# THE BIG PICTURE

## Data Science and Data Analysis

Data analysts carry out their job of engaging data using a well refined, multi-phased process. Their goal is to interpret raw data by converting it into useful and actionable intelligence. What does this process entail exactly?

* **Collecting Data** from various sources
* **Wrangling Data** to make it more reliable
* **Exploring Data** using statistics and visualizations
* **Transforming Data** to prepare it for modeling
* **Modeling Data** using the right machine learning algorithms
* **Evaluating** the results of the data models

Collecting Data :

To be effective at analysis, you are going to need data and a lot of it.

Data might be come from a variety of sources in the physical world

## Machine Learning

Types of Machine Learning:

* **unsupervised learning**
* **supervised learning**
* **reinforcement learning.**

### Unsupervised Learning

Given a lot of data, the computer hasn't the slightest idea what any of it means. figure out if there are any meaningful groupings and patterns within the data, along with instances where that data seems out of place

### Supervised Learning

with supervised learning, the computer is in charge of taking your data and then fitting rules and equations to it. Once it learns these generalized rules through a process called modeling, the rules can be applied to data the computer has never seen before.

## The Possibilities

### Classification

The goal of classification is to identify what class a sample belongs to.

Examples

* Given a list of emails marked spam and not-spam, figure out if a newly received message is actually spam or not.
* Given many different images of your friends, perform facial recognition on a new, never-before seen image of one of your friends.
* After being trained with a few books, decide which of the before seen authors wrote an unidentified article.
* Given a list of physical symptoms, determine what ailment a person has.

Classification falls into the realm of supervised learning because for it to work, you have to guide the computer by proving it with examples of correctly labeled records. Once you're done training the computer, you can test it by seeing how accurately it scores those new records.

### Regression

The goal of regression is to predict continuous-valued features associated with many samples. Continuous-valued meaning small changes in the input result in small changes of the output.

With regression, a mathematical relationship is modeled for your samples so that as you gently alter an input feature, your output feature(s) respond by being altered as well.

Examples

* Calculate an equation to predict the size of a house given its price, or the price of a house given its size.
* Explore if a correlation exists between the hours a student spends studying, spends watching TV, and their final exam score.
* Estimate how many power plants should be constructed in the next 50 years, based upon the historical energy consumption per household.
* Figure out how many days a person has left to live based on the severity of their symptoms.

Regression falls into the realm of supervised learning because you have to provide the computer with labeled samples. It then attempts to fit an equation to the samples' features.

### Clustering

The goal of clustering is to automatically group similar samples into sets.

Since a clustering algorithm has no prior knowledge of how the sets should be defined, and furthermore, since the clustering process is unsupervised, it needs to have a way to tell which samples are the most similar so it can group them accordingly. It does this the same way we humans do: by looking at the various characteristics and features of each sample.

Examples

* Match similar people on a matrimonial website based on their profile answers.
* Through examining search history, recommend houses a prospective home-buyer might be interested in buying.
* Pinpoint the most likely location for a future earthquake using past seismic data.
* Identify new characteristics shared by different people suffering from the same disease.

There are different types of clustering algorithms, some supervised, some unsupervised. There are even semi-supervised clustering methods as well.

In this course, you'll only be dealing with unsupervised clustering. In other words, the clustering algorithm you will use won't need anything except your raw data. No labels hinting at desired clustering outcome will be provided to the algorithm.

### Dimensionality Reduction

The goal of dimensionality reduction is to systematically and intelligently reduce the number of features considered in a dataset.

Examples

* Given a 100 question survey, attempt to find the gist of what is truly being assessed; then rephrase it with just 5 questions.
* Build a robot that can recognize pictures of similar objects, even if they are rotated at odd angles and orientations.
* Compress a video stream by reducing the number of colors.
* Summarize a long book.

Dimensionality reduction falls into the realm of unsupervised learning because you don't instruct the computer which features you want it to create. Rather, the computer infers this information automatically by examining your unlabeled data.

### Reinforcement Learning

The goal of reinforcement learning is to maximize a cumulative reward function--or equivalently, minimize a cost function--given a set of actions and results.

Reinforcement learning is modeled to mimic the way we learn in the real world. We try to solve problems using different techniques. Most of the time, nothing of merit results from our experiments. But occasionally, we stumble upon a set of actions that cause a sweet reward. When this happens, we attempt to repeat those actions that get us rewarded.

If we are rewarded yet again, we further associate those actions with the reward, and this is known as the reinforcement cycle. The entire process is also known as performance maximization.

Examples

* Discover how to fly a quadcopter by minimizing the function which evaluates the chances of it crashing.
* Learn to beat a video game like 'Super Mario Bros.' by minimizing the time it takes to get to the castle.
* Attempt to take a photo and "re-draw" it in the style of a particular artist.
* Automate the trading of stocks and securities by maximizing profit and minimizing transaction fees.

Reinforcement learning is a bit different from regular supervised and unsupervised learning. To illustrate this difference, imagine you had a rubick's cube. With unsupervised learning, none of your data has a label. Due to this, you neither know if the cube is in a solved state, an unsolved state, or even what you need to do with it! All you can do is learn about the structure of the cube--namely that it has six distinct colors, and 9 pieces per face, all belonging to a specific color.

With supervised learning, your aim might be to find if the rubick's cube is in a solved configuration or not. To teach a computer this, you'd have to show it many hundreds of examples of rubick's cubes, each labeled solved or not-solved. We refer to this 'dense' labeling.

With reinforcement learning, your goal might be something like given a rubick's cube in some initial configuration, derive a set of actions which would result in it being solved. Training a computer to do this would involve having the computer attempt to change the state of the cube many times, and then after a while, informing the computer if it's succeeded or failed. With enough trials, the computer will get better at quickly doing those actions which result in a successfully solved puzzle, irrespective of the initial configuration of the cube. As such, reinforcement is a type of 'sparsely' labeled, semi-supervised learning.

# DATA AND FEATURES

## Features Premiere

Features are those quantitative traits(samples) that describe your samples, and they can be numeric or textual.

## Feature Types

There are many synonymous names for features.

* *Attribute* - Features are a quantitative attributes of the samples being observed
* *Axis* - Features form orthogonal axes in their feature-space, if they are linearly independent
* *Column* - Features are represented as columns in your dataset
* *Dimension* - A dataset's features, grouped together can be treated as an n-dimensional coordinate space
* *Input* - Feature values are the input of data-driven, machine learning algorithms
* *Predictor* - Features used to predict other attributes are called predictors
* *View* - Each feature conveys a quantitative trait or perspective about the sample being observed
* *Independent Variable* - Features used to calculate others are like independent variables in algebraic equations

### Continuous Features

Measurable. They are usually a subset of all real numbers. Some example features are distance , time , cost , temperature.

### Categorical Features

there is a specified number of discrete, possible feature values.

#### Nominal

* Car Models
* Colors
* TV Shows

#### Ordinal

* High-Medium-Low
* 1-10 Years Old, 11-20 Years Old, 30-40 Years Old
* Happy, Neutral, Sad

## Determining Features

Your machine learning goal should be to train your algorithms instead of hard coding them.

## Manipulating Data

### Loading Data

To get started with Pandas, import it:

import pandas as pd

There are two data structures in Pandas you need to know how to work with.

#### Series

a one-dimensional, labeled array that represents a single column in your dataset.

all elements in a series must be of the same data type in Panda.

#### Dataframe

a collection of series. load dataset into a dataframe:

from sqlalchemy import create\_engine

engine = create\_engine('sqlite:///:memory:')

sql\_dataframe = pd.read\_sql\_table('my\_table', engine, columns=['ColA', 'ColB'])

xls\_dataframe = pd.read\_excel('my\_dataset.xlsx', 'Sheet1', na\_values=['NA', '?'])

json\_dataframe = pd.read\_json('my\_dataset.json', orient='columns')

csv\_dataframe = pd.read\_csv('my\_dataset.csv', sep=',')

table\_dataframe = pd.read\_html('http://page.com/with/table.html')[0]

* sep
* delimiter
* header
* names
* index\_col
* skipinitialspace
* skiprows
* na\_values
* thousands
* decimal

my\_dataframe.to\_sql('table', engine)

my\_dataframe.to\_excel('dataset.xlsx')

my\_dataframe.to\_json('dataset.json')

my\_dataframe.to\_csv('dataset.csv')

rename column names, use .columns property;

my\_dataframe.columns = ['new', 'column', 'header', 'labels']

get a quick peek at your data by selecting its top or bottom few rows using .head() and .tail():

##### Dataframe Summary

To see a descriptive statistical summary of your dataframe's numeric columns using .describe():

##### View Columns

.columns will display the name of the columns in your dataframe:

>>> df.columns

Index([u'name', u'age', u'location'], dtype='object')

##### View Indices

to display the index values, enter .index:

>>> df.index

RangeIndex(start=0, stop=649, step=1)

In Pandas, axes refers to the two-dimensional, matrix-like shape of your dataframe. Samples span horizontal rows and are stacked vertically on top of one another by index (axis=0). Features are vertical spans that are stacked horizontally next to each other by columns (axis=1)

##### View Column DataTypes

>>> df.dtypes

recency int64

history\_segment object

history float64

mens int64

womens int64

zip\_code object

newbie int64

channel object

segment object

visit int64

conversion int64

spend float64

DM\_category int64

dtype: object

### Slicin

Indexing or slicing

>>> df = pd.read\_csv('Datasets/direct\_marketing.csv')

>>> df

recency history\_segment history mens womens zip\_code newbie

0 10 2) $100 - $200 142.44 1 0 Suburban 0

1 6 3) $200 - $350 329.08 1 1 Rural 1

2 7 2) $100 - $200 180.65 0 1 Suburban 1

3 9 5) $500 - $750 675.83 1 0 Rural 1

4 2 1) $0 - $100 45.34 1 0 Urban 0

5 6 2) $100 - $200 134.83 0 1 Suburban 0

#### Column Indexıng

>>> df.recency

>>> df['recency']

>>> df[['recency']]

>>> df.loc[:, 'recency']

>>> df.loc[:, ['recency']]

>>> df.iloc[:, 0]

>>> df.iloc[:, [0]]

The .loc[] method selects by column label, and .iloc[] selects by column index.

# Results in a series object:

>>> df.recency

>>> df['recency']

>>> df.loc[:, 'recency']

>>> df.iloc[:, 0]

0 10

1 6

2 7

3 9

4 2

5 6

Name: recency, dtype: int64

# Results in a dataframe object:

>>> df[['recency']]

>>> df.loc[:, ['recency']]

>>> df.iloc[:, [0]]

recency

0 10

1 6

2 7

3 9

4 2

5 6

[64000 rows x 1 columns]

#### Row Indexing

You can use either the .loc[] or .iloc[] methods to do selection by row, noting that the expected order is [row\_indexer, column\_indexer]:

>>> df[0:2]

>>> df.iloc[0:2, :]

recency history\_segment history mens womens zip\_code newbie channel

0 10 2) $100 - $200 142.44 1 0 Suburban 0 Phone

1 6 3) $200 - $350 329.08 1 1 Rural 1 Web

The last important difference is that .loc[] is inclusive of the range of values specified, whereas .iloc[] is non-inclusive. In that sense, df.loc[0:1, :] would select the first two rows, but only the first row would be returned using df.iloc[0:1, :].

### Boolean Indexing

dataframes and series can also be indexed with a conditional operation

>>> df.recency < 7

0 False

1 True

2 False

3 False

4 True

5 True

Name: recency, dtype: bool

>>> df[ df.recency < 7 ]

recency history\_segment history mens womens zip\_code newbie

1 6 3) $200 - $350 329.08 1 1 Rural 1

4 2 1) $0 - $100 45.34 1 0 Urban 0

5 6 2) $100 - $200 134.83 0 1 Suburban 0

>>> df[ (df.recency < 7) & (df.newbie == 0) ]

recency history\_segment history mens womens zip\_code newbie

4 2 1) $0 - $100 45.34 1 0 Urban 0

5 6 2) $100 - $200 134.83 0 1 Surburban 0

#### Writing to a Slice

>>> df[df.recency < 7] = -100

>>> df

recency history\_segment history mens womens zip\_code newbie

0 10 2) $100 - $200 142.44 1 0 Suburban 0

1 -100 -100 -100.00 -100 -100 -100 -100

2 7 2) $100 - $200 180.65 0 1 Suburban 1

3 9 5) $500 - $750 675.83 1 0 Rural 1

4 -100 -100 -100.00 -100 -100 -100 -100

5 -100 -100 -100.00 -100 -100 -100 -100

## Feature Representation

### Textual Categorical-Features

In the case of ordinals, you should create a mapping of increasing integers to each possible unique value of the feature. Any entries not found in your designated categories list will be mapped to -1:

>>> ordered\_satisfaction = ['Very Unhappy', 'Unhappy', 'Neutral', 'Happy', 'Very Happy']

>>> df = pd.DataFrame({'satisfaction':['Mad', 'Happy', 'Unhappy', 'Neutral']})

>>> df.satisfaction = df.satisfaction.astype("category",

ordered=True,

categories=ordered\_satisfaction

).cat.codes

>>> df

satisfaction

0 -1

1 3

2 1

3 2

if your feature is nominal;

>>> df = pd.DataFrame({'vertebrates':['Bird', 'Bird', 'Mammal', 'Fish', 'Amphibian', 'Reptile', 'Mammal']})

# Method 1)

>>> df['vertebrates'] = df.vertebrates.astype("category").cat.codes

>>> df

vertebrates vertebrates

0 Bird 1

1 Bird 1

2 Mammal 3

3 Fish 2

4 Amphibian 0

5 Reptile 4

6 Mammal 3

Pandas automatically encodes your nominal entries in alphabetical order.

# Method 2)

>>> df = pd.get\_dummies(df,columns=['vertebrates'])

>>> df

vertebrates\_Amphibian vertebrates\_Bird vertebrates\_Fish \

0 0.0 1.0 0.0

1 0.0 1.0 0.0

2 0.0 0.0 0.0

3 0.0 0.0 1.0

4 1.0 0.0 0.0

5 0.0 0.0 0.0

6 0.0 0.0 0.0

vertebrates\_Mammal vertebrates\_Reptile

0 0.0 0.0

1 0.0 0.0

2 1.0 0.0

3 0.0 0.0

4 0.0 0.0

5 0.0 1.0

6 1.0 0.0

These newly created features are called boolean features because the only values they can contain are either 0 for non-inclusion, or 1 for inclusion. Pandas .get\_dummies() method allows you to completely replace a single, nominal feature with multiple boolean indicator features. This method is quite powerful and has many configurable options, including the ability to return a SparseDataFrame, and other prefixing options.

### Pure Textual Features

Bag of Words model, implemented with the CountVectorizer() method in SciKit-Learn.

>>> from sklearn.feature\_extraction.text import CountVectorizer

>>> corpus = [

... "Authman ran faster than Harry because he is an athlete.",

... "Authman and Harry ran faster and faster.",

... ]

>>> bow = CountVectorizer()

>>> X = bow.fit\_transform(corpus) # Sparse Matrix

>>> bow.get\_feature\_names()

['an', 'and', 'athlete', 'authman', 'because', 'faster', 'harry', 'he', 'is', 'ran', 'than']

>>> X.toarray()

[[1 0 1 1 1 1 1 1 1 1 1]

[0 2 0 1 0 2 1 0 0 1 0]]

### Graphical Features

# Uses the Image module (PIL)

from scipy import misc

# Load the image up

img = misc.imread('image.png')

# Is the image too big? Resample it down by an order of magnitude

img = img[::2, ::2]

# Scale colors from (0-255) to (0-1), then reshape to 1D array per pixel, e.g. grayscale

# If you had color images and wanted to preserve all color channels, use .reshape(-1,3)

X = (img / 255.0).reshape(-1)

# To-Do: Machine Learning with X!

#

::, that is called extended slicing.

.reshape(-1) line. This tells Pandas to take your 2D image and flatten it into a 1D array.

Another method called .ravel() will do the same thing as .reshape(-1), that is unravel a multi-dimensional NDArray into a one dimensional one.

# Uses the Image module (PIL)

from scipy import misc

# Load the image up

dset = []

for fname in files:

img = misc.imread(fname)

dset.append( (img[::2, ::2] / 255.0).reshape(-1) )

dset = pd.DataFrame( dset )

### Audio Features

Audio can be encoded with similar methods as graphical features, with the caveat that your 'audio-image' is already a one-dimensional waveform data type instead of a two-dimensional array of pixels. Rather than looking for graphical attributes, you would look for auditory ones, such as the length of sounds, power and noise ratios, and histogram counts after applying filters.

import scipy.io.wavfile as wavfile

sample\_rate, audio\_data = wavfile.read('sound.wav')

print audio\_data

# To-Do: Machine Learning with audio\_data!

#

## Wrangling Data

Pandas represents missing data internally using Numpy's np.nan.

df.my\_feature.fillna( df.my\_feature.mean() )

df.fillna(0)

df.fillna(method='ffill') # fill the values forward

df.fillna(method='bfill') # fill the values in reverse

df.fillna(limit=5)

If your nans occur at the start or end of your list, interpolation will not be able to help you:

df.interpolate(method='polynomial', order=2)

### Dropping Data

df = df.dropna(axis=0) # remove any row with nans

df = df.dropna(axis=1) # remove any column with nans

# Drop any row with NaNs that has at least 4 non-NaNs within it:

df = df.dropna(axis=0, thresh=4)

# Axis=1 for columns

df = df.drop(labels=['Features', 'To', 'Delete'], axis=1)

duplicate records;

df = df.drop\_duplicates(subset=['Feature\_1', 'Feature\_2'])

Removing duplicate samples will cause gaps to occur in your index count. You can interpolate to fill those holes where appropriate, or alternatively you can reindex your dataframe:

df = df.reset\_index(drop=True)

The drop=True parameter tells Pandas not to keep a backup copy of the original index.

df = df.dropna(axis=0, thresh=2).drop(labels=['ColA', axis=1]).drop\_duplicates(subset=['ColB', 'ColC']).reset\_index()

Pass inplace=True as a parameter to any of the above methods to get that working.

### More Wrangling

>>> df.dtypes

Date object

Name object

Gender object

Height object

Weight object

Age object

Job object

>>> df.Date = pd.to\_datetime(df.Date, errors='coerce')

>>> df.Height = pd.to\_numeric(df.Height, errors='coerce')

>>> df.Weight = pd.to\_numeric(df.Weight, errors='coerce')

>>> df.Age = pd.to\_numeric(df.Age, errors='coerce')

>>> df.dtypes

Date datetime64

Name object

Gender object

Height float64

Weight float64

Age int64

Job object

The errors='coerce' parameter instructs Pandas to enter a NaN at any field where the conversion fails.

see all the unique values present in a particular series.

f you'd like to know how many times each of those unique values are present, you can call .value\_counts().

>>> df.Age.unique()

array([7, 33, 27, 40, 22], dtype=int64)

>>> df.Age.value\_counts()

7 1

22 5

27 1

33 2

40 2

dtype: int64

# EXPLORING DATA

## Visualizations

to understand your data is to visualize it in a pictorial format.

Visualizing your data allows you to interact with it, analyze it in a straightforward way, and identify new patterns, making your would-be complex data more accessible and understandable.

### MatPlotLib

MatPlotLib is a Python data visualization tool that supports 2D and 3D rendering, animation, UI design, event handling, and more.

you will be interacting with MatPlotLib's Pyplot functionality through a Pandas series or dataframe's .plot namespace. Pyplot is a collection of command-style methods that essentially make MatPlotLib's charting methods feel like MATLAB.

## Basic Plots

### Histograms

Histograms help you understand the distribution of a feature in your dataset.

They accomplish this by simultaneously answering the questions where in your feature's domain your records are located at, and how many records exist there. Coincidentally, these two questions are also answered by the .unique() and .value\_counts() methods

Histograms are only really meaningful with categorical data.

If you have a continuous feature, it must first be binned or discretized by transforming the continuous feature into a categorical one by grouping similar values together. To accomplish this, the entire range values is divided into a series of intervals that are usually consecutive, equal in length, and non-overlapping. These intervals will become the categories. Then, a count of how many values fall into each interval serves as the categorical bin count.

call the .plot.hist() method on either a dataframe or series.

import pandas as pd

import matplotlib

matplotlib.style.use('ggplot') # Look Pretty

# If the above line throws an error, use plt.style.use('ggplot') instead

student\_dataset = pd.read\_csv("/Datasets/students.data", index\_col=0)

my\_series = student\_dataset.G3

my\_dataframe = student\_dataset[['G3', 'G2', 'G1']]

my\_series.plot.hist(alpha=0.5)

my\_dataframe.plot.hist(alpha=0.5)

If your interest lies in probabilities per bin rather than frequency counts, set the named parameter normed=True, which will normalize your results as percentages.

### 2D Scatter Plots

2D scatter plots are used to visually inspect if a correlation exist between the charted features.

Both axes of a 2D scatter plot represent a distinct, numeric feature.

They don't have to be continuous, but they must at least be ordinal since each record in your dataset is being plotted as a point with its location along the axes corresponding to its feature values. Without ordering, the position of the plots would have no meaning.

It is possible that either a negative or positive correlation exist between the charted features, or alternatively, none at all. The correlation type can be assessed through the overall diagonal trending of the plotted points.

Positive and negative correlations may further display a linear or non-linear relationship. If a straight line can be drawn through your scatter plot and most of points seem to stick close to it, then it can be said with a certain level of confidence that there is a linear relationship between the plotted features. Similarly, if a curve can be drawn through the points, there is likely a non-linear relationship. If neither a curve nor line adequately seems to fit the overall shape of the plotted points, chances are there is neither a correlation nor relationship between the features, or at least not enough information at present to determine.

import pandas as pd

import matplotlib

matplotlib.style.use('ggplot') # Look Pretty

# If the above line throws an error, use plt.style.use('ggplot') instead

student\_dataset = pd.read\_csv("/Datasets/students.data", index\_col=0)

student\_dataset.plot.scatter(x='G1', y='G3')

you have to call .scatter on a dataframe rather than a series

### 3D Scatter Plots

To follow up from the last section, there surely is a way to visualize the relationship between three variables simultaneously.

import matplotlib

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

import pandas as pd

matplotlib.style.use('ggplot') # Look Pretty

# If the above line throws an error, use plt.style.use('ggplot') instead

student\_dataset = pd.read\_csv("/Datasets/students.data", index\_col=0)

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

ax.set\_xlabel('Final Grade')

ax.set\_ylabel('First Grade')

ax.set\_zlabel('Daily Alcohol')

ax.scatter(student\_dataset.G1, student\_dataset.G3, student\_dataset['Dalc'], c='r', marker='.')

plt.show()

## Higher Dimensionality

### Higher Dimensionality Visualizations

#### Parallel Coordinates

Parallel coordinate plots are similar to scatter plots in that each axis maps to the ordered, numeric domain of a feature.

But instead of having axes aligned in an orthogonal manner, parallel coordinates get their name due to their axes being arranged vertically and in parallel. All that is just a fancy way of saying parallel coordinates are a bunch of parallel, labeled, numeric axes.

Each graphed observation is plotted as a polyline, a series of connected line segments. The joints of the polyline fall on each axis. Since each axis maps to the domain of a numeric feature, the resulting polyline fully describes the value of each of the observation's features.

Parallel coordinates are useful because polylines belonging to similar records tend to cluster together. To graph them with Pandas and MatPlotLib, you have to specify a feature to group by (it can be non-numeric). This results in each distinct value of that feature being assigned a unique color when charted.

from sklearn.datasets import load\_iris

from pandas.tools.plotting import parallel\_coordinates

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib

# Look pretty...

matplotlib.style.use('ggplot')

# If the above line throws an error, use plt.style.use('ggplot') instead

# Load up SKLearn's Iris Dataset into a Pandas Dataframe

data = load\_iris()

df = pd.DataFrame(data.data, columns=data.feature\_names)

df['target\_names'] = [data.target\_names[i] for i in data.target]

# Parallel Coordinates Start Here:

plt.figure()

parallel\_coordinates(df, 'target\_names')

plt.show()

It only supports a single scale for all your axes.

If you have some features that are on a small scale and others on a large scale, you'll have to deal with a compressed plot. For now, your only three options are to:

* Normalize your features before charting them
* Change the scale to a log scale
* Or create separate, multiple parallel coordinate charts. Each one only plotting features with similar domains scales plotted

### Andrew's Curves

helps you visualize higher dimensionality.

multivariate data by plotting each of your dataset's observations as a curve.

from sklearn.datasets import load\_iris

from pandas.tools.plotting import andrews\_curves

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib

# Look pretty...

matplotlib.style.use('ggplot')

# If the above line throws an error, use plt.style.use('ggplot') instead

# Load up SKLearn's Iris Dataset into a Pandas Dataframe

data = load\_iris()

df = pd.DataFrame(data.data, columns=data.feature\_names)

df['target\_names'] = [data.target\_names[i] for i in data.target]

# Andrews Curves Start Here:

plt.figure()

andrews\_curves(df, 'target\_names')

plt.show()

### imshow

.imshow() method generates an image based off of the normalized values stored in a matrix, or rectangular array of float64s.

The properties of the generated image will depend on the dimensions and contents of the array passed in:

* An [X, Y] shaped array will result in a grayscale image being generated
* A [X, Y, 3] shaped array results in a full-color image: 1 channel for red, 1 for green, and 1 for blue
* A [X, Y, 4] shaped array results in a full-color image as before with an extra channel for alpha

Besides being a straightforward way to display .PNG and other images, the .imshow() method has quite a few other use cases. When you use the .corr() method on your dataset, Pandas calculates a correlation matrix for you that measures how close to being linear the relationship between any two features in your dataset are. Correlation values may range from -1 to 1, where 1 would mean the two features are perfectly positively correlated and have identical slopes for all values. -1 would mean they are perfectly negatively correlated, and have a negative slope for one another, again being linear. Values closer to 0 mean there is little to no linear relationship between the two variables at all (e.g., pizza sales and plant growth), and so the further away from 0 the value is, the stronger the relationship between the features:

>>> df = pd.DataFrame(np.random.randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])

>>> df.corr()

a b c d e

a 1.000000 0.007568 0.014746 0.027275 -0.029043

b 0.007568 1.000000 -0.039130 -0.011612 0.082062

c 0.014746 -0.039130 1.000000 0.025330 -0.028471

d 0.027275 -0.011612 0.025330 1.000000 -0.002215

e -0.029043 0.082062 -0.028471 -0.002215 1.000000

import matplotlib.pyplot as plt

plt.imshow(df.corr(), cmap=plt.cm.Blues, interpolation='nearest')

plt.colorbar()

tick\_marks = [i for i in range(len(df.columns))]

plt.xticks(tick\_marks, df.columns, rotation='vertical')

plt.yticks(tick\_marks, df.columns)

plt.show()

.imshow() can help you any time you have a square matrix you want to visualize. Other matrices you might want to visualize include the covariance matrix, the confusion matrix, and in the future once you learn how to use certain machine learning algorithms that generate clusters which live in your feature-space, you'll also be able to use .imshow() to peek into the brain of your algorithms as they run, so long as your features represent a rectangular image!

# Transforming Data

## Transformations

A transformer is any algorithm you apply to your dataset that changes either the feature count or feature values, but does not alter the number of observations. You can use transformers to munge your data as a pre-processing step to clean it up before it's passed to other algorithms. Another popular transformer use is that of dimensionality reduction, where the number of features in your dataset is intelligently reduced to a subset of the original.

## Principal Component Analysis (PCA)

Unsupervised learning aims to discover some type of hidden structure within your data. Without a label or correct answer to test against, there is no metric for evaluating unsupervised learning algorithms. Principal Component Analysis (PCA), a transformation that attempts to convert your possibly correlated features into a set of linearly uncorrelated ones, is the first unsupervised learning algorithm you'll study.

### What is PCA?

PCA falls into the group of functions known as dimensionality reduction algorithms.

If you have reason to believe the question you want solved can be answered using a subset of your collected features, or if the features you've collected are actually many indirect observations of some inherent property you either cannot or do not know how to directly measure, then dimensionality reduction might work for you.

### How doeas PCA work

PCA is one of the most popular techniques for dimensionality reduction, and we recommend you always start with it when you have a complex dataset.

It models **a linear** subspace of your data by capturing its greatest variability. Stated differently, it accesses your dataset's covariance structure directly using matrix calculations and eigenvectors to compute the best unique features that describe your samples.

PCA ensures that each newly computed view (feature) is orthogonal or linearly independent to all previously computed ones**, minimizing these overlaps.** PCA also orders the features by importance, assuming that the more variance expressed in a feature, the more important said feature is. In our telephone pole example, the frontal view had more variance than the bird's-eye view, so it was preferred by PCA

With the newly computed features ordered by importance, dropping the least important features on the list intelligently reduces the number of dimensions needed to represent your dataset, with minimal loss of information. This has many practical uses, including: boiling off high dimensionality observations to just a few key dimensions for visualization purposes, being used as a noise reduction mechanism, and being used as a pre-processing step before sending your data through to other more processor-intensive algorithms. We'll look at more real life use cases in the next unit.

### When to Use PCA

PCA and other dimensionality reduction methods, have three main uses:

* Reducing the dimensionality and thus complexity of your dataset.
* Pre-process your data in preparation for other supervised learning tasks, such as regression and classification.
* To make visualizing your data easier, since we can only perceive three dimensions simultaneously.

PCA is a very fast algorithm and helps you vaporizes redundant features, so when you have a high dimensionality dataset, start by running PCA on it first and then visualizing it. This will better help you understand your data before continuing.

>>> from sklearn.decomposition import PCA

>>> pca = PCA(n\_components=2, svd\_solver='full')

>>> pca.fit(df)

PCA(copy=True, n\_components=2, whiten=False)

>>> T = pca.transform(df)

>>> df.shape

(430, 6) # 430 Student survey responses, 6 questions..

>>> T.shape

(430, 2) # 430 Student survey responses, 2 principal components..

To use PCA effectively, you should be aware of its weaknesses. The first is that it is sensitive to the scaling of your features. PCA maximizes variability based off of variance (the average squared differences of your samples from the mean), and then projects your original data on these directions of maximal variance. If your data has a feature with a large variance and others with small variances, PCA will prefer the larger variance feature. Being a linear transformation, it will rotate and reorient your feature space so it diverts as much of the variance of the larger-variance feature (and some of the other features' variances), into the first few principal components.

PCA is fast, but for very large datasets it might take a while to train. All of the matrix computations required for inversion, eigenvalue decomposition, etc. boggle down the algorithm. Since most real-world datasets are typically very large and will have a level of noise in them, razor-sharp, machine-precision matrix operations aren't always necessary.

RandomizedPCA applies some approximation techniques to speed up large-scale matrix computation. You can use this for your larger datasets. If you would like to perform a randomized singular variable decomposition as the means of estimating PCA, set the svd\_solver parameter to equal the string 'randomized'. Whether or not the randomized approximation actually runs faster than regular PCA depends on a number of factors, including but not limited to:

* If there was any interference from other running sub-processes
* The size of your dataset
* The data types used in your dataset, and
* If the logic introduced to run the randomized selection can complete faster than just running through the dataset regularly.

keep in mind is that PCA is a linear transformation only! In graphical terms, it can rotate and translate the feature space of your samples, but will not skew them. PCA will only, therefore, be able to capture the underlying linear shapes and variance within your data and cannot discern any complex, nonlinear intricacies. For such cases, you will have to make use different dimensionality reduction algorithms, such as Isomap.

## Isomap

### What is Isomap?

Isomap is also an unsupervised learning technique that reduces the dimensionality of your dataset.

No labels or classifications are needed to guide it except your raw data.

PCA is faster than Isomap and works well in most situations, but its limitation is that it assumes a linear relationship exist between your features.

### How Does Isomap Work

Isomap operates by first computing each record's nearest neighbors. This is done by comparing each sample to every other sample in the dataset. Only a sample's K-nearest samples qualify for being included in its nearest-neighborhood samples list. A neighborhood graph is then constructed by linking each sample to its K-nearest neighbors. The result is similar to map of roads that is traversed in order to move from point to point.

### When to Use Isomap

>>> from sklearn import manifold

>>> iso = manifold.Isomap(n\_neighbors=4, n\_components=2)

>>> iso.fit(df)

Isomap(eigen\_solver='auto', max\_iter=None, n\_components=2, n\_neighbors=4,

neighbors\_algorithm='auto', path\_method='auto', tol=0)

**n\_components** is the number of features you want your dataset projected onto

**n\_neighbors** defines the neighborhood size used to create the node neighborhood map

Running Isomap is a lot slower than PCA since a lot more is happening under the hood, particularly for large n\_neighbors values, but it provides a simple way to analyze and manipulate high dimensional samples in terms of its intrinsic nonlinear degrees of freedom. This is because isomap attempts to keep the global structure of your data as it reduces its dimensionality. You can use it in any case where nonlinear geometry degrades the effectiveness of PCA.

Isomap is also a bit more sensitive to noise than PCA. Noisy data can actually act as a conduit to short-circuit the nearest neighborhood map, cause isomap to prefer the 'noisy' but shorter path between samples that lie on the real geodesic surface of your data that would otherwise be well separated.

When using unsupervised dimensionality reduction techniques, be sure to use the feature scaling on all of your features because the nearest-neighbor search that Isomap bases your manifold on will do poorly if you don't, and PCA will prefer features with larger variances. As explained within the labs' source code, SciKit-Learn's StandardScaler is a good-fit for taking care of scaling your data before performing dimensionality reduction.

# Data Modeling

## Clustering

### K-Means

### How Does K-Means Work

Since clustering is an unsupervised algorithm, this similarity metric must be measured automatically and based solely on your data.

to iteratively separate your samples into a user-specified number of "K" cluster groups of roughly equal variance.

Cluster groups are defined by their geometric cluster center, single point referred to as its centroid. Separately, **centroid** and **cluster** are sometimes used interchangeably; but if used together, a **cluster** is a set of similar samples, and a **centroid** is just the mean feature-position of all samples assigned to the cluster.

The centroids are not records in your dataset, however they do 'exist' within your datasets feature-space. This is important because it allows for a meaningful distance measure to be calculated between the centroids and your samples. Every sample in your dataset is assigned to the centroid nearest to it, so if you have a sample that is 10 units away from CusterA's centroid, and 100 units away from ClusterB's, the sample is assigned to ClusterA.

In the case of continuous features, calculating the distance is straightforward.

#### The K-Means Algorithm

K-Means starts by placing a user-specified number of "K" cluster centers in your feature space.

Each cluster then takes ownership of the samples nearest to its centroid, and every sample can only be assigned as single cluster. 'Nearest' is a value that has to be evaluated and in SciKit-Learn, it is defined as the multivariate, n-dimensional Euclidean distance between the sample and the centroid. After this, the centroid location is updated to be **the mean value** of all samples assigned to it. This **mean** value is calculated by feature, so the centroid position ends up being a n-length vector within your feature space.

The assignment and update steps repeat until there are no more changes in either, at which point the algorithm has converged.

### When to Use K-Means

K-Means clustering is best suited when you have a good idea of the number of distinct clusters your unlabeled dataset should be segmented into. Generally, the output of K-Means is used in two ways. To separate your unlabeled data into K groups, which is the clear use case, or to find and use the resulting centroids.

#### Separate Your Data

#### Centroid Usage

Besides divvying up samples, clustering can also provide a layer of abstraction, by directing attention to the cluster and its attributes and not each samples.

You can use the centroid to 'compress' your data. By referring to the centroid rather than the data sample, the number of unique values is reduced, which optimizes the execution speed of other algorithms. Isomap, for instance, uses a nearest neighbors algorithm to calculate the distance from the record you want to transform to every sample in the training dataset. By using the record-to-cluster distance approximation in replacement of the individual record-to-sample distances, since there are far fewer clusters than records, you can achieve unprecedented orders of optimization.

>>> from sklearn.cluster import KMeans

>>> kmeans = KMeans(n\_clusters=5)

>>> kmeans.fit(df)

KMeans(copy\_x=True, init='k-means++', max\_iter=300, n\_clusters=5, n\_init=10,

n\_jobs=1, precompute\_distances='auto', random\_state=None, tol=0.0001,

verbose=0)

>>> labels = kmeans.predict(df)

>>> centroids = kmeans.cluster\_centers\_

xtremely fast to execute. So fast that it's often ran several times over as you saw earlier. Since each successive run of isn't dependent on the results of earlier runs, the execution process lends itself to parallelization, each centroid seeding trial being ran independently. If the clustering job at hand is still taking too long, SciKit-Learn's MiniBatchKMeans further optimizes the process for you.

K-Means is only really suitable when you have a good estimate of the number clusters that exist in your unlabeled data.

Even if you do have the right number of clusters selected, the result produced by K-Means can vary depending on the initial centroid placement. So if you need the same results produced each time, your centroid seeding technique also needs to be able to reliably produce the same placement given the same data. Due to the centroid seed placement having so much of an effect on your clustering outcome, you have to be careful since it is possible to have centroids with only a single sample assigned to them, or even no samples assigned to them in the worst case scenario.

Two other key characteristics of K-Means are that it assumes your samples are length normalized, and as such, is sensitive to feature scaling. It also assumes that the cluster sizes are roughly spherical and similar; this way, the nearest centroid is always the correct assignment.

## Supervised Learning

### Splitting Data

#### Supervised and Unsupervised Learning

* **.transform() :** without changing the number of samples, alters the value of each existing feature by changing its units
* **.predict() :** only with clustering, you could predict the label of the specified sample

The main difference between supervised and unsupervised learning is that with supervised learning, you actually guide the machine's hand at choosing the right answers. By showing the computer examples of what you want it to do, instead of just asking it to tell you something interesting about your data, the computer's responsibility shifts to deriving a set of rules that when applied to raw data, has a decent chance of choosing the answers you've trained it to.

* **.predict() :** After training your machine learning model, you can predict the labels of new and never seen samples
* **.predict\_proba() :** For some estimators, you can further see what the probability of the new sample belonging to each label is
* **.score():** The ability to score how well your model fit the training data

Imagine you had 100 spam emails and 100 regular emails and you trained an estimator to differentiate between the two. And it did so perfectly. What comes next? With such an exact estimator that 100% reliably labels emails, you'd probably want to integrate it into your email client. Do you see a problem here?

Every email that you quizzed your estimator, it's actually seen before during its training! You were telling it, this email is spam, this email isn't spam, etc. What you've essentially done is create a basic model that simply regurgitates the label you've already given for any specific email. That's not machine learning. In fact, that's no more remarkable than opening up a text file, saving something inside of it, and then being amazed when the exact same text is found within the text file upon reopening. This is called overfitting. Your goal with machine learning is to create a generalizable algorithm that can be applied to data it hasn't seen yet and still do the task you've trained it to do. In our case, properly classify the spam status of an email.

To make this possible, of course you'll still need to fit your estimator; but when it comes to testing it, that part will have to be done with data the estimator has never seen before. In other words, all of your training transformation and modeling needs to done using just your training data, without ever seeing your testing data. This will be your way of validating the true accuracy of your model. This is doable by splitting your training data into two portions. One part will actually be used for the training as usual, but the other part of the data is retained and used during testing only. How much data should you hold back? If you hold back too much data then your algorithm's performance is going to suffer, since you didn't train it well. If you don't hold back enough you won't have a statistically sufficient number of 'quiz' questions to gauge your machine learning model with.

### SciKit-Learn Implementation

>>> from sklearn.model\_selection import train\_test\_split

>>> data = [0,1,2,3,4, 5,6,7,8,9] # input dataframe samples

>>> labels = [0,0,0,0,0, 1,1,1,1,1] # the function we're training is " >4 "

>>> data\_train, data\_test, label\_train, label\_test = train\_test\_split(data, labels, test\_size=0.5, random\_state=7)

>>> data\_train

[9, 7, 3, 6, 4]

>>> label\_train

[1, 1, 0, 1, 0]

>>> data\_test

[8, 5, 0, 2, 1]

>>> label\_test

[1, 1, 0, 0, 0]

Unless you specify otherwise, it'll hold back 25% of your data in the validation or testing set, and 75% will stay in the original training set. Each time you run train\_test\_splot() it holds back a randomly shuffled amount of data, so one thing you'll start noticing is that successive trials of your algorithm may actually produce slightly different accuracy levels. This is normal, so do not be alarmed. If you absolutely need the results to come back identically, such as if you're doing a demo, then you can pass in an optional random\_state variable to make the centroid selection reproducible.

After you've trained your model against the training data (data\_train, label\_train), the next step is testing it. You'll use the .predict() method of your model, passing in the testing data (data\_test) to create an array of predictions. And then you'll gauge its accuracy against the true label\_test answers. SciKit-Learn also has a method to help you do that:

from sklearn.metrics import accuracy\_score

# Returns an array of predictions:

>>> predictions = my\_model.predict(data\_test)

>>> predictions

[0, 0, 0, 1, 0]

# The actual answers:

>>> label\_test

[1, 1, 0, 0, 0]

>>> accuracy\_score(label\_test, predictions)

0.4000000000000000

>>> accuracy\_score(label\_test, predictions, normalize=False)

2

## K-Nearest Neighbors

### K-Nearest Neighbors Classification

K-Neighbors is a supervised classification algorithm.

If clustering is the process of separating your samples into groups, then classification would be the process of assigning samples into those groups. Given a set of groups, take a set of samples and mark each sample as being a member of a group. Each group being the correct answer, label, or classification of the sample.

### How Does K-Neighbors Work

Isomap used K-Neighbors in an unsupervised way to build a neighborhood map

K-Nearest Neighbors works by first simply storing all of your training data samples.

Then in the future, when you attempt to check the classification of a new, never-before seen sample, it finds the nearest "K" number of samples to it from within your training data. You must have numeric features in order for 'nearest' to be meaningful. There are other methods you can use for categorical features. For example you can use **bag of words** to vectorize your data. Even so, you may want to experiment with other methods, such as as cosine similarity instead. But at the end of the day, SciKit-Learn's K-Nearest Neighbors only supports numeric features, so you'll have to do whatever has to be done to get your data into that format before proceeding. The distance will be measures as a standard Euclidean

With the nearest neighbors found, K-Neighbors looks at their classes and takes a mode vote to assign a label to the new data point. Further extensions of K-Neighbors can take into account the distance to the samples to weigh their voting power. Each new prediction or classification made, the algorithm has to again find the nearest neighbors to that sample in order to call a vote for it. This process is where a majority of the time is spent, so instead of using brute force to search the training data as if it were stored in a list, **tree structures** are used instead to optimize the search times. **Due to this,the number of classes in dataset doesn't have a bearing on its execution speed. Only the number of records in your training data set.**

#### Decision Boundaries

n-dimensional decision surface.

These boundaries are somewhat similar to the event horizon on a black hole: whenever performing classification, there exist a threshold or region where if superseded, will result in your sample being assigned that class.

For K-Neighbors, generally the higher your "K" value, the smoother and less jittery your decision surface becomes. Higher K values also result in your model providing probabilistic information about the ratio of samples per each class. There is a tradeoff though, as higher K values mean the algorithm is less sensitive to local fluctuations since farther samples are taken into account. This causes it to only model the overall classification function without much attention to detail, and increases the computational complexity of the classification.

### SciKit-Learn and K-Neighbors

It's hard enough coming up with the operators without the added burden of having to figure out which column is the solution the algorithm is supposed to be guided towards. This is why classification algorithms need you to separate out your answers or labels from the rest of your feature columns. When you download labeled datasets from the internet or even make your own, you'll see a classification column that tells you the class of each sample. This column will be right in your feature space. It's your responsibility to splice it out and drop the column once you've finished loading your dataset, but before any transformations, modeling, or test / train splitting:

# Process:

# Load a dataset into a dataframe

X = pd.read\_csv('data.set', index\_col=0)

# Do basic wrangling, but no transformations

# ...

# Immediately copy out the classification / label / class / answer column

y = X['classification'].copy()

X.drop(labels=['classification'], inplace=True, axis=1)

# Feature scaling as necessary

# ...

# Machine Learning

# ...

# Evaluation

# ...

SciKit-Learn's KNeighbors classifier interface is similar to the other machine learning algorithms you've seen so far, but there are a few notes to be aware of. As mentioned, when you fit your model, you now need to supply a second classification array in addition to your (now) unlabeled samples. The labeling vector should be an array of shape [n\_samples], and it should contain the classification or label for each training sample. Alternatively, if you're predicting multiple outputs, your array shape should be [n\_samples, n\_outputs] and again, each array value would contain the respective classification.

The KNeighborsClassifier class constructor takes in a few arguments, most optional:

* **n\_neighbors** The number of neighbors to consider. Keep it odd when doing binary classification, particularly when you use uniform weighting.
* **weights** How to count the votes from the neighbors; does everyone get an equal vote, a weighted vote, or something else?
* **algorithm** You can select an optimization method for searching through your training data set to find the nearest neighbors.

And here is the algorithm in action:

>>> # From now on, you only train on a "portion" of your dset:

>>> X\_train = pd.DataFrame([ [0], [1], [2], [3] ])

>>> y\_train = [0, 0, 1, 1]

>>> from sklearn.neighbors import KNeighborsClassifier

>>> model = KNeighborsClassifier(n\_neighbors=3)

>>> model.fit(X\_train, y\_train)

KNeighborsClassifier(...)

>>> # You can pass in a dframe or an ndarray

>>> model.predict([[1.1]])

>>> model.predict\_proba([[0.9]])

[[ 0.66666667 0.33333333]]

### KNN Gotchas

benefit that if you were to come into contact with more labeled data in the future, you could introduce it into your model without having to do heavy computations to rebuild it. Because of this, K-Neighbors is sometimes referred to as a memory based, 'lazy' machine learning algorithm, as it delays doing calculations until you start classifying.

Some of the caution-points to keep in mind while using **K-Neighbors is that your data needs to be measurable**. If there is no metric for discerning distance between your features, K-Neighbors cannot help you. As with all algorithms dependent on distance measures, it is also sensitive to feature scaling. **K-Neighbors is also sensitive to perturbations and the local structure of your dataset, particularly at lower "K" values.**

**On the other hand, with large "K" values, you have to be more cautious of the overall class distribution of your samples.** If 30% of your dataset is labeled A and 70% of labeled B, with high enough "K" values, you might experience K-Neighbors unjustly giving preference to B labeling, even in those localities of your dataset that should be properly classified as A.

## Regression

### Linear Regression

The main difference between classification and regression algorithms is that regression aims to compute a continuous output, but the goal of classification is to predict a discrete, categorical output.

Using classification, samples get labeled depending on a decision boundary test that separates your data into a range of space.

With regression, a continuous value output is calculated from a best fit curve function that runs through your data. In the special case of linear regression, the curve is restricted such that it is linear.

### How Does Linear Regression Work

y = mx + b.

y = w0 + w1x.

#### So How Do We Find the Weights?

SciKit-Learn uses a technique called ordinary least squares to compute the weights coefficients and intercept needed to solve for the best fitting line that goes through your samples. In the figure above, each of the black dots represents one of your features and of course the green line is your least squares, best fitting line. The red lines represent distances between the true, observed values of your sample compared to the least squares line we're hoping to calculate. Stated differently, these distances are the error between the approximate solution and the actual value. Ordinary least squares works by minimizing the squared sum of all these red line errors to compute the best fitting line. If you're wondering why the squared sum is used instead of the absolute value of the sums, or even just the regular sums, a note about that is included in the Dive Deeper section.

Once you have the equation, you can use it to calculate an expected value for feature y, given that you have feature x. If the x values you plug into the equation happen to lie within the x-domain boundary of those samples you trained your regression with, then this is called **interpolation** or even approximation, because you do have the actual observed values of y for the data in that range. When you use the function to calculate a y-value outside the bounds of your training data's x-domain boundary, that is called **extrapolation**.

#### Multivariate Linear Regression

y = w0 + w1x

Turns into,

y = w0 + w1x1 + w1x2 + ... + wnxn

Where n is the number of features examined. PCA works similar, in a way, to linear regression, but instead of taking one variable as dependent on the rest, PCA makes no assumptions and considers all variables. As a result, it attempts to find minimize the distance between the points and the line itself, which can span over multiple features / dimensions, instead of the way linear regression does it. Ordinary least squares, and by extension, linear regression in its multivariate form, attempts to minimize the sum of squared distances of all independent variables with just the dependent variable. A very cool graphical explanation on the differences between linear regression and PCA has been included in the Dive Deeper section.

### When to Use Linear Regression

Linear regression is widely used in all disciplines for forecasting upcoming feature values by extrapolating the regression line, and for estimating current feature values by interpolating the regression curve over existing data.

It is an extremely well-understood and interpretable technique that run very fast, and produces reasonable results as long as you do not extrapolate too far away from your training data.

One of the main advantages of linear regression over other machine learning algorithms is that even though it's a **supervised learning technique**, it doesn't force you to fine tune a bunch of parameters to get it working. You can literally just dump your data into it and let it produce its results.

You can use linear regression if your features display a significant correlation. The stronger the feature correlation, it being closer to +1 or -1, the better and more accurate the linear regression model for your data will be. The questions linear regression helps you answer are which independent feature inputs relate to the dependent feature output variable, and the degree of that relationship.

In business, linear regression is often used to forecast sales. By finding a correlation between time and the number of sales, a company can predict their near-terms future revenue, which will then help them budget accordingly. Linear regression can also be used to assess risk. Before issuing a loan, most banks will consider many features or aspects about their customers, and run a regression to see if it is worthwhile for them to borrow the money, of if their return on investment is insignificant or even positive for that matter.

In the sciences, geologists train linear regression against historic records to calculate the rate of glacier snow melting, and can use it extrapolate how long it'll take for it to all disappear. Oil engineers do the same while calculating how much is potentially left. When measuring experimental results, chemists use linear regression to empirically calculate and validate concentrations and expected reactions. And of course there are many more uses.

### SciKit-Learn and Linear Regression

* **intercept\_** the scalar constant offset value
* **coef\_** an array of weights, one per input feature, which will act as a scaling factor

>>> from sklearn import linear\_model

>>> model = linear\_model.LinearRegression()

>>> model.fit(X\_train, y\_train)

LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=1, normalize=False)

From here on out, you should separate your data into a training and testing set before fitting your predictor models,

**X\_train** is either a single dimensional array, shaped like **[n\_samples]**, where each sample corresponds to the single feature you're fitting.

Your **y\_train** target can also be a single dimensional **[n\_samples]** list of expected values, or it can be an **[n\_samples, n\_targets]** array if you are using linear regression to compute more than one feature simultaneously. When doing that, your .intercept\_ will also be an array, one value per target.

**.score()** method that returns the R2 coefficient,

>>> # R2 Score

>>> model.score(X\_test, y\_test)

153.244939109

>>> # Sum of Squared Distances

>>> np.sum(model.predict(X\_test) - y\_test) \*\* 2)

5465.15

One thing to keep in mind is that your **R2 coefficient increases the more features you consider when modeling your linear regression, even if those features don't have a good correlation with your dependent feature's values.** Due to this, be selective about which features you choose to use and select just the subset of the most promising ones, otherwise you might be subject to errant **overfitting.**

linear regression works with continuous data, as well as categorical data once numerically encoded.

If you do end up using categorical data and have multiple dummy boolean columns you want to calculate, for example IceCream\_Vanilla, IceCream\_Chocolate, and IceCream\_CookiesNCream, then you should calculate all three of these target regression lines simultaneously. Just increase the number of targets, or your training labels dimension, and then each regression calculation will have its own offset stored in your **.intercepts**\_ array attribute, and the **.coeff\_** attribute will become an array of arrays, one per target. In SciKit-Learn, this is called Multi-Output Linear Regression.

### Linear Regression Gotchas

he last thing to watch out for with linear regression is that the further you extrapolate from the range of your training data, the less reliable the results of the regression become. Keep these thoughts in mind while using linear regression!

# Data Modeling II

## SVC

### SVM and SVC

Support vector machines are a set of supervised learning algorithms that you can use for classification, regression and outlier detection purposes.

In a nutshell, SVC solves the classification problem by finding the equation of the hyperplane (linear surface) which results in the most separation between two classes of samples. This allows you to confidently label your samples in a very fast and efficient way.

### Kernel Trick

### When to Use SVC

you could use it on any classification problem

Other algorithms like K-Neighbors are instantaneous with training, but require you traverse a complicated tree structure for each sample you want to classify.

One of the advantages of SVC is that once you've done the hard work of finding the hyperplane and its supporting vectors, the real job of classifying your samples is as simple as answering what side of the line is the point on? This makes SVC a classifier of choice for problems where classification speed is more critical than training speed.

SVC is extremely effective, even in high dimensional spaces.

There may be cases where number your dataset has more features than the number of samples. Not all machine learning algorithms will be able to work with that, however such datasets aren't an issue for SVC. In fact, at least conceptually, if you use the kernel trick then at some point your data will almost assuredly be at a higher dimensionality than the number of features, depending on which kernel you use. And with the ability to use different kernel functions or even define your own, SVC will prove to be a very versatile classifier for you to have down in your machine learning arsenal.

Lastly, SVC is non-probabilistic. That means the resulting classification is calculated based off of the geometry of your dataset, as opposed to probabilities of occurrences. Once you get to decision trees, you'll see an example of a classifier that works using probabilities and not the geometric nature of your dataset.

### SciKit-Learn and SCV

Unlike linear regression, SVC is a very configurable algorithm.

* **kernel** Defines the type of kernel used with your classifier. The default is the radial basis function rbf, the most popular kernel used with support vector machines generally. SciKit-Learn also supports linear, poly, sigmoid, and precomputed kernels. You can also specify a user defined function to pre-compute the kernel matrix from your sample's feature space, which should be shaped [n\_samples, n\_samples].
* **C** This is the penalty parameter for the error term. Do you want your SVC to never miss a single classification? Or is having a more generalized solution important to you? The lower your C value, the smoother and more generalized your decision boundary is going to be. But if you have a large C value, the classifier will attempt to do whatever is in its power to squiggle and wiggle between each sample to correctly classify it.
* **gamma** This parameter's value is inversely proportional to the extent a single training sample's influence extends. Large gamma values result in each training sample having localized effects only. Smaller values result in each sample affecting a larger area. In essence, the gamma values dictate how pronounced your decision boundary is by varying the influence of your support vector samples.
* **random\_state** SVC and support vector machines are theoretically a deterministic algorithm, meaning if you re-run it against the same input, it should produce identical output each time. However SKLearn's SVC via libsvc implementation randomly shuffles your data during its probability estimation step. So to truly get deterministic execution, set a state seed.

>>> from sklearn.svm import SVC

>>> model = SVC(kernel='linear')

>>> model.fit(X, y)

SVC(C=1.0, cache\_size=200, class\_weight=None, coef0=0.0,

decision\_function\_shape=None, degree=3, gamma='auto', kernel='linear',

max\_iter=-1, probability=False, random\_state=None, shrinking=True,

tol=0.001, verbose=False)

* **support\_** Contains an array of the indices belonging to the selected support vectors
* **support\_vectors\_** The actual samples chosen as the support vectors
* **intercept\_** The constants of the decision function
* **dual\_coef\_** Each support vector's contribution to the decision function, on a per classification basis. This has similarities to the weights of linear regression

### SVC Gotchas

One of SVC's strong points is that since the core of the algorithm is based on a small subset of your dataset's samples, namely the support vectors, even if you have fewer samples than dimensions, so long as the samples you do have are close to the decision boundary, there's a good chance your support vector classifier will do just swell.

Intuitively, those samples that are further away from the decision boundary are more clearly identifiable as belonging to their respective classes. The samples closer to the decision boundary are more vague, and could easily be mistaken as belonging to the wrong class. If you wanted to train a child how to recognize cats from dogs, good training samples would include the most "catly" cat you could find, and the most "dogly" dog. By showing them samples from the two classes that are far away from the decision boundary, they are less likely to look for characteristics and features that might accidentally be misconstrued. Support vector machines behave counter intuitively; they don't care about the samples that are clearly cats, or that are clearly dogs. Rather, they focus on those samples, or support vectors, closest to the decision boundary, so they can compute precisely the smallest change of features that differentiate between the two, perchance it's able to properly classify all of them.

Although SVC can run on a subset of your features, if you get rid of too many, that is, if the dimensionality of your features is much greater than the number of samples, the quality of your decision boundary may still suffer. Again, depending on how far away those samples are from it.

Support vector machines, unfortunately, do not directly give probability estimates for what class a sample belongs to. If a sample is further from the decision boundary than the margin, then the algorithm is intuitively 100% sure of its classification. Any testing found within the margin has some probability of belonging to either class. In SciKit-Learn, to calculate the probability of belonging to either class, you actually have to use an expensive five-fold cross-validation, which we won't discuss at all until the next module.

Since SVC is one of SciKit-Learn's highly configurable predictors, it's easy to start overfitting your models if you're not careful. Furthermore, unlike KNeighbors that does all its processing at the point of predicting, SVC does the majority of its heavy lifting at the point of training, so large training sets can result in sluggish training. If the ability to do realtime training and updating of your model is of great concern to you, you might have to consider another algorithm, depending on the size of your dataset. That said, there are a few mechanisms to speed it up.

## Decision Trees

Decision trees are a supervised, probabilistic, machine learning classifier that are often used as decision support tools.

* The decision tree is structured as a flowchart diagram. It has a single, starting root node, and can contain one or more leaf nodes, and internal nodes.
* Each node of the decision tree represents the test of single feature's value. For example, one node might test the feature named 'age' to see if the user is greater than 18 or not.
* Each branch of the decision tree connects two nodes, an originating node, and a destination node. A branch represents an outcome of the test of the branch's originating node. In the case of the age > 18 test, there are two possible branches: True and False. Each of these two branches would connect the age > 18 node, to two destination nodes.
* Each leaf node of the decision tree represents a classification. The end goal of the tree is to label your input samples, so each leaf fulfills that goal.

### How Do Decision Trees Work

PCA and Isomap work by intelligently simplify your data by either variability or intrinsic geometry. K-Means clusters your data based on feature similarity. K-Neighbors classifies your data by feature similarity. Linear regression models a continuous, linear correlation in your data. And now **decision trees classify your data probabilistically based off of entropy, which can be thought of as 'purity' or information gain.**

### When to Use Decision Trees

Decision trees are one of the oldest and most used machine learning algorithms, perhaps even pre-dating machine learning.

Decision trees, similar to SVC, make use of a clever trick to allow you to do non-linear decision-making by use of a linear decision surface. With SVC it was the table-flipping, kernel trick. With decision trees, you can divide up your feature set into sections and boxes, which otherwise would not have been possible using a single, linear classifier. Due to this, a few other nifty qualities are born to decision trees, such as them being indifferent to feature scaling, unlike KNeighbors or PCA.

Decision trees are a good tool to use when you want backing evidence to support a decision, or even while trying to convince someone of a view-point. By mapping the various probabilities of outcomes, and their consequences, using a decision tree helps you paint the entire picture of the classification results of your features.

### SciKit-Learn and Decision Trees

# You know the drill...

>>> from sklearn import tree

>>> model = tree.DecisionTreeClassifier(max\_depth=9, criterion="entropy")

>>> model.fit(X,y)

DecisionTreeClassifier(class\_weight=None, criterion='entropy', max\_depth=9,

max\_features=None, max\_leaf\_nodes=None, min\_samples\_leaf=1,

min\_samples\_split=2, min\_weight\_fraction\_leaf=0.0,

presort=False, random\_state=None, splitter='best')

# .DOT files can be rendered to .PNGs, if you've already `brew install graphviz`.

>>> tree.export\_graphviz(model.tree\_, out\_file='tree.dot', feature\_names=X.columns)

>>> from subprocess import call

>>> call(['dot', '-T', 'png', 'tree.dot', '-o', 'tree.png'])

* **criterion** By default, SciKit-Learn uses Gini, which is an impurity rating. Alternatively, you could also make use of information gain, or entropy instead.
* **splitter** Lets you control of the algorithm chooses the best split or not. We'll discuss why that's importance once you move to random forest classifier.
* **max\_features** One of the possible splitter options for splitter above is called 'best'. SciKit-Learn runs a bunch of tests on your features to figure out which mechanism should be used when searching for the best split. This parameter limits the number of features to consider while doing this.

After you've gone ahead and trained your tree, you can of course get back all the end-node, leaf classifications that the tree has reached, as well as the entire tree object if you like. For those leaf nodes that aren't 100% pure due to having samples belonging to multiple classes within them, then the end-result class the leaf takes is a weighted mode vote, based on the number of each class label inside of it.

You can also get back a feature\_importances vector that stores, in order of importance, the features that used to make the labeling decisions of your tree.

### Decision Tree Gotchas

Both training and testing speed are fast.

They work with either categorical or continuous features, with or without encoding, and are invariant to feature of scaling.

if configured properly, they can pretty decently handle irrelevant and noisy features. In a nutshell, decision trees help you decide the worst, best and expected classification labels given various scenarios.

While using them, keep in mind that your data still must be multivariate linearly separable for classification to work. The decision surfaces of the tree are still flat, even though they can intersect at angles. Also considering how malleable and how many parameters they have, it's easy to get carried away and overfit your tree unless, particularly if you don't fully test against an independent testing set. A good sign of an overfit tree is its very complex structure, and it having branches that reach out erroneously just to correctly label single records, due to their sensitivity to small, local fluctuations in your training data.

## Random Forest

Decision trees;

* Unlike SVMs, the accuracy of a DTree doesn't decrease when you include irrelevant features
* Unlike KNeighbors, both training and predicting with a DTree are relatively fast operations
* Unlike PCA / IsoMap, DTrees are invariant to monotonic feature scaling and transformations
* Moreover, a trained DTree model is readily human inspectable

As Trevor Hastie indicated, under many circumstances, they're really a miracle machine learning algorithm, but with an Achilles heel—if not carefully pruned, decision trees will deep-learn irregular patterns and data outliers. They do this so well that they rapidly overfit the training set, resulting in excellent training data recall... but poor predictive abilities. This is exactly the problem random forests aim to solve.

This is exactly how random forest work. A single decision tree, tasked to learn a dataset might not be able to perform well due to the outliers, and the breadth and depth complexity of the data. So instead of relying on a single tree, random forests rely on a forest of cleverly grown decision trees. Each tree within the forest is allowed to become highly specialized in a specific area, but still retains some general knowledge about most areas. When a random forest classifier, it is actually each tree in the forest working together to cast votes on what label they think a specific sample should be assigned.

### How Does Random Forest Work

#### Training

Random forests make use of two techniques when training, one occurs at the forest level, and the other at the individual tree level.

First, like any supervised classifier, you'll pass in a training set of samples along with "truth" labels when you create an instance of the class. Instead of sharing the entire dataset with each decision tree, the forest performs an operation which is essential a train / test split of the training data. Each decision tree in the forest randomly samples from the overall training data set. Through doing so, each tree exist in an independent subspace and the variation between trees is controlled. This technique is known as **tree bagging**, or **bootstrap aggregating.**

**Tree bagging** increases the accuracy of decision trees because while an individual decision tree might become hypersensitive to outliers and localized features, once all the results are averaged, the fringe results get blurred out. Therefore using random forests over a single tree decrease the variance of your classification results, without increasing the bias the way KNeighbor does when K is set too high. **This technique only works if the individual trees are not correlated.** If they were all trained on the same training set, they would only reinforce each other's decision. The bootstrapping or randomization of samples each tree is trained upon take care of that.

Random forests also use one more trick. In addition to the tree bagging of training samples at the forest level, each individual decision tree further **'feature bags'** at each node-branch split. This is helpful because some datasets contain a feature that is very correlated to the target (the 'y'-label). By selecting a random sampling of features every split, if such a feature were to exist, it wouldn't show up on as many branches of the tree and there would be more diversity of the features examined.

#### Predicting

After fitting the forest, a prediction can be made for unseen samples by using the majority vote label assigned from each tree.

#### Error Testing

Since each tree within the forest is only trained using a subset of the overall training set, the forest ensemble has the ability to error test itself. It does this by scoring each tree's predictions against that tree's out-of-bag samples. A tree's out of bag samples are those forest training samples that were withheld from a specific tree during training. There's nothing unique about splitting your data between training and testing sets except that you have an independent set of unseen samples to validate the accuracy of your training. Part of the random forest algorithm is the creation of independent sets for the training of each tree, so an overall out-of-bag error metric can be calculated for the forest ensemble. This error value is defined as the mean prediction error for each training samples using only those trees that didn't have the sample in their bootstrap.

One of the advantages of using the out of bag error is it eliminates the need for you to split your data into a training / testing before feeding it into the forest model, since that's part of the forest algorithm. However using the out-of-bag error metric often underestimates the actual performance improvement, and the optimal number of training iterations. Due to this and a few other reason's we'll discuss in the next module, SciKit-Learn recommends you maintain separate training and testing sets.

### SciKit-Learn and Random Forest

* **n\_estimators** Controls the density of the forest ensemble.
* **bootstrap** Also known as bagging. Every trained tree is grown using an independently drawn subset of your input data. As such, training samples not used for training an individual tree are considered out-of-bag for that one tree.
* **obb\_score** Controls whether to use out-of-bag samples to estimate a generalization error. By default, this is turned off, with the assumption that you'll be using random forest just like any other SciKit-Learn estimator, and handling the splitting of your training/testing data manually, along with its scoring.

>>> from sklearn.ensemble import RandomForestClassifier

>>> model = RandomForestClassifier(n\_estimators=10, oob\_score=True)

>>> model.fit(X, y)

RandomForestClassifier(bootstrap=True, class\_weight=None, criterion='gini',

max\_depth=None, max\_features='auto', max\_leaf\_nodes=None,

min\_samples\_leaf=1, min\_samples\_split=2,

min\_weight\_fraction\_leaf=0.0, n\_estimators=10, n\_jobs=1,

oob\_score=True, random\_state=None, verbose=0, warm\_start=False)

>>> print model.oob\_score\_

0.789925345

If, for some reason, you're interested in examining the structure of the individual decision trees that comprise your forest, you can do so using the .estimators\_ attribute of the random forest class. Additionally, if oob\_score=True was set as a parameter, and assuming you had enough trees and data samples for it to actually be possible, the .obb\_score\_ attribute will have calculated an accuracy metric for you using an averaged out-of-bag estimates for your training samples. The out-of-bag samples for a particular tree are those training samples that were withheld during the training of that particular tree.

### Random Forest Gotchas

Random forest almost always boosts your scoring accuracy above that of a single decision tree. Like other ensemble methods, they are effortlessly parallelizable, because you can have independent trees being trained on distinct cores and only need to get a mode vote of their classifications once you attempt to predict or score your data.

Two drawbacks of random forest are that since you're now planting an entire forest as opposed to a single tree, both training and prediction execution times suffer tremendously. Particularly, training is an order of magnitude more time consuming. You also lose the ability to inspect the resulting structure of your classifier as easily. No longer can you just print out a .dot-file flow chart since no single flowchart encompasses the forest, and your results aren't strictly based on following a single logic diagram. Along the same lines, the forest can no longer be linearized into IF...THEN blocks anymore.

Even with all that said, given the added accuracy performance-boost, if a high level of correctly classified samples is what you need, then moving from decision trees to random forest should be an easy choice to make.

# Evaluating Data

## Confusion

### Choosing The Right Estimator

* The dimensionality of your data
* The geometric nature of your data
* The types of features used to represent your data
* The number of training samples you have at your disposal
* The required training and prediction speeds needed for your purposes
* The predictive accuracy level desired
* How configurable you need your model to be
* And much more

### Confusion Matrix

A confusion matrix displays your model's predicted (testing set) outputs against the true observational values.

This helps you see how well your algorithm was able to generalize and identify specific target labels, along with which labels were often confused.

This can be helpful in increasing your accuracy, because you can then engineer additional features that help to better identify highly confusing targets, and then take another run through your data analysis pipeline.

>>> import matplotlib.pyplot as plt

>>> columns = ['Cat', 'Dog', 'Monkey']

>>> confusion = metrics.confusion\_matrix(y\_true, y\_pred)

>>> plt.imshow(confusion, cmap=plt.cm.Blues, interpolation='nearest')

>>> plt.xticks([0,1,2], columns, rotation='vertical')

>>> plt.yticks([0,1,2], columns)

>>> plt.colorbar()

>>> plt.show()

## Cross Validation

### Scoring Metrics

* **True Positive** You predicted it was a desired orange, and it was indeed an orange. Good job!
* **True Negative** You predicted it was an undesired grapefruit, and it was indeed a grapefruit.
* **False Positive** You predicted it was a delicious orange, but behold! It was a actually a sour grapefruit.
* **False Negative** You predicted it was a detested grapefruit, but it actually ended up being the best orange of the season

>>> import sklearn.metrics as metrics

>>> y\_true = [1, 1, 2, 2, 3, 3] # Actual, observed testing datset values

>>> y\_pred = [1, 1, 1, 3, 2, 3] # Predicted values from your model

#### accuracy\_score

>>> metrics.accuracy\_score(y\_true, y\_pred)

0.5

>>> metrics.accuracy\_score(y\_true, y\_pred, normalize=False)

3

#### Recall

To calculate the recall score, or the ratio of true\_positives / (true\_positives + false\_negatives):

>>> metrics.recall\_score(y\_true, y\_pred, average='weighted')

0.5

>>> metrics.recall\_score(y\_true, y\_pred, average=None)

array([ 1. , 0. , 0.5])

#### Precision

You can also calculated the precision score. It is defined very similarly: true\_positives / (true\_positives + false\_positives). The only difference is the very last term in the equation:

>>> metrics.precision\_score(y\_true, y\_pred, average='weighted')

0.38888888888888884

>>> metrics.precision\_score(y\_true, y\_pred, average=None)

array([ 0.66666667, 0. , 0.5 ])

#### F1

The F1 Score is a weighted average of the precision and recall. Defined as 2 \* (precision \* recall) / (precision + recall), the best possible result is 1 and the worst possible score is 0:

>>> metrics.f1\_score(y\_true, y\_pred, average='weighted')

0.43333333333333335

>>> metrics.f1\_score(y\_true, y\_pred, average=None)

array([ 0.8, 0. , 0.5])

#### Full Report

As a convenience, SciKit-Learn also has a built-in method for producing a full report of the above scores for you simultaneously, on a per label basis:

>>> target\_names = ['Fruit 1', 'Fruit 2', 'Fruit 3']

>>> metrics.classification\_report(y\_true, y\_pred, target\_names=target\_names)

precision recall f1-score support

Fruit 1 0.67 1.00 0.80 2

Fruit 2 0.00 0.00 0.00 2

Fruit 3 0.50 0.50 0.50 2

avg / total 0.39 0.50 0.43 6

### Cross Validation

>>> from \*\*sklearn.cross\_validation\*\* import train\_test\_split

>>> X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.5, random\_state=0)

>>> Test how well your model can recall its training data:

>>> model.fit(X\_train, y\_train).score(X\_train, y\_train)

0.943262278808

>>> Test how well your model can predict unseen data:

>>> model.fit(X\_test, y\_test).score(X\_test, y\_test)

0.894716422024

**train\_test\_split()** method selects a random subset of your data to withhold as the testing validation set.

But the most important issue introduced is that with some of the more configurable estimators, such as SVC, you will probably end up running your model many times while tinkering with the various parameters, such as C and gamma.

he way to overcome this is by splitting your data once more. Now you have a training set, a testing set, and a validation set that you don't optimize your models against and only use for scoring. Such a process is too tedious. Having to deal with three sets of data means you must remember to do the same transformations to all sets, otherwise you'll get incorrect shape errors when fitting and training. **cross\_val\_score().**

This method takes as input your model along with your training dataset and performs K-fold cross validations on it. In other words, your training data is first cut into a number of "K" sets. Then, "K" versions of your model are trained, each using an independent K-1 number of the "K" available sets. Each model is evaluated with the last set, it's out-of-bag set. If this sounds super familiar to you, it's because this is the same bootstrapping technique used in random forest.

# 10-Fold Cross Validation on your training data

>>> from sklearn import cross\_validation as cval

>>> cval.cross\_val\_score(model, X\_train, y\_train, cv=10)

array([ 0.93513514, 0.99453552, 0.97237569, 0.98888889, 0.96089385,

0.98882682, 0.99441341, 0.98876404, 0.97175141, 0.96590909])

>>> cval.cross\_val\_score(model, X\_train, y\_train, cv=10).mean()

0.97614938602520218

Cross validation allows you to use all the data you provide as both training and testing, so many resources online will recommend you don't even do the extra step of splitting your data into a training and testing set and just feed the lot directly into your cross validator. There are advantages and disadvantages of this. The main advantage is the overall simplicity of your process. The disadvantage is that it still is possible for some information to leak into your training dataset, as we discussed above with the SVC example. This information leak might even occur prior to you fitting your model, for example it might be at the point of transforming your data using isomap or principle component analysis.

For these reasons, the SciKit-Learn documentation recommends you still keep a completely separate testing set to conduct scoring, after cross validating your models. Make sure you read through the documentation before going through the knowledge checks, particularly for the cv parameter and the different types of cross validator iterators. For example, it's absolutely critical that you know the difference between KFold and StratifiedKFold before you start using cross\_val\_score(), to avoid introducing a serious bug into your machine learning!

In the wild, the best process to use depending on how many samples you have at your disposal and the machine learning algorithms you are using, is either of the following:

1. Split your data into training, validation, and testing sets.
2. Setup a pipeline, and fit it with your training set
3. Access the accuracy of its output using your validation set
4. Fine tune this accuracy by adjusting the hyper-paramters of your pipeline
5. when you're comfortable with its accuracy, finally evaluate your pipeline with the testing set

OR

1. Split your data into training and testing sets.
2. Setup a pipeline with CV and fit / score it with your training set
3. Fine tune this accuracy by adjusting the hyper-paramters of your pipeline
4. When you're comfortable with its accuracy, finally evaluate your pipeline with the testing set

## Power Tuning

### Power-Tuning

**GridSearchCV** takes care of your parameter tuning and also tacks on end-to-end cross validation. This results in more precisely tuned parameter than depending on simple model accuracy scores, and is why the algorithm is name Grid-Search-CV.

### Pipelining

make your code a lot cleaner, SciKit-Learn has created a pipelining class. It wraps around your entire data analysis pipeline from start to finish, and allows you to interact with the pipeline as if it were a single white-box, configurable estimator.

>>> from sklearn.pipeline import Pipeline

>>> svc = svm.SVC(kernel='linear')

>>> pca = RandomizedPCA()

>>> pipeline = Pipeline([

('pca', pca),

('svc', svc)

])

>>> pipeline.set\_params(pca\_\_n\_components=5, svc\_\_C=1, svc\_\_gamma=0.0001)

>>> pipeline.fit(X, y)