Machine Learning: Libraries: Pandas

CPSC 501: Advanced Programming Techniques
Winter 2025

Jonathan Hudson, Ph.D
Assistant Professor (Teaching)
Department of Computer Science
University of Calgary

Thursday, February 13, 2025

UNIVERSITY OF CALGARY

pandas

- **pandas** (2010) is built on **numpy**
- recognition of outside influences and inside software engineering influences
  - Outside -> want array storage to work like spreadsheets/databases
  - Inside -> named data for readability and lookup
- Built around **Series** and **Dataframe** ideas which are essentially 1 dimensional array of data and multidimensional array of data with row/column labels
- Allows us to reference data with names, while maintaining **numpy** speed

Series

- 1D data is a Series, which acts like a specialized dictionary (default indices are integers)

```
import pandas as pd
print(pd.Series([0.25, 0.5, 0.75, 1.0]))
print(pd.Series([0.25, 0.5, 0.75, 1.0], index = ['a','b','c','d']))
data = pd.Series({'a':0.25, 'b':0.5, 'c':0.75, 'd':1.0})
print(data['a'])
```

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
0.25
```

Dataframe

- 2D data is a DataFrame, which acts like a specialized dictionary of Series

```
data_1 = pd.Series({'a':0.25, 'b':0.5, 'c':0.75, 'd':1.0})
data_2 = pd.Series({'a':1, 'b':2, 'c':3, 'd':4})
data = pd.DataFrame({'data_1':data_1, "data_2":data_2})
print(data)
print(data.index)
print(data.columns)
print(data['data_1'])
```

```
   data_1  data_2
a    0.25       1
b    0.50       2
c    0.75       3
d    1.00       4
Index(['a', 'b', 'c', 'd'], dtype='object')
Index(['data_1', 'data_2'], dtype='object')
a    0.25
b    0.50
c    0.75
d    1.00
Name: data_1, dtype: float64
```

Danger Will Robinson!

- **Slices are not copies** (better thought of a views of data)
- This is great for speed as data isn't copied, but it means modification of slice means modifying the original data

```
data = pd.Series({'a':0.25, 'b':0.5, 'c':0.75, 'd':1.0})
print(data)
data['a':'c']['a'] = 5
print(data)
```

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
a    5.00
b    0.50
c    0.75
d    1.00
dtype: float64
```

Indexing?

- We can index using the dictionary keys (index) but sometimes that can be confusing as there is both the explicit index and an always maintained integer (starts at 0) index as well
- **loc**, **iloc** is how we ensure we are doing the numerical type
- reset_index() useful for pushing current index into column and returning to integer index

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
print(data)
print(data[1])
print(data.loc[1])
print(data.iloc[1])
```

```
1    a
3    b
5    c
dtype: object
a
a
b
```

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data = data.reset_index()
print(data)
```

```
   index  0
0      1  a
1      3  b
2      5  c
```

DataFrame operations

- **DataFrame** allow for very quick generation of new columns of data
- Can be transposed quickly and **values()** gives the **numpy** data

```
area = pd.Series({'California': 423967, 'Texas': 695662, 'New York': 141297, 'Florida': 170312, 'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193, 'New York': 19651127, 'Florida': 19552860, 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
print(data)
data['density'] = data['pop']/data['area']
print(data)
print(data.T)
print(data.values)
```

```
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
Florida     170312  19552860
Illinois    149995  12882135
              area       pop     density
California  423967  38332521   90.413926
Texas       695662  26448193   38.018740
New York    141297  19651127  139.076746
Florida     170312  19552860  114.806121
Illinois    149995  12882135   85.883763
           California         Texas      New York       Florida      Illinois
area     4.239670e+05  6.956620e+05  1.412970e+05  1.703120e+05  1.499950e+05
pop      3.833252e+07  2.644819e+07  1.965113e+07  1.955286e+07  1.288214e+07
density  9.041393e+01  3.801874e+01  1.390767e+02  1.148061e+02  8.588376e+01
```

DataFrame operations

- **Wary of missing ()**

```
print(data.loc[(data['density'] > 100) & (data['density'] < 125)])
print(data.loc[(data['density'] > 100) & data['density'] < 125])
```

```
           area       pop     density
Florida  170312  19552860  114.806121
              area       pop     density
California  423967  38332521   90.413926
Texas       695662  26448193   38.018740
New York    141297  19651127  139.076746
Florida     170312  19552860  114.806121
Illinois    149995  12882135   85.883763
```

DataFrame UFuncs

- **numpy Ufuncs** can be applied across a whole **DataFrame**

```python
import pandas as pd
import numpy as np
rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
print(ser)
print(np.exp(ser))
print(df)
print(np.sin(df * np.pi / 4))
```

```
0    6
1    3
2    7
3    4
dtype: int64
0     403.428793
1      20.085537
2    1096.633158
3      54.598150
dtype: float64
   A  B  C  D
0  6  9  2  6
1  7  4  3  7
2  7  2  5  4
          A             B         C             D
0 -1.000000  7.071068e-01  1.000000 -1.000000e+00
1 -0.707107  1.224647e-16  0.707107 -7.071068e-01
2 -0.707107  1.000000e+00 -0.707107  1.224647e-16
```

DataFrame Ufuncs (index alignment)

- **UFuncs will combine on aligned indices**

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967}, name='area')
population = pd.Series({'California': 38332521, 'Texas': 26448193,
                        'New York': 19651127}, name='population')
print(population / area)
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
print(A + B)
```

```
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
0    NaN
1    5.0
2    9.0
3    NaN
dtype: float64
```

Missing Data (None verseus NaN)

- **None** is the typical **Python** type for no data, but in **numpy** it can be only used in '**object**' types as it is no data for an object reference
- Since it is a **Python** level concept it slows down data operations as **numpy** operations do not work **object** types

- **NaN** is a **float** (or similar) level type, the biggest issues with **NaN** is that it is a numerical virus that spreads through any calculation it is used in to produce more **NaN**
- Agreggates likes **sum**() will work on data with **NaN** in it, but the result will be **NaN** which can be disruptive

UNIVERSITY OF CALGARY

Missing Data (None verseus NaN)

- You can make pandas data with None but object type
- If you make data with np.nan the assumed type will be float
- If you give both types then type will be object
- If you give type as float then it will be cast from None to NaN

```
vals1 = np.array([1,None,3,4])
print(vals1)
print(pd.Series(vals1))
vals2 = np.array([1,np.nan,3,4])
print(vals2)
print(pd.Series(vals2))
vals3 = np.array([1,np.nan,3,None])
print(vals3)
print(pd.Series(vals3))
print(pd.Series(vals3, dtype='float64'))
```

```
[1 None 3 4]
0       1
1    None
2       3
3       4
dtype: object
[ 1. nan  3.  4.]
0    1.0
1    NaN
2    3.0
3    4.0
dtype: float64
[1 nan 3 None]
0       1
1     NaN
2       3
3    None
dtype: object
0    1.0
1    NaN
2    3.0
3    NaN
dtype: float64
```

Missing Data (processing it out)

- **pandas** is capable of processing out the **NaN** in order to do **Ufuncs** (unlike **numpy**)
- However sometimes we want to process them out permanently in our data
- We can use **isnull**() **notnull**() to get boolean mask arrays (to use for that)
- Or simply use **fill**, or **drop** operations to remove rows, or put data into rows

```
data = pd.Series([1,np.nan,3,None])
print(data.sum())
print(data.mean())
print(data.isnull())
print(data.dropna())
print(data.fillna(0))
```

```
4.0
2.0
0    False
1     True
2    False
3     True
dtype: bool
0    1.0
2    3.0
dtype: float64
0    1.0
1    0.0
2    3.0
3    0.0
dtype: float64
```

Operations

https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html

- Concatenate - concat
- Joins (Database style) – merge
- Can aggregate data with Ufuncs apply to series and dataframe (like mean, sum, …)
- GroupBy – Can group data and apply aggregates on groups
- Mapping – Apply a lamba or function on series/dataframe
- Sort/Rank
- Describe – distribution properties of columns – describe()
- Correlation matrix – coor()
- Unique
- Count

Reading/Writing Files

- Pandas can do many operations like
- read_csv, read_excel, read_html, read_json, read_pickle, read_sql
- In addition to others for data
- Possible to do many things such as setting names, skipping rows, limit rows, etc.
- to_csv used to write csv data
- Other libraries relevant here Python **csv** module, **json** library, **beautifulsoup/lxml/html5lib** for **HTML**, binary data using **pickle**, web-api generally use **request**, SQL database using **sqlite3/sqlalchemy**

Onward to … matplotlib

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY