CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

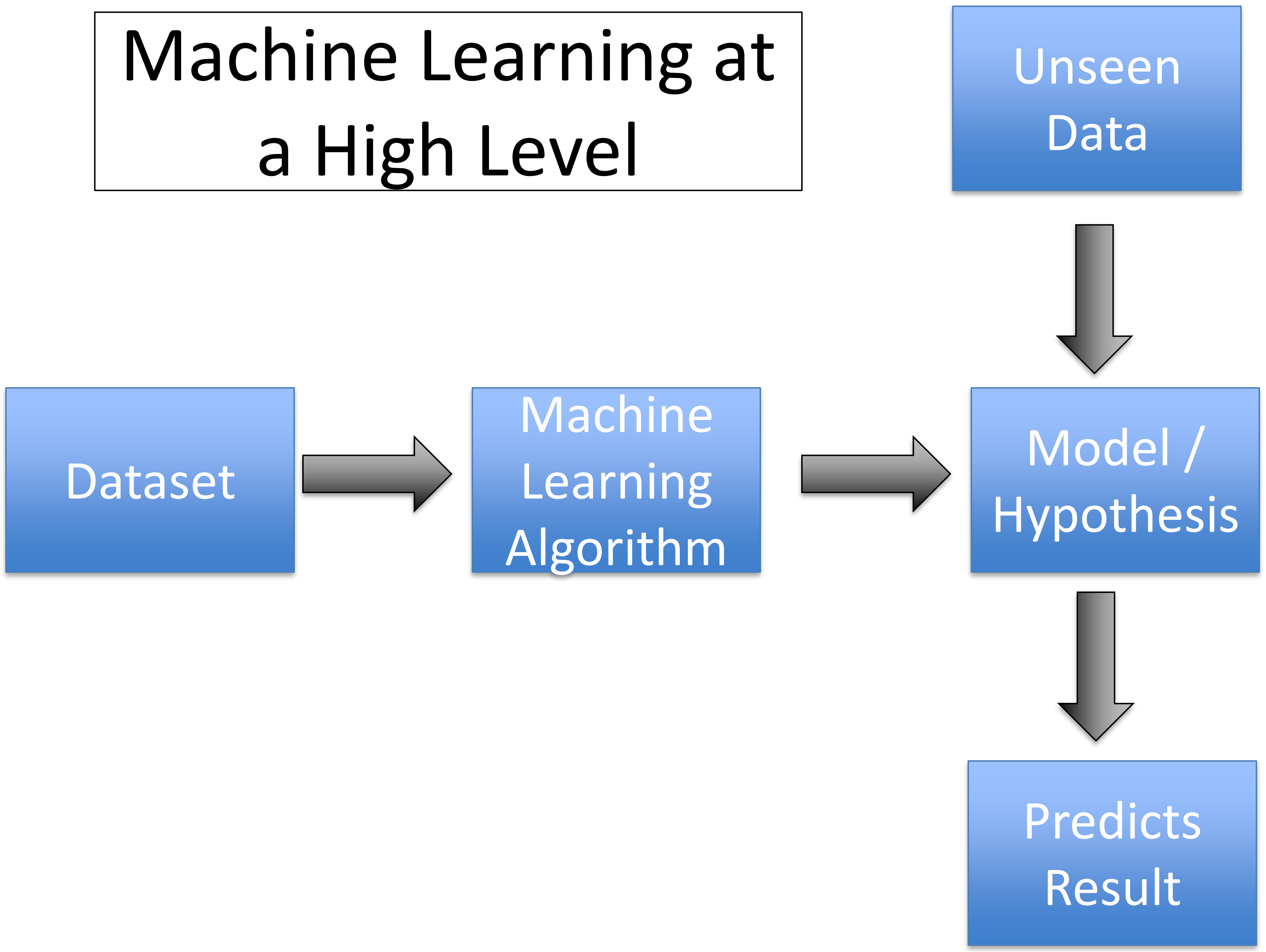
Machine Learning

Machine Learning

Lecture: Introduction to Scikit-Learn

Ted Scully

Machine Learning at a High Level



Machine Learning Process

- The machine learning process is much more involved than the high level work-flow depicted in the previous slide.

- The stages can be broadly defined as follows:
  1. Data exploration
     - Understand the feature data (data types, missing values, outliers, etc)
     - Visualization (boxplots, scatter plots, correlation matrix, etc)
     - Correlations between features and the target
     - Feature engineering (aggregate features)

Machine Learning Process

- The machine learning process is much more involved than the high level work-flow depicted in the previous slide.

- The stages can be broadly defined as follows:

2. Data preparation
    - Dealing with outliers
    - Dealing with missing values
    - Feature encoding (encoding categorical features)
    - Feature selection
    - Feature scaling
    - Handling Imbalance

Machine Learning Process

- The machine learning process is much more involved than the high level work-flow depicted in the previous slide.

- The stages can be broadly defined as follows:

3. Building and Evaluating Models
   - Train many models from different categories (e.g., linear, naïve Bayes, SVM, kNN, decision trees, Random Forest, etc.) using standard parameters.
   - Measure and compare their performance.
   - Debug ML models and analyse the types of errors the models make.

