NumPy, Pandas, and Scikit-Learn

Python's Data Science Libraries

Cameron Derwin

University of Texas at Dallas

Table of contents

local: import xxx as xx
universe: from xxx import *

Introduction

Python's Machine Learning Ecosystem

Python has a lot of Machine Learning:

- SciPy
- NumPy
- Pandas
- Scikit-Learn
- Theano
- Tensorflow
- MatPlotLib
- NLTK
- Keras
- ggplot
- Jupyter
- iPython

It's HUGE!

Today

So we're going to talk about three:

NumPy	Basic data container (arrays) and mathematical computing.
Pandas	Complicated data container (frames) and ability to manipulate it. Sort of like SQL in memory.
Scikit-Learn	The heart of Machine Learning in Python. A collection of *easy-to-use* algorithms optimized for Python. This is the library that actually does the learning.

And show them in action at the end.

NumPy

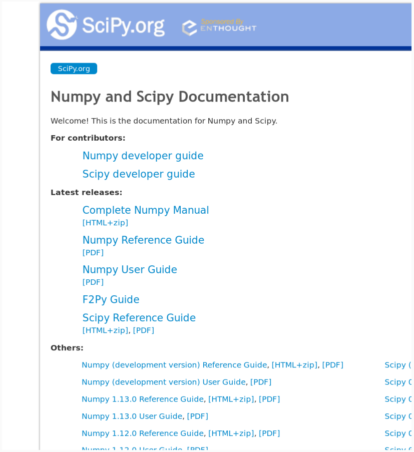
NumPy

- A scientific computing library that provides a powrful array object
- Building block for virtual all scientific computing and machine learning libraries in Python
- Provides versatile utilities to interface with C++/Fortran code
- BSD licensed, and downloaded hundreds of thousands of times

Links

Documentation:

https://docs.scipy.org/doc


SciPy.org

Numpy and Scipy Documentation

Welcome! This is the documentation for Numpy and Scipy.

For contributors:

Numpy developer guide

Scipy developer guide

Latest releases:

Complete Numpy Manual
[HTML+zip]

Numpy Reference Guide
[PDF]

Numpy User Guide
[PDF]

F2Py Guide

Scipy Reference Guide
[HTML+zip], [PDF]

Others:

Numpy (development version) Reference Guide, [HTML+zip], [PDF]

Numpy (development version) User Guide, [PDF]

Numpy 1.13.0 Reference Guide, [HTML+zip], [PDF]

Numpy 1.13.0 User Guide, [PDF]

Numpy 1.12.0 Reference Guide, [HTML+zip], [PDF]

Scipy (

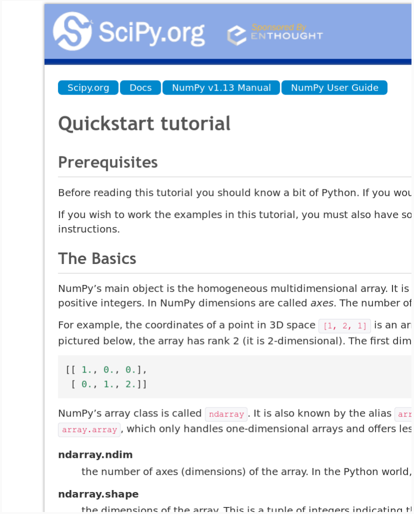
Scipy C

Scipy C

Scipy C

Scipy C

Quickstart:



Quickstart tutorial

Prerequisites

Before reading this tutorial you should know a bit of Python. If you wou

If you wish to work the examples in this tutorial, you must also have so
instructions.

The Basics

NumPy's main object is the homogeneous multidimensional array. It is
positive integers. In NumPy dimensions are called *axes*. The number of

For example, the coordinates of a point in 3D space `[1, 2, 1]` is an ar
pictured below, the array has rank 2 (it is 2-dimensional). The first dim

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

NumPy's array class is called `ndarray`. It is also known by the alias `arr`
`array.array`, which only handles one-dimensional arrays and offers les

ndarray.ndim

the number of axes (dimensions) of the array. In the Python world,

ndarray.shape

3 Parts of NumPy

The NumPy library is split into 3 parts:

Array Objects The core of NumPy is its array object (ndarray). Pandas builds upon this type and Scikit-Learn accepts it an argument for many functions.

Universals Optimized methods for fast computations on arrays of data. Can be used to vectorize python functions.

Scientific Computing NumPy provides utilities for computing FFTs and Linear Algebra quickly in Python. These go well with the SciPy library, but aren't useful for us.

We're going to talk about the first two.

Arrays

```
# Create arrays
>>> a = array([1, 2, 3, 4, 5])
array([1, 2, 3, 4, 5])

# Python type
>>> type(a)
numpy.ndarray

# System type
>>> a.dtype
dtype('int64')
```

```
# Size
>>> size(a)
4
>>> a.size
4

# Shape
>>> shape(a)
(4,)
>>> a.shape
(4,)
```

Basic Array Math

Basic math operations

```
# Multiply by constant
>>> 3 * a
array([ 3, 6, 9 ])

# Add arrays together
>> a + b
array([ 3, 5, 8, 11, 16 ])

# Apply functions
>>> exp(a)
array([ 2.71828183,   7.3890561 ,  20.08553692])
```

Size Errors

Watch out for arrays with different lengths!

```
>>> a = array([1, 2, 3])
>>> b = array([2, 3, 5, 7])

>>> a*array([3, 4, 5])
array([ 3, 8, 15])

>>> a*b
-----------------------------------------------------------
ValueError                 Traceback (most recent call last)

ValueError: operands could not be broadcast together with
shapes (3,) (4,)
```

System Measurements

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

# Bytes per item
>>> a.itemsize
8

# Bytes for whole array
>>> a.nbytes
24

# Dimensions
>>> array([[1, 2], [3, 4], [5, 6]]).ndim
2
```

Setting Values

```
>>> a = array([1, 2, 3, 4, 5])
>>> a[3] = 5
>>> a
array([1, 2, 3, 5, 5])

# Set all elements to zero
>>> a.fill(0)
>>> a
array([0, 0, 0, 0, 0])

# Same with slicing
>>> a[:] = 1
>>> a
array([1, 1, 1, 1, 1])
```

Danger: Casting

NumPy automatically casts data to the datatype of the array!

```
>>> a = arange(5)
>>> a.dtype
dtype('int64')

>>> a[1] = 3.141592
array([ 0, 3, 2, 3, 4])

>>> a.fill(2.7182)
array([ 2, 2, 2, 2, 2])
```

Slicing

```
a[start:stop:step]
```

Arrays support dynamic slicing, much like Python lists.

