College of Engineering & Applied Sciences

CSPB 3022

Introduction To Data Science With Probability And Statistics

Exam Notes

Taylor Larrechea

2024

Exam 1 Notes

Data Frames

A data frame in Pandas is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns). It's akin to a spreadsheet or SQL table and is one of the most commonly used Pandas data structures.

You can create a data frame from various sources, such as:

- Lists
- Dictionaries
- NumPy arrays
- CSV files
- SQL databases

The structure of data frames can be summed up by:

- **Rows**: Each row represents a single observation or record.
- **Columns**: Each column represents a variable or feature. Columns can be of different data types (integer, string, float, etc.).
- **Index**: This is the 'key' for rows, similar to an index in a database. It's an immutable array, allowing fast access to data.

Some operations that can be used with data frames are:

- **Data Manipulation**: Adding, deleting, and modifying both rows and columns.
- **Filtering**: Selecting a subset of rows or columns based on some criteria.
- **Sorting and Grouping**: Organizing data based on values in certain columns.
- **Merging and Joining**: Combining multiple data frames.
- **Handling Missing Data**: Identifying and imputing missing values.

Data Frame Operation Examples

Here are some examples of data frame manipulation in pandas:

Creation

```
import pandas as pd

# Creating a data frame from a dictionary
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'Los Angeles', 'Chicago']}
df = pd.DataFrame(data)

```

Adding A Column

```
# Adding a new column
df['Salary'] = [70000, 80000, 90000]

```

Deleting A Column

```
# Deleting a column
df.drop('Age', axis=1, inplace=True)

```

Filtering Data

```
# Filtering rows where Salary is greater than 75000
high_earners = df[df['Salary'] > 75000]

```

Sorting Data

```
# Sorting data by Salary in descending order
df_sorted = df.sort_values(by='Salary', ascending=False)

```

Merging Data Frames

```
# Creating another data frame
additional_data = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Experience': [5, 10]})

# Merging data frames
merged_df = pd.merge(df, additional_data, on='Name', how='left')

```

Handling Missing Data

```
# Filling missing values with zero
df_filled = df.fillna(0)

```

Reading Data

```
# Reading data from a CSV file
df_from_csv = pd.read_csv('data.csv')

# Writing data to a CSV file
df.to_csv('output.csv', index=False)

```

Combinatorics

Combinatorics is a branch of mathematics dealing with the study of countable, discrete structures and their properties. It's particularly important in computer science, where understanding how to count and arrange objects is crucial for algorithm design, data structure optimization, and problem-solving. Here's a summary of the key concepts in combinatorics:

- **Counting Principles**
  - **The Rule of Sum**: If there are A ways to do something and B ways to do another thing, and these two things cannot happen at the same time, then there are $A + B$ ways to choose one of these actions.
  - **The Rule of Product**: If there are A ways to do something and B ways to do another thing after that, then there are $A \cdot B$ ways to perform both actions.

- **Permutations**
  - Permutations are the arrangements of objects in a specific order.
  - The number of permutations of n distinct objects is $n!$ (n factorial), which is the product of all positive integers up to n.
  - For arranging r objects out of n available objects, the formula is

$$^nP_r = \frac{n!}{(n-r)!}.$$

- **Combinations**
  - Combinations refer to the selection of objects without regard to the order.
  - The number of ways to choose r objects from n different objects is given by

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}.$$

  - Combinations are used when the order doesn't matter.
- **Binomial Theorem**
  - It provides a formula for the expansion of powers of a binomial (sum of two terms).
  - The Binomial Theorem states that:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} \cdot a^{n-k} \cdot b^k$$

    * This means the expansion is a sum of terms, where the exponents of a start at n and decrease to 0, while the exponents of b start at 0 and increase to n. The coefficients of each term are the corresponding binomial coefficients.
  - The coefficients of the terms in the expansion are the binomial coefficients, which can be calculated using combinations.
- **Binomial Distribution**
  - A binomial distribution is a discrete probability distribution that models the number of successes in a fixed number of independent Bernoulli trials. A Bernoulli trial is an experiment with exactly two possible outcomes, typically termed "success" and "failure".
  - In the context of the binomial distribution, $P(x = k)$ denotes the probability of getting exactly k successes in n trials. The formula for this is

$$P(x = k) = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}.$$

    * $\binom{n}{k}$ (read as "n choose k") is the binomial coefficient, representing the number of ways to choose k successes from n trials.
    * p is the probability of success on an individual trial.
    * $1 - p$ is the probability of failure (since the trials are binary, the sum of the probabilities of success and failure is 1).
    * p^k is the probability of having k successes.
    * $(1-p)^{n-k}$ is the probability of having $n - k$ failures.

groupby

The `groupby` method is used to split data into groups based on some criteria, apply a function to each group independently, and then combine the results into a data structure. The process is often summarized as split-apply-combine.

The way `groupby` works is by the following:

- **Split**: The data is divided into groups based on one or more keys. This is done by mapping a function over the index or columns of the DataFrame.
- **Apply**: A function is applied to each group independently. This function could be an aggregation (computing a summary statistic), transformation (standardizing data within a group), or filtration (removing data based on group properties).
- **Combine**: The results of the function application are combined into a new data structure.

