# Advanced Pandas Techniques for Machine Learning

## With Code Example

save for later

# Introduction to Advanced Data Manipulation

Data manipulation is a crucial skill in machine learning and data science. This presentation covers advanced techniques using pandas and Python, focusing on reshaping, transforming, and analyzing complex datasets. We'll explore methods like stacking, unstacking, working with MultiIndex DataFrames, and converting between long and wide data formats.

```python
import pandas as pd
import numpy as np

# Create a sample DataFrame
df = pd.DataFrame({
    'Category': ['A', 'A', 'B', 'B'],
    'Year': [2022, 2023, 2022, 2023],
    'Sales': [100, 120, 90, 110],
    'Profit': [20, 25, 18, 22]
})

print(df)
```

Output:

```
  Category  Year  Sales  Profit
0        A  2022    100      20
1        A  2023    120      25
2        B  2022     90      18
3        B  2023    110      22
```

Swipe next

# Stacking DataFrames

Stacking is the process of transforming columns into rows, creating a hierarchical index. This technique is useful when you want to reshape your data for analysis or visualization.

```python
# Stack the 'Sales' and 'Profit' columns
stacked_df = df.set_index(['Category', 'Year']).stack()

print(stacked_df)
```

Output:

```
Category  Year
A         2022  Sales     100
                Profit     20
          2023  Sales     120
                Profit     25
B         2022  Sales      90
                Profit     18
          2023  Sales     110
                Profit     22
dtype: int64
```

save for later

# Unstacking DataFrames

Unstacking is the reverse operation of stacking, transforming rows into columns. This can be useful for creating pivot tables or reorganizing data for easier analysis.

```python
# Unstack the previously stacked DataFrame
unstacked_df = stacked_df.unstack(level=-1)

print(unstacked_df)
```

Output:

```
              Profit  Sales
Category Year
A        2022      20    100
         2023      25    120
B        2022      18     90
         2023      22    110
```

Swipe next

follow for more

# MultiIndex DataFrames

MultiIndex (or hierarchical index) DataFrames allow you to work with higher-dimensional data in a two-dimensional structure. They are particularly useful for representing complex datasets with multiple levels of categorization.

```
# Create a MultiIndex DataFrame
multi_index_df = pd.DataFrame({
    ('Sales', 'Q1'): [100, 90],
    ('Sales', 'Q2'): [110, 95],
    ('Profit', 'Q1'): [20, 18],
    ('Profit', 'Q2'): [22, 19]
}, index=pd.MultiIndex.from_product([['A', 'B'], [2023]], names=['Category',
'Year']))

print(multi_index_df)
```

Output:

```
              Sales      Profit
                 Q1   Q2     Q1   Q2
Category Year
A        2023   100  110     20   22
B        2023    90   95     18   19
```

Swipe next ⟶

save for later

# Accessing Data in MultiIndex DataFrames

Working with MultiIndex DataFrames requires understanding how to access and manipulate data across different levels of the index hierarchy.

```
# Accessing data using MultiIndex
print(multi_index_df.loc['A', 2023, 'Sales'])
print(multi_index_df.loc[('A', 2023), ('Sales', 'Q1')])

# Selecting specific levels
print(multi_index_df.xs('Q1', axis=1, level=1))
```

Output:

```
Q1    100
Q2    110
Name: (A, 2023, Sales), dtype: int64

100

              Sales  Profit
Category Year
A        2023    100      20
B        2023     90      18
```

Swipe next

# Long vs. Wide Data Formats

follow for more

Data can be represented in long (narrow) or wide formats. Long format is often preferred for analysis and modeling, while wide format can be more readable for humans.

```python
# Wide format (original DataFrame)
print("Wide format:")
print(df)

# Convert to long format
long_df = df.melt(id_vars=['Category', 'Year'], var_name='Metric',
value_name='Value')
print("\nLong format:")
print(long_df)
```

Output:

```
Wide format:
  Category  Year  Sales  Profit
0        A  2022    100      20
1        A  2023    120      25
2        B  2022     90      18
3        B  2023    110      22

Long format:
  Category  Year  Metric  Value
0        A  2022   Sales    100
1        A  2023   Sales    120
2        B  2022   Sales     90
3        B  2023   Sales    110
4        A  2022  Profit     20
5        A  2023  Profit     25
6        B  2022  Profit     18
7        B  2023  Profit     22
```

Swipe next ⟶

# Melting DataFrames

save for later

The melt function is used to transform wide-format data into long-format data. This is particularly useful when preparing data for analysis or visualization that requires data in a "tidy" format.

```python
# Create a wide-format DataFrame
wide_df = pd.DataFrame({
    'Name': ['Alice', 'Bob'],
    'Math': [90, 85],
    'Science': [88, 92],
    'History': [78, 89]
})

# Melt the DataFrame
melted_df = wide_df.melt(id_vars=['Name'], var_name='Subject', value_name='Score')

print("Original wide format:")
print(wide_df)
print("\nMelted long format:")
print(melted_df)
```

Output:

```
Original wide format:
    Name  Math  Science  History
0  Alice    90       88       78
1    Bob    85       92       89

Melted long format:
    Name  Subject  Score
0  Alice     Math     90
1    Bob     Math     85
2  Alice  Science     88
3    Bob  Science     92
4  Alice  History     78
5    Bob  History     89
```

Swipe next →

# Advanced Grouping and Aggregation

save for later

Pandas provides powerful tools for grouping and aggregating data, allowing you to perform complex analyses on your datasets.

```
# Create a sample DataFrame
df = pd.DataFrame({
    'Category': ['A', 'A', 'B', 'B', 'A', 'B'],
    'Subcategory': ['X', 'Y', 'X', 'Y', 'X', 'Y'],
    'Value': [10, 15, 20, 25, 30, 35]
})

# Perform advanced grouping and aggregation
result = df.groupby(['Category', 'Subcategory']).agg({
    'Value': ['sum', 'mean', 'max']
}).reset_index()

print("Original data:")
print(df)
print("\nGrouped and aggregated data:")
print(result)
```

Output:

```
Original data:
  Category Subcategory  Value
0        A           X     10
1        A           Y     15
2        B           X     20
3        B           Y     25
4        A           X     30
5        B           Y     35

Grouped and aggregated data:
  Category Subcategory Value
                         sum mean max
0        A           X   40  20.0  30
1        A           Y   15  15.0  15
2        B           X   20  20.0  20
3        B           Y   60  30.0  35
```

Swipe next

follow for more

# Handling Missing Data

Dealing with missing data is a common task in data manipulation. Pandas offers various methods to handle missing values effectively.

```python
# Create a DataFrame with missing values
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, np.nan, 8],
    'C': [9, 10, 11, 12]
})

print("Original DataFrame:")
print(df)

# Fill missing values
df_filled = df.fillna(method='ffill')
print("\nDataFrame after forward fill:")
print(df_filled)

# Drop rows with any missing values
df_dropped = df.dropna()
print("\nDataFrame after dropping rows with missing values:")
print(df_dropped)
```

Output:

```
Original DataFrame:
     A    B   C
0  1.0  5.0   9
1  2.0  NaN  10
2  NaN  NaN  11
3  4.0  8.0  12

DataFrame after forward fill:
     A    B   C
0  1.0  5.0   9
1  2.0  5.0  10
2  2.0  5.0  11
3  4.0  8.0  12

DataFrame after dropping rows with missing values:
     A    B   C
0  1.0  5.0   9
3  4.0  8.0  12
```

Swipe next →

# Merging and Joining DataFrames

save for later

Combining data from multiple sources is a common operation in data manipulation. Pandas provides various methods to merge and join DataFrames.

```python
# Create two sample DataFrames
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [2, 3, 4], 'Age': [25, 30, 35]})

# Perform an inner join
inner_join = pd.merge(df1, df2, on='ID', how='inner')

# Perform a left join
left_join = pd.merge(df1, df2, on='ID', how='left')

print("DataFrame 1:")
print(df1)
print("\nDataFrame 2:")
print(df2)
print("\nInner Join:")
print(inner_join)
print("\nLeft Join:")
print(left_join)
```

Output:

```
DataFrame 1:
   ID     Name
0   1    Alice
1   2      Bob
2   3  Charlie

DataFrame 2:
   ID  Age
0   2   25
1   3   30
2   4   35

Inner Join:
   ID     Name  Age
0   2      Bob   25
1   3  Charlie   30

Left Join:
   ID     Name   Age
0   1    Alice   NaN
1   2      Bob  25.0
2   3  Charlie  30.0
```

# Time Series Data Manipulation

follow for more

Working with time series data is common in many data analysis tasks. Pandas provides powerful tools for manipulating and analyzing time-based data.

```python
# Create a time series DataFrame
dates = pd.date_range(start='2023-01-01', end='2023-01-10', freq='D')
ts_df = pd.DataFrame({'Date': dates, 'Value': np.random.randn(len(dates))})

# Set the Date column as the index
ts_df.set_index('Date', inplace=True)

# Resample the data to weekly frequency
weekly_data = ts_df.resample('W').mean()

print("Original daily data:")
print(ts_df)
print("\nResampled weekly data:")
print(weekly_data)
```

Output:

```
Original daily data:
               Value
Date
2023-01-01   0.354314
2023-01-02  -0.499253
2023-01-03   1.209451
2023-01-04   0.443120
2023-01-05  -0.509704
2023-01-06   0.332290
2023-01-07  -1.213095
2023-01-08  -0.278596
2023-01-09   0.332784
2023-01-10   0.580333

Resampled weekly data:
               Value
Date
2023-01-08   0.017018
2023-01-15   0.211507
```

Swipe next ⟶

# Real-Life Example: Weather Data Analysis

save for later

Let's analyze a weather dataset to demonstrate the application of advanced data manipulation techniques in a real-world scenario.

```python
# Create a sample weather dataset
dates = pd.date_range(start='2023-01-01', end='2023-12-31', freq='D')
weather_data = pd.DataFrame({
    'Date': dates,
    'Temperature': np.random.uniform(0, 30, len(dates)),
    'Humidity': np.random.uniform(30, 90, len(dates)),
    'Precipitation': np.random.uniform(0, 50, len(dates))
})

# Set Date as index
weather_data.set_index('Date', inplace=True)

# Calculate monthly averages
monthly_avg = weather_data.resample('M').mean()

# Find the hottest day of each month
hottest_days = weather_data.resample('M')['Temperature'].idxmax()

print("Monthly averages:")
print(monthly_avg.head())
print("\nHottest days of each month:")
print(hottest_days.head())
```

Output:

```
Monthly averages:
            Temperature   Humidity  Precipitation
Date
2023-01-31    14.819144  58.741148      24.776924
2023-02-28    15.326223  60.853798      23.865147
2023-03-31    14.606469  59.707280      24.705367
2023-04-30    15.402669  60.167848      25.380250
2023-05-31    15.066451  59.729830      24.911770

Hottest days of each month:
Date
2023-01-31   2023-01-24
2023-02-28   2023-02-25
2023-03-31   2023-03-29
2023-04-30   2023-04-29
2023-05-31   2023-05-27
Freq: M, Name: Temperature, dtype: datetime64[ns]
```

# Real-Life Example: Customer Segmentation

In this example, we'll demonstrate how to use advanced data manipulation techniques for customer segmentation based on purchasing behavior.

```python
import pandas as pd
import numpy as np

# Create a sample customer purchase dataset
np.random.seed(42)
n_customers = 1000
customer_data = pd.DataFrame({
    'CustomerID': range(1, n_customers + 1),
    'TotalPurchases': np.random.randint(1, 100, n_customers),
    'AvgOrderValue': np.random.uniform(10, 200, n_customers),
    'DaysSinceLastPurchase': np.random.randint(1, 365, n_customers)
})

# Perform customer segmentation
customer_data['ValueSegment'] = pd.qcut(customer_data['AvgOrderValue'], q=3,
labels=['Low', 'Medium', 'High'])
customer_data['FrequencySegment'] = pd.qcut(customer_data['TotalPurchases'], q=3,
labels=['Low', 'Medium', 'High'])
customer_data['RecencySegment'] = pd.qcut(customer_data['DaysSinceLastPurchase'],
q=3, labels=['High', 'Medium', 'Low'])

# Analyze segments
segment_analysis = customer_data.groupby(['ValueSegment', 'FrequencySegment',
'RecencySegment']).size().unstack(fill_value=0)

print("Sample of customer data:")
print(customer_data.head())
print("\nSegment analysis:")
print(segment_analysis)
```

# Output

save for later

```
Sample of customer data:
   CustomerID  TotalPurchases  AvgOrderValue  DaysSinceLastPurchase ValueSegment
FrequencySegment RecencySegment
0           1              37     124.472172                    143         High
Medium          Medium
1           2              87     118.256325                    112         High
High          Medium
2           3              52     192.379052                    343         High
High            Low
3           4              69      82.804886                    134       Medium
High          Medium
4           5              66     155.693686                    335         High
High            Low

Segment analysis:
ValueSegment FrequencySegment RecencySegment  High  Low  Medium
High         High             High              29   23      32
                              Low               39   25      35
                              Medium            38   30      32
             Low              High               5   25      18
                              Low                9   24      25
                              Medium             8   23      20
             Medium           High              16   21      30
                              Low               21   24      26
                              Medium            19   24      24
Medium       High             High              31   19      33
                              Low               34   23      32
                              Medium            29   28      26
             Low              High               8   24      22
                              Low                7   25      22
                              Medium            11   18      24
             Medium           High              14   27      25
                              Low               21   20      24
                              Medium            22   19      26
Low          High             High              24   27      26
                              Low               31   23      30
                              Medium            30   24      29
             Low              High               9   17      24
                              Low                9   25      23
                              Medium             8   26      21
             Medium           High              16   25      26
                              Low               16   22      26
                              Medium            17   24      26
```

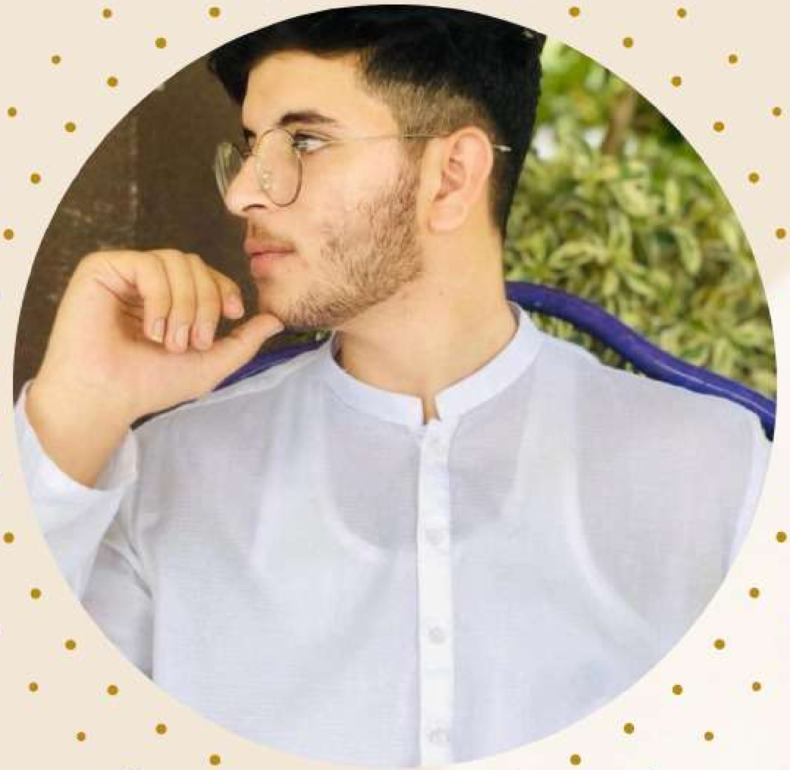
Swipe next

# Additional Resources

For further exploration of advanced data manipulation techniques using pandas and Python, consider the following resources:

1. pandas documentation: https://pandas.pydata.org/docs/
2. "Python for Data Analysis" by Wes McKinney (creator of pandas)
3. DataCamp's "Data Manipulation with pandas" course
4. Real Python's pandas tutorials: https://realpython.com/learning-paths/pandas-data-science/
5. ArXiv paper on data manipulation techniques: "A Survey on Data Preprocessing for Data Mining" (https://arxiv.org/abs/1811.05242)

Data scientist
&ML Engineer



**Follow For More Data Science Content**