```
>>> a = arange(10)

# First three elements
>>> a[1:4]
array([ 1, 2, 3])

# Negative Indices
>>> a[-4:4]
array([ 1, 2, 3])
>>> a[1:-1]
array([ 1, 2, 3])
```

```
# First three elements
>>> a[:3]
array([0, 1, 2])

# Last two elements
>>> a[-2:]

# Skip elements
>>> a[::2]
array([ 0, 2, 4, 6, 8])
```

Multidimensional Arrays

```
>>> a = array([[0, 1], [2, 3],
[4, 5]])

# Get dimensions
>>> a.ndim
2

# Shape
>>> a.shape
(3, 2)

# Count
>>> a.size
6
```

Multidimensional indexing is different:

```
# Getting/setting elements
>>> a[1, 1]
3

>>> a[0, 0] = -1
>>> a
array([[ -1, 1], [2, 3],
[4, 5]])

# Single index will get row
>>> a[1]
array([ 2, 3])
```

Slices are Pointers

Slices are actually just windows into an underlying array. they are implemented with pointers.

Changing slices changes underlying values.

```
>>> a = arange(5)
>>> b = a[1:4]
>>> b
array([ 1, 2, 3])

>>> b[1] = 17
>>> b
array([ 1, 17, 3])

>>> a
array([ 0, 1, 17, 3, 4])
```

Array Slicing

```
>>> a = array([arange(3*k, 3*(k + 1)) for k in range(3)])
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
# Top Right
>>> a[:-1, 1:]
array([[1, 2],
       [4, 5]])
```

```
# Four corners
>>> a[::2, ::2]
array([[0, 2],
       [6, 8]])
```

```
# Retrieve second column
>>> a[:, 1]
array([1, 4, 7])
```

```
# Part of a row
>>> a[~~1~~0, :2]
array([0, 1])
```

More Indexing!

```
# We can slice with an array!
>>> indices = [2, 7, 3]
>>> a = arange(10, 160, 15)
>>> a[indices]
array([ 40, 115,  55])

# Even multidimenisonal arrays!
>>> a = array([arange(k, 6*k, k) for k in range(2, 7)])
>>> a[2, [3, 0, 1]]
array([16,  4,  8])
```

Warning: indexing with arrays copies the underlying array, unlike slices.

Where

Retrieves the indices of elements meeting a boolean condition

```
>>> a = arange(2, 32, 3)
>>> a
array([ 2,  5,  8, 11, 14, 17, 20, 23, 26, 29])

>>> a % 4 == 1
array([False,  True, False, False, False, True,
False, False, False,  True], dtype=bool)

>>> where(a % 4 == 1)
(array([1, 5, 9]),)

>>> a[where(a % 4 == 1)]
array([ 5, 17, 29])
```

Flatten

We can flatten multidimensional lists into 1-d list using the function `flatten` or the iterator `flat`:

```
# Flatten function
>>> a
array([[1, 2, 4],
       [3, 5, 7]])

>>> b = a.flatten()
>>> b
array([1, 2, 4, 3, 5, 7])
```

```
# Flatten copies
>>> b[1:4] = [99, 101, 88]

>>> b
array([  1,  99, 101,  88,
5,   7])

>>> a
array([[1, 2, 4],
       [3, 5, 7]])
```

Flat Iterator

To iterate of a multidimensional list without copying, NumPy provides the `flat` iterator:

```
>>> a.flat
<numpy.flatiter at 0x557dc6776b80>
```

like attribute, independance
wont effect each other

```
>>> a.flat[:]
array([ 1,  2,  4, 17,  5,  7])

>>> a.flat[3] = 17
>>> a
array([[ 1,  2,  4],
       [17,  5,  7]])
```

Reshape

```
>>> a = arange(12)
>>> a
array([ 0, 1, ..., 10, 11])

>>> a.shape
(12,)

# Set shape as 3x4
>>> a.shape = (3, 4,)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
# Reshape copies array
>>> a.reshape(6, 2)
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5],
       [ 6,  7],
       [ 8,  9],
       [10, 11]])

# Make sure sizes match!
>>> a.reshape(4, 4)
ValueError: cannot reshape
array of size 12 into
shape (4,4)
```

Creating Arrays with Non-Default Datatypes

We can create arrays wit different datatypes, both explicitly and implicitly:

```
# Implicitly
>>> a = array([0, 1, 3, 5])
>>> a.dtype
dtype(‘int64’)

>>> b = array([0, 1.0, 3, 5])
>>> b.dtype
dtype(‘float64’)
```

```
# Explicitly
>>> a = array([0, 1, 3, 5],
...           dtype=‘float’)
>>> a.dtype
dtype(‘float64’)

>>> b = array([0, 1, 3, 5],
...           dtype=‘uint8’)
>>> b.dtype
dtype(‘uint8’)
```

Typecasting

Numpy provides to typecasting methods, asarray and astype.

```
>>> a = array([0, 1, 2],
...           dtype=int32)
>>> a
array([0, 1, 2], dtype=int32)

>>> asarray(a, dtype=int64)
array([0, 1, 2])

# Asarray is efficient,
# but dangerous
>>> b = asarray(a, dtype=int32)
>>> b[2] = 17
>>> a
array([0, 1, 17])
```

```
# Astype method is similar
>>> a = array([3, -1],
...           dtype=int64)
>>> a.astype(float)
array([ 3., -1.])

>>> a.astype(uint16)
array([ 3, 65535])

# But safe
>>> b = a.astype(uint8)
>>> b[1] = 256
>>> a
array([3, -1])
```

Sum, Product, Min, and Max

Numpy provides functions to get the sum, product, min, and max of arrays:

```
>>> a = array([[0, 1], [3, -4], [-56, 228]])
>>> a.min()
-56

>>> a.max(axis=1)
array([0, -4, -56])

>>> a.prod()
0

>>> a.sum(axis=0)
array([ -53, 225])
```

Statistical Functions

Numpy also has built-in statistical functions:

```
>>> a = array([1.5, 3.6, 7.8, 9.4, 5.7, 1.5, 2.4])
>>> a.mean()
4.55714285714

>>> average(a)
4.55714285714

>>> average(a, weights=arange(len(a)))
4.3857142857142852

# Standard Deviation and Variance
>>> (a.std(), a.var())
(2.9163857979439176, 8.5053061224489799)
```

Other Mathematical Functions

Numpy also supports tons of other builtin vectorized functions:

- `sin(x)`
- `cos(x)`
- `sinh(x)`
- `cosh(x)`
- `arctan(x)`
- `arctan2(x, y)`
- `dot(x, y)`
- `cross(x, y)`
- `exp(x)`
- `log(x)`
- `absolute(x)`
- `floor(x)`
- `ceiling(x)`
- `fmod(x, y)`
- `conjugate(x)`
- `hypot(x, y)`

And more.

Vectorization

NumPy also supports vectorizing custom pure Python functions.

```
>>> def my_func(n):
...     if n % 2 == 1:
...         return 3*n + 1
...     return n/2

>>> a = arange(10)
>>> my_func(a)
ValueError: The truth value of an array with more...

>>> my_vectorized_func = vectorize(myfunc)
>>> my_vectorized_func(a)
array([  0.,   4.,   1.,  10.,   2.,
         16.,  3.,   22.,  4.,  28.])
```

Reduce

NumPy ops can reduce an array much like the Python builtin `reduce`: the `reduce` method applies an operation to all the elements of an array, keeping a running total.

For example, to calculate $1 + 2 + \cdots + 15$:

```
>>> add.reduce(arange(1, 16))
120
```

We can reduce down a multidimensional array too:

```
>>> a = array([ range(0, 6*k, k) for k in range(1, 6)])
>>> add.reduce(a)
array([ 0, 15, 30, 45, 60, 75])
```

Outer

Outer applies an operation to all pairs off elements from two arrays:

```
>>> a = arange(4)
>>> a
array([ 0, 1, 2, 3])

>>> b = arange(3)
>>> b
array([ 0, 1, 2])

>>> add.outer(a, b)
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])
```

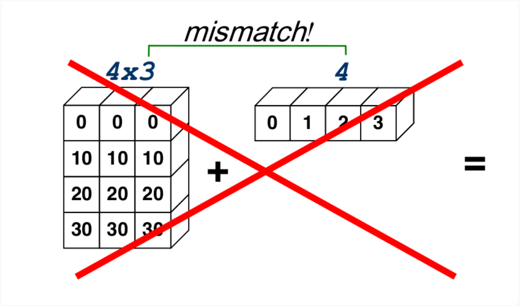
a[0] + b[0]	a[0] + b[1]	a[0] + b[2]
a[1] + b[0]	a[1] + b[1]	a[1] + b[2]
a[2] + b[0]	a[2] + b[1]	a[2] + b[2]
a[3] + b[0]	a[3] + b[1]	a[3] + b[2]

Broadcasting

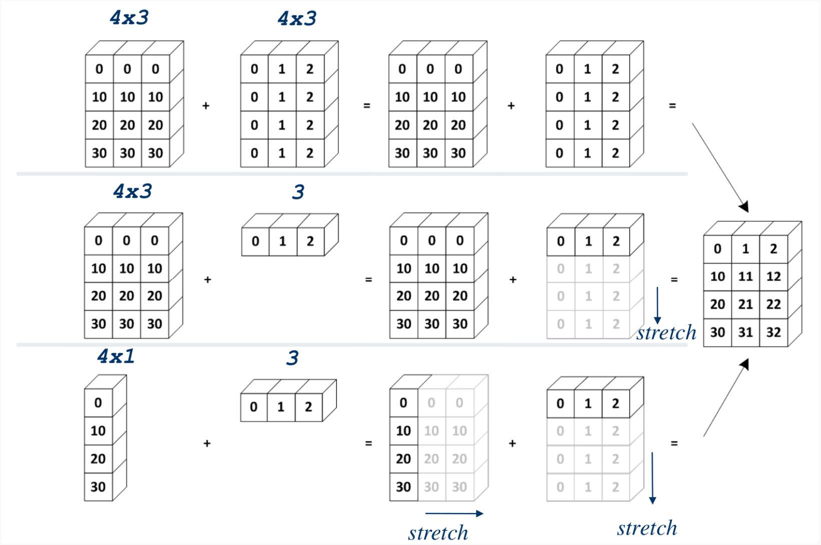
Broadcasting allows you to operate on arrays whose dimensions don't match.

At least one array (sometimes both) will be extending by copying itself in another dimension.

But if arrays have different lengths in a dimension present in both arrays, a mismatch still occurs.



Broadcasting (contd.)



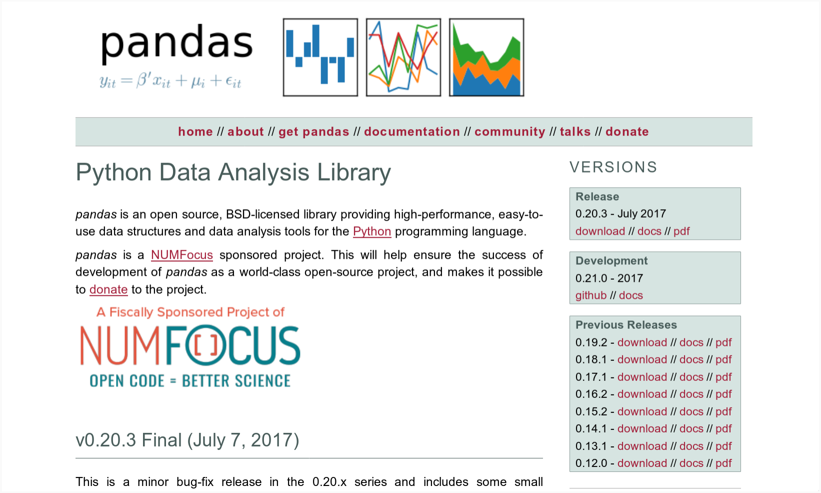
Pandas

Overview

- "high-performance, easy-to-use data structures and data analysis tools"
- Similar abilities to SQL
- Open source, BSD-licensed
- Built on NumPy, SciPy, etc.
- Similar abilities to R, MATLAB, SAS, etc.

Documentation

Url: `http://pandas.pydata.com`



pandas

$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$

home // about // get pandas // documentation // community // talks // donate

Python Data Analysis Library

pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

pandas is a NUMFocus sponsored project. This will help ensure the success of development of *pandas* as a world-class open-source project, and makes it possible to donate to the project.

A Fiscally Sponsored Project of NUMFOCUS OPEN CODE = BETTER SCIENCE

v0.20.3 Final (July 7, 2017)

This is a minor bug-fix release in the 0.20.x series and includes some small

VERSIONS

Release
0.20.3 - July 2017
download // docs // pdf

Development
0.21.0 - 2017
github // docs

Previous Releases
0.19.2 - download // docs // pdf
0.18.1 - download // docs // pdf
0.17.1 - download // docs // pdf
0.16.2 - download // docs // pdf
0.15.2 - download // docs // pdf
0.14.1 - download // docs // pdf
0.13.1 - download // docs // pdf
0.12.0 - download // docs // pdf

Data Structures

Pandas has two principle data structures:

- **Series**: array-like structure built on top of NumPy arrays
- **Data Frames**: table-like structure of data in several dimenions, much like a SQL table

Creating Series

```
>>> import numpy as np
>>> import pandas as pd

>>> pd.Series([1, 2, 3, 5, 7, np.NaN])
0    1.0
1    2.0
2    3.0
3    5.0
4    7.0
5    NaN
dtype: float64
```

Creating Data Frames

```
>>> dates = pd.date_range('20171001', periods=10)

>>> df = pd.DataFrame(
...     np.random.randn(10, 6),
...     index=dates,
...     columns=pd.Series(['a', 'b', 'c', 'd', 'e', 'f'])
... )

>>> type(df)
pandas.core.frame.DataFrame
```

What Dooes a Frame Look Like?

To see what a frame looks like:

```
>>> df.head(3)
                   a         b         c         d
2017-10-01 -0.478320 -0.489404 -0.785154 -0.801605
2017-10-02 -0.844404  0.934958 -0.367757  0.114761
2017-10-03  1.360209 -2.352517  0.041018 -1.213761

>>> df.tail(3)
                   a         b         c         d
2017-10-08 -0.961336  0.460782  0.139607 -0.273383
2017-10-09 -1.565174  0.512675 -1.697653 -0.201862
2017-10-10 -1.110614  0.903145 -0.240492  0.803236
```

Columns and Index

```
>>> df.columns
Index(['a', 'b', 'c', 'd', 'e', 'f'], dtype='object')

>>> df.index
DatetimeIndex(['2017-10-01', '2017-10-02', '2017-10-03',
               '2017-10-04', '2017-10-05', '2017-10-06',
               '2017-10-07', '2017-10-08', '2017-10-09',
               '2017-10-10'],
              dtype='datetime64[ns]', freq='D')
```

Exploration

Get the underlying NumPy array:

```
>>> df.values
array([[-0.47831971, -0.48940388, -0.78515358, -0.8016052 ,  1.1
       [-0.84440431,  0.93495827, -0.36775711,  0.11476138,  0.3
       [ 1.36020854, -2.35251659,  0.04101807, -1.21376139,  0.0
       [ 1.64858949, -1.04773512, -0.69175615, -1.58604607, -0.5
       [ 0.4390159 , -1.08251806,  0.05321245,  1.33927926, -0.2
       [-0.01432173, -1.35310927,  0.27431638,  0.51330786, -0.7
       [-0.8101361 ,  0.85425373, -0.27701164,  0.99856527, -1.0
       [-0.96133623,  0.4607824 ,  0.1396069 , -0.27338295, -0.8
       [-1.56517396,  0.51267507, -1.69765268, -0.20186219,  1.0
       [-1.11061402,  0.90314534, -0.24049214,  0.80323597, -0.1
```

Exploration

```
Get statistiical overview per column:
>>> df.describe()
               a          b          c          d
count  10.000000  10.000000  10.000000  10.000000
mean   -0.233649  -0.265947  -0.355167  -0.030751
std     1.076030   1.157670   0.585439   0.968963
min    -1.565174  -2.352517  -1.697653  -1.586046
25%    -0.932103  -1.073822  -0.610756  -0.669550
50%    -0.644228  -0.014311  -0.258752  -0.043550
75%     0.325681   0.768859   0.050164   0.730754
max     1.648589   0.934958   0.274316   1.339279
```

Sorting

Sort by index:

```
>>> df2 = df.sort_index(axis=1, ascending=False)
>>> df2.head()
                   d         c         b         a
2017-10-01  0.292376 -1.627884 -0.684372 -0.553351
2017-10-02  0.003481  1.681675  0.197446  0.436098
2017-10-03  0.191065 -0.852202  0.421321 -0.039614
2017-10-04 -1.003856 -0.354233  0.166237 -1.150083
2017-10-05  0.372840 -0.193317 -0.798528 -0.786386
```

Sorting

Sorting by value:

```
>>> df2 = df.sort_values(by='c')
>>> df2.head()
                   a         b         c         d
2017-10-01 -0.553351 -0.684372 -1.627884  0.292376
2017-10-09  0.078384  0.558567 -0.893390 -1.877662
2017-10-03 -0.039614  0.421321 -0.852202  0.191065
2017-10-10 -1.238020  1.557588 -0.629652  0.582958
2017-10-08  0.855414  1.766210 -0.365588 -1.432010
```

Fetching a Column

Fetching a column:

```
# Returns series
>>> df.a  # alternatively df['a']
2017-10-01   -0.553351
2017-10-02    0.436098
2017-10-03   -0.039614
2017-10-04   -1.150083
2017-10-05   -0.786386
2017-10-06    1.675462
2017-10-07    0.387097
2017-10-08    0.855414
2017-10-09    0.078384
2017-10-10   -1.238020
Freq: D, Name: a, dtype: float64
```

Selecting Rows

```
>>> df[:3]
                   a         b         c         d
2017-10-01 -0.553351 -0.684372 -1.627884  0.292376
2017-10-02  0.436098  0.197446  1.681675  0.003481
2017-10-03 -0.039614  0.421321 -0.852202  0.191065

# Always returns a DataFrame
>>> df[1:2]
                   a         b         c         d
2017-10-02  0.436098  0.197446  1.681675  0.003481
```

Selecting by Index Label

```
# Returns a Series
>>> dates = df.index
>>> df.loc[dates[0]]
a   -0.553351
b   -0.684372
c   -1.627884
d    0.292376
Name: 2017-10-01 00:00:00, dtype: float64
```

Selecting by Multiple Indices

```
# Can still return a DataFrame
>>> df2 = df.loc[:, ['a', 'c']]
>>> df2.head()
                   a         c
2017-10-01 -0.553351 -1.627884
2017-10-02  0.436098  1.681675
2017-10-03 -0.039614 -0.852202
2017-10-04 -1.150083 -0.354233
2017-10-05 -0.786386 -0.193317

# Can retrieve scalar value
>>> df.loc['20171003', 'b']
0.42132080046038417
```

Select by Numerical Index

Use `df.iloc`:

```
>>> df.iloc[3:5, 1:3]
                   b         c
2017-10-04  0.166237 -0.354233
2017-10-05 -0.798528 -0.193317
```

```
>>> df.iloc[7]
a    0.855414
b    1.766210
c   -0.365588
d   -1.432010
Name: 2017-10-08 00:00:00, dtype: float64
```

Boolean Indexing

Like in NumPy, we can select data by a condition:

```
>>> df[df > 0]
                   a         b         c         d
2017-10-01       NaN       NaN       NaN  0.292376
2017-10-02  0.436098  0.197446  1.681675  0.003481
2017-10-03       NaN  0.421321       NaN  0.191065
2017-10-04       NaN  0.166237       NaN       NaN
2017-10-05       NaN       NaN       NaN  0.372840
2017-10-06  1.675462       NaN  1.092537       NaN
2017-10-07  0.387097       NaN  0.033356       NaN
2017-10-08  0.855414  1.766210       NaN       NaN
2017-10-09  0.078384  0.558567       NaN       NaN
2017-10-10       NaN  1.557588       NaN  0.582958
```

Indexing on a Column

We can also select rows based on value of a single column:

```
>>> df[df.b > 0]
                   a         b         c         d
2017-10-02  0.436098  0.197446  1.681675  0.003481
2017-10-03 -0.039614  0.421321 -0.852202  0.191065
2017-10-04 -1.150083  0.166237 -0.354233 -1.003856
2017-10-08  0.855414  1.766210 -0.365588 -1.432010
2017-10-09  0.078384  0.558567 -0.893390 -1.877662
2017-10-10 -1.238020  1.557588 -0.629652  0.582958
```

Indexing with isin

Say we add a column

```
df['e'] = pd.Series([1, 2, 3, 2, 1, 1, 1, 3, 2, 2])
>>> df
                   a         b         c         d  e
2017-10-01 -0.553351 -0.684372 -1.627884  0.292376  1
2017-10-02  0.436098  0.197446  1.681675  0.003481  2
2017-10-03 -0.039614  0.421321 -0.852202  0.191065  3
2017-10-04 -1.150083  0.166237 -0.354233 -1.003856  2
2017-10-05 -0.786386 -0.798528 -0.193317  0.372840  1
2017-10-06  1.675462 -0.560729  1.092537 -1.037075  1
2017-10-07  0.387097 -0.261000  0.033356 -0.231579  1
2017-10-08  0.855414  1.766210 -0.365588 -1.432010  3
2017-10-09  0.078384  0.558567 -0.893390 -1.877662  2
2017-10-10 -1.238020  1.557588 -0.629652  0.582958  2
```

Indexing with isin

Say we want to select rows with `e = 1` or `e = 3`:

```
>>> df[df.e.isin([1, 3])]
                   a         b         c         d  e
2017-10-01 -0.553351 -0.684372 -1.627884  0.292376  1
2017-10-03 -0.039614  0.421321 -0.852202  0.191065  3
2017-10-05 -0.786386 -0.798528 -0.193317  0.372840  1
2017-10-06  1.675462 -0.560729  1.092537 -1.037075  1
2017-10-07  0.387097 -0.261000  0.033356 -0.231579  1
2017-10-08  0.855414  1.766210 -0.365588 -1.432010  3
```

Setting Columns

```
# We can set columns by creating a labelled series
>>> e = pd.Series([2.0, 3.0, 5.0, 7.0, 11.0, 13.0,
...                17.0, 19.0, 23.0, 27.0],
...      index=df.index)

>>> df['e'] = e
>>> df.head()
                   a         b         c         d     e
2017-10-01 -0.553351 -0.684372 -1.627884  0.292376   2.0
2017-10-02  0.436098  0.197446  1.681675  0.003481   3.0
2017-10-03 -0.039614  0.421321 -0.852202  0.191065   5.0
2017-10-04 -1.150083  0.166237 -0.354233 -1.003856   7.0
2017-10-05 -0.786386 -0.798528 -0.193317  0.372840  11.0
```

Setting Values

```
# We can also set individual values

>>> df.at['20171002', 'e'] = -1.
>>> df.iat[7, 4] = np.NaN
>>> df['e']
2017-10-01     2.0
2017-10-02    -1.0
2017-10-03     5.0
2017-10-04     7.0
2017-10-05    11.0
2017-10-06    13.0
2017-10-07    17.0
2017-10-08     NaN
2017-10-09    23.0
2017-10-10    27.0
Freq: D, Name: e, dtype: float64
```

Missing Values

```
# Drop missing values with ‘dropna‘
>>> df2 = df.dropna()
>>> len(df2)
9

# Fill missing values with ‘fillna‘
>>> df2 = df.fillna(-31415)
>>> df2.loc(’20171008’, ’e’)
-31415.0
```

Statictical Functions

```
# Calculate Mean by column
>>> df.mean()
a    -0.033500
b     0.236274
c    -0.210870
d    -0.413946
e    11.555556
dtype: float64
```

```
# Mean by row
>>> df.mean(1)
2017-10-01   -0.114646
2017-10-02    0.263740
2017-10-03    0.944114
2017-10-04    0.931613
2017-10-05    1.918922
2017-10-06    2.834039
2017-10-07    3.385575
2017-10-08    0.206006
2017-10-09    4.173180
2017-10-10    5.454575
Freq: D, dtype: float64
```

Applying Functions

We can also apply custom functions to each row:

```
>>> def f(x):
...     return 3 * x.max() - 17 * x.min()

>>> df.apply(f)
a    26.072730
b    18.873601
c    32.719049
d    33.669119
e    98.000000
dtype: float64
```

Much more!

- Joins & Merges
- Groupby
- Automatic plotting
- Multi-level Indices
- Time series operations

Machine Learning and scikit-learn

Goals of scikit-learn

- Optimized and easy-to-use implementations of common Machine Learning algorithms
- Full and accessible documentation
- Built on Python best-practices scientificcomputing libraries, e.g. SciPy, NumPy, pandas
- Open source and commercially available (BSD licensed)

But this raises the question **"What is machine learning?"**

Machine Learning in Abstract

- According to Wikipedia: “Machine learning is a field of computer science that gives computers the ability to learn without being explicitly programmed.”
- Russell & Norvig: Machine Learning is used to “to adapt to new circumstances and to detect and extrapolate patterns.”
- Sebastian Raschka: Machine Learning is “a subfield of artificial intelligence that involve[s] the development of self-learning algorithms to gain knowledge from ... data in order to make predictions.”

In other words, machine learning is **modelling**.

Machine Learning in Practice

A typical machine learning workflow might look like this:

1. start with data
2. cleaning/munging data, feature selection
3. build models with a subset of data (¡50%)
4. evaluate models with remaining data
5. select best model or use ensemble model on new data

Feature Engineering

A good model requires good features. This means:

- Features that strongly correlate to outcomes
- Features that are relatively independent
- Features that fit on the same scale

Feature engineering is often underemphasized because there is no universal approach: it requires domain expertise.

An Example

Here is an example centered on Baseball:

Predicting Home Runs Good features would be stats like batting average, slugging percentage, strikeouts, number of doubles, triples, etc. Bad stats would be walks, hit-by-pitches, fielding percentage, etc.

Predicting Steroid Use On the other hand, in predicting steroid use we want to look at changes in performance year over year. Features could include percent increases in home runs, batting average, slugging percentage, etc.

Basic Types of Learning

There are three basic types of learning algorithms.

Classification	Classifies data into one of several predetermined groups. For example, predicting steroid use is a binary classification problem.
Regression	Predicts a value from a continuous spectrum. For example, predicting a given player's number of home runs next season.
Clustering	Separates data into several undetermined groups.

Classification

- Each sample has features $s = (f_1, \ldots, f_n)$ and a classification c_s
- c_s can only take on a finite (usually small) number of values
- Algortihm predicts c_s based on given values of $s = (f_1, \ldots, f_n)$ of a sample
- Considered supervised learning: requires a large set of training data to be classified by hand
- Example: spam detection and email classification. Based on various attributes of an email (sender, length, content) predict whether or not an email should be automatically sent to a particular folder.

Regression

- Each sample has features $s = (f_1, \ldots, f_n)$ and a value $v(s)$
- Values $v(s)$ can take on a large number (often infinite) number of values
- Algorithm predicts the value $v(s)$ given a sample $(f_1, \ldots, f_n)$
- Supervised learning: training data must already have values associated with each sample
- Example: predicting home value in a local market based on square footage, location, year built, etc.

Clustering

- Each sample has features $s = (f_1, \ldots, f_n)$ associated with it, and no other data
- Goal is to partition the samples into n groups in a way that extends well to new data
- Algorithm takes in values $s = (f_1, \ldots, f_n)$ of a sample and assigns it a cluster c_k
- Considered unsupervised learning since no values must be assigned manually first
- Example: Market segmentation: finding which groups of customers share common attributes that can be leveraged to create effective targeted marketing.

Good News

- `scikit-learn` makes it simple to apply algorithms without fully understanding how they work
- `scikit-learn` is production-ready and is considered the industry standard for implementations of many algorithms
- `scikit-learn` has incredible documentation that ranges in technical content – great for learning about algorithms

Bad News

- Algorithms are complicated
- Many require a strong grasp of linear algebra and probability/statistics
- Some requiring tuning parameters and have other technical roadblocks

As a result, I will focus on **how to use** various algorithms included in `scikit-learn`, rather than how the algorithms themselves work.

Resources

If you want to know more about the inner workings of these algorithms, these are great resources:

- “Artificial Intelligence: A Modern Approach”, Stuart Russell & Peter Norvig
- “Python Machine Learning”, Sebastian Raschka

Data

It will be useful to work with a single dataset for the examples in this section.

- We will be using the classical Iris dataset
- It is included in `scikit-learn` and can be accessed through `sklearn.datasets.load_iris`
- It includes 150 samples, each with four features (sepal length, sepal width, petal length, and petal width), classified by species (three species)
- We separate the data into `train_data`, `train_labels`, `test_data`, and `test_labels`, where the length of the test data is 45 and the training data is 105 (.30/.70 split)

The Set Up

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> iris = load_iris()

>>> indices = np.arange(150)
>>> np.random.shuffle(indices)

# We’ll talk about the ‘train_test_fit‘ function in a sec
>>> labels = iris.target_names
>>> (train_data, test_data,
...  train_labels, test_labels) = train_test_split(
...      iris.data, iris.target, test_size=0.3)
```

Running Models in scikit-learn

- Models are classes
- Instances of models can be trained, make predictions, etc.
- Models are intantiated by passing the constructor **hyperparameters**
- Use `fit` to train a model
- Use `predict` for raw predictions or `predict_proba` for probabilities

```
>>> from sklearn import MyModel

# Instantiate with hyperparameters
>>> model = MyModel(*hyperparams)

# Train with ‘fit‘ method
>>> model.fit(X, Y)

# Use model
>>> model.predict(untrained_data)
```

Enhancing Features & Best Practices

- Because feature engineering is hard, there are algorithms to help
- Dimensionality reduction reduces many, inter-correlated features into fewer, independent features
- Feature scaling normalizes features to the same scale (often 0.0 to 1.0). many algorithms are sensitive to unnormalized data.
- Separate training data from test data (randomly) to aviod and recognize **overfitting**
- Evaluate several models and choose best or several

Principal Component Analysis

- Projects data onto new set of orthogonal features that minimizes covariance and maximizes independence
- Mathematically it is formulated using the Singular Value Decomposition (SVD) from linear algebra
- Number of components must be predetermined: `n_components=k` hyperparameter.
- Automatically normalizes variances of the data with `whiten=True`
- Run PCA with `transform` method

Example

```
>>> from sklearn.decomposition import PCA

>>> model = PCA(n_components=4, whiten=True)
>>> model.fit(train_data)
>>> transformed_data = model.transform(train_data)
>>> transformed_test_data = model.transform(test_data)
```

Data Normalization

Some algorithms require all features to be on a similar scale. Features can be scaled by `MaxAbsScaler`:

```
>>> from sklearn.preprocessing import MaxAbsScaler

>>> model = MaxAbsScaler()
>>> transformed_data = model.fit_and_transform(train_data)
>>> transformed_test_data = model.transform(test_data)
```

Data Cleaning

- In addition to normalization, convert boolean and other non-numeric featurs into numbers
- Come up with strategy to handle missing data
- Handle biases in the data (for example, disproportionate data from particular labels and/or value ranges)
- Without these, accuracy of models decreases significantly
- Important to remember to preprocess new data

Cross Validation

- Important to have unbiased way to evaluate models
- Data must be split before any modls are trained to avoid biases and evaluation of overfitting.
- Models must be evaluated in a way that is indepndent of type of learning used

Splitting Data

To split data, we use `sklearn.model_selection.train_test_split`. Takes several arguments (numpy arrays) and the desired split between testing and training data (proportions via the `test_size`/`train_size` parameters):

```
>>> samples, labels
array([...]), array([...])

>>> split = test_train_split(samples, labels, test_size=0.25)

>>> train_data = split[0]
>>> test_data = split[1]

>>> train_labels = split[2]
>>> test_labels = split[3]
```

Scoring Models

Rudimentary scoring for classification: `accuracy_score`. Simply reports percentage of test data correctly classified.

```
>>> model.fit(my_data, my_labels)

>>> predictions = model.predict(test_data)
>>> accuracy_score(predictions, expected_labels)
0.95
```

Honorable Mentions

Tuning Hyperparameter for a model:

- **Exhaustive Grid Search**: test many values in a high-dimensional lattice to determine optimal setting for parameters. Computationally intensive, takes a long time. Easily parallelized.
- **Randomized Parameter Optimization**: draw possible values for parameters from a sample from a given distribution.
- **Model-specific Optimization**: Some models can be exploited to choose best parameters more efficiently.

Algorithms We Won't Talk About

- Least Squares
- Logistic Regression
- Bayesian Regression
- Linear Discriminant Analysis
- Kernel Ridge regression
- Gaussian Processes
- Gradient Tree Boosting
- Isotonic Regression
- Bag of Words
- Variance Scaling
- Voting Classifiers
- Polynomial Regression
- gaussian Mixture Models
- Manifold Learning
- Affinity Propogation
- Mean-shift
- Spectral Clustering
- Latent Semantic Analysis
- Exhaustive Grid Search
- Randomized Parameter Optimization

Algorithms We Will Talk About

- Naive Bayes
- *k*-Nearest Neighbors
- Support Vector Machine (SVM)
- Kernelized SVMs
- Stochastic Gradient Descent
- *k*-Means
- Decision trees
- Random forests
- AdaBoost
- Principle Component Analysis (PCA)
- Cross Validation
- Feature Normalization

Naive Bayes

- Extremely simple mathematically but perhaps most unintuitive algorithm we'll talk about
- Despite simplicity and flawed assumptions, extremely competitive for text classification today
- Assumes features $(f_1, \ldots, f_n)$ are independent
- The probability of a sample $s = f_1, \ldots, f_n)$ being a given class $c_1, \ldots, c_k$ is

$$P(c_i|f_1, \ldots, f_n) = kP(c_i) \prod_{i=1}^{n} P(f_i|y)$$

- $P(c_i)$ and $P(f_i|y)$ can be calculated using relative frequency in the training dataset
- Fast to train and fast to run

Distributions

The differences between implementations in Naive Bayes are in the distributions they assume:

- `GaussianNB`: assumes probabilities are distributed normally (i.e., according to a Bell Curve)
- `MultinomialNB`: Multinomial distribution. Classically very successful in text classification using word counts
- `BernoulliNB`: Bernoulli distribution. Assumes features are binary 1/0 values. If they are not, may convert features to binary.

Gaussian Naive Bayes on Irises

```
>>> from sklearn.naive_bayes import GaussianNB

>>> model = GaussianNB()
>>> model.fit(train_data, train_labels)

>>> predicted_labels = model.predict(test_data)
>>> accuracy_score(predicted_labels, test_labels)
0.977777777778
```

k-Nearest Neighbors

- Classification algorithm
- Training consists of simply storing the dataset (fast but memory intensive)
- Classifies new data points by finding the k nearest training samples to the new data and using the classification of those points
- Algorithm can use a variety of metrics
- Computationally intensive for large training datasets
- Can also be used for regression

Classification using k Nearest Samples

- Majority vote
- Weighted vote
- Regression: average or weighted average of values of neighbors

Alternative to nearest neighbors: select all neighbors with some radius r

Example

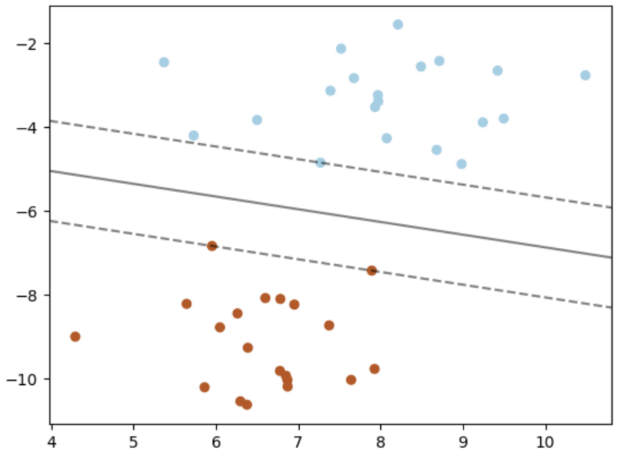
```
# Simple k-NN
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.metrics import accuracy_score

>>> n_neighbors = 5
>>> weights = ‘distance’

>>> model = KNeighborsClassifier(n_neighbors, weights=weights)
>>> model.fit(train_data, train_labels)

>>> predicted_labels = model.predict(test_data)
>>> accuracy_score(predicted_labels, test_labels)
0.97777777777777775
```

Support Vector Machines

- Classification Algorithm
- Can omly be immediately applied to binary classification: two workarounds
- Partitions feature space by a plane
- Uses convex optimization to find optimal plane that separates data
- Training can take a long time for large data sets

Geometric Representation

In 2-dimensions what does an SVM look like?



Multi-class Methods

Suppose we have n classes $c_1, \ldots, c_n$.

One against One Create $\binom{n}{2} = \frac{n(n-1)}{2}$ models that classify a sample as either c_i or c_j for each (i, j). New data is labelled by the class that “wins” most often.

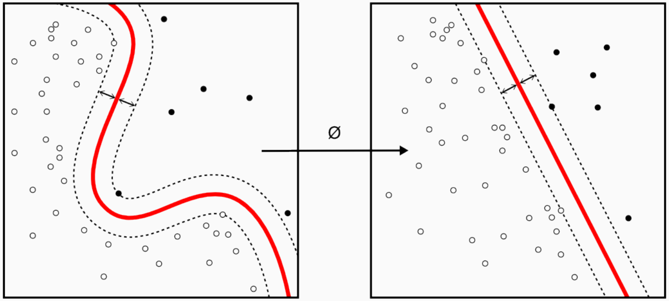
One versus the Rest Create n models that classify a sample as either c_i or any of the rest $c_1 \cup \cdots \cup c_{i-1} \cup c_{i+1} \cup \cdots \cup c_n$. New data is labelled as the class furthest away from the dividing hyperplane.

SVM Classifier in Code

```
>>> from sklearn.svm import LinearSVC, SVC

>>> model1 = LinearSVC()
>>> model1.fit(train_data, train_labels)

>>> model2 = SVC(kernel=‘linear’)
>>> model2.fit(train_data, train_labels)

>>> predicted_labels_1 = model1.predict(test_data)
>>> predicted_labels_2 = model2.predict(test_data)

>>> accuracy_score(predicted_labels_1, test_labels)
0.9555555555555556

>>> accuracy_score(predicted_labels_2, test_labels)
1.0
```

Kernelized SVMs

- As the second example might suggest, there are non-linear SVMs!
- Several different options available for the `SVC` class
- Can take too long to train
- The curse of dimensionality



What Kernels are Available?

- Linear (onbviously)
- Polynomial: `kernel=‘poly’`, with degree `degree` and offset `coeff0`
- Exponential: `kernel=‘rbf’`, with scaling parameter `gamma`
- Sigmoid function (tanh): `kernel=‘sigmoid’`, offset by parameter `coeff0`
- Custom kernels can by specified by Python functions

Example

Iris data using polynomial and exponential kernels:

```
>>> from sklearn.svm import SVC
>>> model1 = SVC(kernel=‘poly’, degree=5)
>>> model1.fit(train_data, train_labels)
>>> model2 = SVC(kernel=‘rbf’)
>>> model2.fit(train_data, train_labels)
>>> predicted_labels_1 = model1.predict(test_data)
>>> predicted_labels_2 = model2.predict(test_data)
>>> accuracy_score(predicted_labels_1, test_labels)
0.933333333333

>>> accuracy_score(predicted_labels_2, test_labels)
0.977777777778
```

Stochastic Gradient Descent

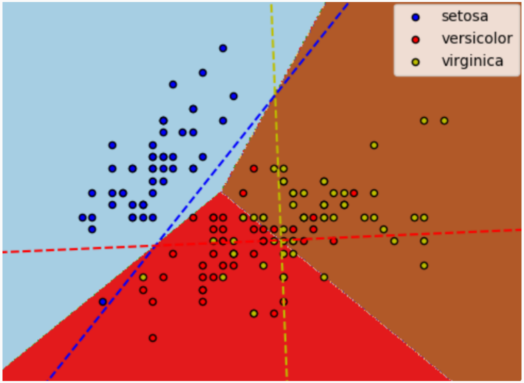
- Alternative to Linear SVMs
- Can be applied to several other models (logistic regression, etc.)
- Significantly faster: uses randomized optimization
- Often scales to feature arrays larger than $10^5 \times 10^5$ (such as in nlp)
- Utilizes several hyperparameters
- Sensitive to scaling (like SVMs)
- Often used when SVMs are *not an option*
- Uses one-versus-all formulticlass classification

SGD Parameters

Linear SVM parameters:

- Loss parameter: `loss='hinge'`
- Penalty parameter: `penalty='l2'`

Multi-class SGD:



Example

Stochastic Gradient descent performs significantly less well on the Iris dataset:

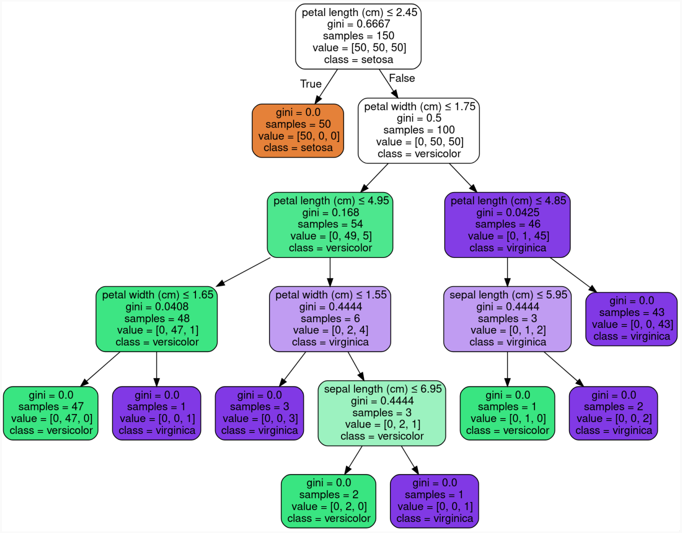
```
>>> from sklearn.linear_model import SGDClassifier

>>> loss = 'hinge'
>>> penalty = 'l2'

>>> model = SGDClassifier(loss=loss, penalty=penalty, max_iter=1
>>> model.fit(train_data, train_labels)

>>> predicted_labels = model.predict(test_data)
>>> accuracy_score(predicted_labels, test_labels)
0.7555555555555554
```

Decision Trees

- Classification and Regression Algorithm
- White Box model (transparent)
- Works by making small decisions at each step
- For example, partitioning data based on values of a single feature
- Tendency to overfit
- Bias in unbalanced data sets
- Hard to represent certain relationships (xor, parity, etc.)

How it Works

A decision tree for the iris problem:



Accuracy on Iris Dataset

```
>>> from sklearn.tree import DecisionTreeClassifier

>>> model = DecisionTreeClassifier(min_samples_split=7, min_samp
>>> model.fit(train_data, train_labels)

>>> predicted_labels = model.predict(test_data)
>>> accuracy_score(predicted_labels, test_labels)
0.933333333333
```

Random Forests

- Randomly generates a “forest” of shallow decision trees
- Randomized by only considering a random subset of features at each branch
- Averages results to predict class and/or value (used for both classification and regression
- Number of trees: `n_estimators=k`. A large number of trees will take significantly longer to traing, and because we average the result the marginal benefit of adding more trees is inversely proportional to the number of trees.
- Number of features: `max_features=l` is the number of features of the model to consider at each branch.
- Easy parellelization: `n_jobs=k`

Extremely Randomized Trees and Total Random Embeddings

Two variations on random forests deserve mention.

Extremely Randomized Trees:

- Additional randomness: instead of picking the optimal threshhold for splitting a sample of randomly chosen features, extremely random trees pick the best of several randomly generated threshholds
- Otherwise, they operate exactly the same as random forests
- Decreased variance and increased bias

Total Random Embeddings:

- Preprocessing technique
- Runs data through an entirely randomly generated forest of trees
- Re-encodes data by indices of the leaves

Performance on the Iris Dataset

```
>>> from sklearn.ensemble import RandomForestClassifier,
...                              ExtraTreesClassifier
>>> n_estimators = 25

>>> model1 = RandomForestClassifier(n_estimators=n_estimators)
>>> model1.fit(train_data, train_labels)
>>> model2 = ExtraTreesClassifier(n_estimators=n_estimators)
>>> model2.fit(train_data, train_labels)

>>> predicted_labels_1 = model1.predict(test_data)
>>> predicted_labels_2 = model2.predict(test_data)
>>> accuracy_score(predicted_labels_1, test_labels)
0.933333333333
>>> accuracy_score(predicted_labels_2, test_labels)
0.955555555556
```

AdaBoost

- Popular ensemble classification and regression algorithm
- “Boosts” samples by weighting features differently in each tree
- Each subsequent tree focuses on predicting errors from previous ones
- Scores new data on weighted average of trees
- Can be applied to weak learners other than shallow trees

Iris Dataset Performance

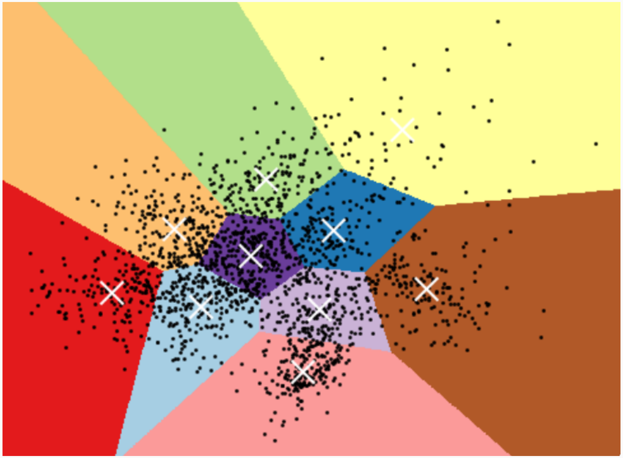
```
>>> from sklearn.ensemble import AdaBoostClassifier

>>> model = AdaBoostClassifier(n_estimators=100)
>>> model.fit(train_data, train_labels)

>>> predicted_labels = model.predict(test_data)
>>> accuracy_score(predicted_labels, test_labels)
0.911111111111
```

k-Means

- Clustering algorithm! (unsupervised learning)
- Start with k randomly selected means
- Assign samples to the cluster corresponding to closest mean
- Recenter means to the centroid (i.e. average) of data points
- Reassign sample points and repeat until means converge
- Assign new points to clusters based closest mean

Voronoi Diagrams

Method can be visualized with Voronoi diagrams: the plane is partitioned into each regions defined by proximity to the means.



How Well Do Iris Clusters Match Species?

```
>>> from sklearn.cluster import KMeans
>>> from sklearn.datasets import load_iris

>>> iris = load_iris()

>>> model = KMeans(n_clusters=3)
>>> model.fit(iris.data)

# Number of irises in clusters corresponding to wrong species
>>> bins = [np.bincount(iris.target[
...     np.where(model.labels_ == i)]) for i in range(3)]
>>> [b.sum() - b.max() for b in bins]
[2, 0, 14]
```