = model.predict(X_val_cv)

        # Calculate evaluation metrics
        mse_train = mean_squared_error(y_train_cv, y_pred_train)
        mae_train = mean_absolute_error(y_train_cv, y_pred_train)
        mse_val = mean_squared_error(y_val_cv, y_pred_val)
        mae_val = mean_absolute_error(y_val_cv, y_pred_val)

        print(f"Training MSE: {mse_train:.2f}")
        print(f"Training MAE: {mae_train:.2f}")
        print(f"Validation MSE: {mse_val:.2f}")
        print(f"Validation MAE: {mae_val:.2f}")

    # Step 7: Final model evaluation
    # Once you are satisfied with the model's performance on the validation set,
    # evaluate it on the test set
    model.fit(X_train_scaled, y_train)
    y_pred_test = model.predict(X_val_scaled)
```

```
    mse_test = mean_squared_error(y_val, y_pred_test)
    mae_test = mean_absolute_error(y_val, y_pred_test)

    print(f"Test MSE: {mse_test:.2f}")
    print(f"Test MAE: {mae_test:.2f}")
    print("----------------------------")

# Step 8: Repeat forecast for different days in a pandas date_range
forecast_dates = pd.date_range(start='2023-06-17', end='2023-06-30',

 freq='D')
for date in forecast_dates:
    for model_name, model in models.items():
        # Preprocess the features for the forecast date
        # ... (preprocess the features for the forecast date accordingly)

        # Make predictions for the forecast date
        # ... (apply necessary transformations to the forecast features, e.g., scaling)
        forecast_features = scaler.transform(forecast_features)
        # Assuming you have preprocessed the forecast features

        forecast = model.predict(forecast_features)
        print(f"Model: {model_name} - Forecast for {date}: {forecast}")
```

In this updated code, you define a dictionary called `models` that contains the models you want to test. Each model is associated with a name as the key. The code then trains and evaluates each model using the TimeSeriesSplit cross-validation. Finally, for each forecast date in the `forecast_dates` range, the code iterates over each model, preprocesses the features specific to that forecast date, and makes predictions using the trained model. The forecasted values are printed for each model and forecast date.

Make sure to adjust the preprocessing and transformation steps for the forecast features to suit your specific requirements.

5 Randomness tests

There are several randomness tests that you can use to assess the randomness of a time series. Here are a few commonly used ones:

- **Ljung-Box Test**: This test checks for autocorrelation in a time series. If significant autocorrelation is detected, it suggests that the time series is not random.
- **Augmented Dickey-Fuller Test**: The ADF test is used to check for the presence of a unit root in a univariate time series. It helps determine if a time series is stationary (non-random) or contains a deterministic trend.
- **Kolmogorov-Smirnov Test**: This test assesses whether the distribution of data in a time series is consistent with a known distribution (e.g., normal distribution). It can help identify departures from randomness in the data distribution.
- **Runs Test**: The runs test checks for the presence of long runs of data points above or below the mean. Deviations from randomness can be indicated by an excessive number of runs.
- **Chi-Square Test**: This test is used to check if the observed distribution of categorical data in a time series is consistent with an expected distribution. It can be used to identify patterns or deviations from randomness in categorical data.
- **Frequency Test**: In a frequency test, you analyze the distribution of values at different frequencies or periods within the time series. Deviations from expected frequencies may indicate non-randomness.
- **Spectral Analysis**: Techniques like the periodogram or spectral density estimation can be used to examine the frequency components of a time series. Unusual peaks or patterns in the spectrum may suggest non-randomness.

The choice of test depends on the nature of your time series data and the specific randomness properties you want to investigate. Different tests are suitable for different types of data and research questions. It's often a good practice to use multiple tests to gain a more comprehensive understanding of the data's randomness or lack thereof.

5.1 Examples of implementations

5.1.1 Ljung-Box Test

The Ljung-Box test is used to determine whether any of a group of autocorrelations of a time series are different from zero, indicating the presence of serial correlation in the data.

Here's how you can perform the Ljung-Box test in Python using the `statsmodels` library:

```
import numpy as np
import statsmodels.api as sm
import statsmodels.stats.diagnostic as diag
import matplotlib.pyplot as plt

# Generate a random time series for testing (replace this with your own data)
np.random.seed(0)
time_series = np.random.randn(100)
```

```
# Perform Ljung-Box test
lags = range(1, 21)  # Choose the number of lags to test
results = diag.acorr_ljungbox(time_series, lags=lags, boxpierce=True)

