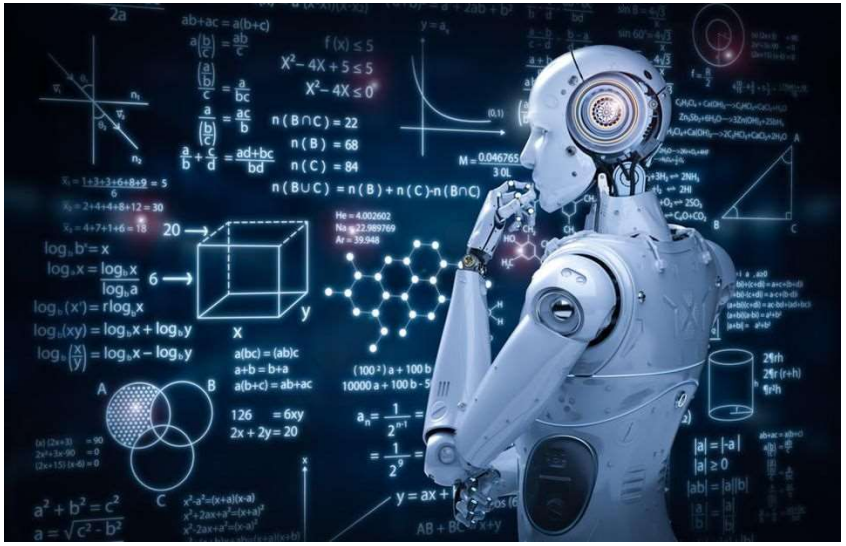
COMP6577 – Machine Learning

Cheatsheet Collection



Johannes Simatupang, S.Kom, M.Kom, COBIT5
NIDN : 0318017001

Python Cheat Sheet

GENERAL

- Python is case sensitive
- Python index starts from 0
- Python uses whitespace (tabs or spaces) to indent code instead of using braces.

HELP

Help Home Page	`help()`
Function Help	`help(str.replace)`
Module Help	`help(re)`

MODULE (AKA LIBRARY)

Python module is simply a '.py' file

List Module Contents	`dir(module1)`
Load Module	`import module1 *`
Call Function from Module	`module1.func1()`

* import statement creates a new namespace and executes all the statements in the associated .py file within that namespace. If you want to load the module's content into current namespace, use `from module1 import *`

SCALAR TYPES

Check data type : `type(variable)`

SIX COMMONLY USED DATA TYPES

1. **int/long*** - Large int automatically converts to long
2. **float*** - 64 bits, there is no 'double' type
3. **bool*** - True or False
4. **str*** - ASCII valued in Python 2.x and Unicode in Python 3
   - String can be in single/double/triple quotes
   - String is a sequence of characters, thus can be treated like other sequences
   - Special character can be done via \ or preface with r

```
str1 = r'this\f?ff'
```

   - String formatting can be done in a number of ways

```
template = '%.2f %s haha %d';
str1 = template % (4.88, 'hola', 2)
```

SCALAR TYPES

* str(), bool(), int() and float() are also explicit type cast functions.

5. **NoneType(None)** - Python 'null' value (ONLY one instance of None object exists)
   - None is not a reserved keyword but rather a unique instance of 'NoneType'
   - None is common default value for optional function arguments :

```
def func1(a, b, c = None)
```

   - Common usage of None :

```
if variable is None :
```

6. **datetime** - built-in python 'datetime' module provides 'datetime', 'date', 'time' types.
   - 'datetime' combines information stored in 'date' and 'time'

Create datetime from String	`dt1 = datetime.strptime('20091031', '%Y%m%d')`
Get 'date' object	`dt1.date()`
Get 'time' object	`dt1.time()`
Format datetime to String	`dt1.strftime('%m/%d/%Y %H:%M')`
Change Field Value	`dt2 = dt1.replace(minute = 0, second = 30)`
Get Difference	`diff = dt1 - dt2` # diff is a 'datetime.timedelta' object

Note : Most objects in Python are mutable except for 'strings' and 'tuples'

DATA STRUCTURES

Note : All non-Get function call (i.e. `list1.sort()`) examples below are in-place (without creating a new object) operations unless noted otherwise.

TUPLE

One dimensional, fixed-length, **immutable** sequence of Python objects of ANY type.

Create Tuple	`tup1 = 4, 5, 6` or `tup1 = (6,7,8)`
Create Nested Tuple	`tup1 = (4,5,6), (7,8)`
Convert Sequence or Iterator to Tuple	`tuple([1, 0, 2])`
Concatenate Tuples	`tup1 + tup2`
Unpack Tuple	`a, b, c = tup1`

Application of Tuple

Swap variables	`b, a = a, b`

DATA STRUCTURES

LIST

One dimensional, variable length, **mutable** (i.e. contents can be modified) sequence of Python objects of ANY type.

Create List	`list1 = [1, 'a', 3]` or `list1 = list(tup1)`
Concatenate Lists*	`list1 + list2` or `list1.extend(list2)`
Append to End of List	`list1.append('b')`
Insert to Specific Position	`list1.insert(posIdx, 'b')` **
Inverse of Insert	`valueAtIdx = list1.pop(posIdx)`
Remove First Value from List	`list1.remove('a')`
Check Membership	`3 in list1 => True` ***
Sort List	`list1.sort()`
Sort with User-Supplied Function	`list1.sort(key = len)` # sort by length

* List concatenation using '+' is expensive since a new list must be created and objects copied over. Thus, `extend()` is preferable.

** Insert is computationally expensive compared with append.

*** Checking that a list contains a value is lot slower than dicts and sets as Python makes a linear scan where others (based on hash tables) in constant time.

Built-in 'bisect module':

- Implements binary search and insertion into a sorted list
- 'bisect.bisect' finds the location, where 'bisect.insort' actually inserts into that location.

♦ WARNING : bisect module functions do not check whether the list is sorted, doing so would be computationally expensive. Thus, using them in an unsorted list will succeed without error but may lead to incorrect results.

SLICING FOR SEQUENCE TYPES†

† Sequence types include 'str', 'array', 'tuple', 'list', etc.

Notation	`list1[start:stop]`
	`list1[start:stop:step]` (if step is used) §

Note :
- 'start' index is included, but 'stop' index is NOT.
- start/stop can be omitted in which they default to the start/end.

§ Application of 'step' :

Take every other element	`list1[::2]`
Reverse a string	`str1[::-1]`

DICT (HASH MAP)

Create Dict	`dict1 ={'key1' :'value1',2 :[3, 2]}`
Create Dict from Sequence	`dict(zip(keyList, valueList))`
Get/Set/Insert Element	`dict1['key1']` `dict1['key1'] = 'newValue'`
Get with Default Value	`dict1.get('key1', defaultValue)` **
Check if Key Exists	`'key1' in dict1`
Delete Element	`del dict1['key1']`
Get Key List	`dict1.keys()` ***
Get Value List	`dict1.values()` ***
Update Values	`dict1.update(dict2)` # dict1 values are replaced by dict2

* 'KeyError' exception if the key does not exist.

** 'get()' by default (aka no 'defaultValue') will return 'None' if the key does not exist.

*** Returns the lists of keys and values in the same order. However, the order is not any particular order, aka it is most likely not sorted.

Valid dict key types

- Keys have to be immutable like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable too)
- The technical term here is 'hashability', check whether an object is hashable with the `hash('this is string'), hash([1, 2])` - this would fail.

SET

- A set is an **unordered** collection of UNIQUE elements.
- You can think of them like dicts but keys only.

Create Set	`set([3, 6, 3])` or `{3, 6, 3}`
Test Subset	`set1.issubset(set2)`
Test Superset	`set1.issuperset(set2)`
Test sets have same content	`set1 == set2`

- **Set operations :**

Union(aka 'or')	`set1 | set2`
Intersection (aka 'and')	`set1 & set2`
Difference	`set1 - set2`
Symmetric Difference (aka 'xor')	`set1 ^ set2`

Python For Data Science *Cheat Sheet*

NumPy Basics

Learn Python for Data Science Interactively at www.DataCamp.com

NumPy

The **NumPy** library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Use the following import convention:

```
>>> import numpy as np
```

NumPy Arrays

Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],
                 dtype = float)
```

Initial Placeholders

```
>>> np.zeros((3,4))                    Create an array of zeros
>>> np.ones((2,3,4),dtype=np.int16)    Create an array of ones
>>> d = np.arange(10,25,5)             Create an array of evenly
                                       spaced values (step value)
>>> np.linspace(0,2,9)                 Create an array of evenly
                                       spaced values (number of samples)
>>> e = np.full((2,2),7)               Create a constant array
>>> f = np.eye(2)                      Create a 2X2 identity matrix
>>> np.random.random((2,2))            Create an array with random values
>>> np.empty((3,2))                    Create an empty array
```

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

```
>>> np.int64       Signed 64-bit integer types
>>> np.float32     Standard double-precision floating point
>>> np.complex     Complex numbers represented by 128 floats
>>> np.bool        Boolean type storing TRUE and FALSE values
>>> np.object      Python object type
>>> np.string_     Fixed-length string type
>>> np.unicode_    Fixed-length unicode type
```

Inspecting Your Array

```
>>> a.shape          Array dimensions
>>> len(a)           Length of array
>>> b.ndim           Number of array dimensions
>>> e.size           Number of array elements
>>> b.dtype          Data type of array elements
>>> b.dtype.name     Name of data type
>>> b.astype(int)    Convert an array to a different type
```

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

```
>>> g = a - b                          Subtraction
  array([[-0.5,  0. ,  0. ],
         [-3. , -3. , -3. ]])
>>> np.subtract(a,b)                   Subtraction
>>> b + a                              Addition
  array([[ 2.5,  4. ,  6. ],
         [ 5. ,  7. ,  9. ]])
>>> np.add(b,a)                        Addition
>>> a / b                              Division
  array([[ 0.66666667,  1.        ,  1.        ],
         [ 0.25      ,  0.4       ,  0.5       ]])
>>> np.divide(a,b)                     Division
>>> a * b                              Multiplication
  array([[  1.5,   4. ,   9. ],
         [  4. ,  10. ,  18. ]])
>>> np.multiply(a,b)                   Multiplication
>>> np.exp(b)                          Exponentiation
>>> np.sqrt(b)                         Square root
>>> np.sin(a)                          Print sines of an array
>>> np.cos(b)                          Element-wise cosine
>>> np.log(a)                          Element-wise natural logarithm
>>> e.dot(f)                           Dot product
  array([[ 7.,  7.],
         [ 7.,  7.]])
```

Comparison

```
>>> a == b                             Element-wise comparison
  array([[False,  True,  True],
         [False, False, False]], dtype=bool)
>>> a < 2                              Element-wise comparison
  array([True, False, False], dtype=bool)
>>> np.array_equal(a, b)               Array-wise comparison
```

Aggregate Functions

```
>>> a.sum()                            Array-wise sum
>>> a.min()                            Array-wise minimum value
>>> b.max(axis=0)                      Maximum value of an array row
>>> b.cumsum(axis=1)                   Cumulative sum of the elements
>>> a.mean()                           Mean
>>> b.median()                         Median
>>> a.corrcoef()                       Correlation coefficient
>>> np.std(b)                          Standard deviation
```

Copying Arrays

```
>>> h = a.view()          Create a view of the array with the same data
>>> np.copy(a)            Create a copy of the array
>>> h = a.copy()          Create a deep copy of the array
```

Sorting Arrays

```
>>> a.sort()              Sort an array
>>> c.sort(axis=0)        Sort the elements of an array's axis
```

Subsetting, Slicing, Indexing

Also see Lists

Subsetting

```
>>> a[2]
  3
```

Select the element at the 2nd index

```
>>> b[1,2]
  6.0
```

Select the element at row 0 column 2 (equivalent to `b[1][2]`)

Slicing

```
>>> a[0:2]
  array([1, 2])
```

Select items at index 0 and 1

```
>>> b[0:2,1]
  array([ 2. ,  5.])
```

Select items at rows 0 and 1 in column 1

```
>>> b[:1]
  array([[1.5, 2., 3.]])
```

Select all items at row 0 (equivalent to `b[0:1, :]`)

```
>>> c[1,...]
  array([[ 3. ,  2. ,  1. ],
         [ 4. ,  5. ,  6. ]])
```

Same as `[1, :, :]`

```
>>> a[ ::-1]
  array([3, 2, 1])
```

Reversed array `a`

Boolean Indexing

```
>>> a[a<2]
  array([1])
```

Select elements from `a` less than 2

Fancy Indexing

```
>>> b[[1, 0, 1, 0],[0, 1, 2, 0]]
  array([ 4. ,  2. ,  6. ,  1.5])
>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]
  array([[ 4. ,  5. ,  6. ,  4. ],
         [ 1.5,  2. ,  3. ,  1.5],
         [ 4. ,  5. ,  6. ,  4. ],
         [ 1.5,  2. ,  3. ,  1.5]])
```

Select elements `(1,0),(0,1),(1,2)` and `(0,0)`

Select a subset of the matrix's rows and columns

Array Manipulation

Transposing Array

```
>>> i = np.transpose(b)    Permute array dimensions
>>> i.T                    Permute array dimensions
```

Changing Array Shape

```
>>> b.ravel()              Flatten the array
>>> g.reshape(3,-2)        Reshape, but don't change data
```

Adding/Removing Elements

```
>>> h.resize((2,6))        Return a new array with shape (2,6)
>>> np.append(h,g)         Append items to an array
>>> np.insert(a, 1, 5)     Insert items in an array
>>> np.delete(a,[1])       Delete items from an array
```

Combining Arrays

```
>>> np.concatenate((a,d),axis=0)    Concatenate arrays
  array([ 1,  2,  3, 10, 15, 20])
>>> np.vstack((a,b))                Stack arrays vertically (row-wise)
  array([[ 1. ,  2. ,  3. ],
         [ 1.5,  2. ,  3. ],
         [ 4. ,  5. ,  6. ]])
>>> np.r_[e,f]                      Stack arrays vertically (row-wise)
>>> np.hstack((e,f))                Stack arrays horizontally (column-wise)
  array([[ 7.,  7.,  1.,  0.],
         [ 7.,  7.,  0.,  1.]])
>>> np.column_stack((a,d))          Create stacked column-wise arrays
  array([[ 1, 10],
         [ 2, 15],
         [ 3, 20]])
>>> np.c_[a,d]                      Create stacked column-wise arrays
```

Splitting Arrays

```
>>> np.hsplit(a,3)                  Split the array horizontally at the 3rd
  [array([1]),array([2]),array([3])] index
>>> np.vsplit(c,2)                  Split the array vertically at the 2nd index
[array([[[ 1.5,  2. ,  1. ],
         [ 4. ,  5. ,  6. ]]]),
 array([[[ 3. ,  2. ,  1. ],
         [ 4. ,  5. ,  6. ]]])]
```

Python For Data Science *Cheat Sheet*

SciPy - Linear Algebra

Learn More Python for Data Science Interactively at www.datacamp.com

SciPy

The **SciPy** library is one of the core packages for scientif c computing that provides mathematical algorithms and convenience functions built on the NumPy extension of Python.

Interacting With NumPy

Also see NumPy

```
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> b = np.array([(1+5j,2j,3j), (4j,5j,6j)])
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]])
```

Index Tricks

`>>> np.mgrid[0:5,0:5]`	Create a dense meshgrid
`>>> np.ogrid[0:2,0:2]`	Create an open meshgrid
`>>> np.r_[3,[0]*5,-1:1:10j]`	Stack arrays vertically (row-wise)
`>>> np.c_[b,c]`	Create stacked column-wise arrays

Shape Manipulation

`>>> np.transpose(b)`	Permute array dimensions
`>>> b.fatten()`	Flat en the array
`>>> np.hstack((b,c))`	Stack arrays horizontally (column-wise)
`>>> np.vstack((a,b))`	Stack arrays vertically (row-wise)
`>>> np.hsplit(c,2)`	Split the array horizontally at the 2nd index
`>>> np.vpslit(d,2)`	Split the array vertically at the 2nd index

Polynomials

`>>> from numpy import poly1d`	
`>>> p = poly1d([3,4,5])`	Create a polynomial object

Vectorizing Functions

```
>>> def myfunc(a):
        if a < 0:
          return a*2
        else:
          return a/2
>>> np.vectorize(myfunc)
```

Vectorize functions

Type Handling

`>>> np.real(b)`	Return the real part of the array elements
`>>> np.imag(b)`	Return the imaginary part of the array elements
`>>> np.real_if_close(c,tol=1000)`	Return a real array if complex parts close to 0
`>>> np.cast['f'](np.pi)`	Cast object to a data type

Other Useful Functions

`>>> np.angle(b,deg=True)`	Return the angle of the complex argument
`>>> g = np.linspace(0,np.pi,num=5)`	Create an array of evenly spaced values (number of samples)
`>>> g [3:] += np.pi`	
`>>> np.unwrap(g)`	Unwrap
`>>> np.logspace(0,10,3)`	Create an array of evenly spaced values (log scale)
`>>> np.select([c<4],[c*2])`	Return values from a list of arrays depending on conditions
`>>> misc.factorial(a)`	Factorial
`>>> misc.comb(10,3,exact=True)`	Combine N things taken at k time
`>>> misc.central_diff_weights(3)`	Weights for Np-point central derivative
`>>> misc.derivative(myfunc,1.0)`	Find the n-th derivative of a function at a point

Linear Algebra

Also see NumPy

You'll use the `linalg` and `sparse` modules. Note that `scipy.linalg` contains and expands on `numpy.linalg`.

```
>>> from scipy import linalg, sparse
```

Creating Matrices

```
>>> A = np.matrix(np.random.random((2,2)))
>>> B = np.asmatrix(b)
>>> C = np.mat(np.random.random((10,5)))
>>> D = np.mat([[3,4], [5,6]])
```

Basic Matrix Routines

Inverse

`>>> A.I`	Inverse
`>>> linalg.inv(A)`	Inverse

Transposition

`>>> A.T`	Tranpose matrix
`>>> A.H`	Conjugate transposition

Trace

`>>> np.trace(A)`	Trace

Norm

`>>> linalg.norm(A)`	Frobenius norm
`>>> linalg.norm(A,1)`	L1norm (max column sum)
`>>> linalg.norm(A,np.inf)`	L inf norm (max row sum)

Rank

`>>> np.linalg.matrix_rank(C)`	Matrix rank

Determinant

`>>> linalg.det(A)`	Determinant

Solving linear problems

`>>> linalg.solve(A,b)`	Solver for dense matrices
`>>> E = np.mat(a).T`	Solver for dense matrices
`>>> linalg.lstsq(F,E)`	Least-squares solution to linear matrix equation

Generalized inverse

`>>> linalg.pinv(C)`	Compute the pseudo-inverse of a matrix (least-squares solver)
`>>> linalg.pinv2(C)`	Compute the pseudo-inverse of a matrix (SVD)

Creating Sparse Matrices

`>>> F = np.eye(3, k=1)`	Create a 2X2 identity matrix
`>>> G = np.mat(np.identity(2))`	Create a 2x2 identity matrix
`>>> C[C > 0.5] = 0`	
`>>> H = sparse.csr_matrix(C)`	Compressed Sparse Row matrix
`>>> I = sparse.csc_matrix(D)`	Compressed Sparse Column matrix
`>>> J = sparse.dok_matrix(A)`	Dictionary Of Keys matrix
`>>> E.todense()`	Sparse matrix to full matrix
`>>> sparse.isspmatrix_csc(A)`	Identify sparse matrix

Sparse Matrix Routines

Inverse

`>>> sparse.linalg.inv(I)`	Inverse

Norm

`>>> sparse.linalg.norm(I)`	Norm

Solving linear problems

`>>> sparse.linalg.spsolve(H,I)`	Solver for sparse matrices

Sparse Matrix Functions

`>>> sparse.linalg.expm(I)`	Sparse matrix exponential

Matrix Functions

Addition

`>>> np.add(A,D)`	Addition

Subtraction

`>>> np.subtract(A,D)`	Subtraction

Division

`>>> np.divide(A,D)`	Division

Multiplication

`>>> A @ D`	Multiplication operator (Python 3)
`>>> np.multiply(D,A)`	Multiplication
`>>> np.dot(A,D)`	Dot product
`>>> np.vdot(A,D)`	Vector dot product
`>>> np.inner(A,D)`	Inner product
`>>> np.outer(A,D)`	Outer product
`>>> np.tensordot(A,D)`	Tensor dot product
`>>> np.kron(A,D)`	Kronecker product

Exponential Functions

`>>> linalg.expm(A)`	Matrix exponential
`>>> linalg.expm2(A)`	Matrix exponential (Taylor Series)
`>>> linalg.expm3(D)`	Matrix exponential (eigenvalue decomposition)

Logarithm Function

`>>> linalg.logm(A)`	Matrix logarithm

Trigonometric Functions

`>>> linalg.sinm(D)`	Matrix sine
`>>> linalg.cosm(D)`	Matrix cosine
`>>> linalg.tanm(A)`	Matrix tangent

Hyperbolic Trigonometric Functions

`>>> linalg.sinhm(D)`	Hyperbolic matrix sine
`>>> linalg.coshm(D)`	Hyperbolic matrix cosine
`>>> linalg.tanhm(A)`	Hyperbolic matrix tangent

Matrix Sign Function

`>>> np.signm(A)`	Matrix sign function

Matrix Square Root

`>>> linalg.sqrtm(A)`	Matrix square root

Arbitrary Functions

`>>> linalg.funm(A, lambda x: x*x)`	Evaluate matrix function

Decompositions

Eigenvalues and Eigenvectors

`>>> la, v = linalg.eig(A)`	Solve ordinary or generalized eigenvalue problem for square matrix
`>>> l1, l2 = la`	Unpack eigenvalues
`>>> v[:,0]`	First eigenvector
`>>> v[:,1]`	Second eigenvector
`>>> linalg.eigvals(A)`	Unpack eigenvalues

Singular Value Decomposition

`>>> U,s,Vh = linalg.svd(B)`	Singular Value Decomposition (SVD)
`>>> M,N = B.shape`	
`>>> Sig = linalg.diagsvd(s,M,N)`	Construct sigma matrix in SVD

LU Decomposition

`>>> P,L,U = linalg.lu(C)`	LU Decomposition

Sparse Matrix Decompositions

`>>> la, v = sparse.linalg.eigs(F,1)`	Eigenvalues and eigenvectors
`>>> sparse.linalg.svds(H, 2)`	SVD

Asking For Help

```
>>> help(scipy.linalg.diagsvd)
>>> np.info(np.matrix)
```

DataCamp
Learn Python for Data Science Interactively

Data Analysis with PANDAS

CHEAT SHEET

Based on the Pandas Documentation

DATA STRUCTURES

SERIES (1D)

One-dimensional array-like object containing an array of data (of any **NumPy** data type) and an associated array of data labels, called its **"Index"**. If index of data is not specified, then a default one consisting of the integers 0 through N-1 is created.

Create Series	`series1 = pd.Series ([1, 2], index = ['a', 'b'])` `series1 = pd.Series(dict1)*`
Get Series Values	`series1.values`
Get Values by Index	`series1['a']` `series1[['b','a']]`
Get Series Index	`series1.index`
Get Name Attribute (None is default)	`series1.name` `series1.index.name`
** Common Index Values are Added	`series1 + series2`
Unique But Unsorted	`series2 = series1.unique()`

* Can think of Series as a fixed-length, ordered dict. Series can be substituted into many functions that expect a dict.

** Auto-align differently-indexed data in arithmetic operations

Get Columns and Row Names	`df1.columns` `df1.index`
Get Name Attribute (None is default)	`df1.columns.name` `df1.index.name`
Get Values	`df1.values` # returns the data as a 2D ndarray, the dtype will be chosen to accomrandate all of the columns
** Get Column as Series	`df1['state'] or df1.state`
** Get Row as Series	`df1.ix['row2'] or df1.ix[1]`
Assign a column that doesn't exist will create a new column	`df1['eastern'] = df1.state == 'Ohio'`
Delete a column	`del df1['eastern']`
Switch Columns and Rows	`df1.T`

* Dicts of Series are treated the same as Nested dict of dicts.

** Data returned is a 'view' on the underlying data, NOT a copy. Thus, any in-place modifications to the data will be reflected in df1.

DATAFRAME (2D)

Tabular data structure with ordered collections of columns, each of which can be different value type.

Data Frame (DF) can be thought of as a dict of Series.

Create DF
(from a dict of equal-length lists or NumPy arrays)
	`dict1 = {'state': ['Ohio', 'CA'], 'year': [2000, 2010]}`
	`df1 = pd.DataFrame(dict1)`
	# columns are placed in sorted order
	`df1 = pd.DataFrame(dict1, index = ['row1', 'row2'])`
	# specifying index
	`df1 = pd.DataFrame(dict1, columns = ['year', 'state'])`
	# columns are placed in your given order

* Create DF (from nested dict of dicts) The inner keys as row indices	`dict1 = {'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 3, 'row2': 4} }` `df1 = pd.DataFrame(dict1)`

PANEL DATA (3D)

Create Panel Data : (Each item in the Panel is a DF)

```
import pandas.datareader.data as web
panel1 = pd.Panel({stk : web.get_data_
yahoo(stk, '1/1/2000', '1/1/2010')
for stk in ['AAPL', 'IBM']})
```

panel1 Dimensions : 2 (item) * 861 (major) * 6 (minor)

"Stacked" DF form : (Useful way to represent panel data)

```
panel1 = panel1.swapaxes('item', 'minor')
panel1.ix[:, '6/1/2003', :].to_frame() *
```

=> Stacked DF (with **hierarchical indexing** **):

#		Open	High	Low	Close	Volume	Adj-Close
# major	minor						
# 2003-06-01	AAPL						
#	IBM						
# 2003-06-02	AAPL						
#	IBM						

DATA STRUCTURES CONTINUED

* DF has a "to_panel()" method which is the inverse of "to_frame()".

** Hierarchical indexing makes N-dimensional arrays unnecessary in a lot of cases. Aka prefer to use Stacked DF, not Panel data.

INDEX OBJECTS

Immutable objects that hold the axis labels and other metadata (i.e. axis name)

- i.e. Index, MultiIndex, DatetimeIndex, PeriodIndex
- Any sequence of labels used when constructing Series or DF internally converted to an Index.
- Can functions as fixed-size set in additional to being array-like.

HIERARCHICAL INDEXING

Multiple index levels on an axis : A way to work with higher dimensional data in a lower dimensional form.

Multiindex :

```
series1 = Series(np.random.randn(6), index =
[['a', 'a', 'a', 'b', 'b', 'b'], [1, 2, 3,
1, 2, 3]])
series1.index.names = ['key1', 'key2']
```

Series Partial Indexing	`series1['b']` # Outer Level `series1[:, 2]` # Inner Level

DF Partial Indexing	`df1['outerCol3','InnerCol2']`
	Or
	`df1['outerCol3'][['InnerCol2']]`

Swaping and Sorting Levels

Swap Level (level interchanged) *	`swapSeries1 = series1.swaplevel('key1', 'key2')`
Sort Level	`series1.sortlevel(1)` # sorts according to first inner level
Common Ops : Swap and Sort **	`series1.swaplevel(0, 1).sortlevel(0)` # the order of rows also change

* The order of the rows do not change. Only the two levels got swapped.

** Data selection performance is much better if the index is sorted starting with the outermost level, as a result of calling `sortlevel(0)` or `sort_index()`.

Summary Statistics by Level

Most stats functions in DF or Series have a "level" option that you can specify the level you want on an axis.

Sum rows (that have same 'key2' value)	`df1.sum(level = 'key2')`
Sum columns ..	`df1.sum(level = 'col3', axis = 1)`

- Under the hood, the functionality provided here utilizes panda's **'groupby'**.

DataFrame's Columns as Indexes

DF's "set_index" will create a new DF using one or more of its columns as the index.

New DF using columns as index	`df2 = df1.set_index(['col3', 'col4'])` * ‡ # col3 becomes the outermost index, col4 becomes inner index. Values of col3, col4 become the index values.

* "reset_index" does the opposite of "set_index", the hierarchical index are moved into columns.

‡ By default, 'col3' and 'col4' will be removed from the DF, though you can leave them by option : `'drop = False'`.

MISSING DATA

Python	`NaN = np.nan` (not a number)
Pandas *	`NaN` or python built-in `None` mean missing/NA values

* Use `pd.isnull()`, `pd.notnull()` or `series1/df1.isnull()` to detect missing data.

FILTERING OUT MISSING DATA

`dropna()` returns with ONLY non-null data, source data NOT modified.

`df1.dropna()` # drop any row containing missing value	
`df1.dropna(axis = 1)` # drop any column containing missing values	
`df1.dropna(how = 'all')` # drop row that are all missing	
`df1.dropna(thresh = 3)` # drop any row containing < 3 number of observations	

FILLING IN MISSING DATA

`df2 = df1.fillna(0)` # fill all missing data with 0	
`df1.fillna(inplace = True)` # modify in-place	
Use a different fill value for each column :	`df1.fillna({'col1' : 0, 'col2' : -1})`
Only forward fill the 2 missing values in front :	`df1.fillna(method = 'ffill', limit = 2)`
i.e. for column1, if row 3-6 are missing, so 3 and 4 get filled with the value from 2, NOT 5 and 6.	

Python For Data Science *Cheat Sheet*

Matplotlib

Learn Python interactively at www.DataCamp.com

Matplotlib

Matplotlib is a Python 2D plotting library which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms.

matplotlib

1 Prepare The Data

Also see Lists & NumPy

1D Data

```
>>> import numpy as np
>>> x = np.linspace(0, 10, 100)
>>> y = np.cos(x)
>>> z = np.sin(x)
```

2D Data or Images

```
>>> data = 2 * np.random.random((10, 10))
>>> data2 = 3 * np.random.random((10, 10))
>>> Y, X = np.mgrid[-3:3:100j, -3:3:100j]
>>> U = -1 - X**2 + Y
>>> V = 1 + X - Y**2
>>> from matplotlib.cbook import get_sample_data
>>> img = np.load(get_sample_data('axes_grid/bivariate_normal.npy'))
```

2 Create Plot

```
>>> import matplotlib.pyplot as plt
```

Figure

```
>>> fig = plt.figure()
>>> fig2 = plt.figure(figsize=plt.figaspect(2.0))
```

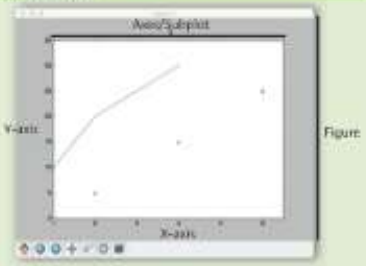
Axes

All plotting is done with respect to an `Axes`. In most cases, a subplot will fit your needs. A subplot is an axes on a grid system.

```
>>> fig.add_axes()
>>> ax1 = fig.add_subplot(221) # row-col-num
>>> ax3 = fig.add_subplot(212)
>>> fig3, axes = plt.subplots(nrows=2,ncols=2)
>>> fig4, axes2 = plt.subplots(ncols=3)
```

Plot Anatomy & Workflow

Plot Anatomy



Workflow

The basic steps to creating plots with matplotlib are:

1 Prepare data 2 Create plot 3 Plot 4 Customize plot 5 Save plot 6 Show plot

```
>>> import matplotlib.pyplot as plt
>>> x = [1,2,3,4]                  Step 1
>>> y = [10,20,25,30]
>>> fig = plt.figure()             Step 2
>>> ax = fig.add_subplot(111)      Step 3
>>> ax.plot(x, y, color='lightblue', linewidth=3)  Step 4
>>> ax.scatter([2,4,6],
               [5,15,25],
               color='darkgreen',
               marker='^')
>>> ax.set_xlim(1, 6.5)
>>> plt.savefig('foo.png')
>>> plt.show()                     Step 6
```

4 Customize Plot

Colors, Color Bars & Color Maps

```
>>> plt.plot(x, x, x, x**2, x, x**3)
>>> ax.plot(x, y, alpha = 0.4)
>>> ax.plot(x, y, c='k')
>>> fig.colorbar(im, orientation='horizontal')
>>> im = ax.imshow(img,
                   cmap='seismic')
```

Markers

```
>>> fig, ax = plt.subplots()
>>> ax.scatter(x,y,marker=".")
>>> ax.plot(x,y,marker="o")
```

Linestyles

```
>>> plt.plot(x,y,linewidth=4.0)
>>> plt.plot(x,y,ls='solid')
>>> plt.plot(x,y,ls='--')
>>> plt.plot(x,y,'--',x**2,y**2,'-.')
>>> plt.setp(lines,color='r',linewidth=4.0)
```

Text & Annotations

```
>>> ax.text(1,
            -2.1,
            'Example Graph',
            style='italic')
>>> ax.annotate("Sine",
                xy=(0, 0),
                xycoords='data',
                xytext=(10.5, 0),
                textcoords='data',
                arrowprops=dict(arrowstyle="->",
                                connectionstyle="arc3"),)
```

Mathtext

```
>>> plt.title(r'$sigma_i=15$', fontsize=20)
```

Limits, Legends & Layouts

Limits & Autoscaling

```
>>> ax.margins(x=0.0,y=0.1)          Add padding to a plot
>>> ax.axis('equal')                 Set the aspect ratio of the plot to 1
>>> ax.set(xlim=[0,10.5],ylim=[-1.5,1.5])  Set limits for x-and y-axis
>>> ax.set_xlim(0,10.5)              Set limits for x-axis
```

Legends

```
>>> ax.set(title='An Example Axes',   Set a title and x-and y-axis labels
           ylabel='Y-Axis',
           xlabel='X-Axis')
>>> ax.legend(loc='best')            No overlapping plot elements
```

Ticks

```
>>> ax.xaxis.set(ticks=range(1,5),    Manually set x-ticks
                 ticklabels=[3,100,-12,"foo"])
>>> ax.tick_params(axis='y',          Make y-ticks longer and go in and out
                   direction='inout',
                   length=10)
```

Subplot Spacing

```
>>> fig3.subplots_adjust(wspace=0.5,  Adjust the spacing between subplots
                         hspace=0.3,
                         left=0.125,
                         right=0.9,
                         top=0.9,
                         bottom=0.1)
>>> fig.tight_layout()                Fit subplot(s) in to the figure area
```

Axis Spines

```
>>> ax1.spines['top'].set_visible(False)   Make the top axis line for a plot invisible
>>> ax1.spines['bottom'].set_position(('outward',10))  Move the bottom axis line outward
```

3 Plotting Routines

1D Data

```
>>> lines = ax.plot(x,y)              Draw points with lines or markers connecting them
>>> ax.scatter(x,y)                   Draw unconnected points, scaled or colored
>>> axes[0,0].bar([1,2,3],[3,4,5])    Plot vertical rectangles (constant width)
>>> axes[1,0].barh([0.5,1,2.5],[0,1,2])  Plot horizontal rectangles (constant height)
>>> axes[1,1].axhline(0.45)           Draw a horizontal line across axes
>>> axes[0,1].axvline(0.65)           Draw a vertical line across axes
>>> ax.fill(x,y,color='blue')         Draw filled polygons
>>> ax.fill_between(x,y,color='yellow')  Fill between y-values and 0
```

Vector Fields

```
>>> axes[0,1].arrow(0,0,0.5,0.5)      Add an arrow to the axes
>>> axes[1,1].quiver(y,z)             Plot a 2D field of arrows
>>> axes[0,1].streamplot(X,Y,U,V)     Plot 2D vector fields
```

Data Distributions

```
>>> ax1.hist(y)                       Plot a histogram
>>> ax3.boxplot(y)                    Make a box and whisker plot
>>> ax3.violinplot(z)                 Make a violin plot
```

2D Data or Images

```
>>> fig, ax = plt.subplots()
>>> im = ax.imshow(img,               Colormapped or RGB arrays
                   cmap='gist_earth',
                   interpolation='nearest',
                   vmin=-2,
                   vmax=2)
```

```
>>> axes2[0].pcolor(data2)            Pseudocolor plot of 2D array
>>> axes2[0].pcolormesh(data)         Pseudocolor plot of 2D array
>>> CS = plt.contour(Y,X,U)           Plot contours
>>> axes2[2].contourf(data1)          Plot filled contours
>>> axes2[2]= ax.clabel(CS)           Label a contour plot
```

5 Save Plot

Save figures

```
>>> plt.savefig('foo.png')
```

Save transparent figures

```
>>> plt.savefig('foo.png', transparent=True)
```

6 Show Plot

```
>>> plt.show()
```

Close & Clear

```
>>> plt.cla()                         Clear an axis
>>> plt.clf()                         Clear the entire figure
>>> plt.close()                       Close a window
```

Python For Data Science *Cheat Sheet*

Scikit-Learn

Learn Python for data science Interactively at www.DataCamp.com

Scikit-learn

Scikit-learn is an open source Python library that implements a range of machine learning, preprocessing, cross-validation and visualization algorithms using a unified interface.

A Basic Example

```
>>> from sklearn import neighbors, datasets, preprocessing
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target
>>> X_train, X_test, y_train, y_test= train_test_split(X, y, random_state=33)
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>> knn.fit(X_train, y_train)
>>> y_pred = knn.predict(X_test)
>>> accuracy_score(y_test, y_pred)
```

Loading The Data

Also see NumPy & Pandas

Your data needs to be numeric and stored as NumPy arrays or SciPy sparse matrices. Other types that are convertible to numeric arrays, such as Pandas DataFrame, are also acceptable.

```
>>> import numpy as np
>>> X = np.random.random((10,5))
>>> y = np.array(['M','M','F','F','M','F','M','M','F','F','F'])
>>> X[X < 0.7] = 0
```

Training And Test Data

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X,
                                                        y,
                                                        random_state=0)
```

Create Your Model

Supervised Learning Estimators

Linear Regression

```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression(normalize=True)
```

Support Vector Machines (SVM)

```
>>> from sklearn.svm import SVC
>>> svc = SVC(kernel='linear')
```

Naive Bayes

```
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
```

KNN

```
>>> from sklearn import neighbors
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
```

Unsupervised Learning Estimators

Principal Component Analysis (PCA)

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=0.95)
```

K Means

```
>>> from sklearn.cluster import KMeans
>>> k_means = KMeans(n_clusters=3, random_state=0)
```

Model Fitting

Supervised learning	
`>>> lr.fit(X, y)`	Fit the model to the data
`>>> knn.fit(X_train, y_train)`	
`>>> svc.fit(X_train, y_train)`	
Unsupervised Learning	
`>>> k_means.fit(X_train)`	Fit the model to the data
`>>> pca_model = pca.fit_transform(X_train)`	Fit to data, then transform it

Prediction

Supervised Estimators	
`>>> y_pred = svc.predict(np.random.random((2,5)))`	Predict labels
`>>> y_pred = lr.predict(X_test)`	Predict labels
`>>> y_pred = knn.predict_proba(X_test)`	Estimate probability of a label
Unsupervised Estimators	
`>>> y_pred = k_means.predict(X_test)`	Predict labels in clustering algos

Preprocessing The Data

Standardization

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(X_train)
>>> standardized_X = scaler.transform(X_train)
>>> standardized_X_test = scaler.transform(X_test)
```

Normalization

```
>>> from sklearn.preprocessing import Normalizer
>>> scaler = Normalizer().fit(X_train)
>>> normalized_X = scaler.transform(X_train)
>>> normalized_X_test = scaler.transform(X_test)
```

Binarization

```
>>> from sklearn.preprocessing import Binarizer
>>> binarizer = Binarizer(threshold=0.0).fit(X)
>>> binary_X = binarizer.transform(X)
```

Encoding Categorical Features

```
>>> from sklearn.preprocessing import LabelEncoder
>>> enc = LabelEncoder()
>>> y = enc.fit_transform(y)
```

Imputing Missing Values

```
>>> from sklearn.preprocessing import Imputer
>>> imp = Imputer(missing_values=0, strategy='mean', axis=0)
>>> imp.fit_transform(X_train)
```

Generating Polynomial Features

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly = PolynomialFeatures(5)
>>> poly.fit_transform(X)
```

Evaluate Your Model's Performance

Classification Metrics

Accuracy Score

```
>>> knn.score(X_test, y_test)
```
Estimator score method

```
>>> from sklearn.metrics import accuracy_score
>>> accuracy_score(y_test, y_pred)
```
Metric scoring functions

Classification Report

```
>>> from sklearn.metrics import classification_report
>>> print(classification_report(y_test, y_pred))
```
Precision, recall, f1-score and support

Confusion Matrix

```
>>> from sklearn.metrics import confusion_matrix
>>> print(confusion_matrix(y_test, y_pred))
```

Regression Metrics

Mean Absolute Error

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2]
>>> mean_absolute_error(y_true, y_pred)
```

Mean Squared Error

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(y_test, y_pred)
```

R^2 Score

```
>>> from sklearn.metrics import r2_score
>>> r2_score(y_true, y_pred)
```

Clustering Metrics

Adjusted Rand Index

```
>>> from sklearn.metrics import adjusted_rand_score
>>> adjusted_rand_score(y_true, y_pred)
```

Homogeneity

```
>>> from sklearn.metrics import homogeneity_score
>>> homogeneity_score(y_true, y_pred)
```

V-measure

```
>>> from sklearn.metrics import v_measure_score
>>> metrics.v_measure_score(y_true, y_pred)
```

Cross-Validation

```
>>> from sklearn.cross_validation import cross_val_score
>>> print(cross_val_score(knn, X_train, y_train, cv=4))
>>> print(cross_val_score(lr, X, y, cv=2))
```

Tune Your Model

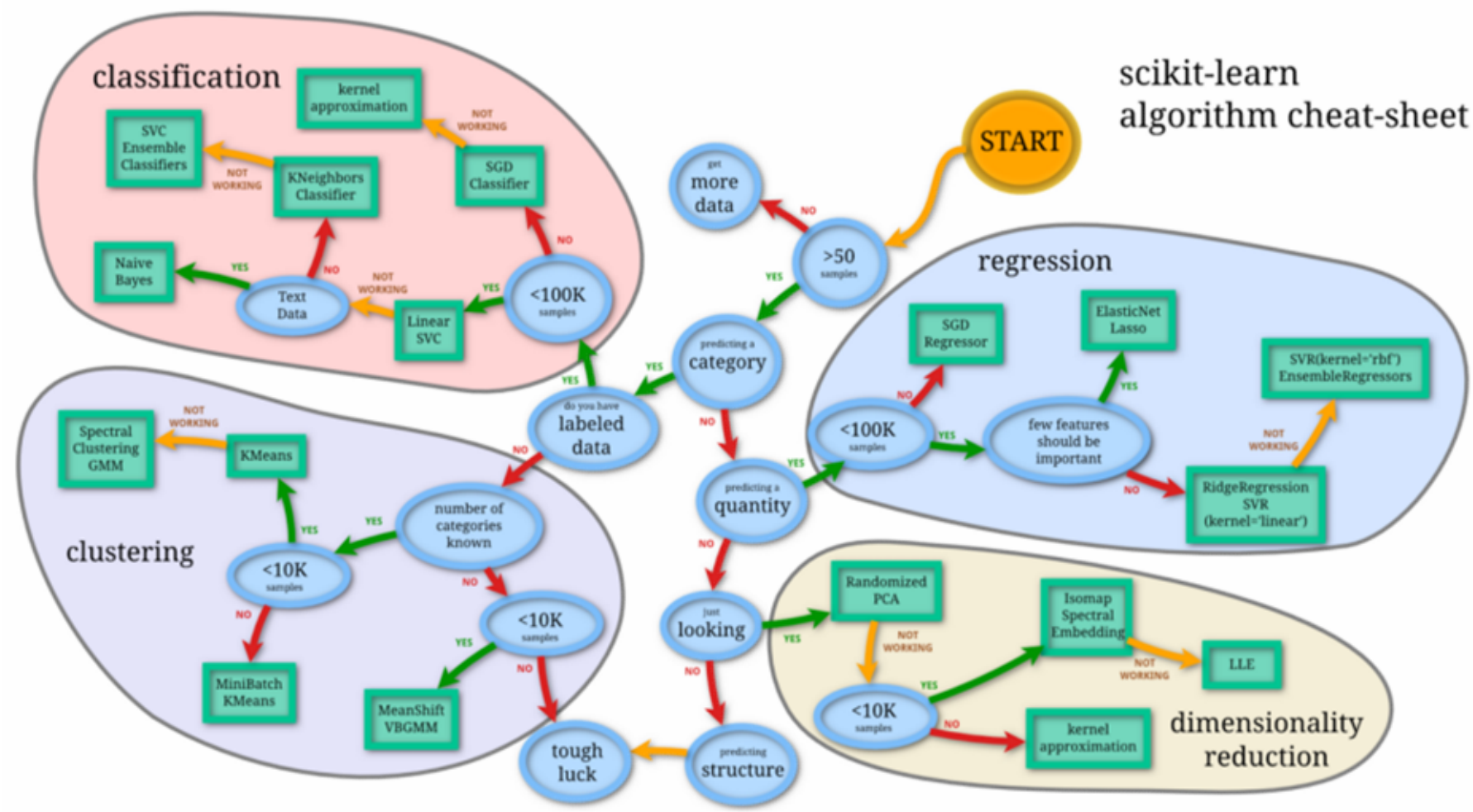
Grid Search

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {"n_neighbors": np.arange(1,3),
              "metric": ["euclidean", "cityblock"]}
>>> grid = GridSearchCV(estimator=knn,
                        param_grid=params)
>>> grid.fit(X_train, y_train)
>>> print(grid.best_score_)
>>> print(grid.best_estimator_.n_neighbors)
```

Randomized Parameter Optimization

```
>>> from sklearn.grid_search import RandomizedSearchCV
>>> params = {"n_neighbors": range(1,5),
              "weights": ["uniform", "distance"]}
>>> rsearch = RandomizedSearchCV(estimator=knn,
                                 param_distributions=params,
                                 cv=4,
                                 n_iter=8,
                                 random_state=5)
>>> rsearch.fit(X_train, y_train)
>>> print(rsearch.best_score_)
```

DataCamp
Learn Python for Data Science Interactively


scikit-learn
algorithm cheat-sheet
START
get more data
>50 samples
predicting a category
do you have labeled data
predicting a quantity
just looking
predicting structure
tough luck
classification
kernel approximation
SVC Ensemble Classifiers
KNeighbors Classifier
SGD Classifier
Naive Bayes
Text Data
Linear SVC
<100K samples
regression
SGD Regressor
ElasticNet Lasso
SVR(kernel='rbf') EnsembleRegressors
<100K samples
few features should be important
RidgeRegression SVR (kernel='linear')
clustering
Spectral Clustering GMM
KMeans
number of categories known
<10K samples
MiniBatch KMeans
<10K samples
MeanShift VBGMM
dimensionality reduction
Randomized PCA
Isomap Spectral Embedding
LLE
<10K samples
kernel approximation
YES
NO
NOT WORKING

Good Luck