4. Fine Tuning and Optimization
   - Perform hyper-parameter optimization
   - Incorporate transformation choices from part 2 as part of the hyper-parameter optimization
   - Try Ensemble methods
   - Finally assess the generalization capability of your model on the test set.

Machine Learning Process with Scikit-Learn

- We will be looking at the steps 2-4 outlined in the previous two slides using Scikit Learn.

- However, before we start on this process we will briefly introduce Scikit learn and how to use it to perform basic classification and regression modelling.

- Therefore, the structure of what we cover will be:
  - Introduction to Scikit Learn
  - Data Preparation
  - Build and Evaluating Models
  - Fine Tuning and Optimization

Part 1 - Introduction to SciKit Learn

- Scikit-learn provides a range of supervised and unsupervised learning algorithms in Python.

The library is largely written in Python but also makes extensive use of **NumPy** and **SciPy**. It is also designed to be used easily with **Matplotlib** for visualization.

- The library is focused on modelling data. It is not focused on loading, manipulating and summarizing data.
- Often you may find that you will need to use techniques from **NumPy and Pandas**.

- The most recent release was in May 2019 (scikit-learn 0.21.0). We will be using 0.21.x throughout this module.

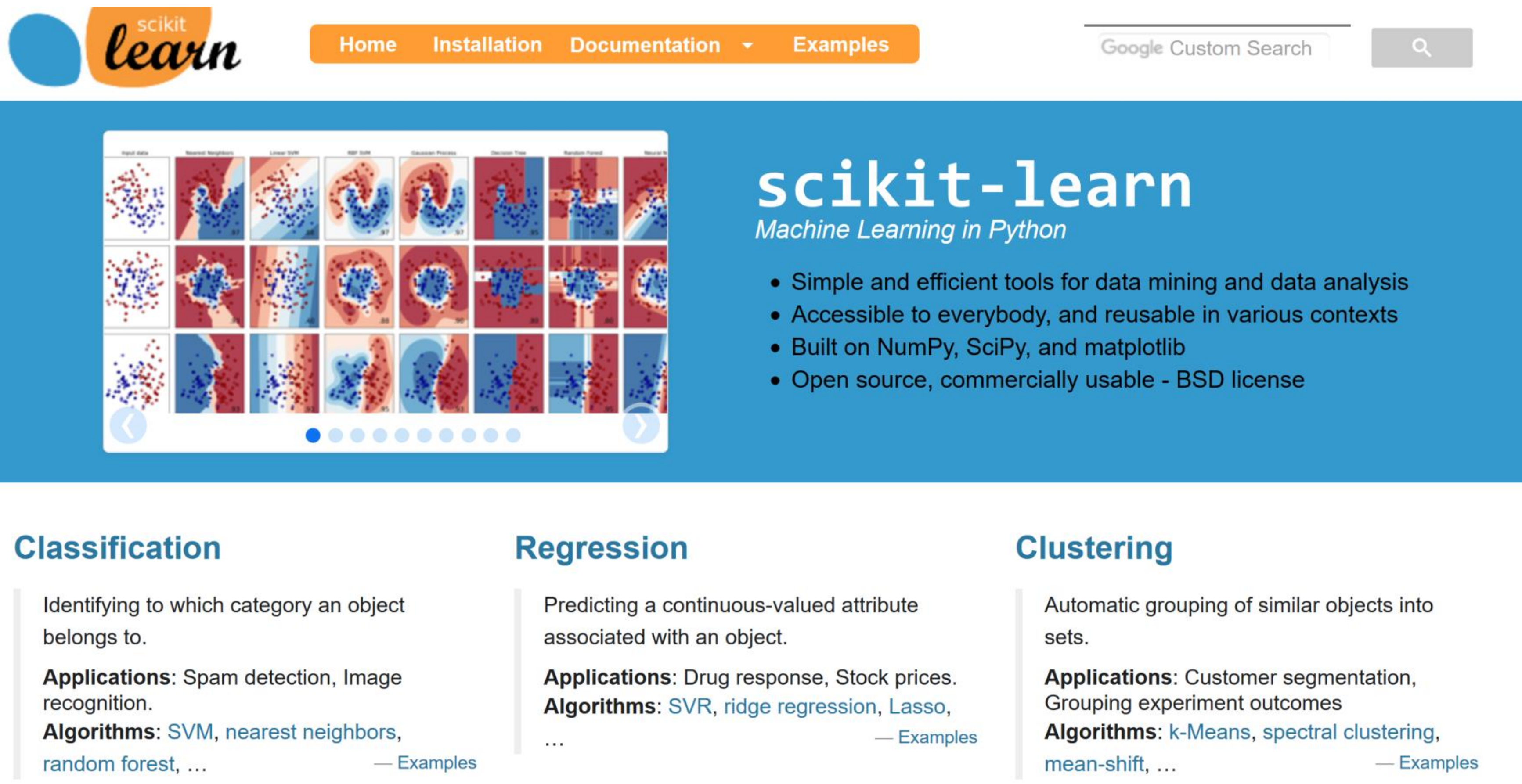
```
import sklearn
print(sklearn.__version__)
```

