Learning in Graphs
Applications to Community Mining

Graph Mining and Analysis with Python

Fragkiskos Malliaros, Michalis Vazirgiannis

INFMDI341 Machine Learning Avance

Monday, 2 June 2014

Outline

- Brief Introduction to Python
- Graph Mining with NetworkX

Overview of Python

- Python is an interpreted language, meaning there is no compilation or linking
- Python can be used in two different modes
  - **Interactive mode** makes it easy to experiment with the language
  - **Standard mode** is for running executable scripts and programs
- Typically, Python programs are much shorter than equivalent C, C++, or Java programs
  - High-level language and data types allow expressing complex operations concisely
  - Grouping of statements is done by **indentation** (e.g., tabs) instead of using brackets
  - No variable declarations

Modules and Built-in Numeric Types

- Python modules are **libraries** of code
  - They are imported using the **`import`** function, which runs the file

```
>>> from math import pi, sqrt

>>> import math
>>> math.pi
>>> math.e
```

- Python provides integers, floating-point numbers, complex numbers, etc.
  - **int**
  - **float**
  - **long** (long integers have unlimited precision)
  - **complex** (they have a real and imaginary part, which are each a floating point number)

Operators

Operation	Result
`x + y`	sum of *x* and *y*
`x - y`	difference of *x* and *y*
`x * y`	product of *x* and *y*
`x / y`	quotient of *x* and *y*
`x // y`	(floored) quotient of *x* and *y*
`x % y`	remainder of x / y
`-x`	*x* negated
`abs(x)`	absolute value or magnitude of *x*
`int(x)`	*x* converted to integer
`long(x)`	*x* converted to long integer
`float(x)`	*x* converted to floating point
`complex(re,im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
`pow(x, y), x ** y`	*x* to the power *y*

Comparisons

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

Examples of Operations

```
>>> 125 + 25
>>> 125 * 25
>>> 125 ** 25
>>> 4 1/3
>>> 1/ float (3)
>>> 1/3.0

>>> import math
>>> math.sqrt (math .pi)
>>> math.sin (_)
>>> 1 + _
```

- In the interactive mode, the _ operator contains the result of the last operation, which is very handy for subsequent operations

Python Statements (1/3)

The Python if statement

```
if test:
     block of code
elif test:
     block of code
else:
     block of code
```

Example: Computation of the absolute value

```
x = 4
y = 5

if x > y:
     absval = x - y
elif y > x:
     absval = y - x
else:
     absval = 0

Print "The absolute value is ", absval
```

Python Statements (2/3)

The Python for statement

```
for target in sequence:
     block of code
```

Examples

```
names = ['Peter ', 'John ', 'Mary ', 'Helen ', 'Tom ',
'Nicholas ']

for name in names:
     print name

for x in [0 ,1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10]:
     print x

for x in range (11):
     print x

for x in range (10 ,21) : print x
```

The range function creates a list of integers [start, stop) with the following syntax

```
range([ start ,] stop [, step ])
```

Python Statements (3/3)

The Python while statement

```
while expression:
     block of code
```

Example

```
temperature = 60

while (temperature > 40):
     print('The water is hot enough')
     temperature = temperature - 1

print ('Water's temperature is now OK')
```

How to execute python programs

Save a block of code in a file with extension "py": test1.py

Example

```
#test1.py
temperature = 60
while (temperature > 40):
     print('The water is hot enough')
     temperature = temperature - 1
print ('Water's temperature is now OK')
```

Execute the program from the OS

```
$python test1.py
```

Execute the program from within the python environment

```
>>> execfile(test1.py)
```

Outline

- Brief Introduction to Python
- **Graph Mining with NetworkX**

Graph Mining with NetworkX

- **NetworkX** is a Python package for creating and manipulating graphs and networks

  http://networkx.github.io/

- Main features
  - Python language data structures for graphs, digraphs, and multigraphs
  - Nodes can be **anything** (e.g., text, images, XML records)
  - Edges can hold arbitrary data (e.g. weights, time-series)
  - Generators for classic graphs, random graphs, and synthetic networks
  - Standard graph algorithms
  - Network structure and analysis measures
  - Open source BSD license
  - Well tested: more than 1800 unit tests, >90% code coverage
  - Additional benefits from Python: fast prototyping, easy to teach, multi-platform

Creating Undirected Graphs

```
>>> import networkx as nx
```

Import library

```
>>> G = nx.Graph()
```

Create a new undirected graph

```
>>> G.add_node("Jim")
>>> G.add_node("Jenny")
>>> G.add_node("David")
>>> G.add_edge("Jim", "David")
>>> G.add_edge("Jenny", "David")
```

Add new nodes and edges

```
>>> print G.number_of_nodes()
>>> print G.number_of_edges()
>>> print G.nodes()
>>> print G.edges()
```

Print the basic characteristics of the graph

```
>>> print G.degree("Jim")
>>> print G.degree()
```

Compute the degree of a specific node or of all the nodes in the graph

Creating Directed Graphs

```
>>> G = nx.DiGraph()
```

Create a new directed graph

```
>>> G.add_edges_from([("A","B"), ("C","A")])
```

Add nodes and edges edges

```
>>> print G.in_degree()
>>> print G.out_degree()
```

Print the in-degree and out-degree of the nodes

```
>>> print G.neighbors("A")
>>> print G.neighbors("B")
```

Print the neighborhood nodes of A and B

```
>>> U = G.to_undirected()
>>> print U.neighbors("B")
```

Convert the directed graph to undirected and print the neighbors of B

Loading/Writing Graphs from/to File

test.txt

```
a b
b c
b d
c d
```

Suppose that you have a graph stored in a text file in the **edge list** format: node pairs, one edge per line

```
>>> G = nx.read_edge_list("test.txt")
----
```

Syntax:

```
>>> G = nx.read_adjlist("adjlist.txt", comments='#',
delimiter=' ', nodetype=int)
```

Especially for character type nodes:

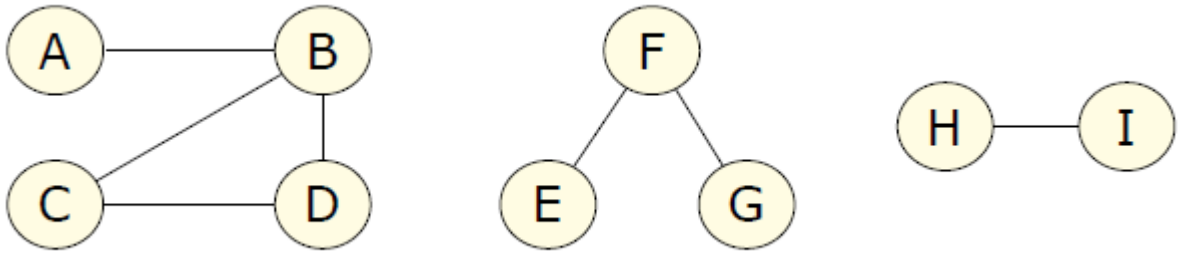
```
G1=nx.read_edgelist("test.txt", comments='#', delimiter=' ',
nodetype=None, create_using=nx.Graph())
```

Add a new edge between nodes **a** and **c** of G and save the new graph into the "edgelist.txt" file

```
>>> G.add_edge(a,c)
>>> nx.write_edgelist (G, "edgelist.txt", delimiter=' ')
```

Graph Connectivity

- A graph is connected if there is a path between every pair of nodes in the graph
- A connected component is a subset of the nodes where
  - A path exists between every pair in the subset



Example graph with 3 connected components

Connectivity in NetworkX

Define the graph

```
G = nx.Graph()
G.add_edges_from([("a","b"),("b","c"),("b","d"),("c","d")])
G.add_edges_from([("e","f"),("f","g"),("h","i")])
```

Examine if the graph is connected and if not, find the number of con. components

```
>>> print nx.is_connected(G)
False
>>> print nx.number_connected_components(G)
3
```

Find all connected components and print their nodes

```
>>> comps = nx.connected_component_subgraphs(G)
>>> print comps[0].nodes()
['a', 'c', 'b', 'd']
>>> print comps[0].nodes()
['e', 'g', 'f']
>>> print comps[2].nodes()
['i', 'h']
```

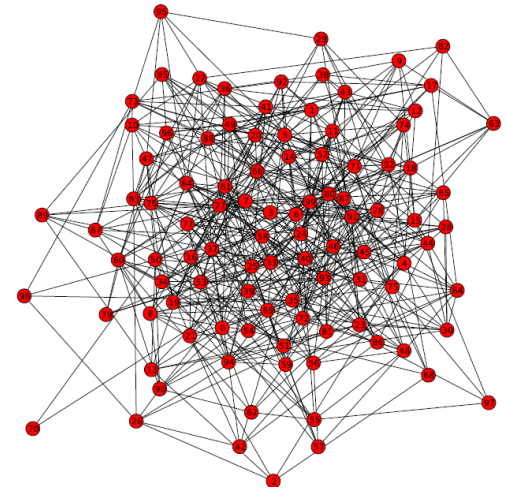
Network Visualization

- NetworkX uses the **Matplotlib** module for some simple network visualizations

Create a random graph based on the Erdos-Renyi model and visualize it

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.erdos_renyi_graph(100,0.11)
plt.figure(figsize =(10 ,10))
nx.draw(G)
plt.show()
```



More examples of graph visualization at:
http://networkx.github.io/documentation/latest/gallery.html