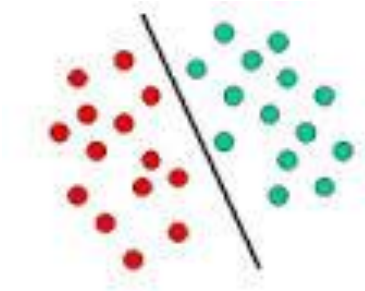
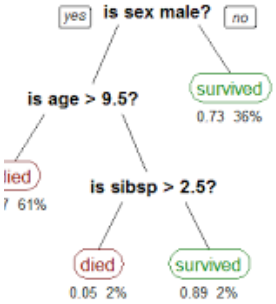
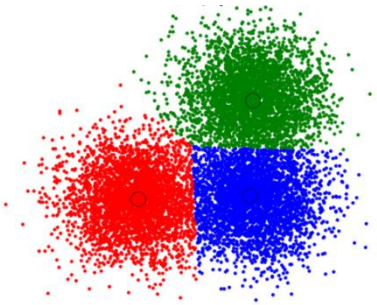
Machine Learning with Python



Classification
(LR, SVM…)



Trees



Clustering

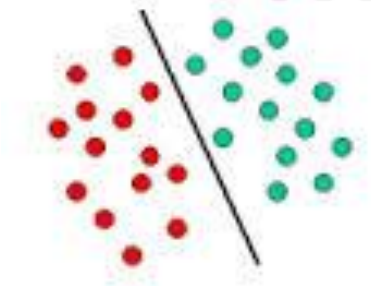
… …

Luís Garmendia

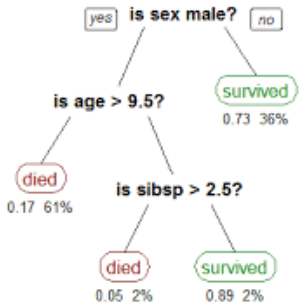
Index

- *Introduction to Data Sciences and Machine Learning*
- *Scikit-Learn: A Python Machine Learning Library*
    - *Supervised Linear regression*
    - *Supervised Classification*
    - *Unsupervised dimensionality reduction*
    - *Unsupervised Clustering*
    - *Classification of images: confusion matrix*
- *Model Validation: Bias-variance tradeoff*
- *Learning curves*
- *Feature Engineering: Transform data*
- *Imputation of missing data*

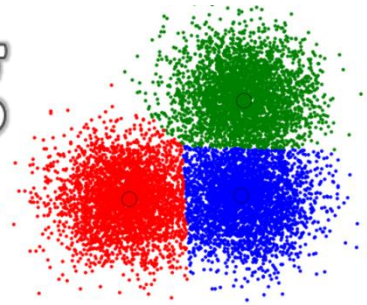
Data Sciences
Machine Learning



Classification
(LR, SVM…)



Trees



Clustering … …

Luis Garmendia

Data Sciences

[data science is]... the process of discovering what is true and useful in mountains of data

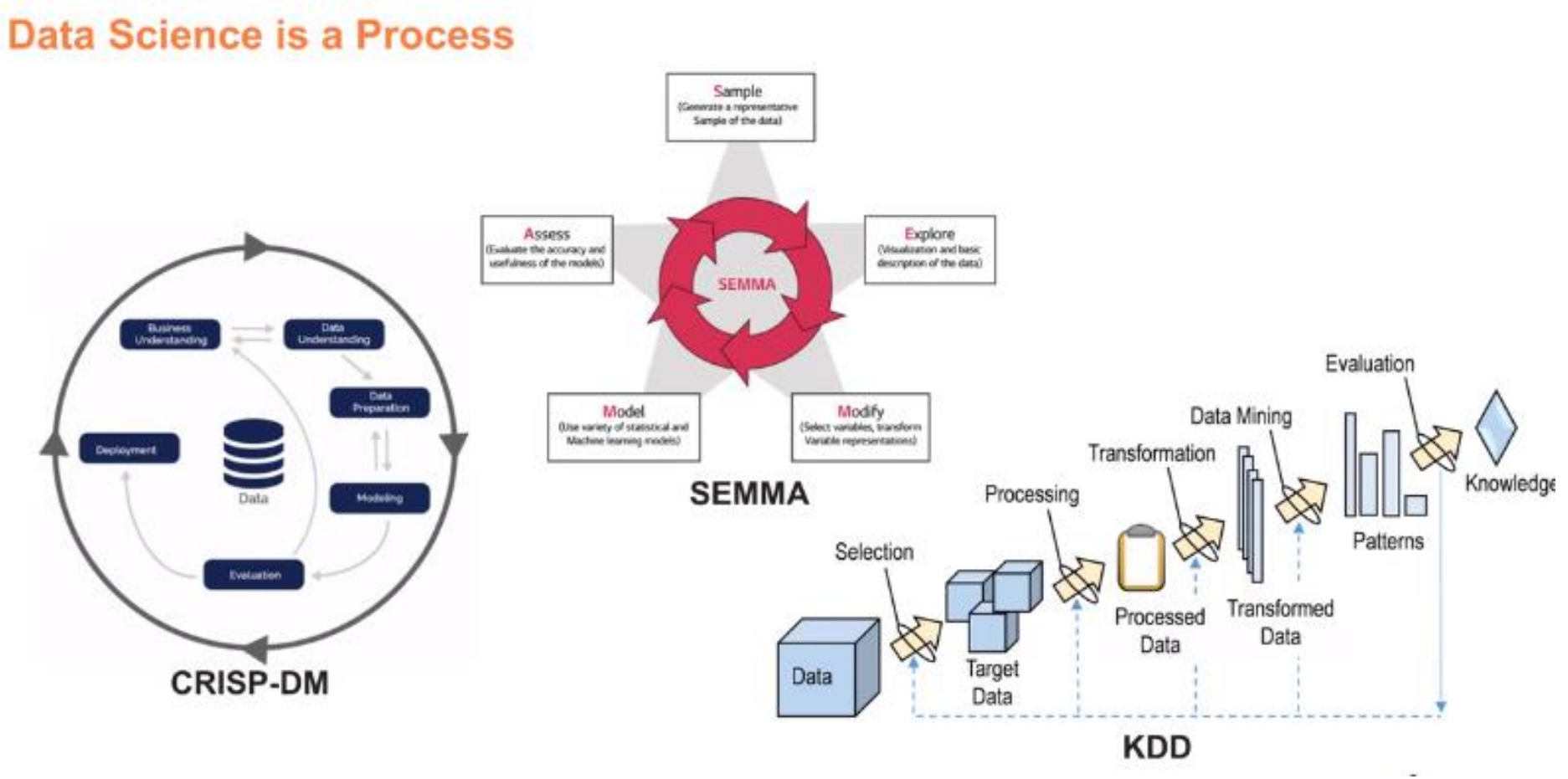
Gregory Piatetsky-Shapiro

Any effective data science process is iterative

- You may have many iterations of trial and error
- You may need to move backward from any given step to a prior step

End goal: Operationalize your analytic models/findings that meet your business success criteria and goals/objectives.

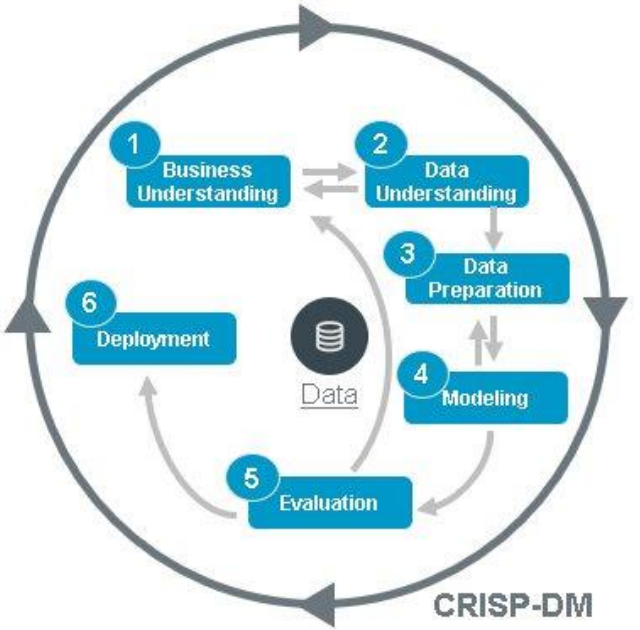
Data Sciences is a process



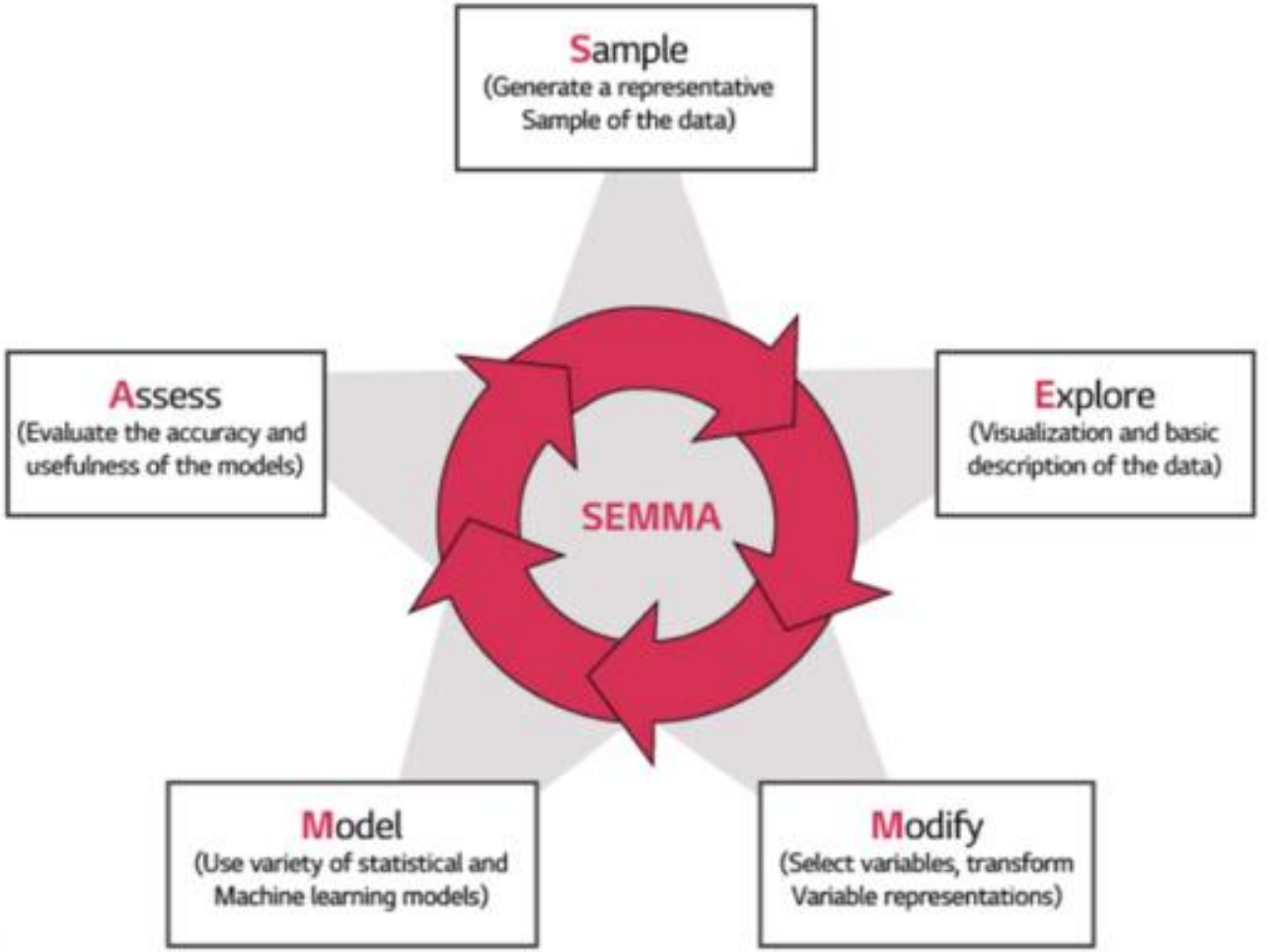
CRISP-DM

The published **CRISP-DM** is a 70 page document with important information. For getting started with Data Science projects

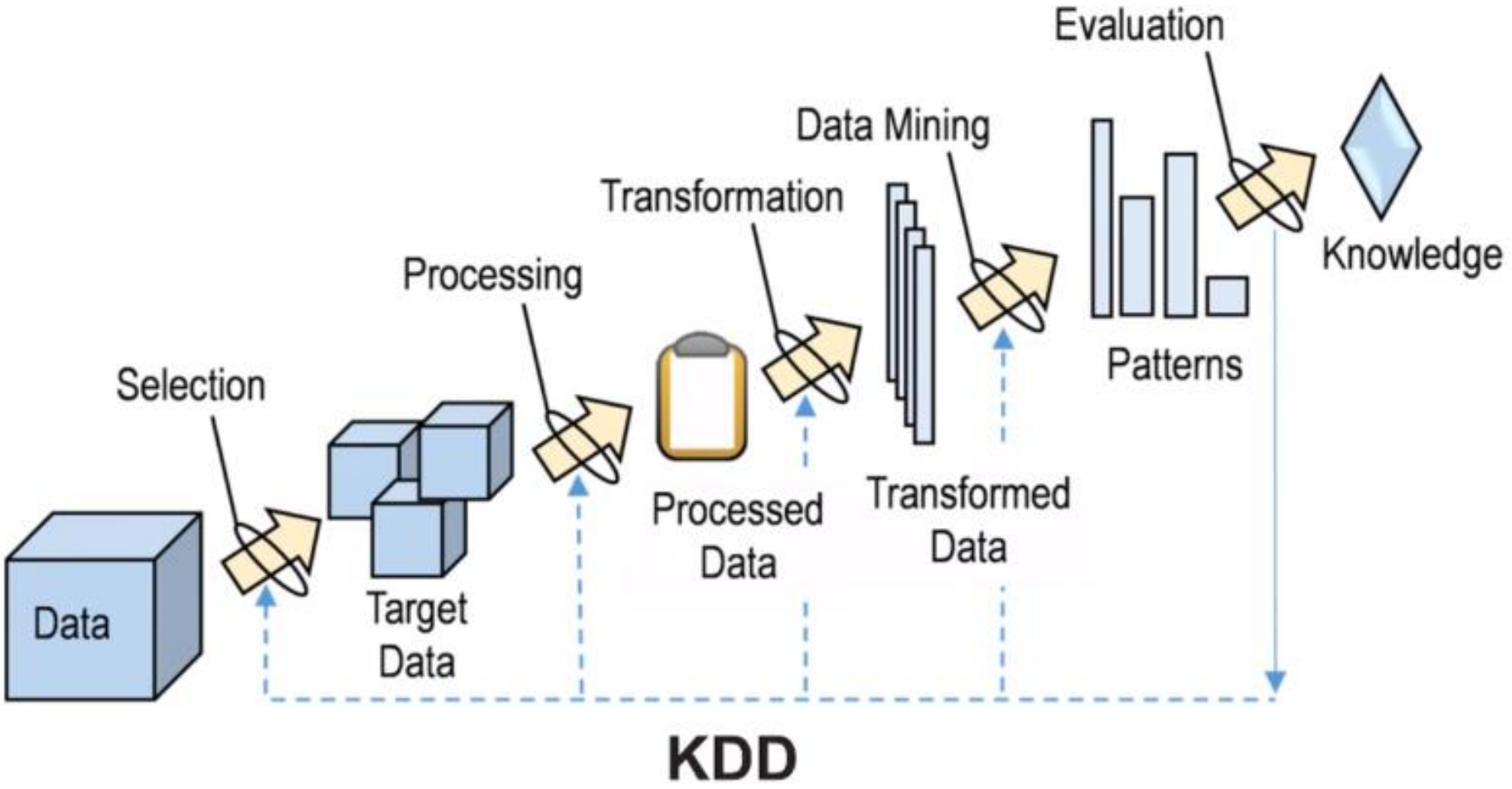
Since the term "data mining" is now outdated and often referred to as "Data Science", we call it the **Data Science Process**.



SEMMA


Sample
(Generate a representative Sample of the data)
Explore
(Visualization and basic description of the data)
Modify
(Select variables, transform Variable representations)
Model
(Use variety of statistical and Machine learning models)
Assess
(Evaluate the accuracy and usefulness of the models)
SEMMA
SEMMA
Pr

KDD


Evaluation
Data Mining
Transformation
Processing
Selection
Knowledge
Patterns
Transformed Data
Processed Data
Target Data
Data
KDD

1- Bussiness Understanding

Be clear on the business goals, success criteria and Data Science goals, and plan out project tools and workflow

- Six Data Science Process (CRISP-DM) steps:

  a) Business goal(s) – e.g. increase profit by $1M in 2020 via improved customer retention

  b) Success criteria – e.g. reduce churn rate from current 10% to 5% in 2020 (based on customer annual profitability calculations)

  c) Data Science goal(s) (with understanding you don't know what you don't know) - e.g. Identify 80k customers with annual customer value of at least $100

  d) Data Science planning - Make first assessment of what type(s) or combinations of Modeling functions will be required to satisfy business objective: predictive modeling, clustering, affinity, etc.

  e) Plan tools and map out workflow

2- Data Understanding

Know the underlying data

Next, it's important to get an understanding of the data you'll be working with. Ask these questions and perform these tasks.

a) Multiple data sources?

b) Vendor data source?

c) Which object type?

d) Is it accessible?

e) Which Schema?

f) Describe and explore data

g) Verify data adequacy

h) Verify data quality

3- Data preparation

Does data need preparing?

Often, data preparation is needed as determined in the previous step. In fact, sometimes 80% of the time on a Data Science project is spent on data preparation. Answer these questions.

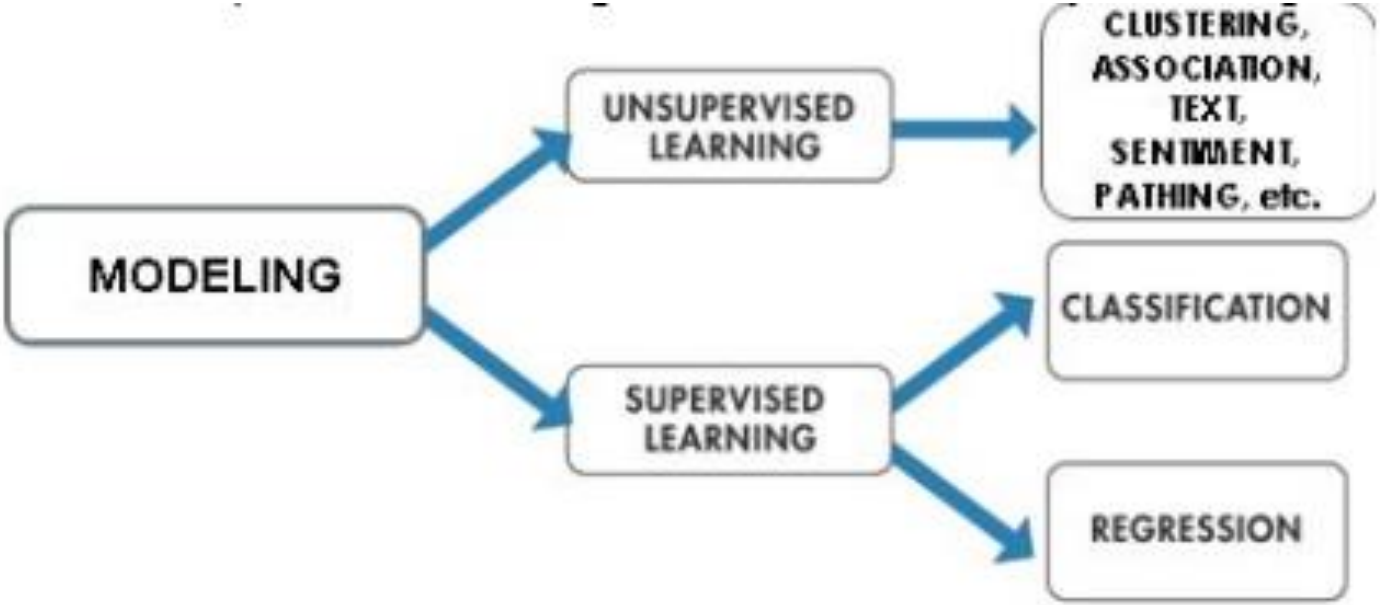
Needs Preparing? Yes/No

Integrate data from different sources?

- If prediction, have target variable? If not, construct from available data and append to each record
- Normal Distribution?
- Remove Outliers?
- Missing Data? Remove or replace (e.g. with average, with median)?
- Combine variables? (e.g. convert Length and Width to Area)
- Categorical converted to Numeric or vice versa
- Binning (e.g. putting data into buckets)
- Aggregate data? (e.g. monthly or quarterly sales)

4 - Modeling

Which predictive / analyitc functions (supervised/unsupervised), which arguments, and assess accuracy of findings



4 - Modeling

Create the Model (supervised) / Perform Analysis (unsupervised)

- Is this Supervised or Unsupervised Learning?
  - If Supervised, Create Model: is it Regression or Classification?
  - Select Regression/Classification functions
  - Split input data into training and test/validation sets
  - Create the Model(s) on training set by executing Functions
  - Assess which arguments produce best results on training set

  - If Unsupervised, Perform Analysis:
  - Execute Functions on full input data set
  - Assess which arguments produce best results

4 - Modeling

Assess Model Performance – Predictive Model Accuracy / Analysis Output

- Analysis Output - For Unsupervised analysis (descriptive models with no target variable such as association, clustering)
  - View Output – Is analysis working /producing meaningful output that meet Data Science goals?
    e.g. check if interesting clusters or useful affinity lift values
  - Use visualization to help assess reasonableness of analysis output
    e.g., for affinity does Chart clearly show/differentiate strengths of association between items?
- Predictive Model Accuracy - For Supervised models (predictive Models with target variable)
  - Validate predictive model on test set and review prediction accuracy (score model)
    e.g. using Confusion Matrix function to get accuracy metrics

5 - Evaluation

Evaluate findings against business goals and success criteria

Once you have done your modeling and have an idea of the accuracy/performance, you will now evaluate against the business goals/success criteria established in step 1.

a) Assess business value of the Model/analysis output

Is Output Actionable (Does it sufficiently meet business goals/success criteria?), nice to know or new starting point?

'Pivot' as needed (often the output is the start of a new overall goal. Repeat starting at Step 1. Business Understanding or Step 2: Data Understanding)

Is there a business reason model/analysis is deficient?

e.g. A model may not predict accurately enough in disease prediction to become a standard test

If time and budget permit, use the model/analysis in a pilot application before fully operationalizing

6 - Deployment

Operationalize, monitor and manage/revisit over time

The end goal is to "operationalize" the analytic findings. Taking analytics from insight to impact – the process of deploying analytics results within the business environment for use/reuse, to meet business goals, and ultimately improve the operation of the business (e.g. improve customer retention). Ask these questions and perform these tasks.

a) Plan deployment (how to operationalize) - If output is actionable (determined in step 5. Evaluation), what is the plan to deploy/operationalize? How are you going to operationalize the model/analysis in your business to achieve business goals/success criteria?

Operationalizing the analytic findings:

For descriptive analytics, this may involve creating reports/tables for internal business customers that document results /recommendations.

- For predictive analytics, it may involve providing executable predictive models that can be embedded within operational systems.

6 - Deployment

Operationalize, monitor and manage/revisit over time

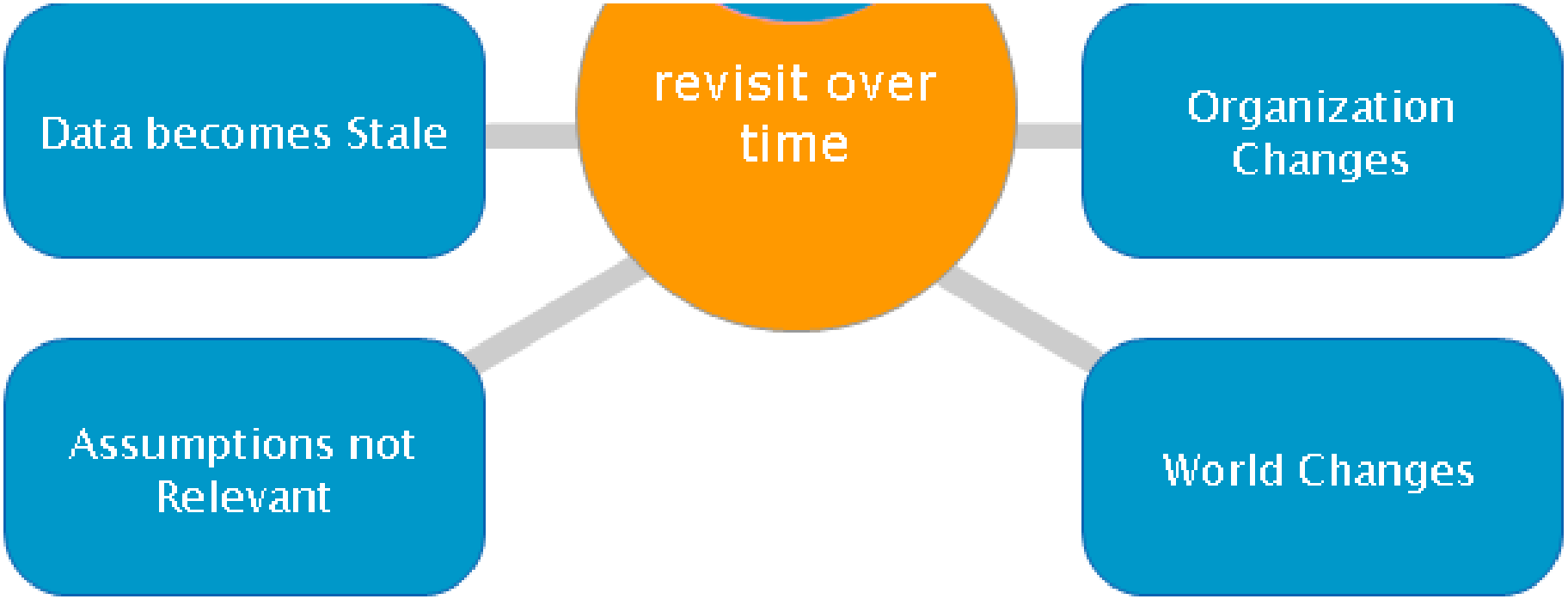
b) Plan monitoring and maintenance – After the analytic findings have been "operationalized", it is important to have a monitoring and maintenance plan in place, to ensure business goals continue to be achieved. **Are the analytic findings still relevant as time marches on?**

This is especially important with regard to predictive Models

- Models may become 'stale' decreasing their predictive accuracy
- The 'state of the world' may change in the future; e.g., market influences, consumer preferences and behaviors, economic realities, etc.
- Underlying data may change, new data may become available, etc.

Models may need to be rebuilt using new data (returning to Step 4. Modeling to use more recent data or to Step 2. Data Understanding if new sources of input data have become available.)

6 - Deployment



https://the-modeling-agency.com/crisp-dm.pdf

Data Sciences Test

Question 7. Which type of modeling finds patterns in unknown data; there is no dependent variable to match. The output is information about the data under analysis, not an executable model.

Unsupervised Learning

Question 8. Which type of modeling are these: clustering, association, text, sentiment, and pathing.

Unsupervised Learning

Question 9. Which type of modeling involves splitting the data into training and test data sets and performs classification and prediction, matching a set of independent variables with one or more target/dependent variable(s) in each input record.

Supervised Learning

Question 10. In which Data Science Process - CRISP-DM step might you determine you need additional data preparation based on the algortihm/function you are using and also assess the model performance - predictive model accuracy or analysis output?

4. Modeling

Data Sciences

Tools, Languages, Data Types, and Beyond

Analytic tools

Analytic languages

New data types and formats

Various data sources

Data Sciences

Data Sciences

Exploring Your Data

Exploratory Data Analysis on Raw (and Transformed) Data

Data Profiling / EDA in Vantage

- Univariate Statistics
- Distributions and Distribution Matching
- Percentiles
- Moving Aggregates
- Correlation Analysis
- Pathing Analysis
- Identity Match
- Outlier Detection
- 100's of others

Data Sciences

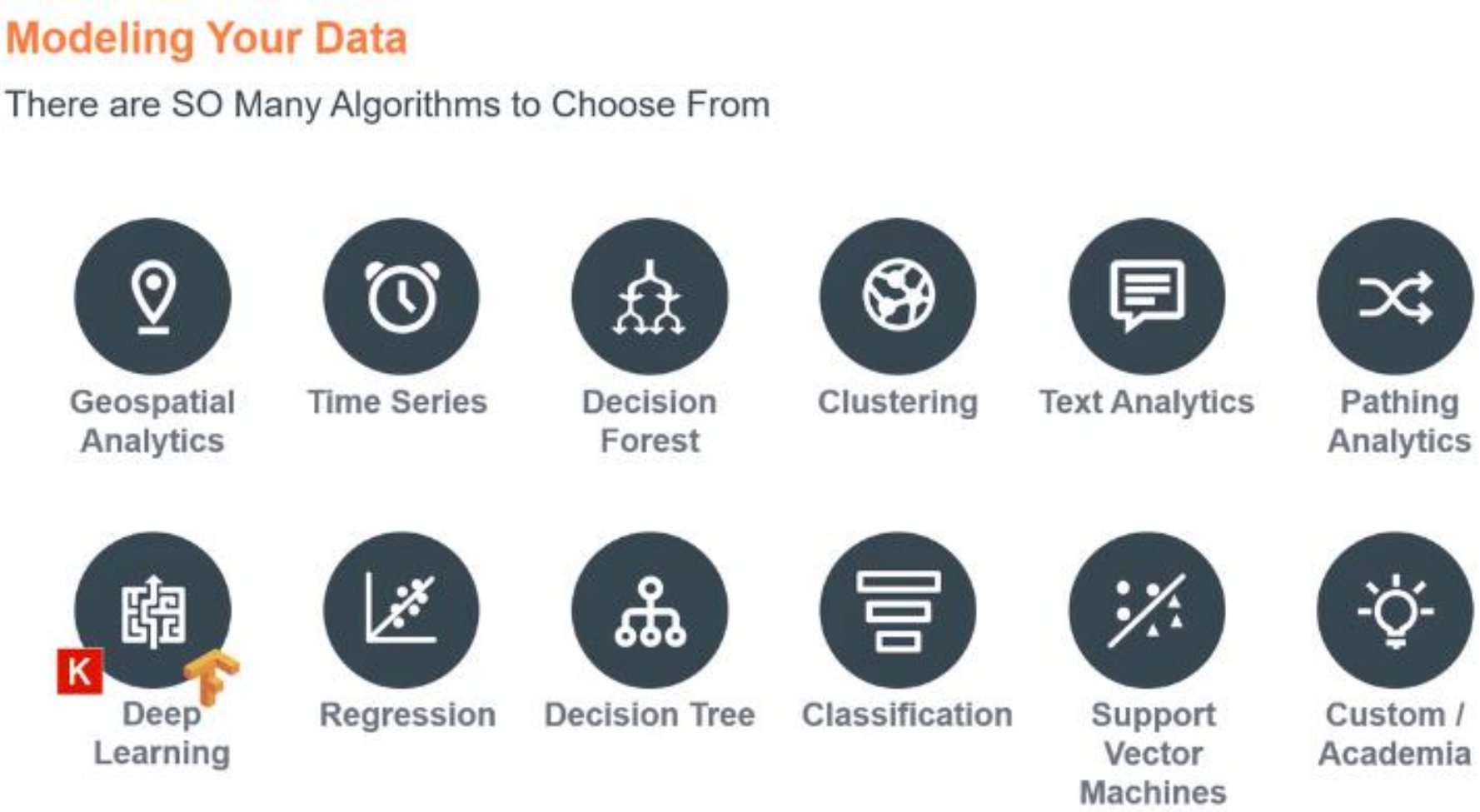
Transforming Your Data

From Data to Variables

Feature Engineering in Vantage

- SQL is a powerful Variable Creation tool!
  - Formulas, Joining, Dimensioning
- Pathing Analysis
- Pivoting / UnPivoting
- Parsing
  - JSON, XML, Apache Logs
- Scaling/Rescaling
- Design Coding
- Sessionizing
- 100's of others

Data Sciences Algorithms



Data Sciences Algorithms

Most Widely Used Categories of Analytical Techniques

All Supported by Vantage

- **Classification / Regression (Supervised Learning)**
  - Predictive models that use relationships in historical data to predict future outcomes. Classification is used to predict which class a discrete data point is part of, while regression is used to predict a continuous value.
- **Clustering / Segmentation (Unsupervised Learning)**
  - Group together individuals in a population by their similarity, with no pre-conceived hypothesis of clustering factors. The objects within clusters are similar to one another, while objects across clusters are dissimilar.
- **Time Series / Path Analysis**
  - Time series analysis is used for analyzing data that varies over time. Path analysis is a specific type of time series analysis that can be used in various time-based applications such as identifying the "golden path" that customers take leading to a purchase.
- **Text Analytics**
  - Obtain meaning from natural language text. Text information is converted into word frequencies, which then become numbers that can be analyzed.

Analytics Library

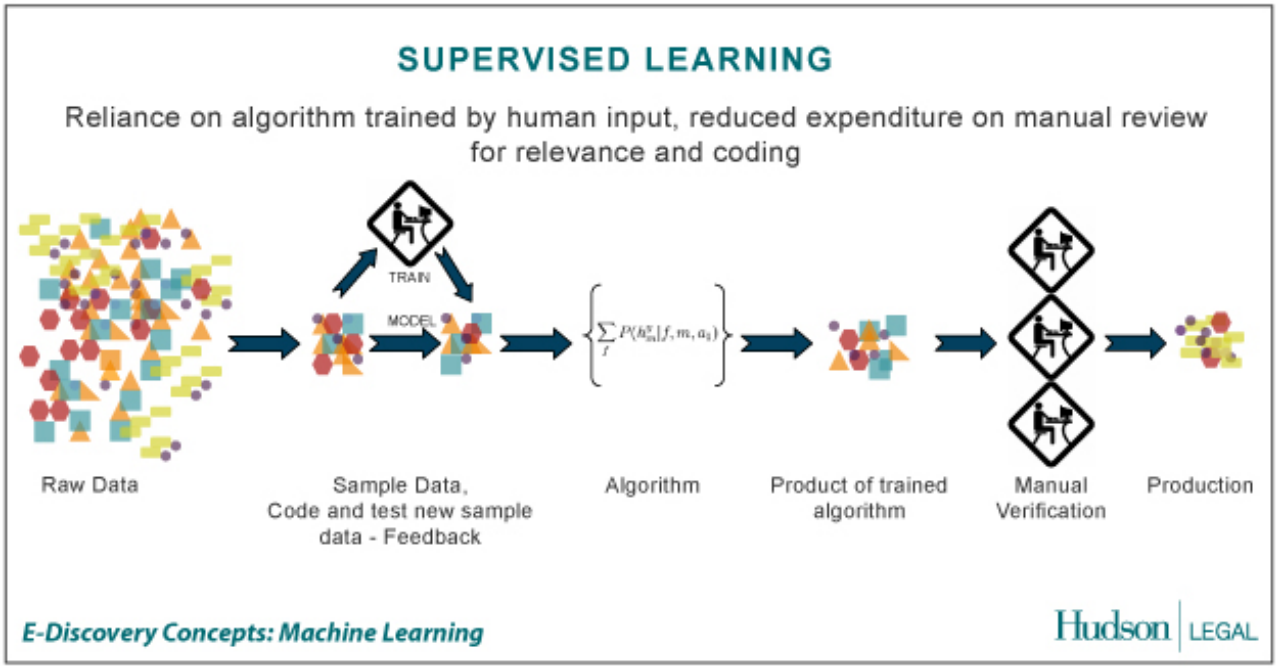
The Analytics Library consists of a Java XSP (a SQL generator), 2 table operators (to support the KMeans and Decision Tree algorithms) and statistical test lookup tables that enable Data Scientists to analyze large datasets at scale. It operates entirely inside Vantage SQL Engine (SQLE) and no additional infrastructure is required to run. The Analytics Library behaves like a database version of an API and is compatible with Vantage SQLE 16.20 and 17.0 and 17.05. It is easy to install and requires only 40MB of space in the SQLE database for the functions.The Analytics Library provides data scientists with the following capabilities, directly in the Vantage SQLE:

- Descriptive Statistics
  - Data Quality/Values
  - Univariate Statistics
  - Frequency
  - Histogram
  - RI Analysis (column overlap)
  - Text Field Analysis
- Data Transformation
  - Bin Code
  - Formula Derivation
  - One-Hot Encoding
  - Null Value Replacement
  - Recode/Rescale
  - Sigmoid/Z-Score
- Hypothesis Tests
  - Parametric Tests
  - Binomial Tests
  - Contingency Tests
  - KS Tests
  - Rank Tests
- Model Building & Scoring
  - Matrix (ESSCP/Covariance/Correlation)
  - Linear Regression
  - Logistic Regression
  - Factor Analysis
  - K-Means
  - Decision Tree
  - Association Rules/Sequence Analysis

https://downloads.teradata.com/download/database/analytics-library

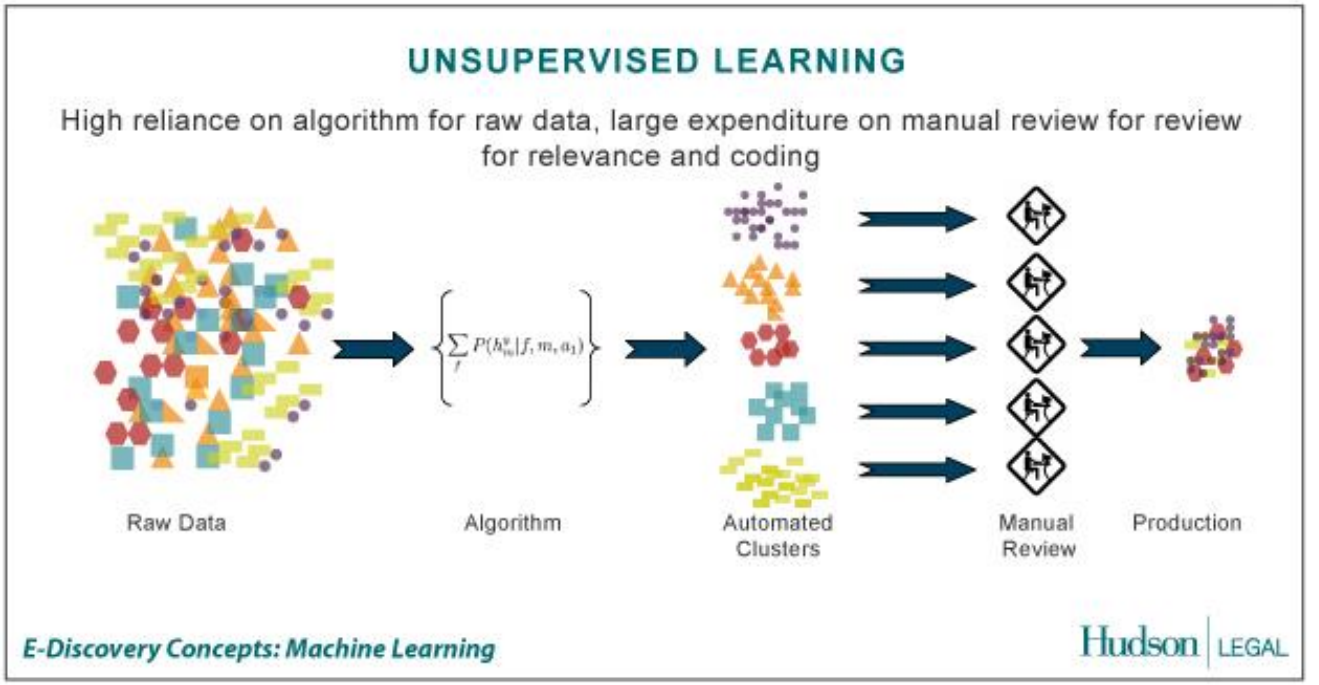
Aprendizaje supervisado

Se conocen las clases. Los datos se etiquetan.
Ejemplo: los correos son spam o legítimos.

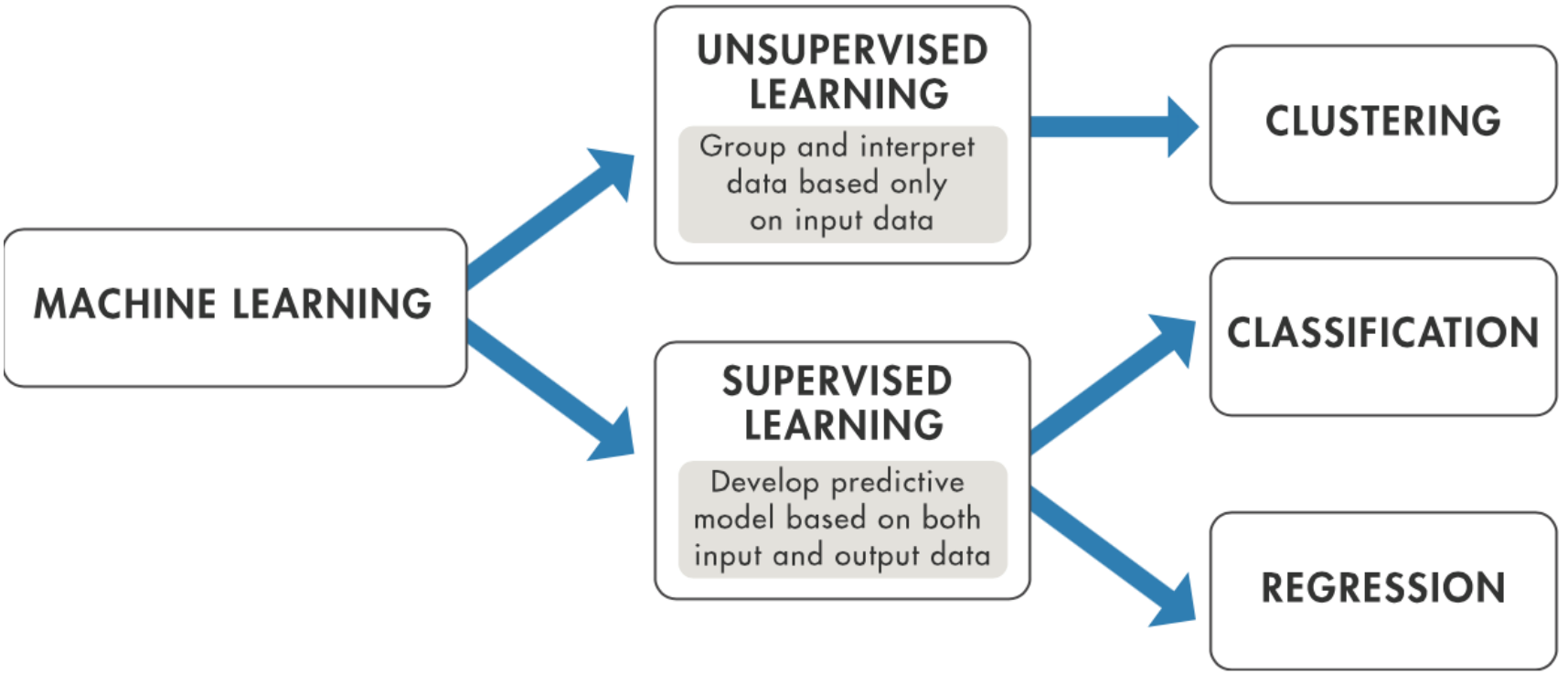


Aprendizaje no supervisado

No se conocen las clases. Los datos no se etiquetan. No se conoce estructura o forma de organizar los datos.

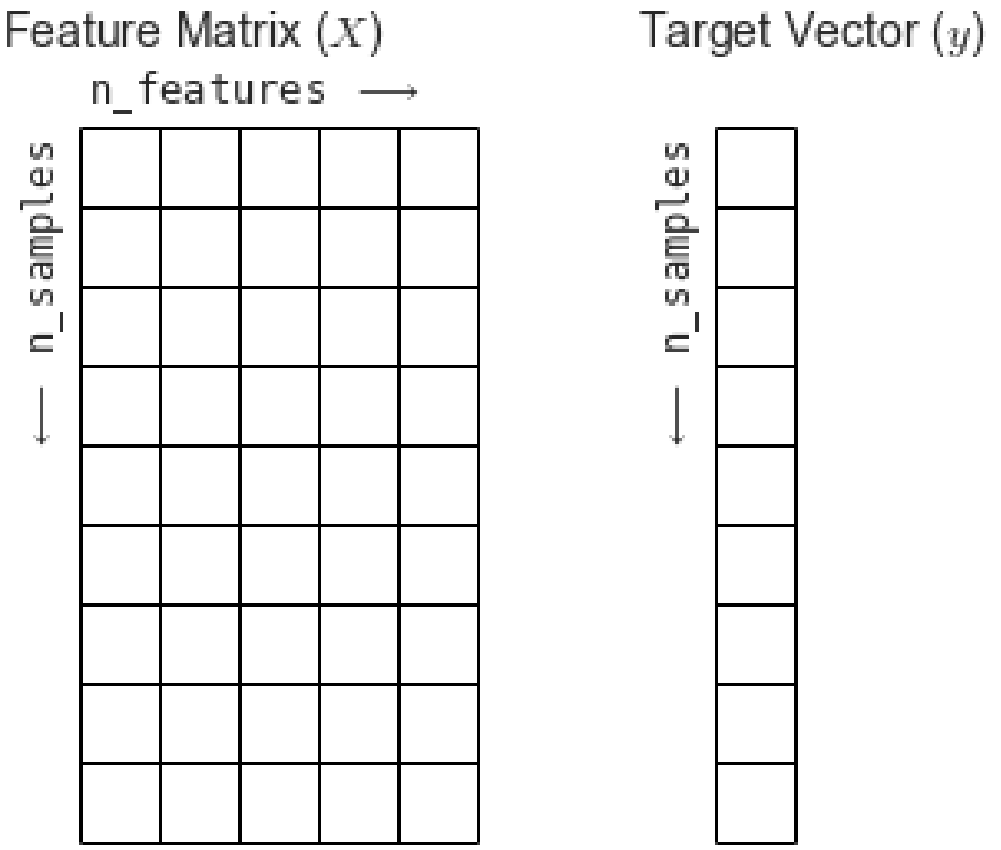


Machine Learning



Matrix and target vector

the expected layout of features and target values is:



Target array

In addition to the feature matrix X, we also generally work with a *label* or *target* array, which by convention we will usually call y.

The target array is usually one dimensional, with length n_samples, and is generally contained in a NumPy array or Pandas Series.

The target array may have continuous numerical values, or discrete classes/labels.

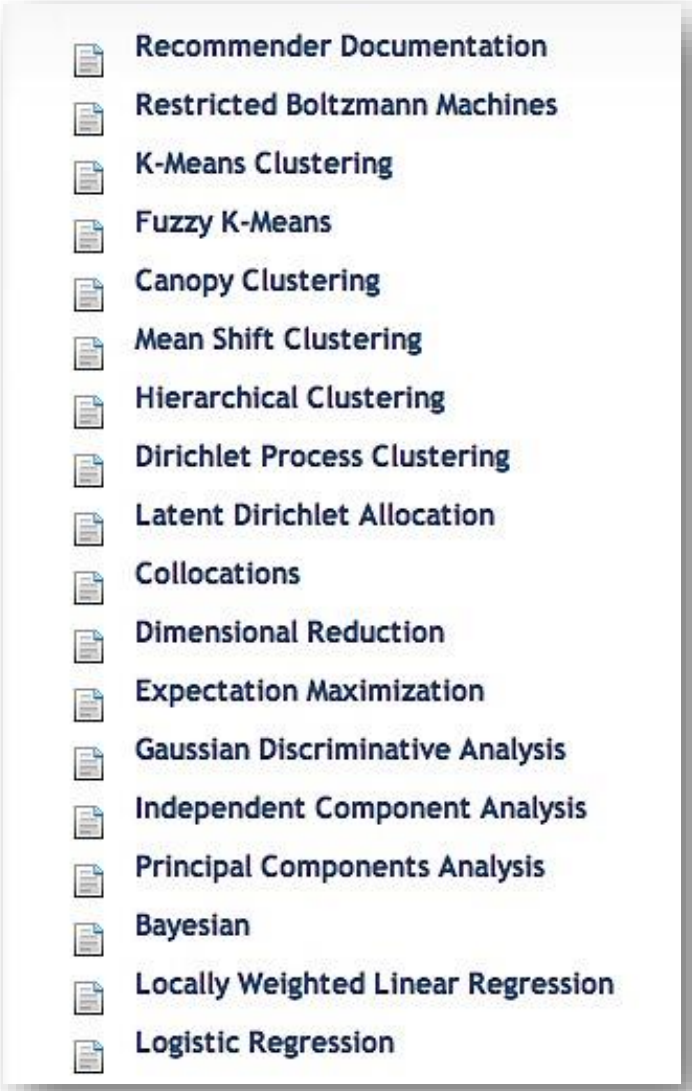
The distinguishing feature of the target array is that it is usually the quantity **we want to *predict from the data***; in statistical terms, it is the **dependent variable**.