# Extract p-values from the results
p_values = results['lb_pvalue']

# Set a significance level (e.g., 0.05)
alpha = 0.05

# Plot p-values
plt.figure(figsize=(10, 4))
plt.plot(lags, p_values, marker='o', linestyle='-', color='blue')
plt.axhline(y=alpha, color='red', linestyle='--')
plt.xlabel('Lag')
plt.ylabel('P-Value')
plt.title('Ljung-Box Test')
plt.grid(True)
plt.show()

# Check if any p-value is less than alpha
if any(p_values < alpha):
    print("The time series is not random.")
else:
    print("The time series appears to be random.")
```

In this example, we generate a random time series and perform the Ljung-Box test for a range of lags. We then plot the p-values and check if any p-value is less than a chosen significance level (`alpha`). If any p-value is below the significance level, we conclude that the time series is not random.

You can replace the `time_series` variable with your own time series data to perform the test on your data.

5.1.2 Augmented Dickey-Fuller Test

The Augmented Dickey-Fuller (ADF) test is used to test for the presence of a unit root in a time series, which indicates whether the time series is stationary or contains a deterministic trend. Here's a Python implementation of the ADF test using the `statsmodels` library:

```
import pandas as pd
import numpy as np
import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller

# Generate or load your time series data
# For this example, let's create a random walk time series
np.random.seed(42)
n = 100
```

```
random_walk = np.cumsum(np.random.randn(n))

# Define a function to perform the ADF test
def adf_test(time_series):
    result = adfuller(time_series)
    print("ADF Statistic:", result[0])
    print("p-value:", result[1])
    print("Critical Values:")
    for key, value in result[4].items():
        print(f"{key}: {value}")

# Perform the ADF test on the random walk time series
adf_test(random_walk)
```

In this code:

1. We import the necessary libraries, including `statsmodels` for the ADF test.
2. We generate or load your time series data. For this example, we created a random walk time series.
3. We define a function `adf_test` that takes a time series as input and performs the ADF test using `adfuller` from `statsmodels`.
4. We print out the ADF Statistic, p-value, and critical values.

You can replace the `random_walk` variable with your own time series data to perform the ADF test on your dataset. The key result to interpret is the p-value. If the p-value is less than a significance level (e.g., 0.05), you can reject the null hypothesis of a unit root, indicating that the time series is stationary.

5.1.3 Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov test is a non-parametric test used to check if a dataset follows a specific probability distribution or if two datasets have the same distribution. Here's a Python implementation of the Kolmogorov-Smirnov test using the `scipy` library:

```
import numpy as np
from scipy import stats

# Generate two example datasets
np.random.seed(42)
data1 = np.random.normal(0, 1, 1000)  # Sample data 1 from a normal distribution
data2 = np.random.normal(0, 1, 1000)  # Sample data 2 from a normal distribution

# Perform the Kolmogorov-Smirnov test for two-sample comparison
ks_statistic, ks_pvalue = stats.ks_2samp(data1, data2)

print("Kolmogorov-Smirnov Statistic:", ks_statistic)
print("p-value:", ks_pvalue)

# Interpret the results
alpha = 0.05  # Significance level
```

```
if ks_pvalue < alpha:
    print("Reject the null hypothesis: The two datasets have different distributions.")
else:
    print("Fail to reject the null hypothesis: The two datasets have similar distributions.")
```

In this code:

1. We import the necessary libraries, including `numpy` and `scipy.stats`.
2. We generate two example datasets `data1` and `data2` sampled from normal distributions.
3. We perform the Kolmogorov-Smirnov test for two-sample comparison using `ks_2samp` from `scipy.stats`.
4. We print out the Kolmogorov-Smirnov statistic and the p-value.
5. We interpret the results based on a predefined significance level (`alpha`). If the p-value is less than `alpha`, we reject the null hypothesis, indicating that the two datasets have different distributions.

You can replace `data1` and `data2` with your own datasets to perform the Kolmogorov-Smirnov test for your specific data.

5.1.4 Runs Test

The Runs Test is a non-parametric statistical test used to assess the randomness of a binary or dichotomous sequence. It checks if there is a significant clustering of similar values in the sequence. Here's a Python implementation of the Runs Test:

```
import numpy as np
from scipy import stats

# Generate an example binary sequence (0s and 1s)
np.random.seed(42)
sequence = np.random.randint(0, 2, 100)

# Define the runs test function
def runs_test(sequence):
    runs = [0]  # Initialize the list of runs with 0
    for i in range(1, len(sequence)):
        if sequence[i] != sequence[i - 1]:
            runs.append(1)  # A run of a different value starts
        else:
            runs[-1] += 1  # Continue the current run

    observed_runs = len(runs)
    expected_runs = (2 * len(sequence) - 1) / 3  # Expected number of runs for a random sequen

    # Calculate the Z-score and p-value
    z_score = (observed_runs - expected_runs) / np.sqrt((2 * len(sequence) * (2 * len(sequence)
    p_value = 2 * (1 - stats.norm.cdf(abs(z_score)))  # Two-tailed test

    return z_score, p_value
```

```
# Perform the runs test
z_score, p_value = runs_test(sequence)

print("Z-Score:", z_score)
print("p-value:", p_value)

# Interpret the results
alpha = 0.05  # Significance level
if p_value < alpha:
    print("Reject the null hypothesis: The sequence is not random.")
else:
    print("Fail to reject the null hypothesis: The sequence is random.")
```

In this code:

1. We generate an example binary sequence `sequence` consisting of 0s and 1s.
2. We define a `runs_test` function that calculates the number of runs in the sequence, calculates the expected number of runs for a random sequence, and then computes the Z-score and p-value for the Runs Test.
3. We perform the Runs Test on the `sequence` and calculate the Z-score and p-value.
4. We interpret the results based on a predefined significance level (`alpha`). If the p-value is less than `alpha`, we reject the null hypothesis, indicating that the sequence is not random. Otherwise, we fail to reject the null hypothesis, suggesting that the sequence is random.

You can replace the `sequence` variable with your own binary sequence data to perform the Runs Test on your specific data.

5.1.5 Chi-Square Test

The Chi-Square Test is used to determine if there is a significant association between two categorical variables. It is often used to test the independence of two categorical variables in a contingency table. Here's how you can implement the Chi-Square Test in Python using the `scipy` library:

```
import numpy as np
from scipy.stats import chi2_contingency

# Create a contingency table (replace this with your own data)
observed_table = np.array([[10, 20, 30], [5, 15, 25]])

# Perform the Chi-Square Test
chi2, p, dof, expected = chi2_contingency(observed_table)

# Interpret the results
alpha = 0.05  # Significance level

print("Chi-Square Statistic:", chi2)
print("Degrees of Freedom:", dof)
print("p-value:", p)
```

```
if p < alpha:
    print("Reject the null hypothesis: There is a significant association between the variables
else:
    print("Fail to reject the null hypothesis: There is no significant association between the
```

In this code:

1. You need to create a contingency table (`observed_table`) with your categorical data. Each row represents one category of one variable, and each column represents one category of the other variable. Replace the values in `observed_table` with your own data.
2. We use the `chi2_contingency` function from the `scipy.stats` library to perform the Chi-Square Test. This function returns the Chi-Square statistic, p-value, degrees of freedom, and the expected frequency table.
3. We interpret the results based on a predefined significance level (`alpha`). If the p-value is less than `alpha`, we reject the null hypothesis, indicating that there is a significant association between the variables. Otherwise, we fail to reject the null hypothesis, suggesting that there is no significant association between the variables.

Make sure to replace the `observed_table` variable with your own contingency table data to perform the Chi-Square Test on your specific data.

5.1.6 Frequency Test

A frequency test is often used to check if a set of data follows an expected distribution. One common test for this purpose is the Chi-Square Goodness-of-Fit Test. Here's how you can implement it in Python:

```
import numpy as np
from scipy.stats import chi2_contingency

# Define your observed frequencies (counts)
observed_frequencies = np.array([25, 35, 40])  # Replace with your observed counts

# Define your expected frequencies (expected counts)
expected_frequencies = np.array([30, 30, 40])  # Replace with your expected counts

# Perform the Chi-Square Goodness-of-Fit Test
chi2_statistic, p_value = chi2_contingency([observed_frequencies, expected_frequencies])

# Set your significance level (alpha)
alpha = 0.05

# Interpret the results
print("Chi-Square Statistic:", chi2_statistic)
print("p-value:", p_value)

if p_value < alpha:
    print("Reject the null hypothesis: The data does not follow the expected distribution.")
```

```
else:
    print("Fail to reject the null hypothesis: The data follows the expected distribution.")
```

In this code:

- `observed_frequencies` should contain your observed counts for each category.
- `expected_frequencies` should contain your expected counts for each category under the null hypothesis.
- The Chi-Square Goodness-of-Fit Test is performed using `chi2_contingency` from `scipy.stats`.
- The Chi-Square statistic and p-value are calculated, and the results are interpreted based on a predefined significance level (`alpha`).

You can replace the values in `observed_frequencies` and `expected_frequencies` with your specific data and expected distribution to perform the test.

[ ]: