#17

Data Representation for Visualization

Using NumPy and Pandas

MIS561 Data Visualization
Original Author : Lusi Yang

Data Wrangling for Visualization

Where **data wrangling** is used?

- To draw conclusions from visualized data, we need to handle our data and transform it into the best possible representation.

Data wrangling – the discipline of augmenting, transforming, and enriching data in a way that allows it to be displayed and understood by visualization tools or machine learning algorithms.

Tools and libraries for visualization

example

Non-coding tool

tableau software

Coding tool

python

MATLAB

our focus

NumPy

When handling data, we often need a way to work with multidimensional arrays. We also have to apply some basic mathematical and statistical operations on that data.

NumPy provides support for large n-dimensional arrays and is the built-in support for many high-level mathematical and statistical operations.

Basic NumPy Operations

```python
# importing the necessary dependencies
import numpy as np
```

```python
# loading the Dataset
dataset = np.genfromtxt('./data/normal_distribution.csv', delimiter=',')
```

Indexing

- Indexing elements in a NumPy **array**, on a high level, works the same as with built-in Python Lists. We are able to index elements in multi-dimensional matrices.

```python
dataset[0]        # index single element in outermost dimension
dataset[-1]       # index in reversed order in outermost dimension
dataset[1, 1]     # index single element in two-dimensional data
dataset[-1, -1]   # index in reversed order in two-dimensional data
```

Overview of Indexing 2D Arrays

Two dimensional arrays

To get a single element from a 2 dimensional array, I have to provide two indexes.

$$
\begin{array}{rlcl}
 & \text{Second index} & & \\
 & \quad\;\; 0 \quad\;\; 1 \quad\;\; 2 & & \\
\text{arr} = \text{np.array}(& [[2, 3, 4], & 0 & \\
 & \;[1, 2, 5], & 1 & \text{First index} \\
 & \;[3, 4, 3]]) & 2 &
\end{array}
$$

(Image by author)

Source: Towardsdatascience.com

Basic NumPy Operations

Slicing

- Slicing is adapted from Python's Lists. When handling large amounts of data, being able to easily slice parts of lists into new ndarrays is very helpful.

```
[20] dataset[1:3]       # rows 2 and 3
     dataset[:2, :2]    # 2x2 subset of the data
     dataset[-1, ::-1] # last row with elements reversed
```

Case #2: Slice index 0 to index 2 (not included) – Vertically and Horizontally

Case #3: Negative signs are used for direction, double colon used for skipping.

Basic NumPy Operations

Splitting

- There are two ways of splitting your data, horizontally and vertically. Horizontal splitting can be done with the **hsplit** method. Vertical splitting can be done with the **vsplit** method.

```
np.hsplit(dataset, (3))  # split horizontally in 3 equal lists
np.vsplit(dataset, (2))  # split vertically in 2 equal lists
```

Splitting data can be helpful in many situations, from plotting only half of your time-series data to separating test and training data for machine learning algorithms.

Basic NumPy Operations

Iterating

- Iterating steps over the whole list of data one after another, visiting every single element in the ndarray once.

nditer is a multi-dimensional iterator object that iterates over a given number of arrays.

```
# iterating over whole dataset (each value in each row)
for x in np.nditer(dataset):
    print(x)
```

ndenumerate will give us exactly this index, thus returning (**0**, **1**) for the second value in the first row.

```
# iterating over whole dataset with indices matching the position in the
dataset
for index, value in np.ndenumerate(dataset):
    print(index, value)
```

Advanced NumPy Operations

Filtering

- Filtering is a very powerful tool that can be used to clean up your data if you want to avoid outlier values. It's also helpful to get some better insights into your data.

In addition to the **dataset[dataset > 10]** shorthand notation, we can use the built-in NumPy **extract** method, which does the same thing using a different notation.

```
dataset[dataset > 10]                       # values bigger than 10
np.extract((dataset < 3), dataset)          # alternative - values smaller than 3
dataset[(dataset > 5) & (dataset < 10)] # values bigger 5 and smaller 10
```

Advanced NumPy Operations

Sorting

- Sorting each row of a dataset can be really useful. Using NumPy, we are also able to sort on other dimensions, such as columns.

argsort gives us the possibility to get a list of indices, which would result in a sorted list.

```
np.sort(dataset)          # values sorted on last axis
np.sort(dataset, axis=0)  # values sorted on axis 0
np.argsort(dataset)       # indices of values in sorted list
```

Advanced NumPy Operations

Combining

- Stacking rows and columns onto an existing dataset can be helpful when you have two datasets of the same dimension saved to different files.

Given two datasets, we use **vstack** to "stack" **dataset_1** on top of **dataset_2**, which will give us a combined dataset with all the rows from **dataset_1**, followed by all the rows from **dataset_2**.

If we use **hstack**, we stack our datasets "next to each other," meaning that the elements from the first row of **dataset_1** will be followed by the elements of the first row of **dataset_2**.

```
np.vstack([dataset_1, dataset_2])        # combine datasets vertically
np.hstack([dataset_1, dataset_2])        # combine datasets horizontally
```

Advanced NumPy Operations

Reshaping

- Reshaping might help you to reduce dimensionality to make visualization easier.

```
dataset.reshape(-1, 2)       # reshape dataset to two columns x rows
np.reshape(dataset, (1, -1)) # reshape dataset to one row x columns
```

Here, **-1** is an unknown dimension that NumPy identifies automatically.

Hands-on time

- Execute and understand the codes from Activity 1 to Activity 3 in Jupyter Lab

Activity 1: Using NumPy to compute the mean, median, and variance for the given numbers

Activity 2: Indexing, slicing and iterating

Activity 3: Filtering, sorting, combining and reshaping

Pandas

The **pandas** Python library offers data structures and methods to manipulate different types of data, such as numerical and temporal. These operations are easy to use and highly optimized for performance.

Data is represented in **DataFrame** in Pandas.

Data formats such as CSV, JSON, and databases can be used for **DataFrame** creation. DataFrames are the internal representation of data and are very similar to tables, but are more powerful.

Advantages of pandas over NumPy

High level of abstraction

- Pandas has a higher abstraction level than NumPy, which gives it a simpler interface for users to interact with. It abstracts away some of the more complex concepts and makes it easier to use and understand.

Less intuition

- Many methods, such as joining, selecting, and loading files, are usable without much intuition and without taking away much of the powerful nature of pandas.

Faster processing

- The internal representation of DataFrames allows faster processing for some operations. Of course, this always depends on the data and its structure.

Easy DataFrames design

- DataFrames are designed for operations with and on large datasets.

Basic operations of pandas

```python
# importing the necessary dependencies
import pandas as pd
```

```python
# loading the Dataset
dataset = pd.read_csv('./data/world_population.csv', index_col=0)
```

Indexing

- Indexing with pandas is a bit more complex than with NumPy. We can only access columns with the single bracket. To use the indices of the rows to access them, we need the **iloc** method.

```python
dataset["2000"]                     # index the 2000 col
dataset.iloc[-1]                    # index the last row
dataset.loc["Germany"]              # index the row with index Germany
dataset[["2015"]].loc[["Germany"]]  # index row Germany and column 2015
```

Loc = label
Iloc = integer location

Basic operations of pandas

Slicing

- Slicing with pandas is even more powerful. We can use the default slicing syntax in NumPy or use multi-selection. If we want to slice different rows or columns by name, we can simply pass a list into the bracket.

```
dataset.iloc[0:10]                  # slice of the first 10 rows

dataset.loc[["Germany", "India"]]  # slice of rows Germany and India

# subset of Germany and India with years 1970/90

dataset.loc[["Germany", "India"]][["1970", "1990"]]
```

Basic operations of pandas

Iterating

- Iterating DataFrames is also possible. Considering that they can have several dimensions and dtypes, the indexing is very high level and iterating over each row has to be done separately.

```
# iterating the whole dataset
for index, row in dataset.iterrows():
    print(index, row)
```

Basic operations of pandas

Series

- A pandas Series is a one-dimensional labelled array that is capable of holding any type of data. We can create a Series by loading datasets from a .csv file, Excel spreadsheet, or SQL database. There are many different ways to create them.

For example:

```
# import pandas
import pandas as pd
# import numpy
import numpy as np
# creating a numpy array
numarr = np.array(['p','y','t','h','o','n'])
ser = pd.Series(numarr)
print(ser)
```

```
0    p
1    y
2    t
3    h
4    o
5    n
dtype: object
```

NumPy arrays

```
# import pandas
import pandas as pd
# creating a pandas list
plist = ['p','y','t','h','o','n']
ser = pd.Series(plist)
print(ser)
```

```
0    p
1    y
2    t
3    h
4    o
5    n
dtype: object
```

pandas arrays

Advanced pandas Operations

Filtering

- Filtering in pandas has a higher-level interface than NumPy. You can still use the "simple" brackets-based **conditional filtering**.

- You're also able to use more complex queries, for example, filter rows based on a regular expression:

```
dataset.filter(items=["1990"])        # only column 1990
dataset[(dataset["1990"] < 10)]        # countries'population density < 10 in
1990
dataset.filter(like="8", axis=1)      # years containing an 8
dataset.filter(regex="a$", axis=0)  # countries ending with a
```

Advanced pandas Operations

Sorting

- With pandas, we are able to do sorting easily. Sorting in ascending and descending order can be done using the parameter known as **ascending**. You can do more complex sorting by providing more than one value in the **by = [ ]** list.

```
dataset.sort_values(by=["1999"])        # values sorted by 1999

# values sorted by 1999 descending

dataset.sort_values(by=["1994"], ascending=False)
```

Sorting each row or column based on a given row or column will help you get better insights into your data and find the ranking of a given dataset.

Advanced pandas Operations

Reshaping

- Reshaping can be crucial for easier visualization and algorithms. However, depending on your data, this can get really complex:

Country Code	ABW	AFG	AGO	ALB	AND	ARB	ARE	ARG	ARM
Year									
1999	494.466667	29.161566	11.712507	113.459051	136.512766	20.221913	34.499856	13.391379	108.669477

Steps to shaping the data – handle the index then pivot the table

```
dataset2 = dataset #Create a dataset

dataset2.index=[0]*len(dataset2) #Squash the index into single value

dataset2=dataset2.pivot(index=None, columns="Country Code", values = "1999") #Use Pivot
#Designate Column values using the values from Country Code
#Assign values from year 1999
```

Steps to shaping the data – handle the index then pivot the table

```
dataset2 = dataset #Create a dataset

dataset2.index=[0]*len(dataset2) #Squash the index into single value

dataset2=dataset2.pivot(index=None, columns="Country Code", values = "1999") #Use Pivot
#Designate Column values using the values from Country Code
#Assign values from year 1999
```

```
dataset.index.names=['Year'] # Assign the Index Namae
dataset.rename(index={0: '1999'}) #Assign values to the index (previously calculated as 0)
```

Country Code	ABW	AFG	AGO	ALB	AND	ARB	ARE	ARG	ARM
Year									
1999	494.466667	29.161566	11.712507	113.459051	136.512766	20.221913	34.499856	13.391379	108.669477

Hands-on time

- Execute and understand the codes from Activity 4 to Activity 6 in Jupyter Lab

Activity 4: Using pandas to compute the mean, median, and variance for the given numbers

Activity 5: Indexing, slicing and iterating with pandas

Activity 6: Filtering, sorting, and reshaping

Summary

What we have learnt today?

- Using NumPy and Pandas for data representation before visualization, e.g., indexing, slicing, iterating, filtering, sorting, and reshaping.

  -> Mastering these skills improve the quality of visualizations.

- Use NumPy and Pandas to get information from data, e.g., mean, median, and variance.

  -> This information enriches our visualizations.