The basic syntax for how `groupby` works is as follows:

```
df.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=NoDefault.
no_default, observed=False, dropna=True)

```

- **by**: Specifies the grouping criteria. Can be a function, column name, or list of column names.
- **axis**: Determines whether to group by rows (0) or columns (1).
- **level**: If the axis is a MultiIndex (hierarchical), groups by a particular level or levels.
- **as_index**: For aggregated output, returns object with group labels as the index. Setting it to False will return group labels in the columns.
- **sort**: Sorts group keys. By default, it's set to True.

groupby Example

Here is a simple example of utilizing `groupby` in Pandas:

```
import pandas as pd

# Example DataFrame
data = {
    'Date': ['2023-01-01', '2023-01-01', '2023-01-02', '2023-01-02', '2023-01-03'],
    'Product': ['A', 'B', 'A', 'A', 'B'],
    'Sales': [100, 200, 150, 100, 250]
}
df = pd.DataFrame(data)

# Group by 'Product' and sum up the sales
grouped_df = df.groupby('Product')['Sales'].sum()

print(grouped_df)

```

agg

The `agg` method in Pandas, when used with groupby, is a versatile tool for performing multiple aggregation operations on your grouped data. It allows for more flexibility than just applying a single aggregate function like `sum` or `mean` directly. With `agg`, you can apply different aggregation functions to your data simultaneously, and even specify custom functions to suit your analysis needs.

After grouping your DataFrame with the `groupby` method, you can use `agg` to specify one or more aggregation operations to apply to the grouped data. `agg` can take a variety of inputs:

- A single aggregation function as a string (e.g. `'sum'`, `'mean'`).

- A list of functions (e.g., `['sum', 'mean', 'max']`), applying each function to each column of each group.
- A dictionary where keys are column names and values are functions or lists of functions, allowing different aggregation for different columns.

agg Example

Expanding on the previous example that was used with `groupby`, the example below encapsulates how to use the `agg` function with `groupby`:

```
import pandas as pd

# Example DataFrame
data = {
    'Date': ['2023-01-01', '2023-01-01', '2023-01-02', '2023-01-02', '2023-01-03'],
    'Product': ['A', 'B', 'A', 'A', 'B'],
    'Sales': [100, 200, 150, 100, 250]
}
df = pd.DataFrame(data)

# Group by 'Product' and apply multiple aggregation functions to 'Sales'
grouped_df = df.groupby('Product')['Sales'].agg(['sum', 'mean', 'max'])

print(grouped_df)

```

This would output the total, average, and maximum sales for each product.

More advanced usage can be applied to the `agg` function, below is an example with the previous data frame in the last example of this advanced usage:

```
grouped_df = df.groupby('Product').agg({
    'Sales': ['sum', 'mean'],
    'Date': ['min', 'max']
})

```

This performs a sum and mean aggregation on the `Sales` and finds the minimum and maximum `Date` for each `Product`.

Joining DataFrames

Joining data frames is a fundamental operation in data analysis, allowing you to combine data from different sources based on common identifiers. Pandas offers several methods for joining DataFrames, akin to SQL joins, including `merge`, `join`, and `concat`. Understanding these methods and when to use each is key to effective data manipulation.

1. **merge**: The `merge` function is the most versatile method for joining two DataFrames. It allows you to perform inner, outer, left, and right joins by specifying how you want the DataFrames to be merged.
   - **Syntax**: `pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False)`
   - **Parameters**:
     - `left, right`: The DataFrames you want to join.
     - `how`: Specifies the type of join `('left', 'right', 'outer', 'inner')`.
     - `on`: The column(s) to join on. Must be found in both DataFrames.
     - `left_on, right_on`: Columns from the left and right DataFrames to use as keys if they have different names.
     - `left_index, right_index`: If True, use the index (row labels) from the left or right DataFrame as its join key(s).

merge Example

Below is a simple example of how `merge` works by joining DataFrames:

```
import pandas as pd

# Example DataFrames
df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'], 'value': range(4)})
df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'], 'value': range(4, 8)})

# Inner join on 'key'
inner_joined = pd.merge(df1, df2, on='key', how='inner')

# Outer join on 'key'
outer_joined = pd.merge(df1, df2, on='key', how='outer')

```

2. **join**: The `join` method is a convenient method for combining DataFrames based on their indexes or on a key column. It's a simpler interface than `merge` for index-based joining.

   - **Syntax**: `DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='')`
   - **Parameters**:
     - `other`: The DataFrame to join with.
     - `on`: The column or index level names to join on in the `other` DataFrame. Must be found in both the calling DataFrame and `other`.
     - `how`: Type of join `('left', 'right', 'outer', 'inner')`.
     - `lsuffix, rsuffix`: Suffixes to apply to overlapping column names in the DataFrames.

join Example

Below is a simple example of how `join` works by joining DataFrames:

```
# Assuming df1 and df2 from the previous example
# Join df2 to df1 using the index of df1 and the 'key' column of df2
joined = df1.join(df2.set_index('key'), on='key', how='left', lsuffix='_df1', rsuffix='_df2')

```

3. **concat**: The `concat` function is used for concatenating DataFrames along a particular axis (row-wise or column-wise). It's useful for stacking DataFrames vertically or horizontally.

   - **Syntax**: `pd.concat(objs, axis=0, join='outer')`
   - **Parameters**:
     - `objs`: A sequence of DataFrames to concatenate.
     - `axis`: The axis to concatenate along (0 for rows, 1 for columns).
     - `join`: How to handle indexes on other axis `('outer', 'inner')`.

concat Example

Below is a simple example of how `concat` works by joining DataFrames:

```
# Vertical concatenation
vertical_concat = pd.concat([df1, df2])

# Horizontal concatenation, assuming same indexes
horizontal_concat = pd.concat([df1, df2], axis=1)

```