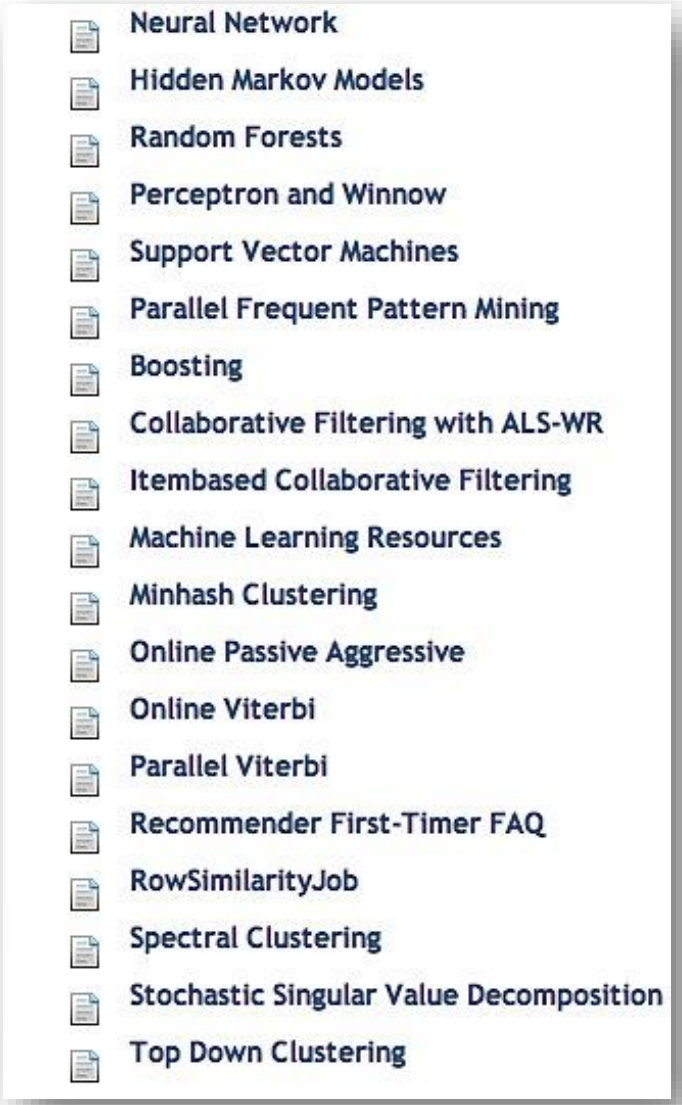
Machine learning steps

Most commonly, the ML are as follows:

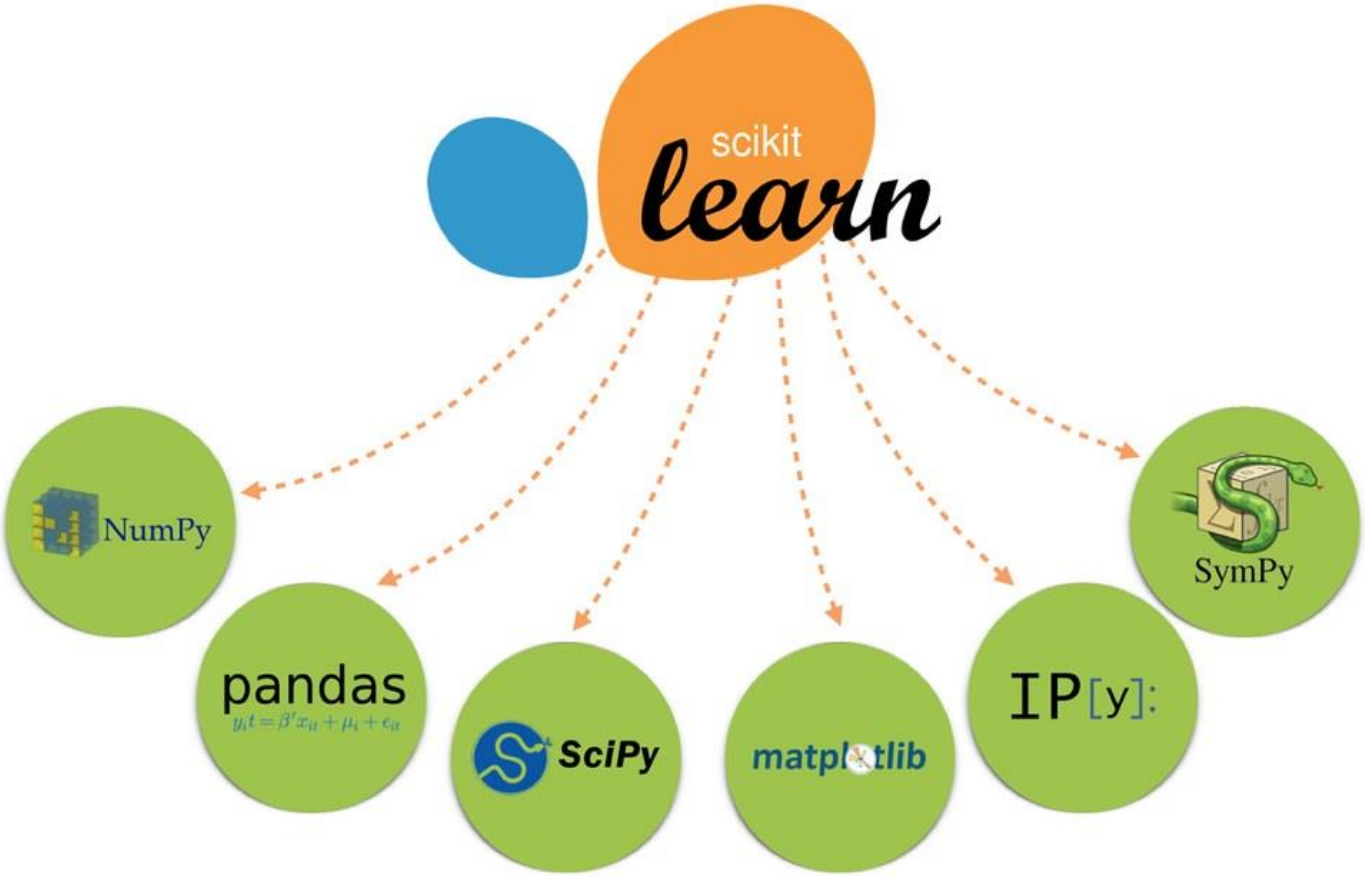
1. Choose a class of **model** by importing the appropriate estimator
2. Choose model hyperparameters by instantiating this class with desired values.
3. Arrange data into a features **matrix and target vector**
4. Fit the model to your data by calling the **fit()** method of the model instance.
5. Apply the Model to new data:
    1. For supervised learning, often we predict labels for unknown data using the **predict()** method.
    2. For unsupervised learning, we often transform or **infer properties** of the data using the **transform() or predict()** method.

Machine Learning





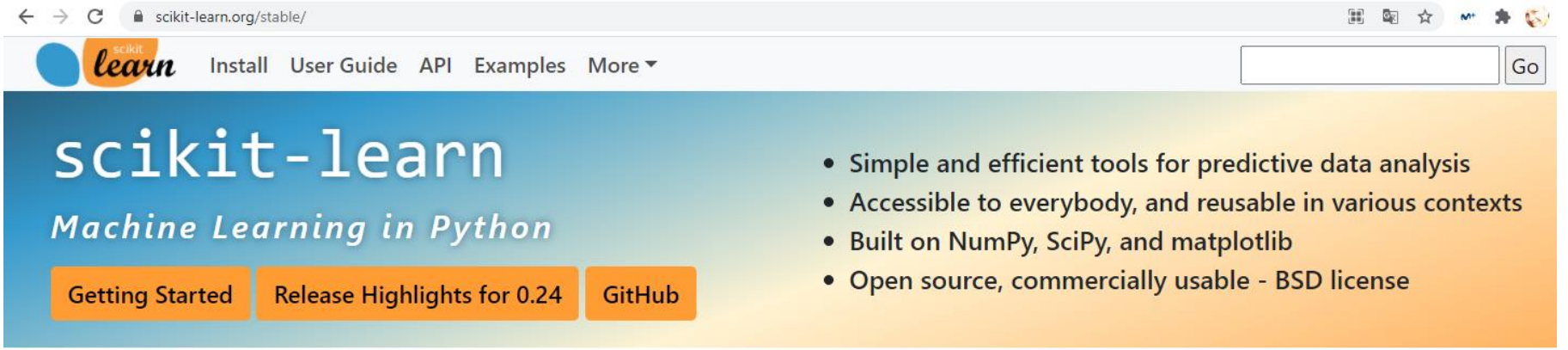
Scikit-learn



Luís Garmendia

Scikit-learn

There are several Python libraries which provide solid implementations of a range of machine learning algorithms. One of the best known is Scikit-Learn, a package that provides efficient versions of a large number of common algorithms.



https://scikit-learn.org/stable/

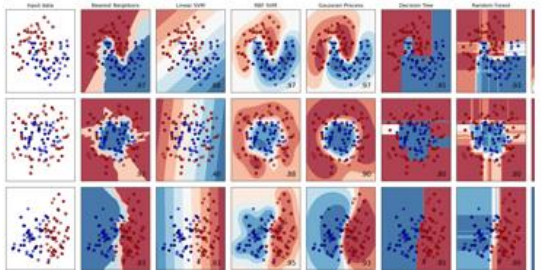
Scikit-learn

https://scikit-learn.org/stable/

Classification

Identifying which category an object belongs to.

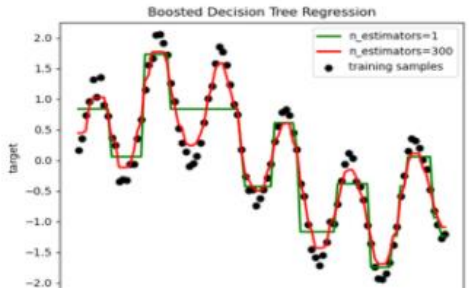
Applications: Spam detection, image recognition.
Algorithms: SVM, nearest neighbors, random forest, and more...



Regression

Predicting a continuous-valued attribute associated with an object.

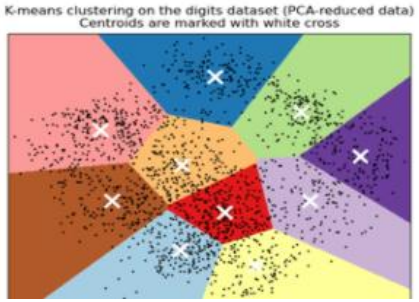
Applications: Drug response, Stock prices.
Algorithms: SVR, nearest neighbors, random forest, and more...



Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: k-Means, spectral clustering, mean-shift, and more...



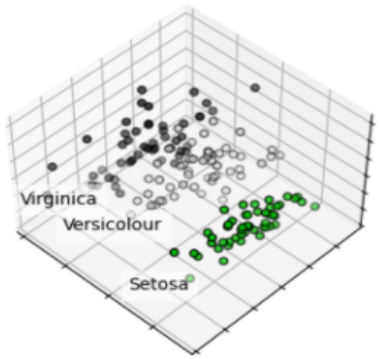
Scikit-learn

https://scikit-learn.org/stable/

Dimensionality reduction

Reducing the number of random variables to consider.

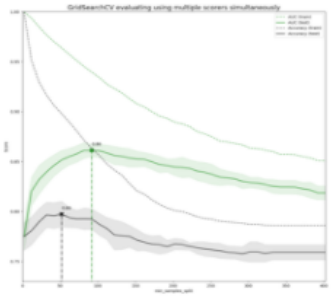
Applications: Visualization, Increased efficiency
Algorithms: k-Means, feature selection, non-negative matrix factorization, and more...



Model selection

Comparing, validating and choosing parameters and models.

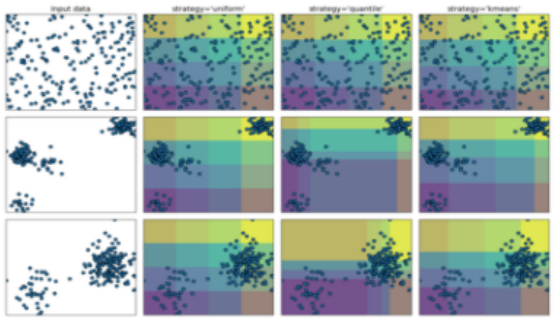
Applications: Improved accuracy via parameter tuning
Algorithms: grid search, cross validation, metrics, and more...



Preprocessing

Feature extraction and normalization.

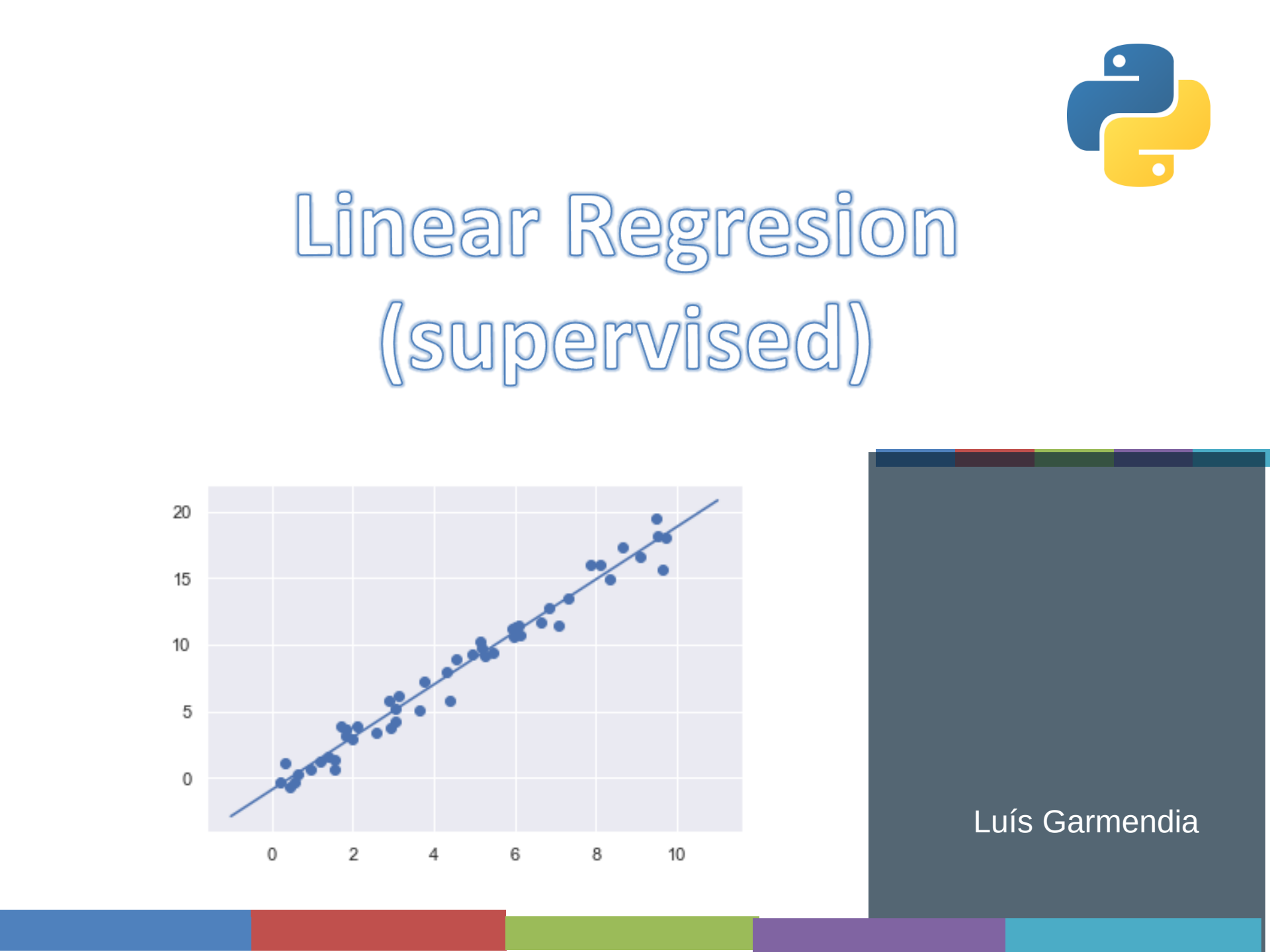
Applications: Transforming input data such as text for use with machine learning algorithms.
Algorithms: preprocessing, feature extraction, and more...



Exercise : Machine learning steps

Most commonly, the ML are as follows:

1. Choose a class of **model** by importing the appropriate estimator
2. Choose model hyperparameters by instantiating this class with desired values.
3. Arrange data into a features **matrix and target vector**
4. Fit the model to your data by calling the **fit()** method of the model instance.
5. Apply the Model to new data:
    1. For supervised learning, often we predict labels for unknown data using the **predict()** method.
    2. For unsupervised learning, we often transform or **infer properties** of the data using the **transform() or predict()** method.


Linear Regresion
(supervised)
20
15
10
5
0
0
2
4
6
8
10
Luís Garmendia

Appying a supervised ML model

Basic recipe for applying a supervised machine learning model:

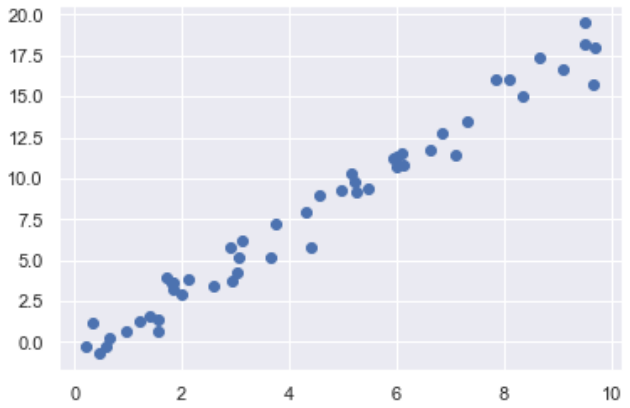
1. Choose a class of model
2. Choose model hyperparameters
3. Fit the model to the training data
4. Use the model to predict labels for new data

Linear Regression (supervised)

As an example of this process, let's consider a simple linear regression—that is, the common case of fitting a line to (x,y) data.

```
import matplotlib.pyplot as plt
import numpy as np

rng = np.random.RandomState(42)
x = 10 * rng.rand(50)
y = 2 * x - 1 + rng.randn(50)
plt.scatter(x, y);
```

```
In [20]: import matplotlib.pyplot as plt
         import numpy as np

         rng = np.random.RandomState(42)
         x = 10 * rng.rand(50)
         y = 2 * x - 1 + rng.randn(50)
         plt.scatter(x, y);
```




```
In [21]: x
Out[21]: array([3.74540119, 9.50714306, 7.31993942, 5.98658484, 1.5601864 ,
                1.5599452 , 0.58083612, 8.66176146, 6.01115012, 7.08072578,
```

Install sklearn
pip install sklearn

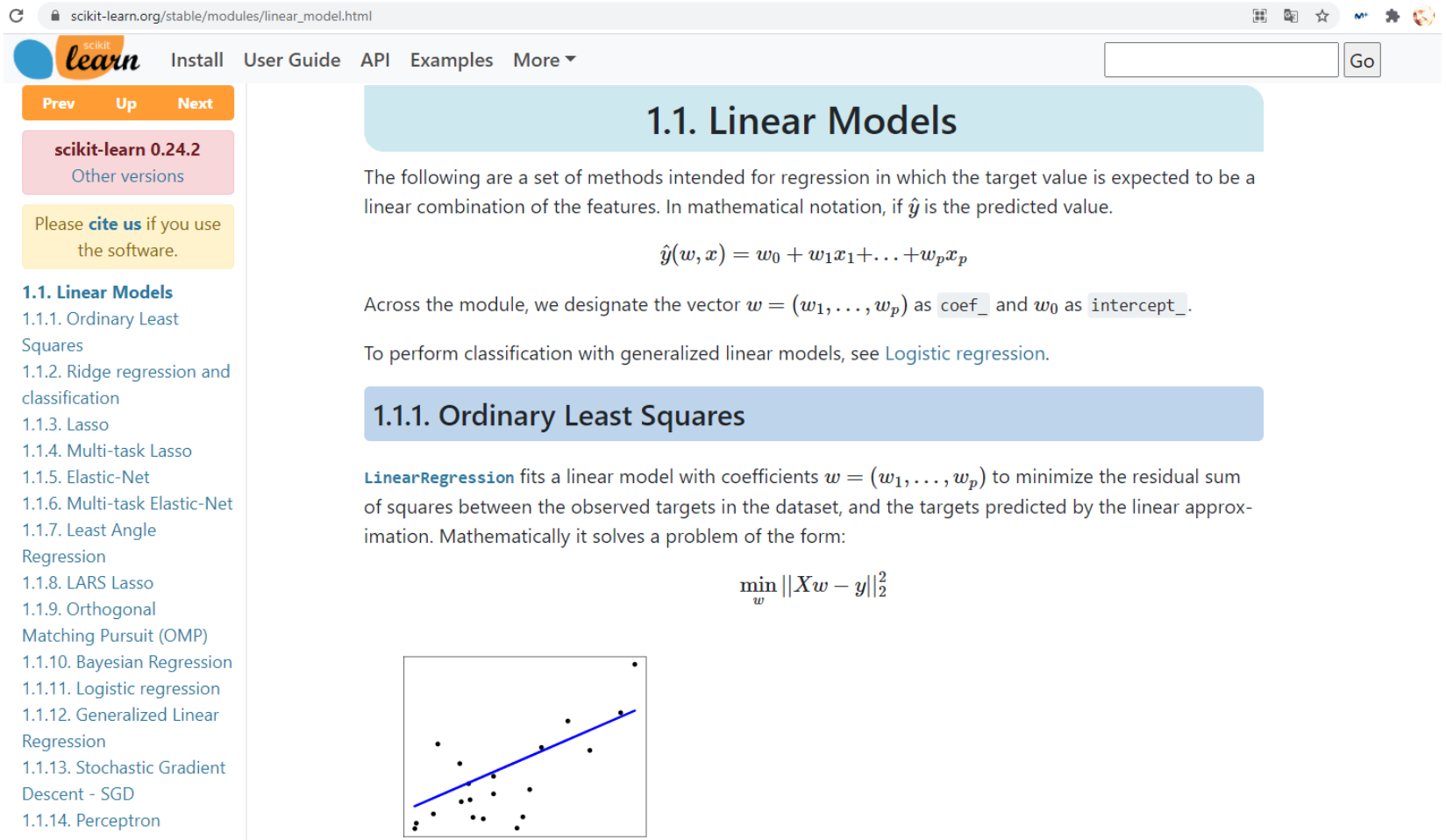
```
In [23]: pip install sklearn
```

```
Collecting sklearn
  Downloading sklearn-0.0.tar.gz (1.1 kB)
Requirement already satisfied: scikit-learn in c:\users\luis\anaconda3\lib\site-packages (from sklearn) (0.22.1)
Requirement already satisfied: scipy>=0.17.0 in c:\users\luis\anaconda3\lib\site-packages (from scikit-learn->sklearn) (1.4.1)
Requirement already satisfied: numpy>=1.11.0 in c:\users\luis\anaconda3\lib\site-packages (from scikit-learn->sklearn) (1.18.1)
Requirement already satisfied: joblib>=0.11 in c:\users\luis\anaconda3\lib\site-packages (from scikit-learn->sklearn) (0.14.1)
Building wheels for collected packages: sklearn
  Building wheel for sklearn (setup.py): started
  Building wheel for sklearn (setup.py): finished with status 'done'
  Created wheel for sklearn: filename=sklearn-0.0-py2.py3-none-any.whl size=1320 sha256=264abce7084e9dc81b0a059edb1b2ac99385b71
d0ca342a6e5b642fed1b82dc7
  Stored in directory: c:\users\luis\appdata\local\pip\cache\wheels\46\ef\c3\157e41f5ee1372d1be90b09f74f82b10e391eaacca8f22d33e
Successfully built sklearn
Installing collected packages: sklearn
Successfully installed sklearn-0.0
Note: you may need to restart the kernel to use updated packages.
```

```
In [24]: from sklearn.linear_model import LinearRegression
```

Read documentation

https://scikit-learn.org/stable/modules/linear_model.html



Choose model hyperparameters

An important point is that *a class of model is not the same as an instance of a model*.

Once we have decided on our model class, there are still some options open to us. Depending on the model class we are working with, we might need to answer one or more questions like the following:

- Would we like to fit for the offset (i.e., *y*-intercept)?
- Would we like the model to be normalized?
- Would we like to preprocess our features to add model flexibility?
- What degree of regularization would we like to use in our model?
- How many model components would we like to use?

1 - Instantiate the model

We would like to fit the intercept using the fit_intercept hyperparameter:

```
from sklearn.linear_model import LinearRegression
model = LinearRegression(fit_intercept=True)
model
```

```
In [24]: from sklearn.linear_model import LinearRegression

In [25]: model = LinearRegression(fit_intercept=True)
         model

Out[25]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

2 - Arrange data into matrix X and target vector y

We need to coerce these *x* values into a [n_samples, n_features] matrix

X = x[:, np.newaxis]

X.Shape

```
In [28]: x

Out[28]: array([3.74540119, 9.50714306, 7.31993942, 5.98658484, 1.5601864 ,
                1.5599452 , 0.58083612, 8.66176146, 6.01115012, 7.08072578,
                0.20584494, 9.69909852, 8.32442641, 2.12339111, 1.81824967,
                1.8340451 , 3.04242243, 5.24756432, 4.31945019, 2.9122914 ,
                6.11852895, 1.39493861, 2.92144649, 3.66361843, 4.56069984,
                7.85175961, 1.99673782, 5.14234438, 5.92414569, 0.46450413,
                6.07544852, 1.70524124, 0.65051593, 9.48885537, 9.65632033,
                8.08397348, 3.04613769, 0.97672114, 6.84233027, 4.40152494,
                1.22038235, 4.9517691 , 0.34388521, 9.09320402, 2.58779982,
                6.62522284, 3.11711076, 5.20068021, 5.46710279, 1.84854456])

In [29]: X = x[:, np.newaxis]
         X.shape

Out[29]: (50, 1)

In [30]: X

Out[30]: array([[3.74540119],
                [9.50714306],
                [7.31993942],
                [5.98658484],
                [1.5601864 ],
                [1.5599452 ],
                [0.58083612],
                [8.66176146],
```

3 - Fit de model

model.fit(X, y)

slope and intercept of the simple linear fit:

model.coef_

model.intercept_

```
In [31]: model.fit(X, y)
Out[31]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

In [32]: model.coef_
Out[32]: array([1.9776566])

In [33]: model.intercept_
Out[33]: -0.9033107255311164
```

4 - Predict labels for unknown data

xfit = np.linspace(-1, 11)

we need to coerce these *x* values into a [n_samples, n_features]

Xfit = xfit[:, np.newaxis]

feed it to the model

yfit = model.predict(Xfit)

```
In [41]: xfit = np.linspace(-1, 11)

In [42]: Xfit = xfit[:, np.newaxis]
         yfit = model.predict(Xfit)

In [43]: Xfit.shape

Out[43]: (50, 1)

In [44]: yfit.shape

Out[44]: (50,)

In [45]: Xfit

Out[45]: array([[-1.        ],
                [-0.75510204],
                [-0.51020408],
                [-0.26530612],
                [-0.02040816],
                [ 0.2244898 ],
```

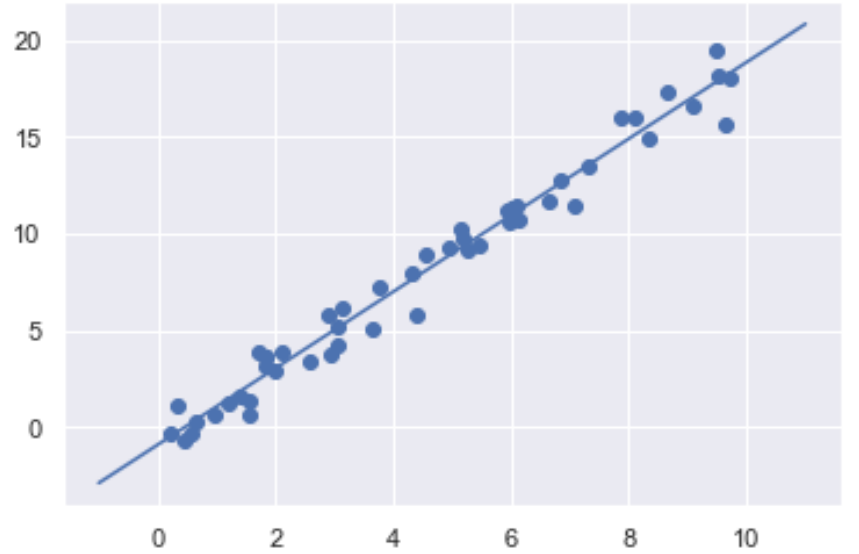
5 - visualize the data

visualize the results by plotting first the raw data, and then this model fit

plt.scatter(x, y)

plt.plot(xfit, yfit);

```
In [46]: plt.scatter(x, y)
         plt.plot(xfit, yfit);
```



Data as table

A basic table is a two-dimensional grid of data, in which the rows represent individual elements of the dataset

```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
```

```
In [18]: import seaborn as sns
         iris = sns.load_dataset('iris')
         iris.head()
```

Out[18]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Prepare iris X and y

```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()

X_iris = iris.drop('species', axis=1)
X_iris.shape

y_iris = iris['species']
y_iris.shape
```

In [51]:
```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
```

Out[51]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [52]:
```
X_iris = iris.drop('species', axis=1)
X_iris.shape
```

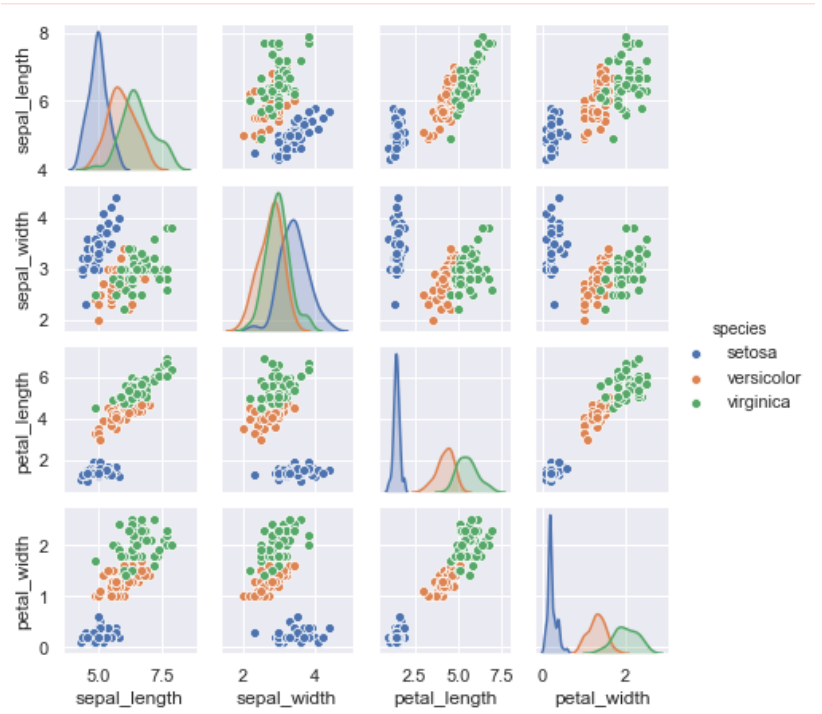
Out[52]: (150, 4)

In [53]:
```
y_iris = iris['species']
y_iris.shape
```

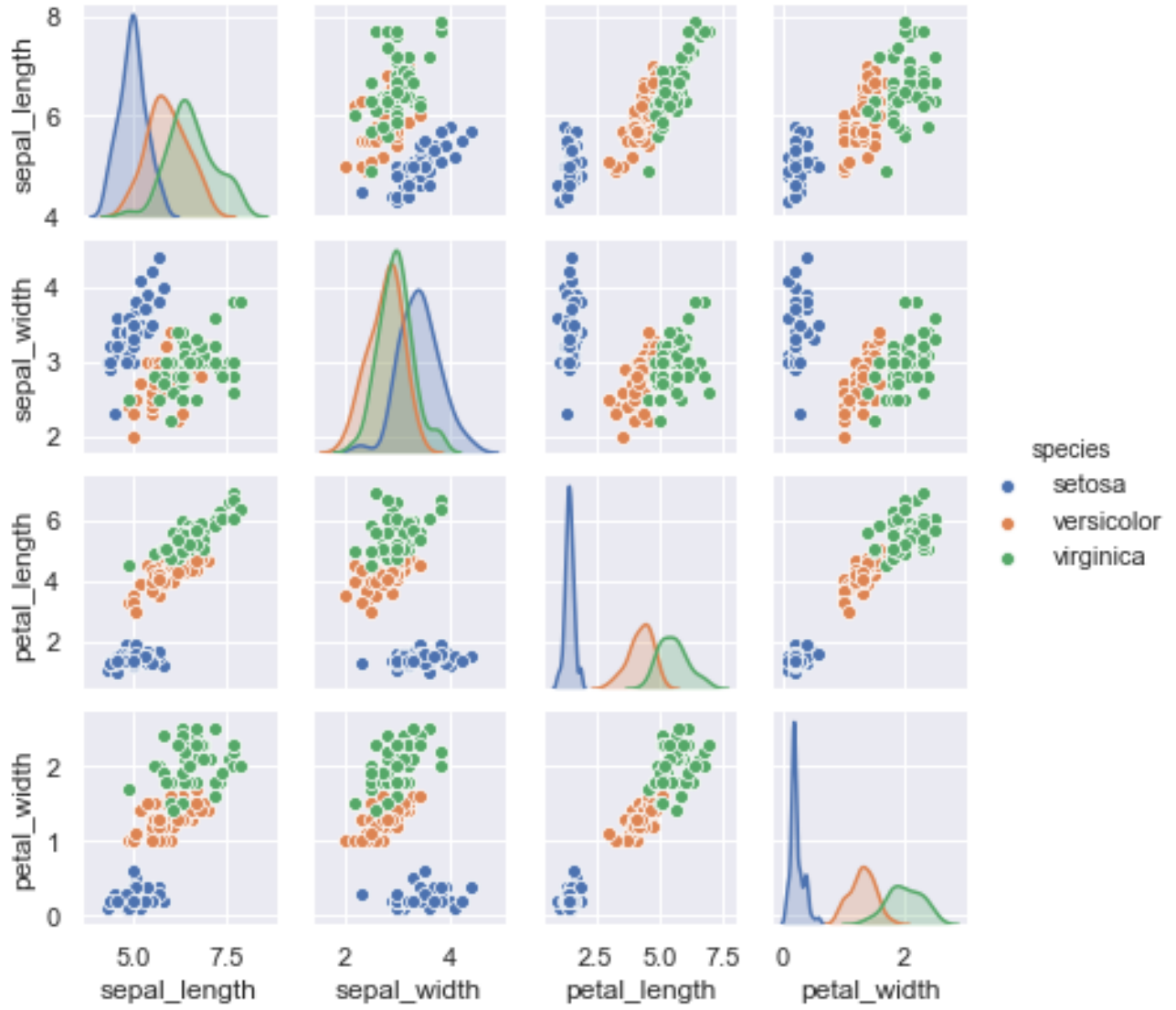
Out[53]: (150,)

Data visualization

```
%matplotlib inline
import seaborn as sns;
sns.set()
sns.pairplot(iris, hue='species', size=1.5);
```



Data visualization

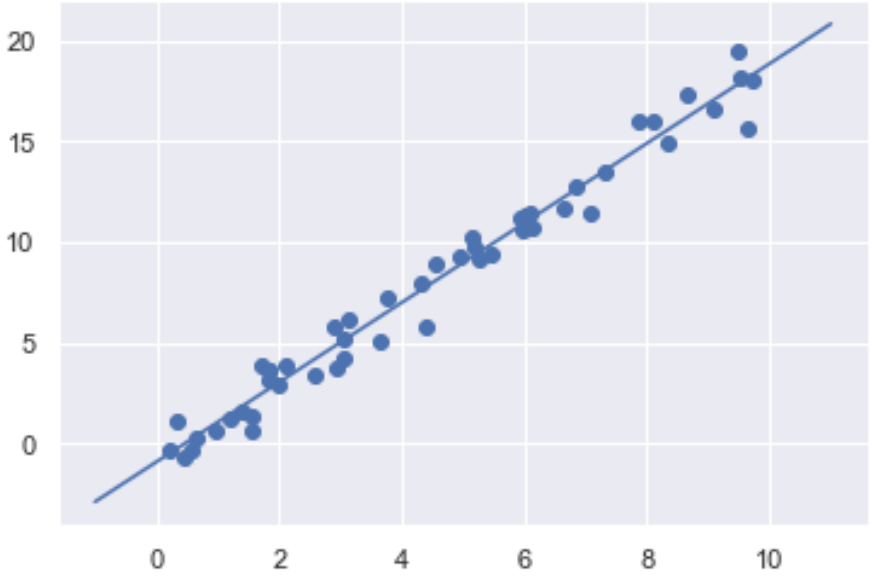


Exercise : Machine learning steps

Most commonly, the ML are as follows:

1. Choose a class of **model** by importing the appropriate estimator
2. Choose model hyperparameters by instantiating this class with desired values.
3. Arrange data into a features **matrix and target vector**
4. Fit the model to your data by calling the **fit()** method of the model instance.
5. Apply the Model to new data:
    1. For supervised learning, often we predict labels for unknown data using the **predict()** method.
    2. For unsupervised learning, we often transform or **infer properties** of the data using the **transform() or predict()** method.

Training Sets
Naive Bayes Classification
(supervised)



Luís Garmendia

Naïve Bayes

Naive Bayes models are a group of extremely fast and simple classification algorithms that are often suitable for very high-dimensional datasets. Because they are so fast and have so few tunable parameters, they end up being very useful as a quick-and-dirty baseline for a classification problem.

Naïve Bayes

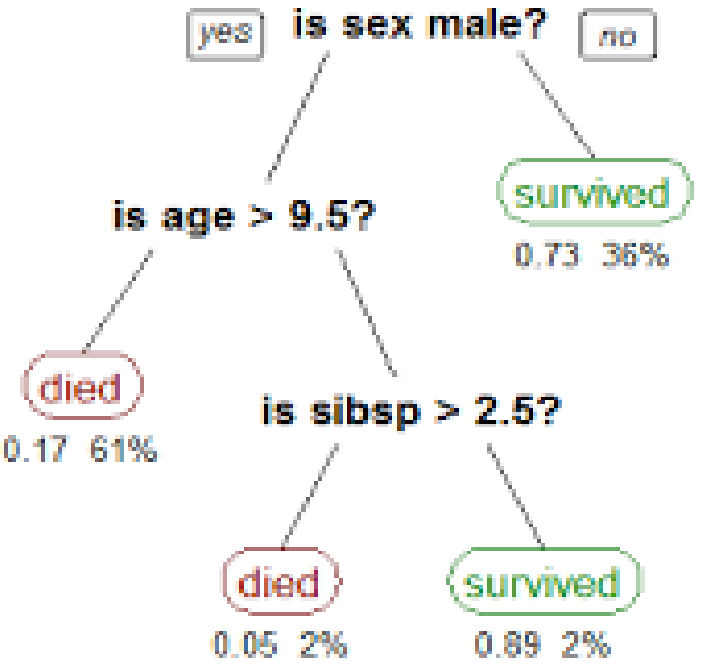
Naive Bayes models are a group of extremely fast and simple classification algorithms that are often suitable for very high-dimensional datasets. Because they are so fast and have so few tunable parameters, they end up being very useful as a quick-and-dirty baseline for a classification problem.

$$P(L \mid \text{features}) = \frac{P(\text{features} \mid L)P(L)}{P(\text{features})}$$

Naïve Bayes

Naive Bayes models are a group of extremely fast and simple classification algorithms that are often suitable for very high-dimensional datasets. Because they are so fast and have so few tunable parameters, they end up being very useful as a quick-and-dirty baseline for a classification problem.

$$P(L \mid \text{features}) = \frac{P(\text{features} \mid L)P(L)}{P(\text{features})}$$



train_test_split

We would like to evaluate the model on data it has not seen before, and so we will split the data into a *training set* and a *testing set*. This could be done by hand, but it is more convenient to use the train_test_split utility function

```
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> X, y = np.arange(10).reshape((5, 2)), range(5)
>>> X
array([[0, 1],
[2, 3],
[4, 5],
[6, 7],
[8, 9]])
 >>> list(y)
[0, 1, 2, 3, 4]
```

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

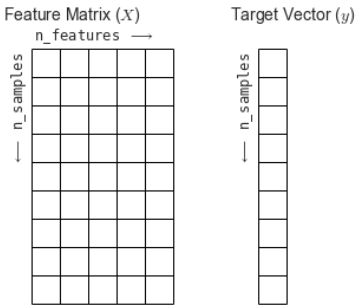
train_test_split

```
>>> X_train, X_test, y_train, y_test = train_test_split( ... X, y, test_size=0.33, random_state=42)
...
>>> X_train
array([[4, 5],
[0, 1],
[6, 7]])
>>> y_train
[2, 0, 3]
>>> X_test
array([[2, 3],
[8, 9]])
>>> y_test
[1, 4]
```

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

Prepare iris X and y

```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()

X_iris = iris.drop('species', axis=1)
X_iris.shape

y_iris = iris['species']
y_iris.shape
```




```
In [51]: import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
```

Out[51]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
In [52]: X_iris = iris.drop('species', axis=1)
X_iris.shape
```

Out[52]: (150, 4)

```
In [53]: y_iris = iris['species']
y_iris.shape
```

Out[53]: (150,)

Iris train_test_split

```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris, random_state=1)
```

```
In [54]: from sklearn.model_selection import train_test_split
         Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,
                                                         random_state=1)

In [55]: Xtrain.shape

Out[55]: (112, 4)

In [56]: ytrain.shape

Out[56]: (112,)

In [57]: Xtest.shape

Out[57]: (38, 4)
```

Instantiate and fit the model

```
from sklearn.naive_bayes import GaussianNB
# 1. choose model class
model = GaussianNB()
# 2. instantiate model
model.fit(Xtrain, ytrain)
 # 3. fit model to data
y_model = model.predict(Xtest)
```

Accuracy_score

Finally, we can use the accuracy_score utility to see the fraction of predicted labels that match their true value:

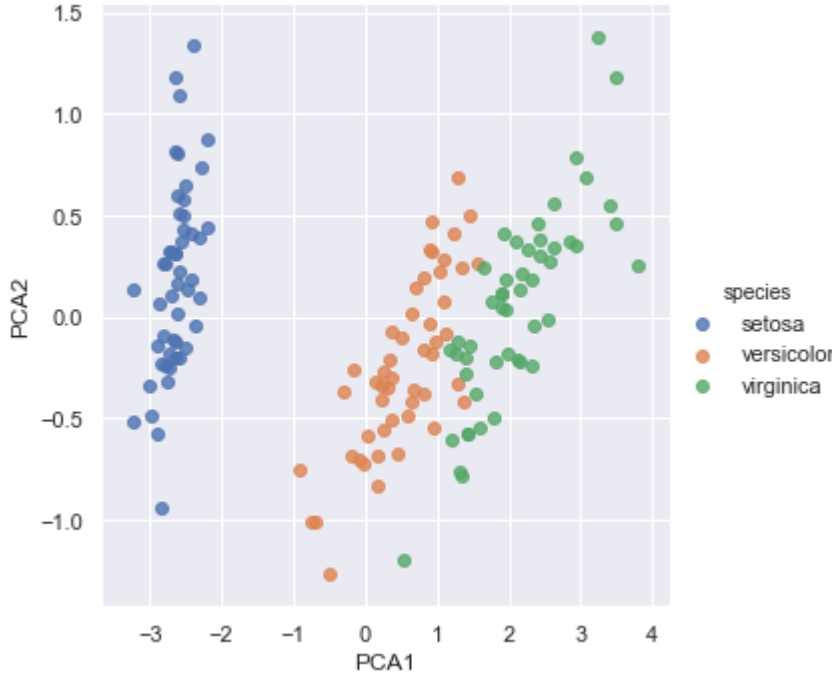
from sklearn.metrics import accuracy_score

accuracy_score(ytest, y_model)

```
In [59]: from sklearn.naive_bayes import GaussianNB # 1. choose model class
         model = GaussianNB()                       # 2. instantiate model
         model.fit(Xtrain, ytrain)                  # 3. fit model to data
         y_model = model.predict(Xtest)

In [60]: from sklearn.metrics import accuracy_score
         accuracy_score(ytest, y_model)

Out[60]: 0.9736842105263158
```

The naive classification algorithm is 97% effective for this particular dataset!

Unsupervised example: Iris dimensionality



Luís Garmendia

PCA: Principal component analysis

In this section, we explore what is perhaps one of the most broadly used of unsupervised algorithms, principal component analysis (PCA). PCA is fundamentally a **dimensionality reduction algorithm**, but it can also be useful as a tool for visualization, for noise filtering, for feature extraction and engineering, and much more.

We will ask the model to return two components—that is, a two-dimensional representation of the data.

PCA: Principal component analysis

```
from sklearn.decomposition import PCA # 1. Choose the model class
model = PCA(n_components=2)    # 2. Instantiate the model with hyperparameters
model.fit(X_iris)              # 3. Fit to data. Notice y is not specified!
X_2D = model.transform(X_iris) # 4. Transform the data to two dimensions
```

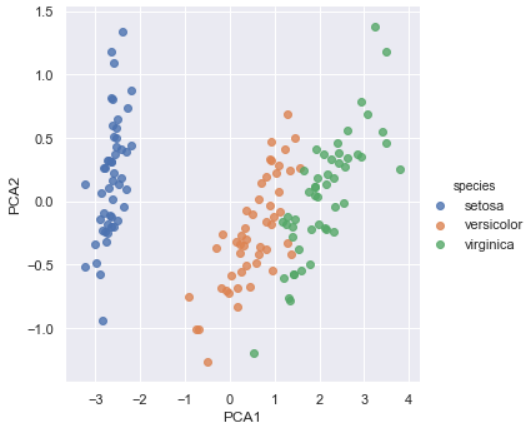
```
In [61]: from sklearn.decomposition import PCA  # 1. Choose the model class
         model = PCA(n_components=2)           # 2. Instantiate the model with hyperparameters
         model.fit(X_iris)                     # 3. Fit to data. Notice y is not specified!
         X_2D = model.transform(X_iris)        # 4. Transform the data to two dimensions

In [62]: X_2D.shape

Out[62]: (150, 2)
```

PCA: Principal component visualize results

Now let's plot the results. A quick way to do this is to insert the results into the original Iris DataFrame, and use Seaborn's lmplot to show the results:

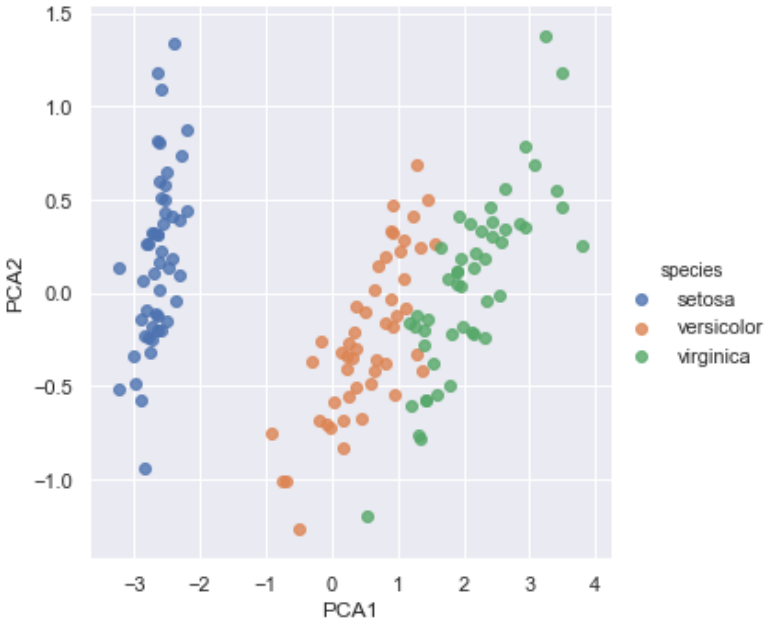
```
iris['PCA1'] = X_2D[:, 0]
iris['PCA2'] = X_2D[:, 1]
sns.lmplot("PCA1", "PCA2", hue='species', data=iris, fit_reg=False);
```

```
In [65]: iris['PCA1'] = X_2D[:, 0]
         iris['PCA2'] = X_2D[:, 1]
         sns.lmplot("PCA1", "PCA2", hue='species', data=iris, fit_reg=False);
```



PCA: Principal component visualize results

```
In [65]: iris['PCA1'] = X_2D[:, 0]
         iris['PCA2'] = X_2D[:, 1]
         sns.lmplot("PCA1", "PCA2", hue='species', data=iris, fit_reg=False);
```

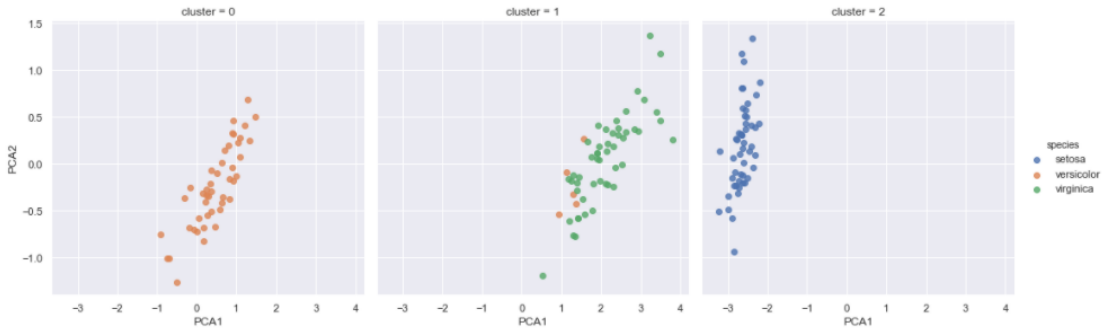


The species are fairly well separated, even though the PCA algorithm had no knowledge of the species labels! This indicates to us that a relatively straightforward classification will probably be effective on the dataset

Unsupervised Iris Custering GMM: Gaussian Mixture Models

In [70]:
```
from sklearn import mixture      # 1. Choose the model class
model = mixture.GaussianMixture(n_components=3,
            covariance_type='full')  # 2. Instantiate the model with hyperparameters
model.fit(X_iris)                    # 3. Fit to data. Notice y is not specified!
y_gmm = model.predict(X_iris)
```

In [71]:
```
iris['cluster'] = y_gmm
sns.lmplot("PCA1", "PCA2", data=iris, hue='species',
           col='cluster', fit_reg=False);
```



Luís Garmendia

GMM: Gaussian Mixture Models

The non-probabilistic nature of *k*-means and its use of simple distance-from-cluster-center to assign cluster membership leads to poor performance for many real-world situations. In this section we will take a look at Gaussian mixture models (GMMs), which can be viewed as an extension of the ideas behind *k*-means, but can also be a powerful tool for estimation beyond simple clustering.

Given simple, well-separated data, *k*-means finds suitable clustering results.

GMM: Gaussian Mixture Models

```
from sklearn import mixture
model = mixture.GaussianMixture(n_components=3, covariance_type='full')
model.fit(X_iris)
y_gmm = model.predict(X_iris)
```

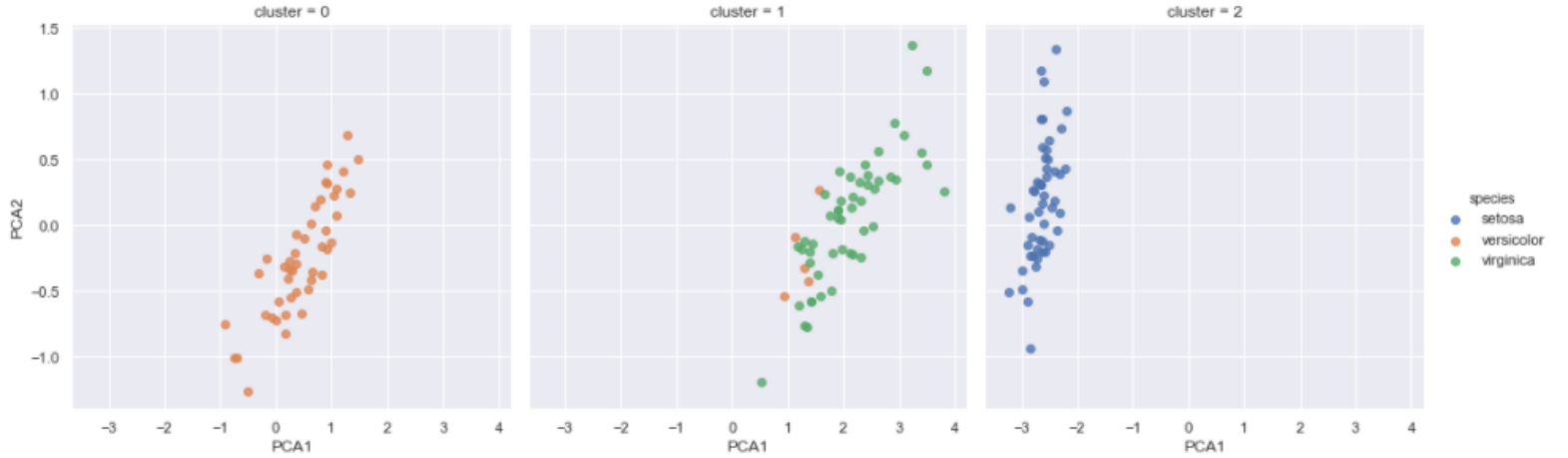
As before, we will add the cluster label to the Iris DataFrame and use Seaborn to plot the results:

```
iris['cluster'] = y_gmm
sns.lmplot("PCA1", "PCA2", data=iris, hue='species', col='cluster',
fit_reg=False);
```

GMM: Gaussian Mixture Models

In [70]:
```
from sklearn import mixture      # 1. Choose the model class
model = mixture.GaussianMixture(n_components=3,
            covariance_type='full')  # 2. Instantiate the model with hyperparameters
model.fit(X_iris)                    # 3. Fit to data. Notice y is not specified!
y_gmm = model.predict(X_iris)
```

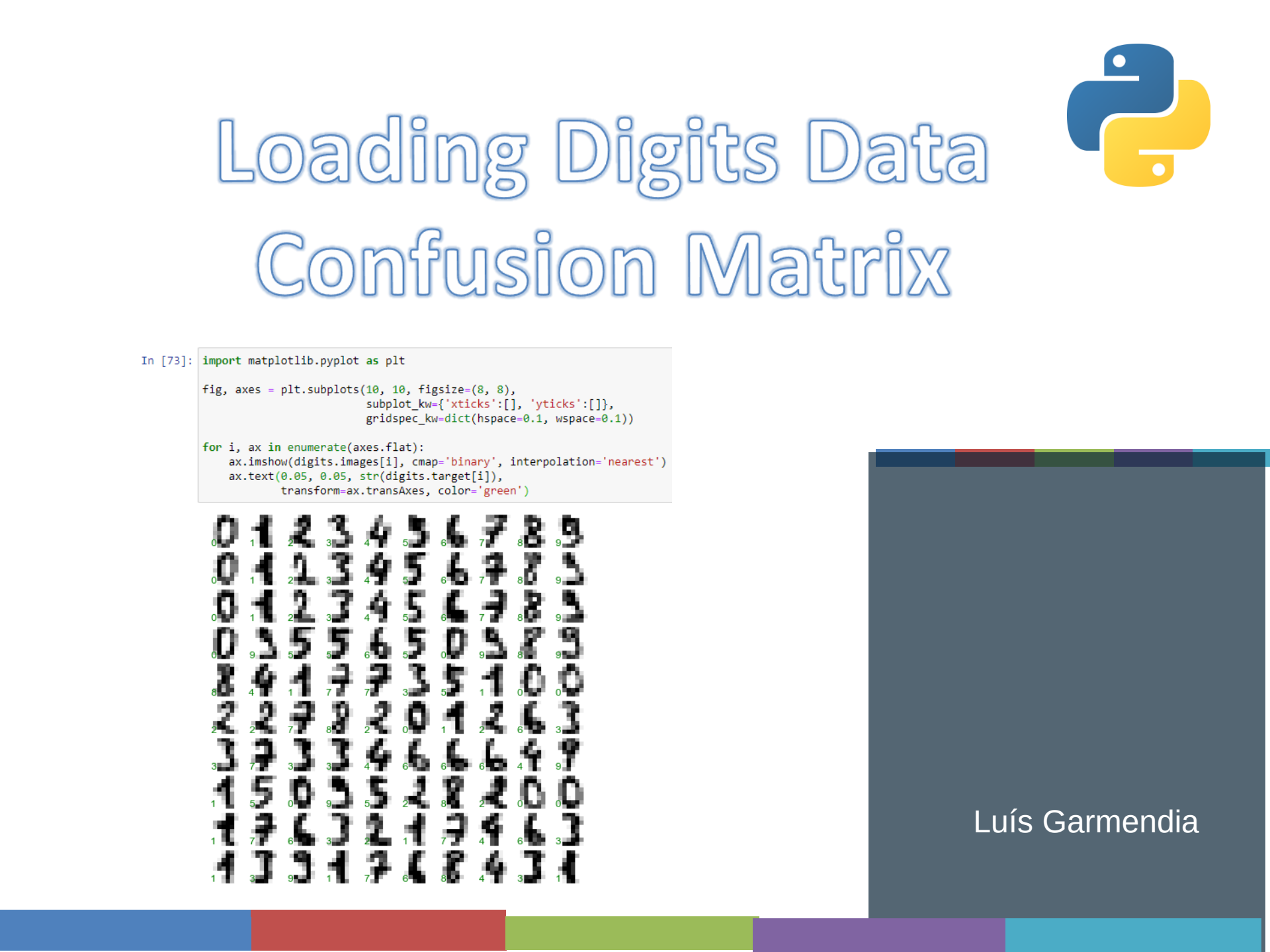
In [71]:
```
iris['cluster'] = y_gmm
sns.lmplot("PCA1", "PCA2", data=iris, hue='species',
           col='cluster', fit_reg=False);
```



GMM: Gaussian Mixture Models

By splitting the data by cluster number, we see exactly how well the GMM algorithm has recovered the underlying label: the *setosa* species is separated perfectly within cluster 0, while there remains a small amount of mixing between *versicolor* and *virginica*.
This means that even without an expert to tell us the species labels of the individual flowers, the measurements of these flowers are distinct enough that we could *automatically* identify the presence of these different groups of species with a simple clustering algorithm!

Loading Digits Data
Confusion Matrix



In [73]:
```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
    ax.text(0.05, 0.05, str(digits.target[i]),
            transform=ax.transAxes, color='green')
```


Luís Garmendia

Loading digits data

The images data is a three-dimensional array: 1,797 samples each consisting of an 8 × 8 grid of pixels.

from sklearn.datasets import load_digits

digits = load_digits()

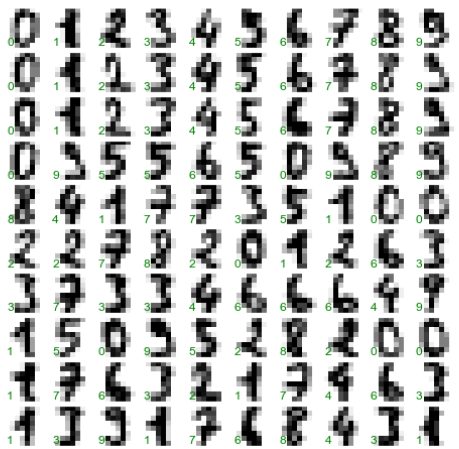
digits.images.shape

```
In [72]: from sklearn.datasets import load_digits
         digits = load_digits()
         digits.images.shape

Out[72]: (1797, 8, 8)
```

Loading digits data

```
import matplotlib.pyplot as plt
fig, axes = plt.subplots(10, 10, figsize=(8, 8),
    subplot_kw={'xticks':[], 'yticks':[]}, gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i, ax in enumerate(axes.flat):
   ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
   ax.text(0.05, 0.05, str(digits.target[i]), transform=ax.transAxes, color='green')
```

```
In [73]: import matplotlib.pyplot as plt

         fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                                  subplot_kw={'xticks':[], 'yticks':[]},
                                  gridspec_kw=dict(hspace=0.1, wspace=0.1))

         for i, ax in enumerate(axes.flat):
             ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
             ax.text(0.05, 0.05, str(digits.target[i]),
                     transform=ax.transAxes, color='green')
```



[n_samples, n_features] representation

In order to work with this data within Scikit-Learn, we need a two-dimensional, . We can accomplish this by treating each pixel in the image as a feature: that is, by flattening out the pixel arrays so that we have a length-64 array of pixel values representing each digit. Additionally, we need the target array, which gives the previously determined label for each digit. These two quantities are built into the digits dataset under the data and target attributes, respectively:

```
X = digits.data
X.shape
y = digits.target
y.shape
```

```
In [74]: X = digits.data
         X.shape

Out[74]: (1797, 64)

In [75]: y = digits.target
         y.shape

Out[75]: (1797,)
```

Dimensional reduction

We'd like to visualize our points within the 64-dimensional parameter space, but it's difficult to effectively visualize points in such a high-dimensional space. Instead we'll reduce the dimensions to 2, using an unsupervised method. Here, we'll make use of a manifold learning algorithm called *Isomap*

X = digits.data

X.shape

y = digits.target

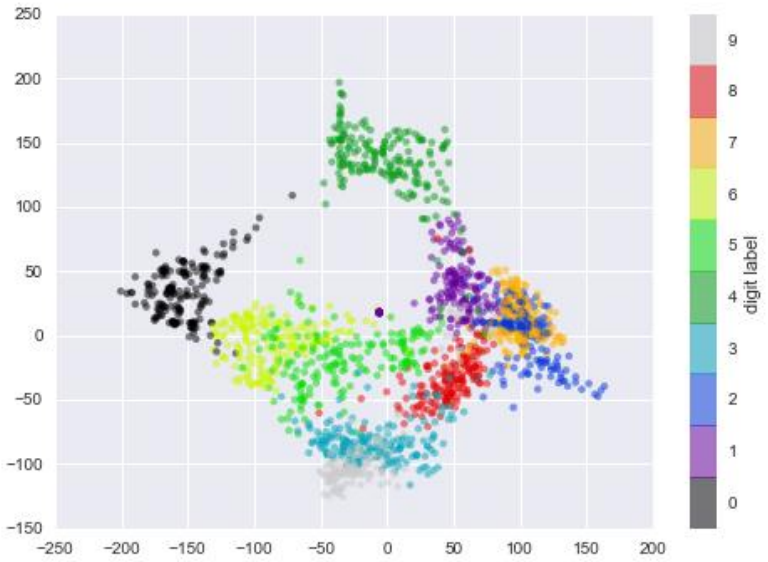
y.shape

```
In [76]: from sklearn.manifold import Isomap
         iso = Isomap(n_components=2)
         iso.fit(digits.data)
         data_projected = iso.transform(digits.data)
         data_projected.shape

Out[76]: (1797, 2)
```

View Dimensional reduction

```
plt.scatter(data_projected[:, 0], data_projected[:, 1], c=digits.target, edgecolor='none',
alpha=0.5, cmap=plt.cm.get_cmap('rainbow', 10))
plt.colorbar(label='digit label', ticks=range(10))
plt.clim(-0.5, 9.5);
```

```
In [27]: plt.scatter(data_projected[:, 0], data_projected[:, 1], c=digits.target,
                     edgecolor='none', alpha=0.5,
                     cmap=plt.cm.get_cmap('spectral', 10))
         plt.colorbar(label='digit label', ticks=range(10))
         plt.clim(-0.5, 9.5);
```



View Dimensional reduction

This plot gives us some good intuition into how well various numbers are separated in the larger 64-dimensional space

Overall, however, the different groups appear to be fairly well separated in the parameter space: this tells us that even a very straightforward supervised classification algorithm should perform suitably on this data.

Clasify digits with Naïve Bayes

```
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)

from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(Xtrain, ytrain)
y_model = model.predict(Xtest)

from sklearn.metrics import accuracy_score
accuracy_score(ytest, y_model)
```

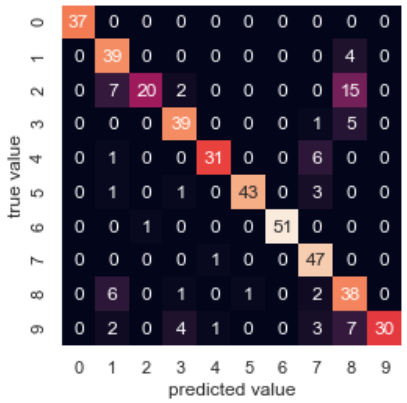
```
In [80]: Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)

In [81]: from sklearn.naive_bayes import GaussianNB
         model = GaussianNB()
         model.fit(Xtrain, ytrain)
         y_model = model.predict(Xtest)

In [82]: from sklearn.metrics import accuracy_score
         accuracy_score(ytest, y_model)

Out[82]: 0.8333333333333334
```

Confusion Matrix

With even this extremely simple model, we find about 80% accuracy for classification of the digits! However, this single number doesn't tell us *where* we've gone wrong—one nice way to do this is to use the *confusion matrix*

```
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(ytest, y_model)
sns.heatmap(mat, square=True, annot=True, cbar=False)
plt.xlabel('predicted value')
plt.ylabel('true value');
```

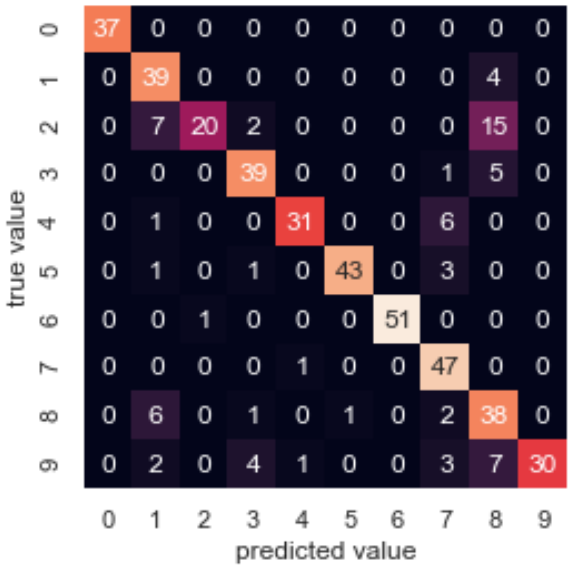
```
In [83]: from sklearn.metrics import confusion_matrix

         mat = confusion_matrix(ytest, y_model)

         sns.heatmap(mat, square=True, annot=True, cbar=False)
         plt.xlabel('predicted value')
         plt.ylabel('true value');
```



Confusion Matrix

This shows us where the mis-labeled points tend to be: for example, a large number of twos here are mis-classified as either ones or eights. Another way to gain intuition into the characteristics of the model is to plot the inputs again, with their predicted labels.

In [83]:
```
from sklearn.metrics import confusion_matrix

mat = confusion_matrix(ytest, y_model)

sns.heatmap(mat, square=True, annot=True, cbar=False)
plt.xlabel('predicted value')
plt.ylabel('true value');
```



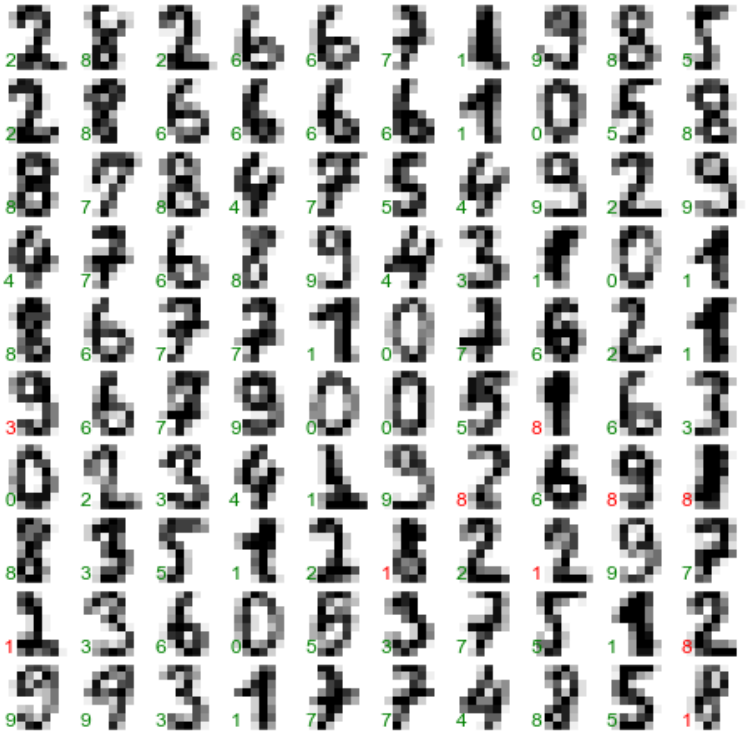
Confusion Matrix

We'll use green for correct labels, and red for incorrect labels:

```
fig, axes = plt.subplots(10, 10, figsize=(8, 8), subplot_kw={'xticks':[], 'yticks':[]},
gridspec_kw=dict(hspace=0.1, wspace=0.1))

test_images = Xtest.reshape(-1, 8, 8)

for i, ax in enumerate(axes.flat):
    ax.imshow(test_images[i], cmap='binary', interpolation='nearest')
    ax.text(0.05, 0.05, str(y_model[i]),
            transform=ax.transAxes,
            color='green' if (ytest[i] == y_model[i]) else 'red')
```

Confusion Matrix

In [84]:
```
fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))

test_images = Xtest.reshape(-1, 8, 8)

for i, ax in enumerate(axes.flat):
    ax.imshow(test_images[i], cmap='binary', interpolation='nearest')
    ax.text(0.05, 0.05, str(y_model[i]),
            transform=ax.transAxes,
            color='green' if (ytest[i] == y_model[i]) else 'red')
```

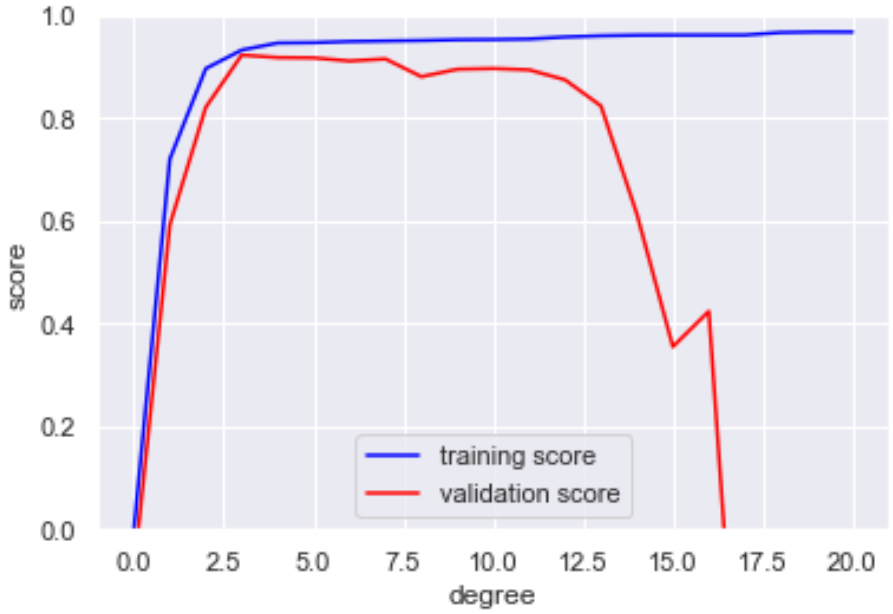


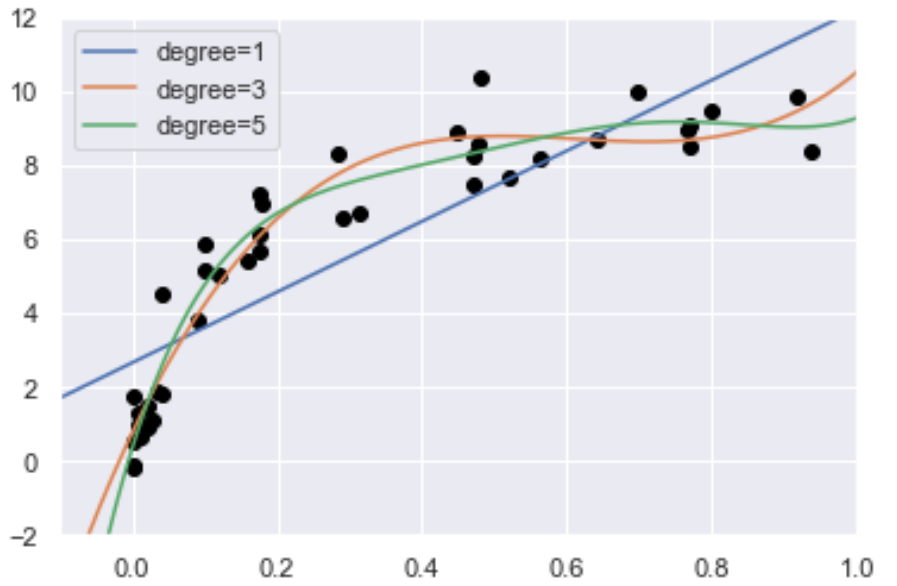
Conclusion

To go beyond our 80% classification rate, we might move to a more sophisticated algorithm such as support vector machines, random forests or another classification approach.

NEXT: how to select and validate your model

Model Validation





Luís Garmendia

Model validation

- In principle, model validation is very simple: after choosing a model and its hyperparameters, we can estimate how effective it is by applying it to some of the training data and comparing the prediction to the known value.
- But sometimes it fails

Wrong model validation

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=1)
model.fit(X, y)
y_model = model.predict(X)
from sklearn.metrics import accuracy_score
accuracy_score(y, y_model)
```

1.0

Model_selection.train_test_split

holdout set: we hold back some subset of the data from the training of the model, and then use this holdout set to check the model performance.

This splitting can be done using the train_test_split:

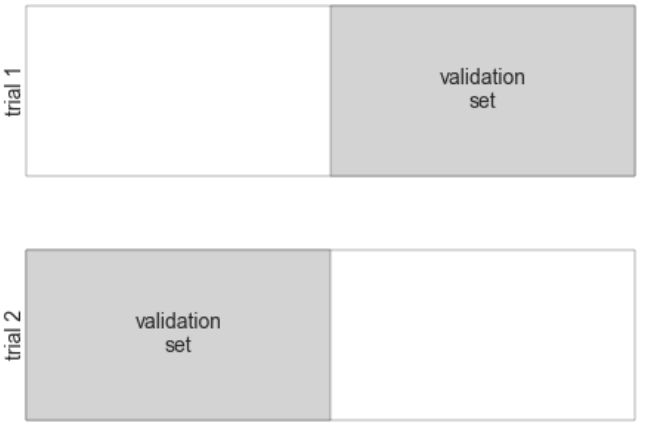
```
from sklearn.model_selection import train_test_split
X1, X2, y1, y2 = train_test_split(X, y, random_state=0, train_size=0.5)
model.fit(X1, y1)
y2_model = model.predict(X2)
accuracy_score(y2, y2_model)
```

0.9066666666666662

Cross-validation model validation

One disadvantage of using a holdout set for model validation is that we have lost a portion of our data to the model training.

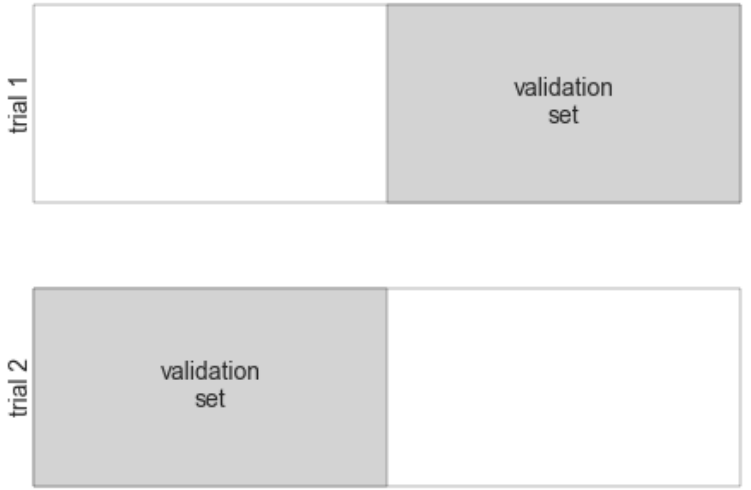
One way to address this is to use *cross-validation*; that is, to do a sequence of fits where each subset of the data is used both as a training set and as a validation set.



2 validation trials

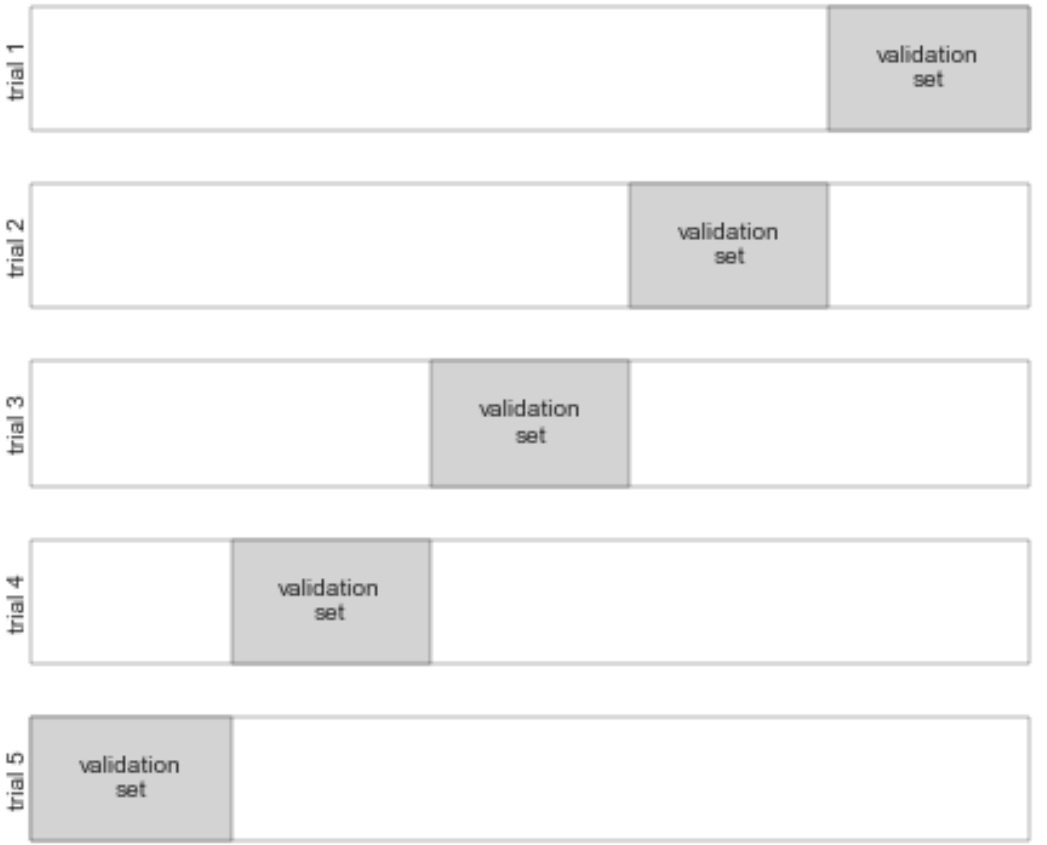
```
y2_model = model.fit(X1, y1).predict(X2)
y1_model = model.fit(X2, y2).predict(X1)
accuracy_score(y1, y1_model), accuracy_score(y2, y2_model)
```

(0.9599999999999996, 0.9666666666666662)



many validation trials

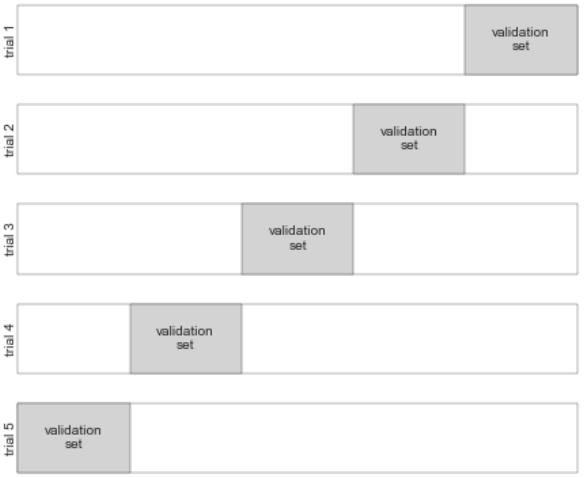
five-fold cross-validation:



many validation trials cross_val_score

five-fold cross-validation:

```
from sklearn.model_selection import cross_val_score
cross_val_score(model, X, y, cv=5)
array([ 0.96666667, 0.96666667, 0.93333333, 0.93333333, 1. ])
```



Leave one out validation trial

The extreme case in which our number of folds is equal to the number of data points: that is, we train on all points but one in each trial.

```
from sklearn.model_selection import LeaveOneOut
scores = cross_val_score(model, X, y, cv=LeaveOneOut())
scores
```

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

Leave one out validation trial

Because we have 150 samples, the leave one out cross-validation yields scores for 150 trials, and the score indicates either successful (1.0) or unsuccessful (0.0) prediction. Taking the mean of these gives an estimate of the error rate:

```
scores.mean()
```

0.9599999999999996

Selecting the best model

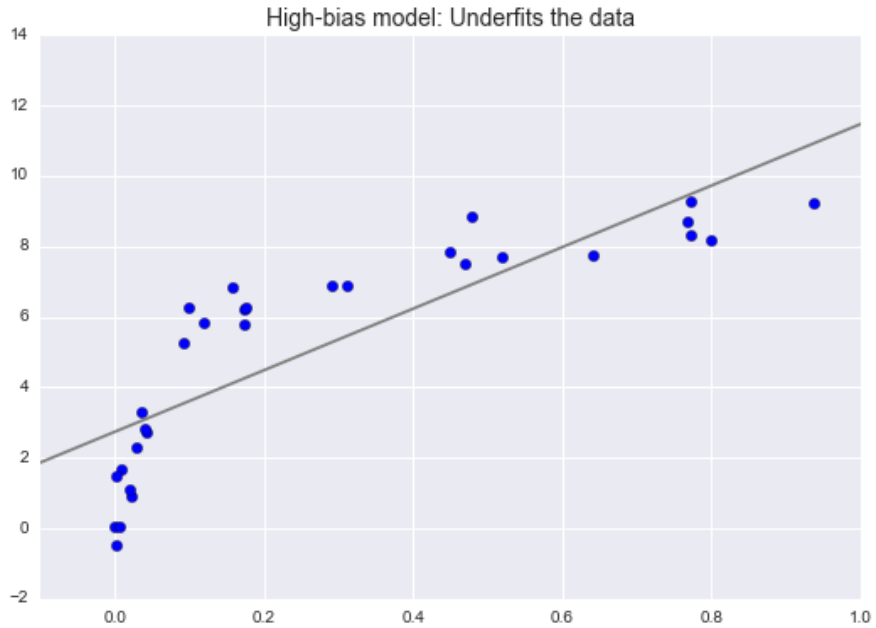
Model selection and selection of **hyperparameters** are some of the most important aspects of the practice of machine learning.

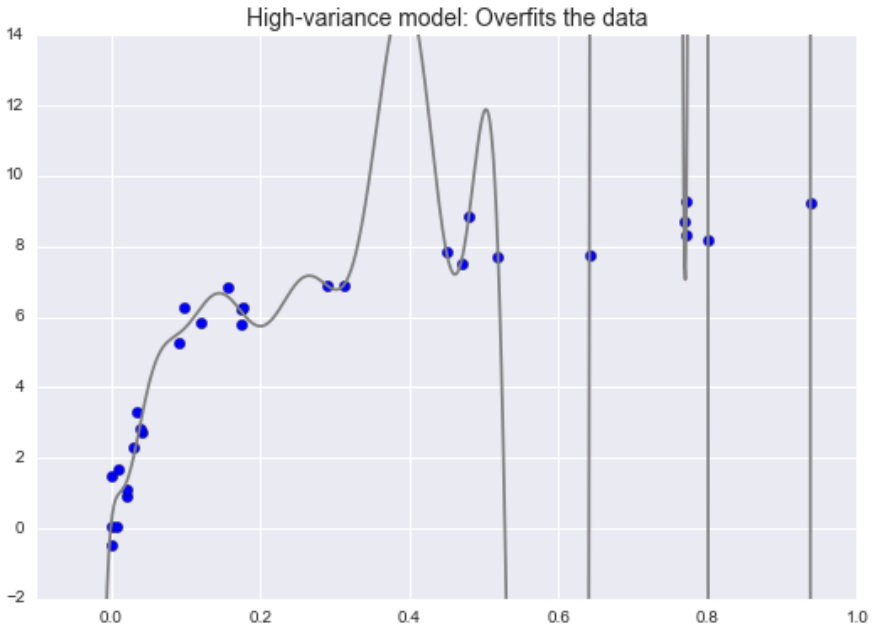
Of core importance is the following question: *if our estimator is underperforming, how should we move forward?* There are several possible answers:

- Use a more complicated/more flexible model
- Use a less complicated/less flexible model
- Gather more training samples
- Gather more data to add features to each sample

The bias-variance trade-of

Two regression fits to the same dataset:

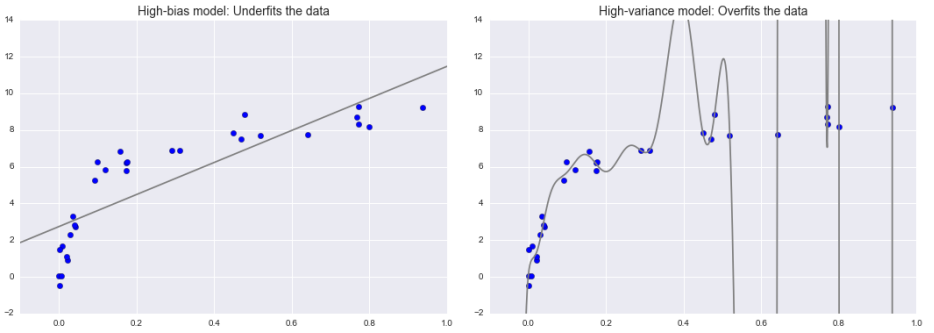




The bias-variance trade-of

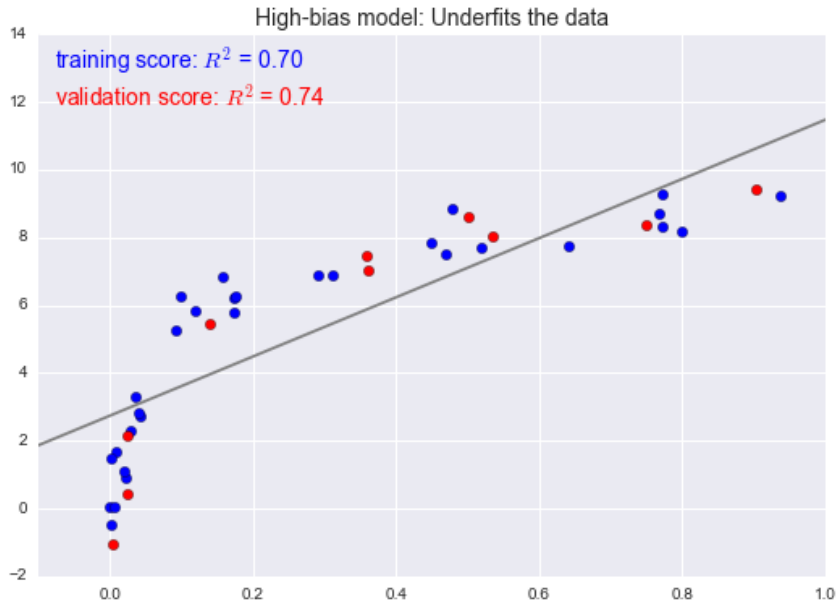
The straight-line model will never be able to describe this dataset well. Such a model is said to ***underfit*** the data: that is, it does not have enough model flexibility to suitably account for all the features in the data; another way of saying this is that the model has **high *bias***.

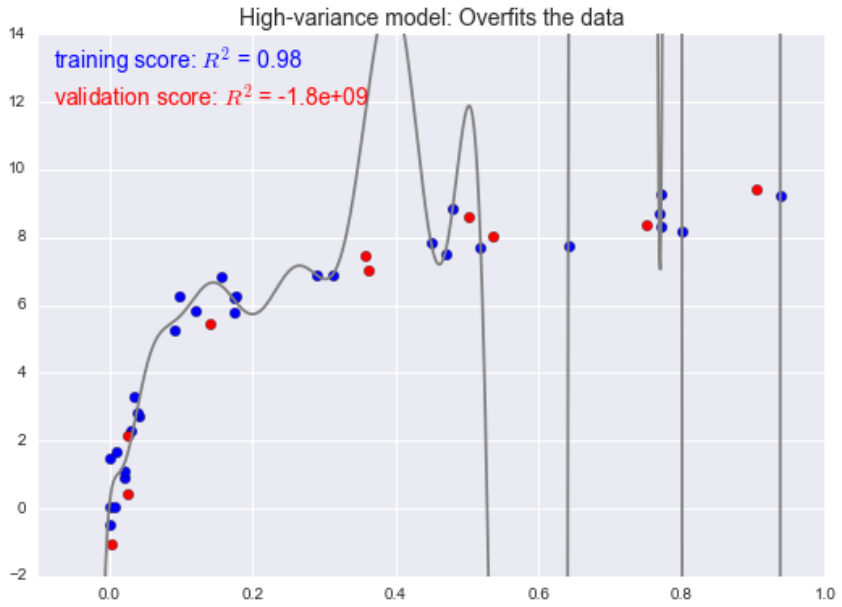
The model on the right attempts to fit a high-order polynomial through the data. Here the model fit has enough flexibility to nearly perfectly account for the fine features in the data, but even though it very accurately describes the training data, its precise form seems to be more reflective of the particular noise properties of the data rather than the intrinsic properties of whatever process generated that data. Such a model is said to ***overfit*** the data



The bias-variance trade-of

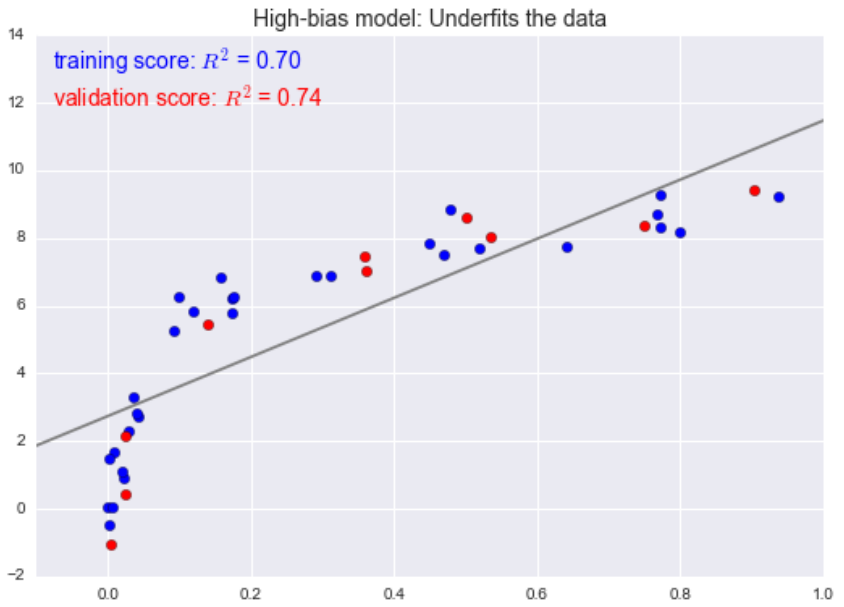
What happens if we use these two models to predict the y-value for some new data. In the following diagrams, the red/lighter points indicate data that is omitted from the training set

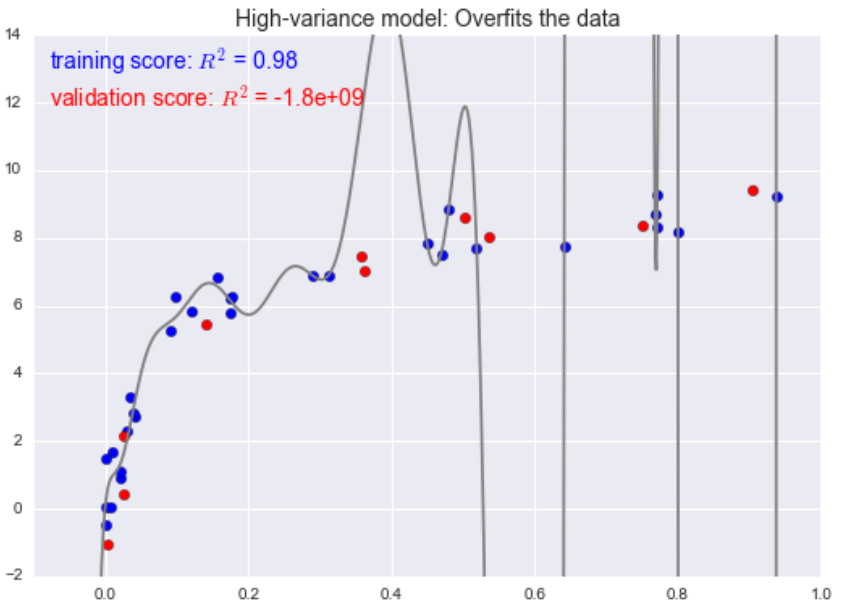




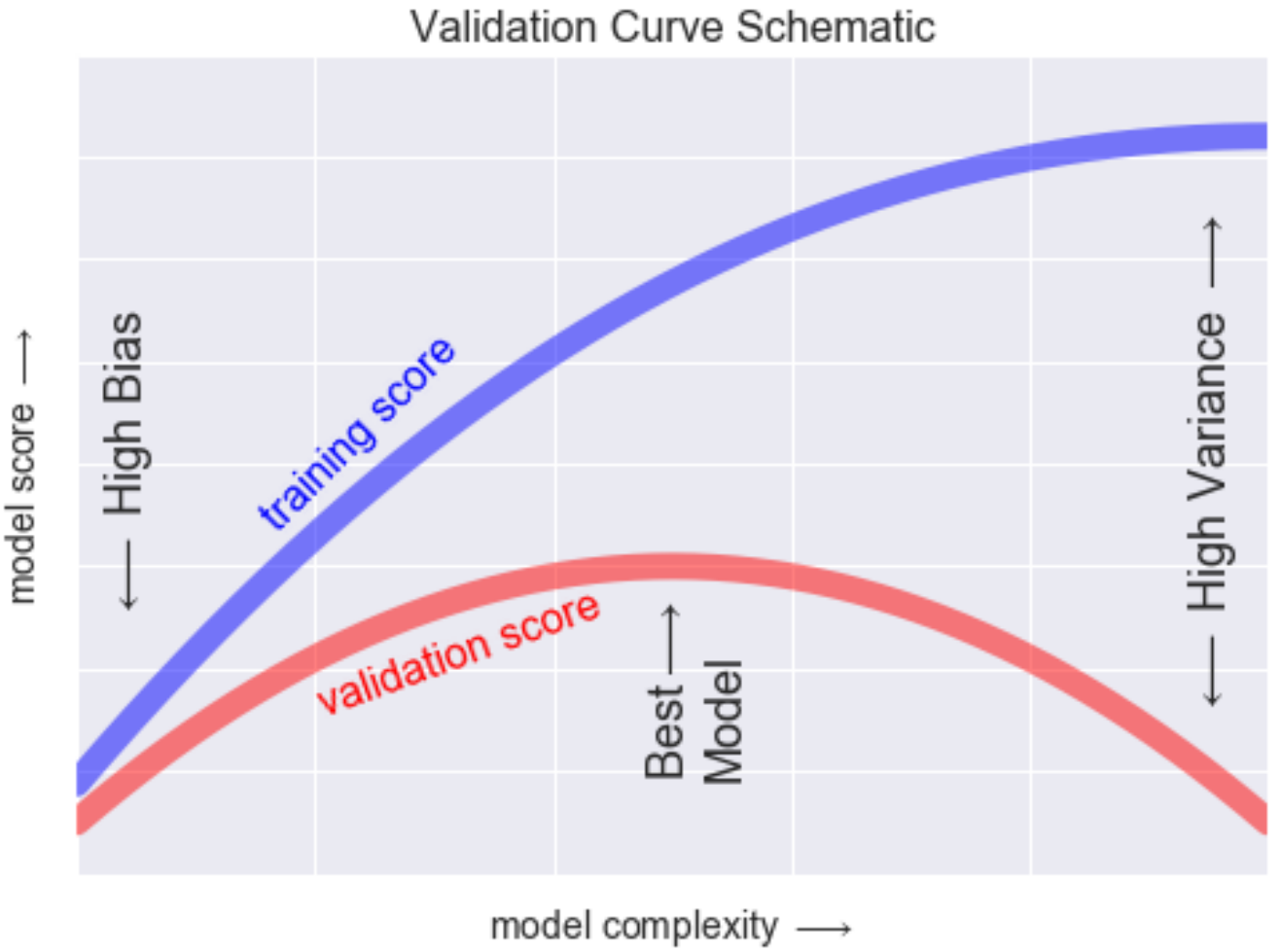
The bias-variance trade-of

- For high-bias models, the performance of the model on the validation set is similar to the performance on the training set.
- For high-variance models, the performance of the model on the validation set is far worse than the performance on the training set.





Validation curve



Validation curve

- The training score is everywhere higher than the validation score. This is generally the case: the model will be a better fit to data it has seen than to data it has not seen.
- For very low model complexity (a high-bias model), the training data is under-fit, which means that the model is a poor predictor both for the training data and for any previously unseen data.
- For very high model complexity (a high-variance model), the training data is over-fit, which means that the model predicts the training data very well, but fails for any previously unseen data.
- For some intermediate value, the validation curve has a maximum. This level of complexity indicates a suitable trade-off between bias and variance.

Coefficient of determination

The score here is the R^2 score, or coefficient of determination, which measures how well a model performs relative to a simple mean of the target values.

- $R^2 = 1$ indicates a perfect match,
- $R^2 = 0$ indicates the model does no better than simply taking the mean of the data
- negative values mean even worse models.

Polinomial regression model

Linear $y = ax+b$

A degree-3 polynomial $y = ax^3+bx^2+cx+d$

…

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
def PolynomialRegression(degree=2, **kwargs):
    return make_pipeline(PolynomialFeatures(degree),
                         LinearRegression(**kwargs))
```

Polinomial regression model

Create some data

```
import numpy as np
def make_data(N, err=1.0, rseed=1):
        # randomly sample the data
        rng = np.random.RandomState(rseed)
        X = rng.rand(N, 1) ** 2
        y = 10 - 1. / (X.ravel() + 0.1)
        if err > 0:
            y += err * rng.randn(N)
        return X, y
X, y = make_data(40)
```

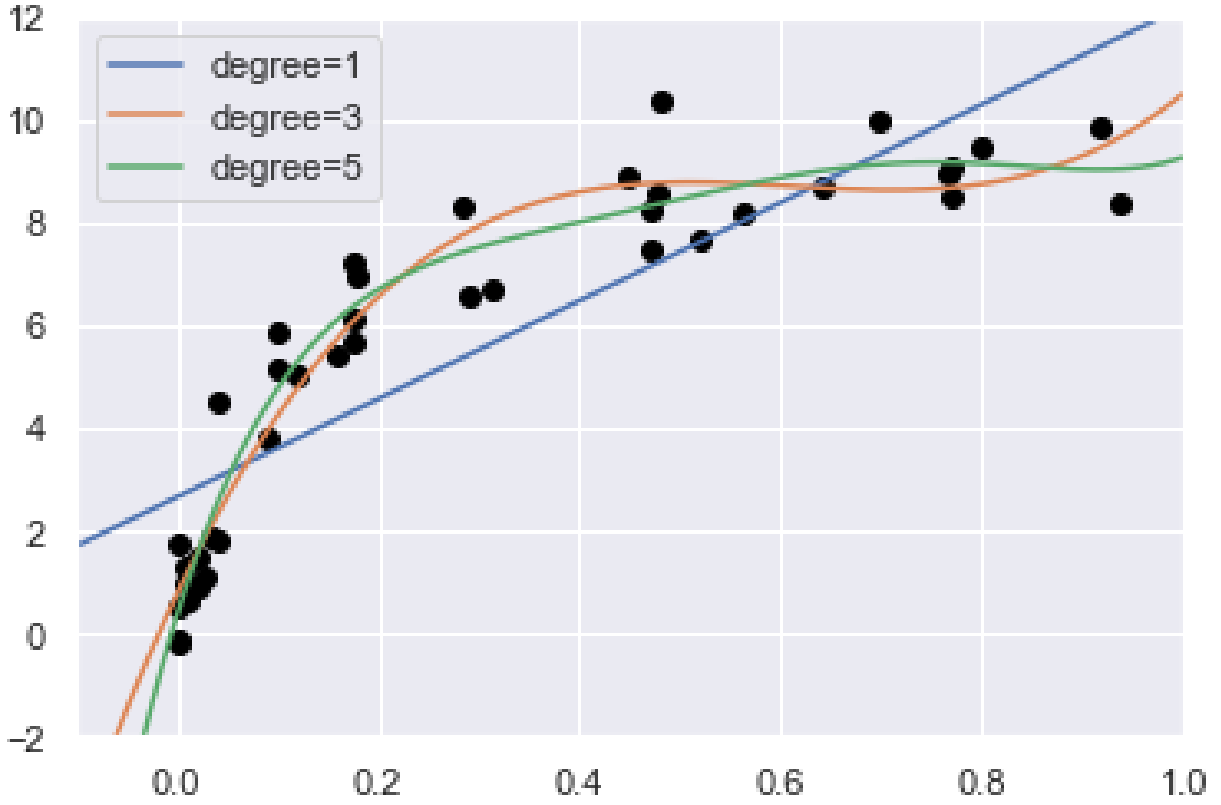
Polinomial regression model

Visualize our data, along with polynomial fits of several degrees

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # plot formatting
X_test = np.linspace(-0.1, 1.1, 500)[:, None]
plt.scatter(X.ravel(), y, color='black')
axis = plt.axis()
for degree in [1, 3, 5]:
    y_test = PolynomialRegression(degree).fit(X, y).predict(X_test)
    plt.plot(X_test.ravel(), y_test, label='degree={0}'.format(degree))
plt.xlim(-0.1, 1.0)
plt.ylim(-2, 12)
plt.legend(loc='best');
```

Polinomial regression model

Visualize our data, along with polynomial fits of several degrees



Validation curve

A useful question to answer is this: what degree of polynomial provides a suitable trade-off between bias (under-fitting) and variance (over-fitting)

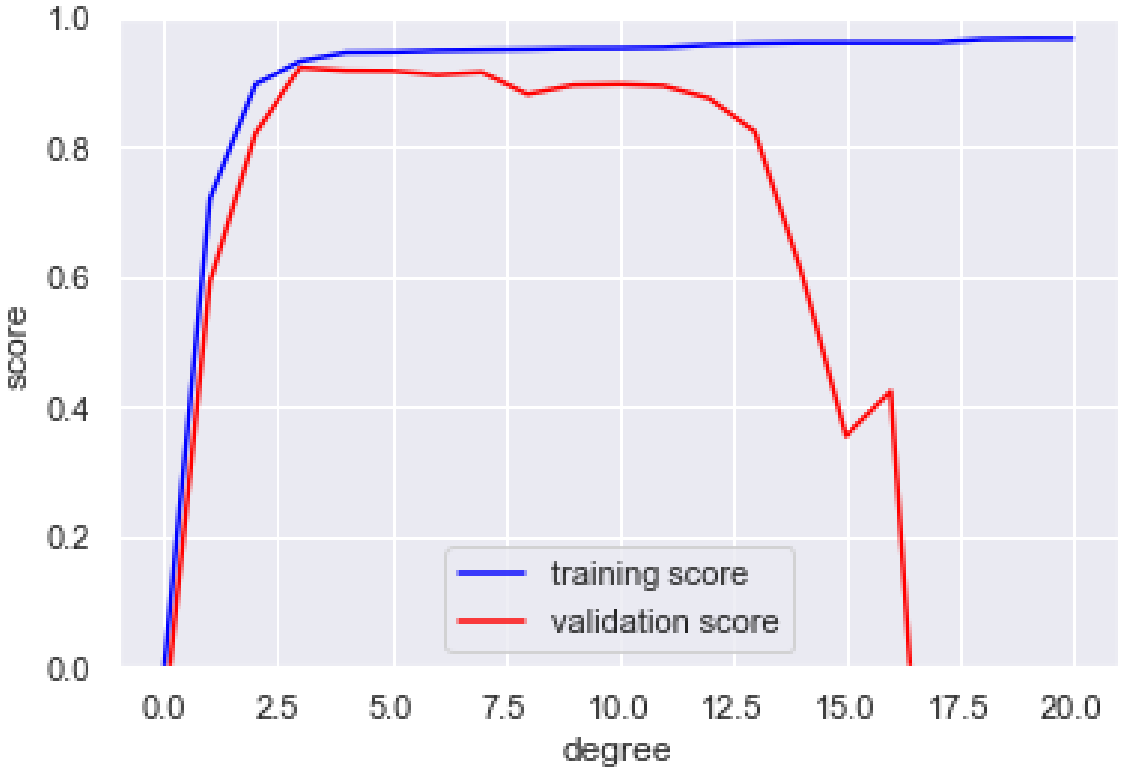
This can be done straightforwardly using the validation_curve

```
from sklearn.model_selection import validation_curve
degree = np.arange(0, 21)
train_score, val_score = validation_curve(PolynomialRegression(), X, y,
    'polynomialfeatures__degree', degree, cv=7)
plt.plot(degree, np.median(train_score, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score, 1), color='red', label='validation score')
plt.legend(loc='best')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

Validation curve

A useful question to answer is this: what degree of polynomial provides a suitable trade-off between bias (under-fitting) and variance (over-fitting)

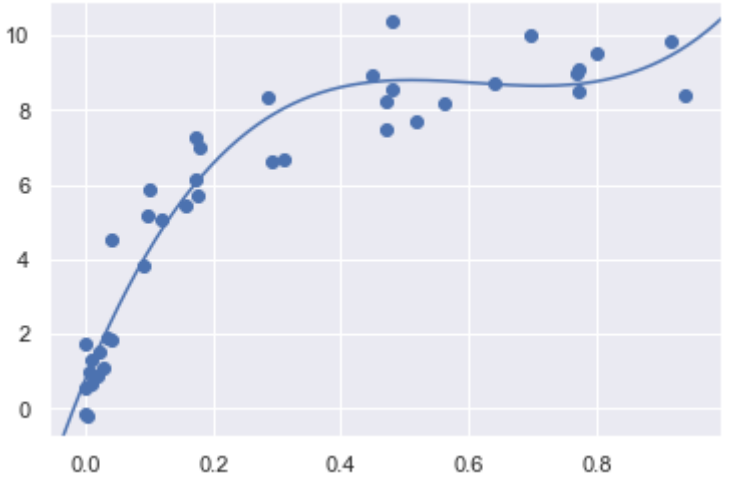
This can be done straightforwardly using the validation_curve



Validation curve

We can read-off that the optimal trade-off between bias and variance is found for a third-order polynomial; we can compute and display this fit over the original data as follows

```
plt.scatter(X.ravel(), y)
lim = plt.axis()
y_test = PolynomialRegression(3).fit(X, y).predict(X_test)
plt.plot(X_test.ravel(), y_test);
plt.axis(lim);
```



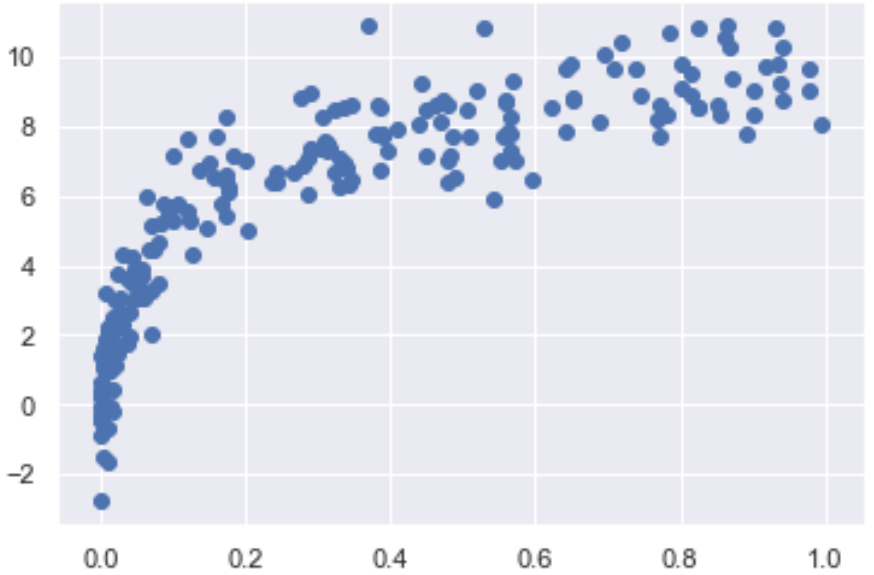
Size of the training data

The optimal model will generally depend on the size of your training data.

X2, y2 = make_data(200)

plt.scatter(X2.ravel(), y2);

```
X2, y2 = make_data(200)
plt.scatter(X2.ravel(), y2);
```



Size of the training data

We will duplicate the preceding code to plot the validation curve for this larger dataset;

```
degree = np.arange(21)
train_score2, val_score2 = validation_curve(PolynomialRegression(), X2, y2,
'polynomialfeatures__degree', degree, cv=7)
plt.plot(degree, np.median(train_score2, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score2, 1), color='red', label='validation score')
plt.plot(degree, np.median(train_score, 1), color='blue', alpha=0.3,
                    linestyle='dashed')
plt.plot(degree, np.median(val_score, 1), color='red', alpha=0.3,
linestyle='dashed')
plt.legend(loc='lower center')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

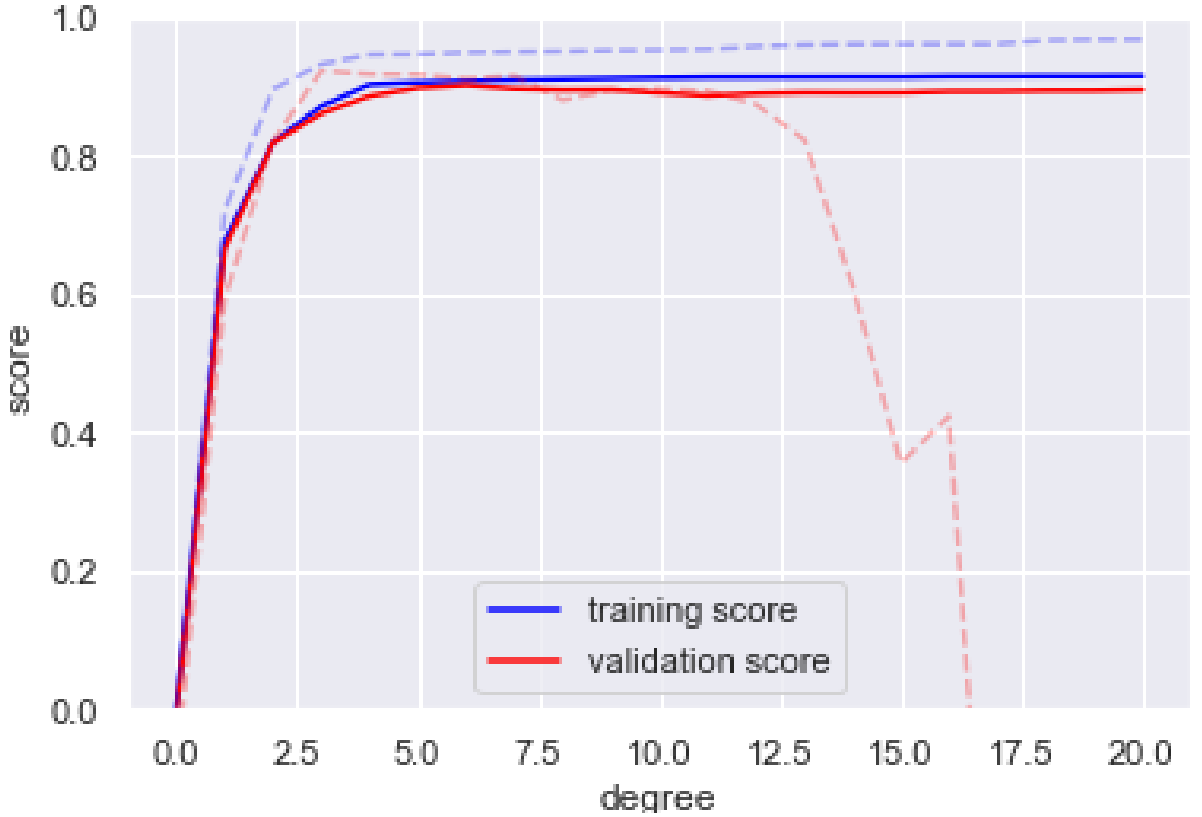
Size of the training data

We will duplicate the preceding code to plot the validation curve for this larger dataset

```
degree = np.arange(21)
train_score2, val_score2 = validation_curve(PolynomialRegression(), X2, y2,
'polynomialfeatures__degree', degree, cv=7)
plt.plot(degree, np.median(train_score2, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score2, 1), color='red', label='validation score')
plt.plot(degree, np.median(train_score, 1), color='blue', alpha=0.3,
linestyle='dashed')
plt.plot(degree, np.median(val_score, 1), color='red', alpha=0.3, linestyle='dashed')
plt.legend(loc='lower center')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

Size of the training data

We will duplicate the preceding code to plot the validation curve for this larger dataset;



Training curve behaviour

Thus we see that the behavior of the validation curve has not one but two important inputs:

- the model complexity
- the number of training points.

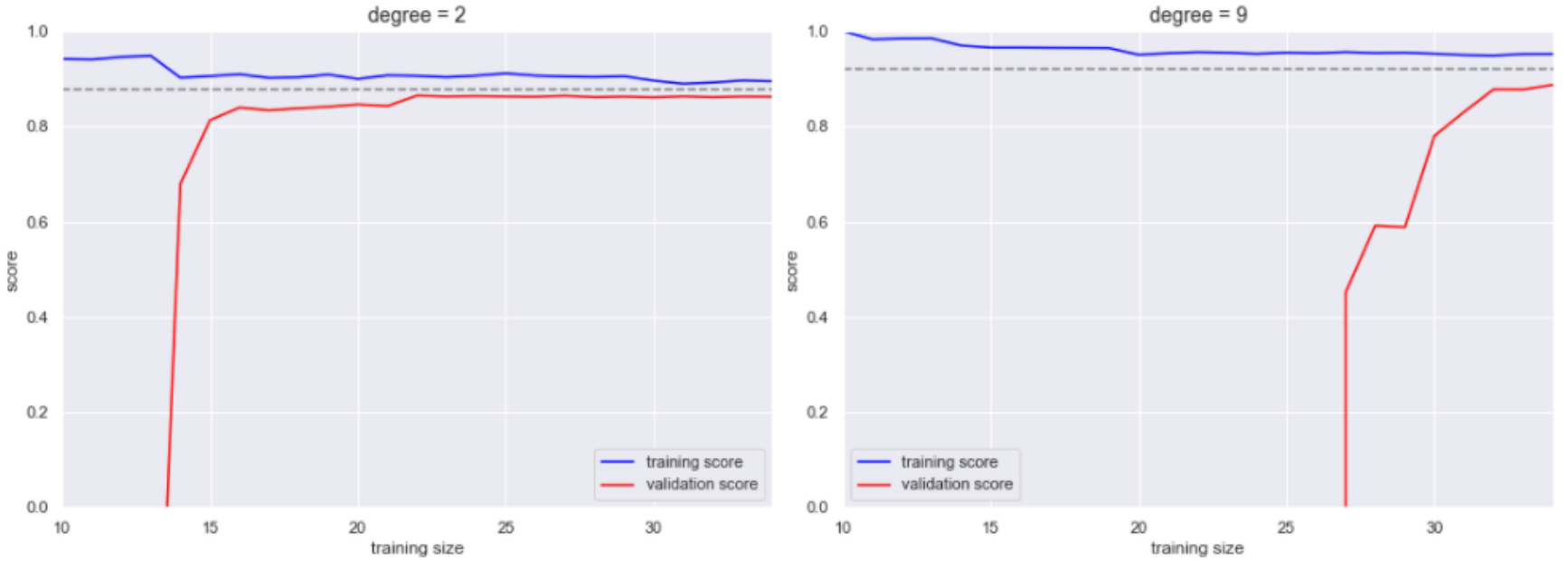
Training curve behaviour

The general behavior we would expect from a learning curve is this:

- A model of a given complexity will *overfit* a small dataset: this means the training score will be relatively high, while the validation score will be relatively low.
- A model of a given complexity will *underfit* a large dataset: this means that the training score will decrease, but the validation score will increase.
- A model will never, except by chance, give a better score to the validation set than the training set: this means the curves should keep getting closer together but never cross.

Adding more training data will not significantly improve the fit

learning curve for our original dataset with a second-order polynomial model and a ninth-order polynomial



Adding more training data will not significantly improve the fit

learning curve for our original dataset with a second-order polynomial model and a ninth-order polynomial

```
from sklearn.model_selection import learning_curve
fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
for i, degree in enumerate([2, 9]):
        N, train_lc, val_lc = learning_curve(PolynomialRegression(degree), X, y, cv=7,
train_sizes=np.linspace(0.3, 1, 25))
        ax[i].plot(N, np.mean(train_lc, 1), color='blue', label='training score')
        ax[i].plot(N, np.mean(val_lc, 1), color='red', label='validation score')
        ax[i].hlines(np.mean([train_lc[-1], val_lc[-1]]), N[0], N[-1],  color='gray', linestyle='dashed')
        ax[i].set_ylim(0, 1)
        ax[i].set_xlim(N[0], N[-1])
        ax[i].set_xlabel('training size')
        ax[i].set_ylabel('score')
        ax[i].set_title('degree = {0}'.format(degree), size=14)
        ax[i].legend(loc='best')
```

Training curve behaviour

The general behavior we would expect from a learning curve is this:

- A model of a given complexity will *overfit* a small dataset: this means the training score will be relatively high, while the validation score will be relatively low.
- A model of a given complexity will *underfit* a large dataset: this means that the training score will decrease, but the validation score will increase.
- A model will never, except by chance, give a better score to the validation set than the training set: this means the curves should keep getting closer together but never cross.

Maximize score automated tools Grid Search

Here is an example of using grid search to find the optimal polynomial model. We will explore a three-dimensional grid of model features; namely the polynomial degree, the flag telling us whether to fit the intercept, and the flag telling us whether to normalize the problem. This can be set up using Scikit-Learn's GridSearchCV meta-estimator

```
from sklearn.model_selection import GridSearchCV
param_grid = {'polynomialfeatures__degree':
      np.arange(21),'linearregression__fit_intercept': [True, False],
      'linearregression__normalize':      [True, False]}
grid = GridSearchCV(PolynomialRegression(), param_grid, cv=7)
grid.fit(X, y);
grid.get_params
{'linearregression__fit_intercept': False,
'linearregression__normalize': True,
'polynomialfeatures__degree': 4}
```

Feature Engineering
Vectorization

Luís Garmendia

Feature Engineering

Feature engineering: that is, taking whatever information you have about your problem and turning it into numbers that you can use to build your feature matrix.

In this section, we will cover a few common examples of feature engineering tasks: features for representing ***categorical data***, features for representing ***text***, and features for representing ***images***.

Additionally, we will discuss *derived features* for **increasing model complexity** and *imputation* of missing data. Often this process is known as ***vectorization***, as it involves converting arbitrary data into well-behaved vectors.

Categorical Features

One common type of non-numerical data is *categorical* data. For example, imagine you are exploring some data on housing prices, and along with numerical features like "price" and "rooms", you also have "neighborhood" information.

```
data = [
    {'price': 850000, 'rooms': 4, 'neighborhood': 'Queen Anne'},
    {'price': 700000, 'rooms': 3, 'neighborhood': 'Fremont'},
    {'price': 650000, 'rooms': 3, 'neighborhood': 'Wallingford'},
    {'price': 600000, 'rooms': 2, 'neighborhood': 'Fremont'} ]
```

You might be tempted to encode this data with a straightforward numerical mapping:

```
    {'Queen Anne': 1, 'Fremont': 2, 'Wallingford': 3};
```

Categorical Features

```
from sklearn.feature_extraction import DictVectorizer
vec = DictVectorizer(sparse=False, dtype=int)
vec.fit_transform(data)
```

```
In [111]: data = [
              {'price': 850000, 'rooms': 4, 'neighborhood': 'Queen Anne'},
              {'price': 700000, 'rooms': 3, 'neighborhood': 'Fremont'},
              {'price': 650000, 'rooms': 3, 'neighborhood': 'Wallingford'},
              {'price': 600000, 'rooms': 2, 'neighborhood': 'Fremont'}
          ]
```

```
In [112]: from sklearn.feature_extraction import DictVectorizer
          vec = DictVectorizer(sparse=False, dtype=int)
          vec.fit_transform(data)
```

```
Out[112]: array([[     0,      1,      0, 850000,      4],
                 [     1,      0,      0, 700000,      3],
                 [     0,      0,      1, 650000,      3],
                 [     1,      0,      0, 600000,      2]])
```

Categorical Features

To see the meaning of each column, you can inspect the feature names:

vec.get_feature_names()

```
In [113]: vec.get_feature_names()

Out[113]: ['neighborhood=Fremont',
           'neighborhood=Queen Anne',
           'neighborhood=Wallingford',
           'price',
           'rooms']
```

Text Features
word count

Another common need in feature engineering is to convert text to a set of representative numerical values. For example, most automatic mining of social media data relies on some form of encoding the text as numbers. One of the simplest methods of encoding data is by *word counts*:

```
sample = ['problem of evil', 'evil queen', 'horizon problem']
```

For a vectorization of this data based on word count, we could construct a column representing the word "problem," the word "evil," the word "horizon," and so on. While doing this by hand would be possible, the tedium can be avoided by using Scikit-Learn's CountVectorizer

```
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer()
X = vec.fit_transform(sample)
X
```

Text Features
word count

The result is a sparse matrix recording the number of times each word appears; it is easier to inspect if we convert this to a DataFrame with labeled columns:

```
import pandas as pd
pd.DataFrame(X.toarray(), columns=vec.get_feature_names())
```

```
In [114]: sample = ['problem of evil',
                    'evil queen',
                    'horizon problem']

In [115]: from sklearn.feature_extraction.text import CountVectorizer

          vec = CountVectorizer()
          X = vec.fit_transform(sample)
          X

Out[115]: <3x5 sparse matrix of type '<class 'numpy.int64'>'
                  with 7 stored elements in Compressed Sparse Row format>

In [116]: import pandas as pd
          pd.DataFrame(X.toarray(), columns=vec.get_feature_names())
```

Out[116]:

	evil	horizon	of	problem	queen
0	1	0	1	1	0
1	1	0	0	0	1
2	0	1	0	1	0

Term frequency-inverse *TF–IDF*

There are some issues with this approach, however: the raw word counts lead to features which put too much weight on words that appear very frequently, and this can be sub-optimal in some classification algorithms.

One approach to fix this is known as *term frequency-inverse* which weights the word counts by a measure of how often they appear in the documents.

```
from sklearn.feature_extraction.text import TfidfVectorizer
vec = TfidfVectorizer()
X = vec.fit_transform(sample)
pd.DataFrame(X.toarray(), columns=vec.get_feature_names())
```

```
In [117]: from sklearn.feature_extraction.text import TfidfVectorizer
          vec = TfidfVectorizer()
          X = vec.fit_transform(sample)
          pd.DataFrame(X.toarray(), columns=vec.get_feature_names())
```

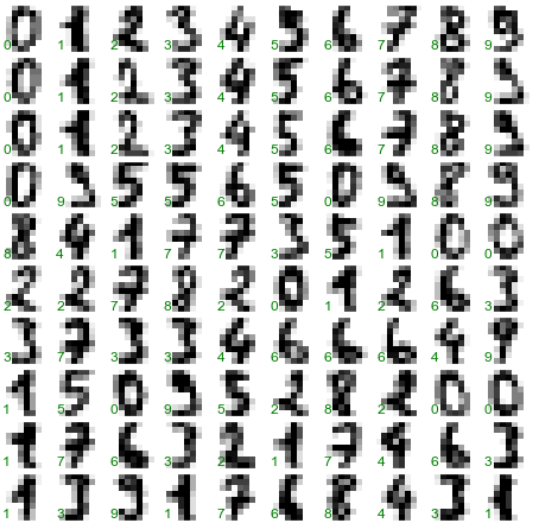
Out[117]:

	evil	horizon	of	problem	queen
0	0.517856	0.000000	0.680919	0.517856	0.000000
1	0.605349	0.000000	0.000000	0.000000	0.795961
2	0.000000	0.795961	0.000000	0.605349	0.000000

Image Feature

The simplest approach is using the pixel values themselves. But depending on the application, such approaches may not be optimal.

```
In [73]: import matplotlib.pyplot as plt

fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
    ax.text(0.05, 0.05, str(digits.target[i]),
            transform=ax.transAxes, color='green')
```



Derived Feature
basis function regression

Another useful type of feature is one that is mathematically derived from some input features.

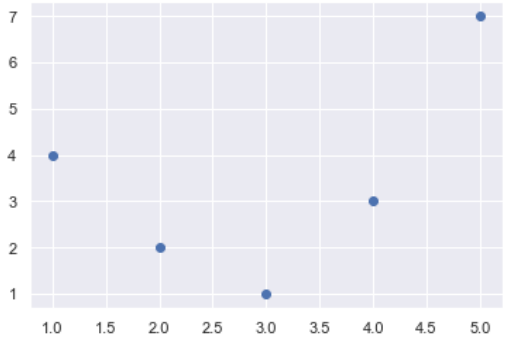
We could convert a linear regression into a polynomial regression not by changing the model, but by **transforming the input**! This is sometimes known as *basis function regression*

Derived Feature

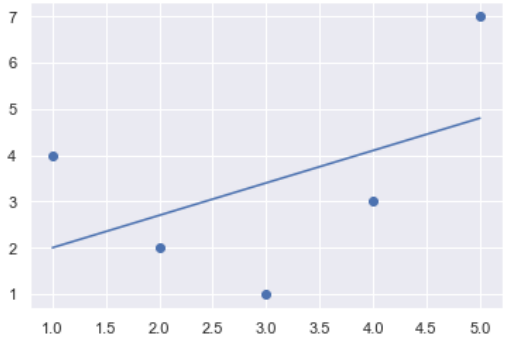
Linear is not representing the relationbetbeer and y

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
x = np.array([1, 2, 3, 4, 5])
y = np.array([4, 2, 1, 3, 7])
plt.scatter(x, y);
from sklearn.linear_model import LinearRegression
X = x[:, np.newaxis]
model = LinearRegression().fit(X, y)
yfit = model.predict(X)
plt.scatter(x, y)
plt.plot(x, yfit);
```

```
In [118]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

x = np.array([1, 2, 3, 4, 5])
y = np.array([4, 2, 1, 3, 7])
plt.scatter(x, y);
```




```
In [119]: from sklearn.linear_model import LinearRegression
X = x[:, np.newaxis]
model = LinearRegression().fit(X, y)
yfit = model.predict(X)
plt.scatter(x, y)
plt.plot(x, yfit);
```



Derived Feature

One approach to this is to transform the data, adding extra columns of features to drive more flexibility in the model.

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=3, include_bias=False)
X2 = poly.fit_transform(X)
print(X2)
```

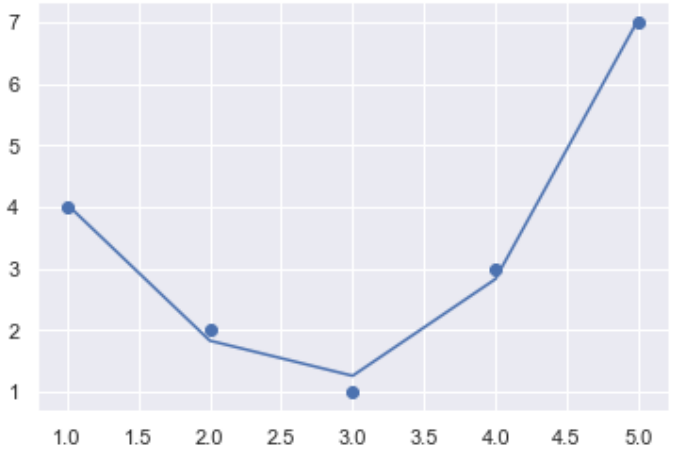
```
In [120]: from sklearn.preprocessing import PolynomialFeatures
          poly = PolynomialFeatures(degree=3, include_bias=False)
          X2 = poly.fit_transform(X)
          print(X2)
```

```
[[  1.   1.   1.]
 [  2.   4.   8.]
 [  3.   9.  27.]
 [  4.  16.  64.]
 [  5.  25. 125.]]
```

Derived Feature

Computing a linear regression on this expanded input gives a much closer fit to our data:

```
In [121]: model = LinearRegression().fit(X2, y)
          yfit = model.predict(X2)
          plt.scatter(x, y)
          plt.plot(x, yfit);
```




```
model = LinearRegression().fit(X2, y)
yfit = model.predict(X2)
plt.scatter(x, y)
plt.plot(x, yfit);
```

This idea of improving a model not by changing the model, but by transforming the inputs, is fundamental to many of the more powerful machine learning methods.

Missing data Imputation

Another common need in feature engineering is handling of missing data.

```
from numpy import nan
X = np.array([[ nan, 0, 3 ],
       [ 3, 7, 9 ],
       [ 3, 5, 2 ],
       [ 4, nan, 6 ],
       [ 8, 8, 1 ]])
y = np.array([14, 16, -1, 8, -5])
```

When applying a typical machine learning model to such data, we will need to first replace such missing data with some appropriate fill value. This is known as ***imputation*** of missing values

Missing data Imputation

Strategies range from simple (e.g., replacing missing values with the **mean of the column**) to sophisticated (e.g., using matrix completion or a robust model to handle such data).

For a baseline imputation approach, using the mean, median, or most frequent value, Scikit-Learn provides the Imputer class:

```
from sklearn.impute import SimpleImputer
imp = SimpleImputer(strategy='mean')
X2 = imp.fit_transform(X)
X2
```

```
In [127]: from numpy import nan
          X = np.array([[ nan, 0,   3  ],
                        [ 3,   7,   9  ],
                        [ 3,   5,   2  ],
                        [ 4,   nan, 6  ],
                        [ 8,   8,   1  ]])
          y = np.array([14, 16, -1,  8, -5])
```

```
In [128]: from sklearn.impute import SimpleImputer
          imp = SimpleImputer(strategy='mean')
          X2 = imp.fit_transform(X)
          X2
```

```
Out[128]: array([[4.5, 0. , 3. ],
                 [3. , 7. , 9. ],
                 [3. , 5. , 2. ],
                 [4. , 5. , 6. ],
                 [8. , 8. , 1. ]])
```

Feature pipelines

It can quickly become tedious to do the transformations by hand, especially if you wish to string together multiple steps.

For example, we might want a processing pipeline that looks something like this:

1. Impute missing values using the mean
2. Transform features to quadratic
3. Fit a linear regression

To streamline this type of processing pipeline, Scikit-Learn provides a `Pipeline` object

Feature pipelines

```
from sklearn.pipeline import make_pipeline
model = make_pipeline(SimpleImputer(strategy='mean'),
      PolynomialFeatures(degree=2),
      LinearRegression())
```

1. Impute missing values using the mean
2. Transform features to quadratic
3. Fit a linear regression

This pipeline looks and acts like a standard Scikit-Learn object, and will apply all the specified steps to any input data.

```
model.fit(X, y) # X with missing values print(y)
print(model.predict(X))
```

References

https://ipython.org/

https://numpy.org/

https://jupyter.org/try

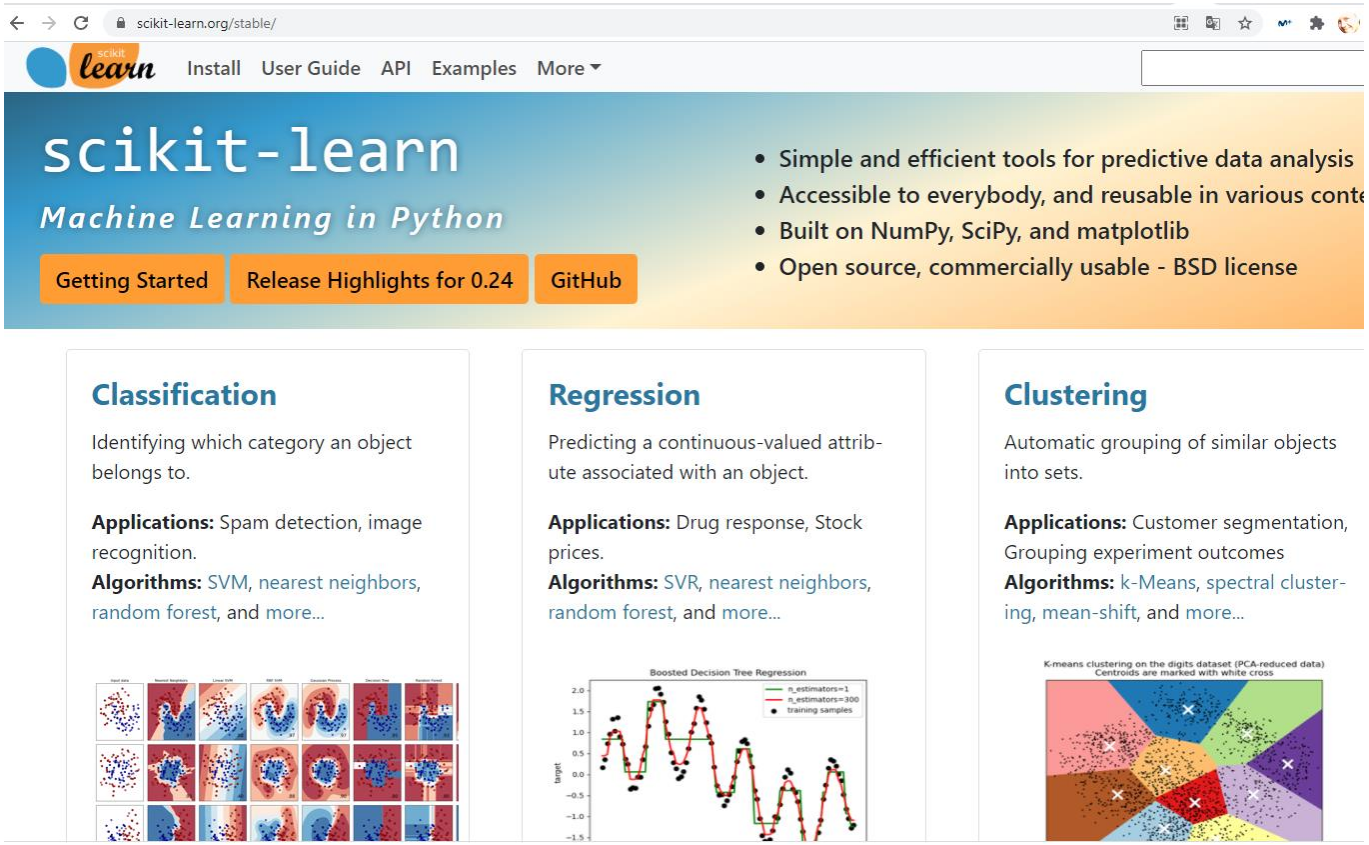
https://pandas.pydata.org/

http://seaborn.pydata.org/

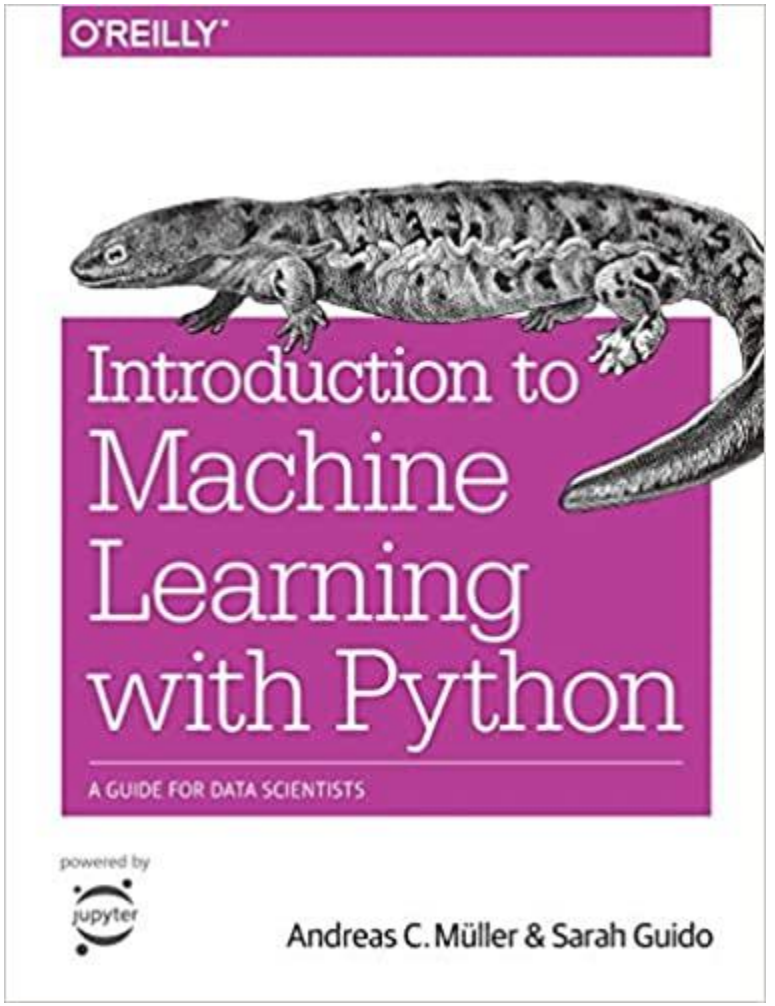
https://scikit-learn.org/stable/

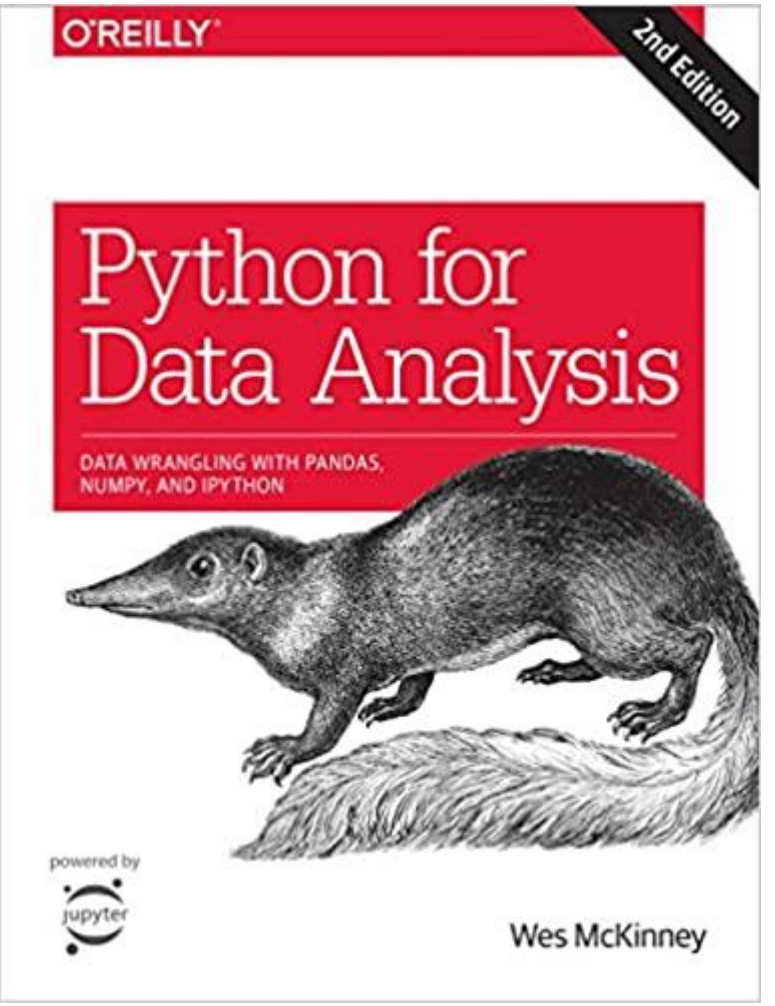
References

https://scikit-learn.org/stable/

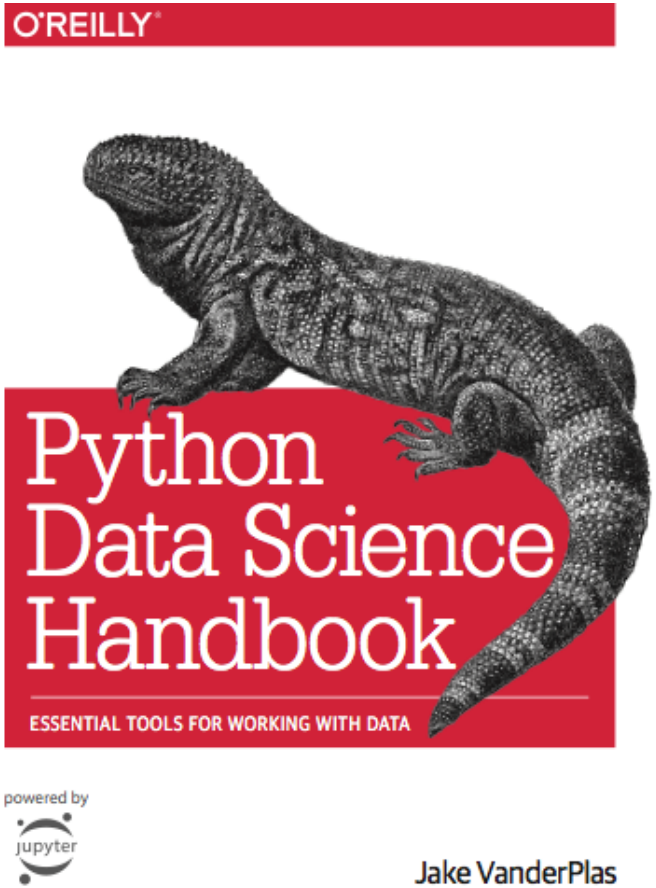


References


O'REILLY
Introduction to
Machine
Learning
with Python
A GUIDE FOR DATA SCIENTISTS
powered by
jupyter
Andreas C. Müller & Sarah Guido


O'REILLY
2nd Edition
Python for
Data Analysis
DATA WRANGLING WITH PANDAS,
NUMPY, AND IPYTHON
powered by
jupyter
Wes McKinney

References



https://jakevdp.github.io/PythonDataScienceHandbook/