Data Wrangling

with pandas Cheat Sheet

http://pandas.pydata.org

Pandas API Reference Pandas User Guide

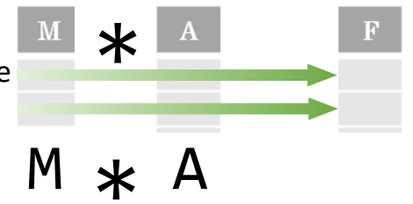
Tidy Data – A foundation for wrangling in pandas

In a tidy data set:

Each **variable** is saved in its own **column**

Each **observation** is saved in its own **row**

Tidy data complements pandas's **vectorized operations**. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.



Creating DataFrames

	a	b	c
1	4	7	10
2	5	8	11
3	6	9	12

```
df = pd.DataFrame(
        {"a" : [4, 5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
       index = [1, 2, 3])
```

Specify values for each column.

```
df = pd.DataFrame(
    [[4, 7, 10],
     [5, 8, 11],
     [6, 9, 12]],
    index=[1, 2, 3],
    columns=['a', 'b', 'c'])
```

Specify values for each row.

		a	b	c
N	v			
D	1	4	7	10
	2	5	8	11
e	2	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
index = pd.MultiIndex.from_tuples(
       [('d', 1), ('d', 2),
        ('e', 2)], names=['n', 'v']))
```

Create DataFrame with a MultiIndex

Method Chaining

Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

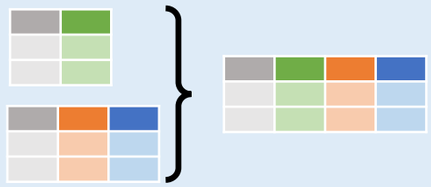
```
df = (pd.melt(df)
        .rename(columns={
                'variable':'var',
                'value':'val'})
        .query('val >= 200')
    )
```

Reshaping Data – Change layout, sorting, reindexing, renaming

pd.melt(df)
Gather columns into rows.

df.pivot(columns='var', values='val')
Spread rows into columns.

pd.concat([df1,df2])
Append rows of DataFrames



pd.concat([df1,df2], axis=1)
Append columns of DataFrames

df.sort_values('mpg')
Order rows by values of a column (low to high).

df.sort_values('mpg', ascending=False)
Order rows by values of a column (high to low).

df.rename(columns = {'y':'year'})
Rename the columns of a DataFrame

df.sort_index()
Sort the index of a DataFrame

df.reset_index()
Reset index of DataFrame to row numbers, moving index to columns.

df.drop(columns=['Length', 'Height'])
Drop columns from DataFrame

Subset Observations - rows

df[df.Length > 7]
Extract rows that meet logical criteria.

df.drop_duplicates()
Remove duplicate rows (only considers columns).

df.sample(frac=0.5)
Randomly select fraction of rows.

df.sample(n=10) Randomly select n rows.

df.nlargest(n, 'value')
Select and order top n entries.

df.nsmallest(n, 'value')
Select and order bottom n entries.

df.head(n)
Select first n rows.

df.tail(n)
Select last n rows.

Subset Variables - columns

df[['width', 'length', 'species']]
Select multiple columns with specific names.

df['width'] *or* **df.width**
Select single column with specific name.

df.filter(regex='*regex*')
Select columns whose name matches regular expression *regex*.

Using query

query() allows Boolean expressions for filtering rows.

```
df.query('Length > 7')
df.query('Length > 7 and Width < 8')
df.query('Name.str.startswith("abc")',
         engine="python")
```

Subsets - rows and columns

Use **df.loc[]** and **df.iloc[]** to select only rows, only columns or both.
Use **df.at[]** and **df.iat[]** to access a single value by row and column.
First index selects rows, second index columns.

df.iloc[10:20]
Select rows 10-20.

df.iloc[:, [1, 2, 5]]
Select columns in positions 1, 2 and 5 (first column is 0).

df.loc[:, 'x2':'x4']
Select all columns between x2 and x4 (inclusive).

df.loc[df['a'] > 10, ['a', 'c']]
Select rows meeting logical condition, and only the specific columns .

df.iat[1, 2] Access single value by index

df.at[4, 'A'] Access single value by label

Logic in Python (and pandas)

<	Less than	!=	Not equal to
>	Greater than	df.column.isin(*values*)	Group membership
==	Equals	pd.isnull(*obj*)	Is NaN
<=	Less than or equals	pd.notnull(*obj*)	Is not NaN
>=	Greater than or equals	&,|,~,^,df.any(),df.all()	Logical and, or, not, xor, any, all

regex (Regular Expressions) Examples

'\.'	Matches strings containing a period '.'
'Length$'	Matches strings ending with word 'Length'
'^Sepal'	Matches strings beginning with the word 'Sepal'
'^x[1-5]$'	Matches strings beginning with 'x' and ending with 1,2,3,4,5
'^(?!Species$).*'	Matches strings except the string 'Species'

Cheatsheet for pandas (http://pandas.pydata.org/ originally written by Irv Lustig, Princeton Consultants, inspired by Rstudio Data Wrangling Cheatsheet

Summarize Data

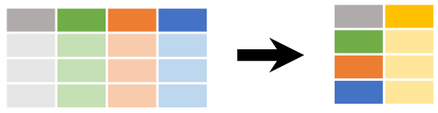
`df['w'].value_counts()`
Count number of rows with each unique value of variable

`len(df)`
# of rows in DataFrame.

`df.shape`
Tuple of # of rows, # of columns in DataFrame.

`df['w'].nunique()`
# of distinct values in a column.

`df.describe()`
Basic descriptive and statistics for each column (or GroupBy).



pandas provides a large set of summary functions that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups. When applied to a DataFrame, the result is returned as a pandas Series for each column. Examples:

`sum()`
Sum values of each object.

`count()`
Count non-NA/null values of each object.

`median()`
Median value of each object.

`quantile([0.25,0.75])`
Quantiles of each object.

`apply(function)`
Apply function to each object.

`min()`
Minimum value in each object.

`max()`
Maximum value in each object.

`mean()`
Mean value of each object.

`var()`
Variance of each object.

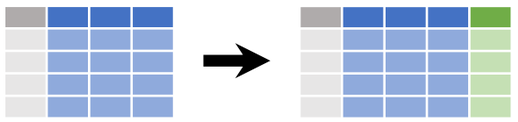
`std()`
Standard deviation of each object.

Handling Missing Data

`df.dropna()`
Drop rows with any column having NA/null data.

`df.fillna(value)`
Replace all NA/null data with value.

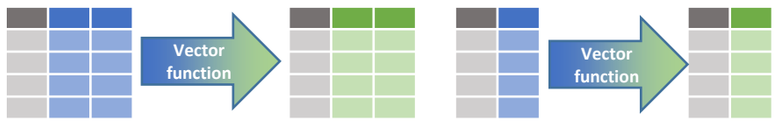
Make New Columns



`df.assign(Area=lambda df: df.Length*df.Height)`
Compute and append one or more new columns.

`df['Volume'] = df.Length*df.Height*df.Depth`
Add single column.

`pd.qcut(df.col, n, labels=False)`
Bin column into n buckets.



pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (a pandas Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

`max(axis=1)`
Element-wise max.

`clip(lower=-10,upper=10)`
Trim values at input thresholds

`min(axis=1)`
Element-wise min.

`abs()`
Absolute value.

Group Data

`df.groupby(by="col")`
Return a GroupBy object, grouped by values in column named "col".

`df.groupby(level="ind")`
Return a GroupBy object, grouped by values in index level named "ind".

All of the summary functions listed above can be applied to a group. Additional GroupBy functions:

`size()`
Size of each group.

`agg(function)`
Aggregate group using function.

The examples below can also be applied to groups. In this case, the function is applied on a per-group basis, and the returned vectors are of the length of the original DataFrame.

`shift(1)`
Copy with values shifted by 1.

`rank(method='dense')`
Ranks with no gaps.

`rank(method='min')`
Ranks. Ties get min rank.

`rank(pct=True)`
Ranks rescaled to interval [0, 1].

`rank(method='first')`
Ranks. Ties go to first value.

`shift(-1)`
Copy with values lagged by 1.

`cumsum()`
Cumulative sum.

`cummax()`
Cumulative max.

`cummin()`
Cumulative min.

`cumprod()`
Cumulative product.

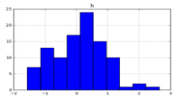
Windows

`df.expanding()`
Return an Expanding object allowing summary functions to be applied cumulatively.

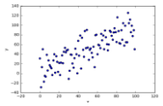
`df.rolling(n)`
Return a Rolling object allowing summary functions to be applied to windows of length n.

Plotting

`df.plot.hist()`
Histogram for each column



`df.plot.scatter(x='w',y='h')`
Scatter chart using pairs of points



Combine Data Sets

adf

x1	x2
A	1
B	2
C	3

+

bdf

x1	x3
A	T
B	F
D	T

=

Standard Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NaN

`pd.merge(adf, bdf, how='left', on='x1')`
Join matching rows from bdf to adf.

x1	x2	x3
A	1.0	T
B	2.0	F
D	NaN	T

`pd.merge(adf, bdf, how='right', on='x1')`
Join matching rows from adf to bdf.

x1	x2	x3
A	1	T
B	2	F

`pd.merge(adf, bdf, how='inner', on='x1')`
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NaN
D	NaN	T

`pd.merge(adf, bdf, how='outer', on='x1')`
Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

`adf[adf.x1.isin(bdf.x1)]`
All rows in adf that have a match in bdf.

x1	x2
C	3

`adf[~adf.x1.isin(bdf.x1)]`
All rows in adf that do not have a match in bdf.

ydf

x1	x2
A	1
B	2
C	3

+

zdf

x1	x2
B	2
C	3
D	4

=

Set-like Operations

x1	x2
B	2
C	3

`pd.merge(ydf, zdf)`
Rows that appear in both ydf and zdf (Intersection).

x1	x2
A	1
B	2
C	3
D	4

`pd.merge(ydf, zdf, how='outer')`
Rows that appear in either or both ydf and zdf (Union).

x1	x2
A	1

```
pd.merge(ydf, zdf, how='outer',
         indicator=True)
.query('_merge == "left_only"')
.drop(columns=['_merge'])
```
Rows that appear in ydf but not zdf (Setdiff).

datacamp

Python For Data Science

Pandas Basics Cheat Sheet

Learn Pandas Basics online at www.DataCamp.com

Pandas

The **Pandas** library is built on NumPy and provides easy-to-use **data structures** and **data analysis** tools for the Python programming language.

Use the following import convention:

```
>>> import pandas as pd
```

> Pandas Data Structures

Series

A **one-dimensional** labeled array capable of holding any data type

Index →

a	3
b	-5
c	7
d	4

```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

Dataframe

A **two-dimensional** labeled data structure with columns of potentially different types

Columns →

Index →	Country	Capital	Population
0	Belgium	Brussels	11190846
1	India	New Delhi	1303171035
2	Brazil	Brasilia	207847528

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
            'Capital': ['Brussels', 'New Delhi', 'Brasília'],
            'Population': [11190846, 1303171035, 207847528]}
>>> df = pd.DataFrame(data,
                      columns=['Country', 'Capital', 'Population'])
```

> Dropping

```
>>> s.drop(['a', 'c']) #Drop values from rows (axis=0)
>>> df.drop('Country', axis=1) #Drop values from columns(axis=1)
```

> Asking For Help

```
>>> help(pd.Series.loc)
```

> Sort & Rank

```
>>> df.sort_index() #Sort by labels along an axis
>>> df.sort_values(by='Country') #Sort by the values along an axis
>>> df.rank() #Assign ranks to entries
```

> I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> df.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> df.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')
```

Read multiple sheets from the same file

```
>>> xlsx = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

Read and Write to SQL Query or Database Table

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:')
>>> pd.read_sql("SELECT * FROM my_table;", engine)
>>> pd.read_sql_table('my_table', engine)
>>> pd.read_sql_query("SELECT * FROM my_table;", engine)
```

read_sql() **is a convenience wrapper around** read_sql_table() **and** read_sql_query()

```
>>> df.to_sql('myDf', engine)
```

> Selection

Also see NumPy Arrays

Getting

```
>>> s['b'] #Get one element
 -5
>>> df[1:] #Get subset of a DataFrame
   Country Capital Population
 1 India New Delhi 1303171035
 2 Brazil Brasília 207847528
```

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc[[0],[0]] #Select single value by row & column
 'Belgium'
>>> df.iat([0],[0])
 'Belgium'
```

By Label

```
>>> df.loc[[0], ['Country']] #Select single value by row & column labels
 'Belgium'
>>> df.at([0], ['Country'])
 'Belgium'
```

By Label/Position

```
>>> df.ix[2] #Select single row of subset of rows
 Country Brazil
 Capital Brasília
 Population 207847528
>>> df.ix[:,'Capital'] #Select a single column of subset of columns
 0 Brussels
 1 New Delhi
 2 Brasília
>>> df.ix[1,'Capital'] #Select rows and columns
 'New Delhi'
```

Boolean Indexing

```
>>> s[~(s > 1)] #Series s where value is not >1
>>> s[(s < -1) | (s > 2)] #s where value is <-1 or >2
>>> df[df['Population']>1200000000] #Use filter to adjust DataFrame
```

Setting

```
>>> s['a'] = 6 #Set index a of Series s to 6
```

> Retrieving Series/DataFrame Information

Basic Information

```
>>> df.shape #(rows,columns)
>>> df.index #Describe index
>>> df.columns #Describe DataFrame columns
>>> df.info() #Info on DataFrame
>>> df.count() #Number of non-NA values
```

Summary

```
>>> df.sum() #Sum of values
>>> df.cumsum() #Cummulative sum of values
>>> df.min()/df.max() #Minimum/maximum values
>>> df.idxmin()/df.idxmax() #Minimum/Maximum index value
>>> df.describe() #Summary statistics
>>> df.mean() #Mean of values
>>> df.median() #Median of values
```

> Applying Functions

```
>>> f = lambda x: x*2
>>> df.apply(f) #Apply function
>>> df.applymap(f) #Apply function element-wise
```

> Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
 a 10.0
 b NaN
 c 5.0
 d 7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_values=0)
 a 10.0
 b -5.0
 c 5.0
 d 7.0
>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```

datacamp

Python For Data Science
NumPy Cheat Sheet

Learn NumPy online at www.DataCamp.com

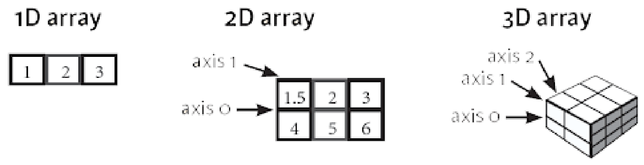
Numpy

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=" ")
```

> Asking For Help

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

> 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
```

> 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
```

> 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
>>> np.median(b) #Median
>>> np.corrcoef(a) #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

Subsetting

```
>>> a[2] #Select the element at the 2nd index
 3
>>> b[1,2] #Select the element at row 1 column 2 (equivalent to b[1][2])
 6.0
```

Slicing

```
>>> a[0:2] #Select items at index 0 and 1
 array([1, 2])
>>> b[0:2,1] #Select items at rows 0 and 1 in column 1
 array([ 2., 5.])
>>> b[:1] #Select all items at row 0 (equivalent to b[0:1, :])
 array([[1.5, 2., 3.]])
>>> c[1,...] #Same as [1,:,:]
 array([[[ 3., 2., 1.],
         [ 4., 5., 6.]]])
>>> a[ : :-1] #Reversed array a array([3, 2, 1])
```

Boolean Indexing

```
>>> a[a<2] #Select elements from a less than 2
 array([1])
```

Fancy Indexing

```
>>> b[[1, 0, 1, 0],[0, 1, 2, 0]] #Select elements (1,0),(0,1),(1,2) and (0,0)
 array([ 4. , 2. , 6. , 1.5])
>>> b[[1, 0, 1, 0]][:,[0,1,2,0]] #Select a subset of the matrix's rows and columns
 array([[ 4. ,5. , 6. , 4. ],
        [ 1.5, 2. , 3. , 1.5],
        [ 4. , 5. , 6. , 4. ],
        [ 1.5, 2. , 3. , 1.5]])
```

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

datacamp

Python For Data Science
Scikit-Learn Cheat Sheet

Learn Scikit-Learn online 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)
```

> 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)
```

> 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)
```

> Evaluate Your Model's Performance

Classification Metrics

Accuracy Score

```
>>> knn.score(X_test, y_test) #Estimator score method
>>> from sklearn.metrics import accuracy_score #Metric scoring functions
>>> accuracy_score(y_test, y_pred)
```

Classification Report

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

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))
```

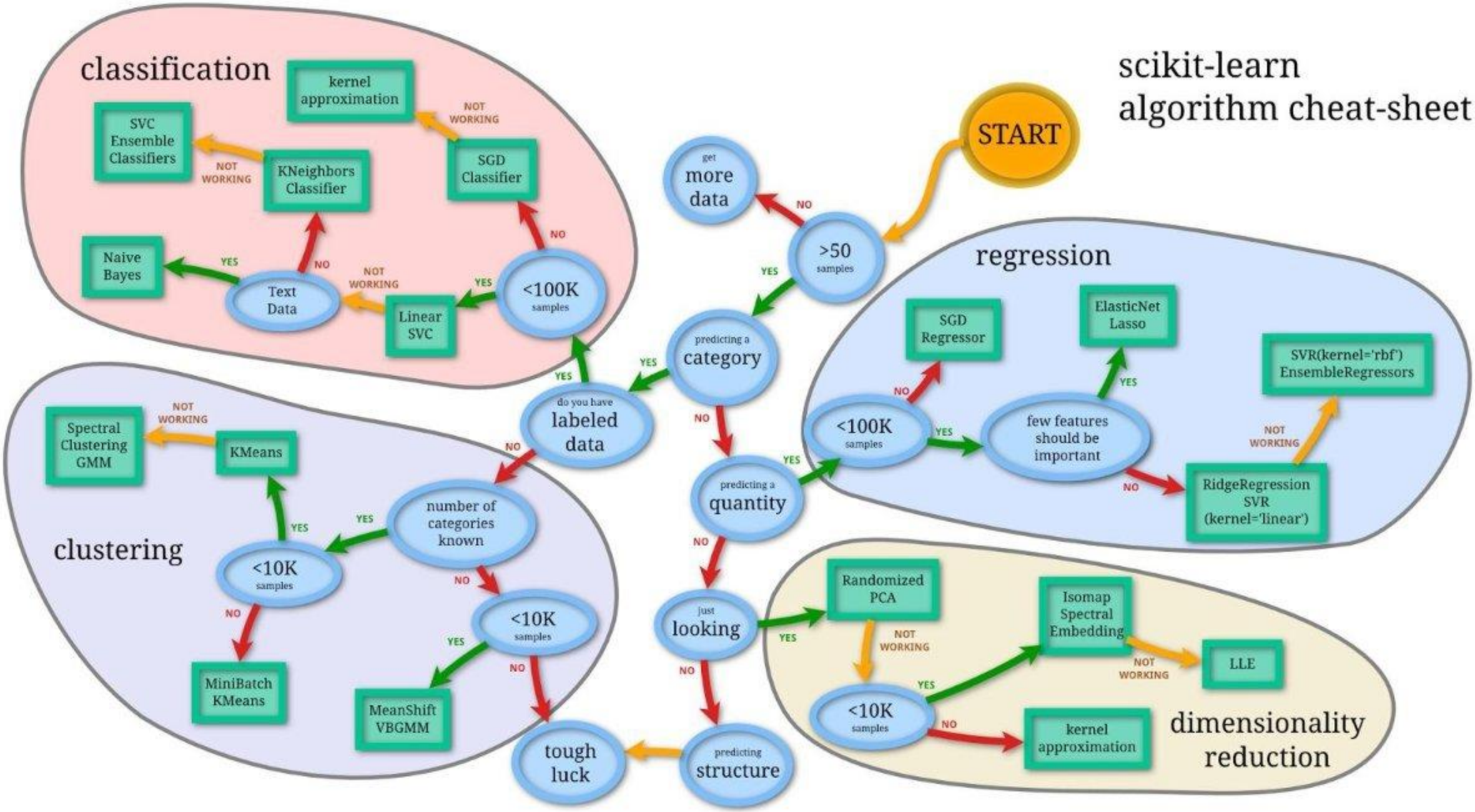
> Tune Your Model

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_)
```


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