Introduction to Scikit Learn

- **Scikit Learn is well organized and there are extensive tutorials and API pages, which can be accessed here.**

- **The functionally offered by Scikit can be broken into the following :**

1. **Classification**: a large collection of learning algorithms such as naive bayes, lazy methods, support vector machines, decision trees, ensembles etc.
2. **Clustering**: for grouping unlabelled data such as Kmeans, DBScan, etc.
3. **Regression**: libraries for predicting real-valued attributes such as multiple linear regression, ridge regression, etc.
4. **Pre-processing**: Feature selection, 0utlier detection, normalization, encoding categorical features, etc.
5. **Dimensionality Reduction**: Reduces the number of features that you need to consider in your dataset.
6. **Model Selection**: Cross- validation, metrics, hyper-parameter optimization.



scikit-learn

Machine Learning in Python

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Classification

Identifying to which category an object belongs to.

Applications: Spam detection, Image recognition.
Algorithms: SVM, nearest neighbors, random forest, …

— Examples

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.
Algorithms: SVR, ridge regression, Lasso, …

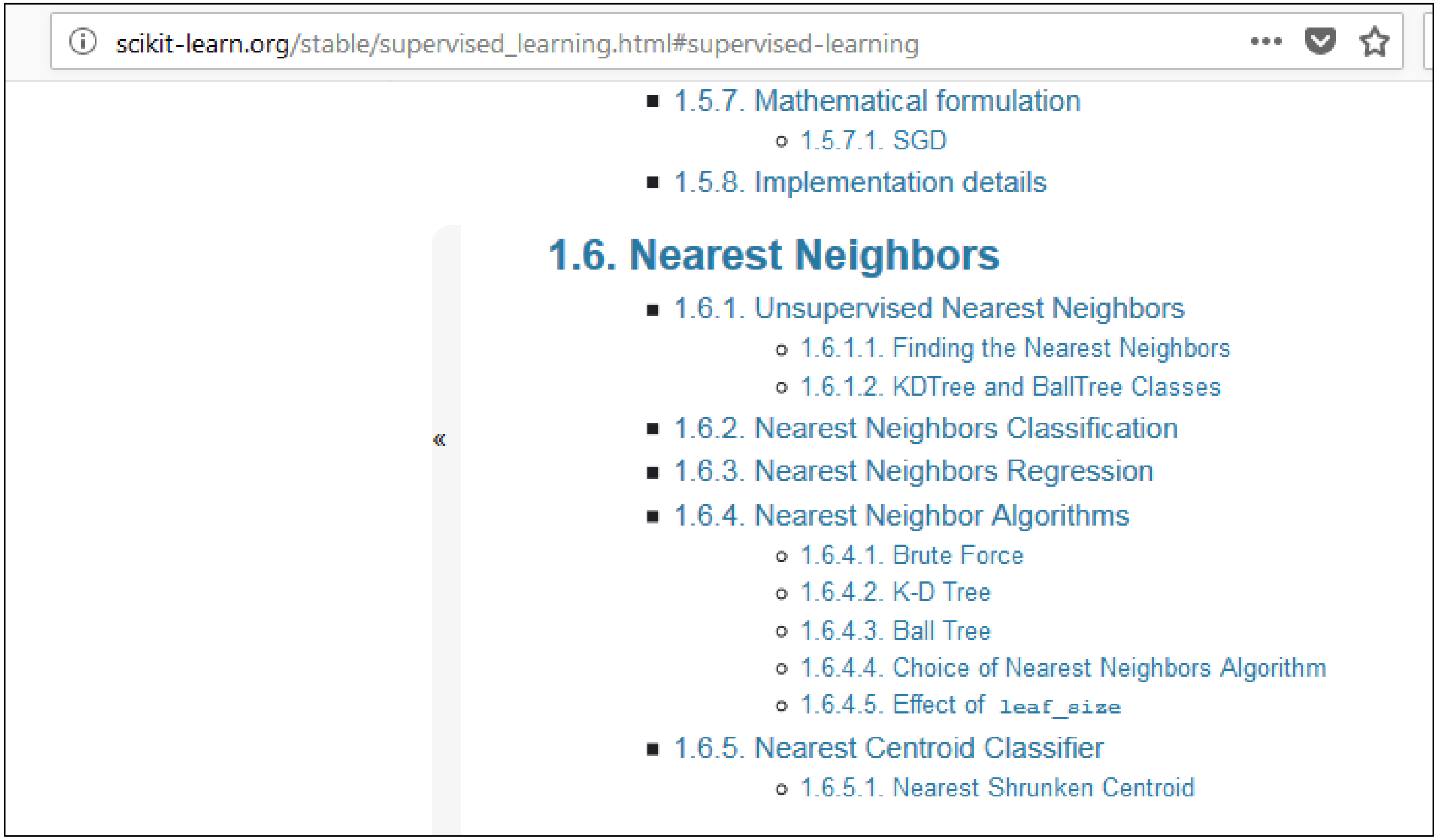
— Examples

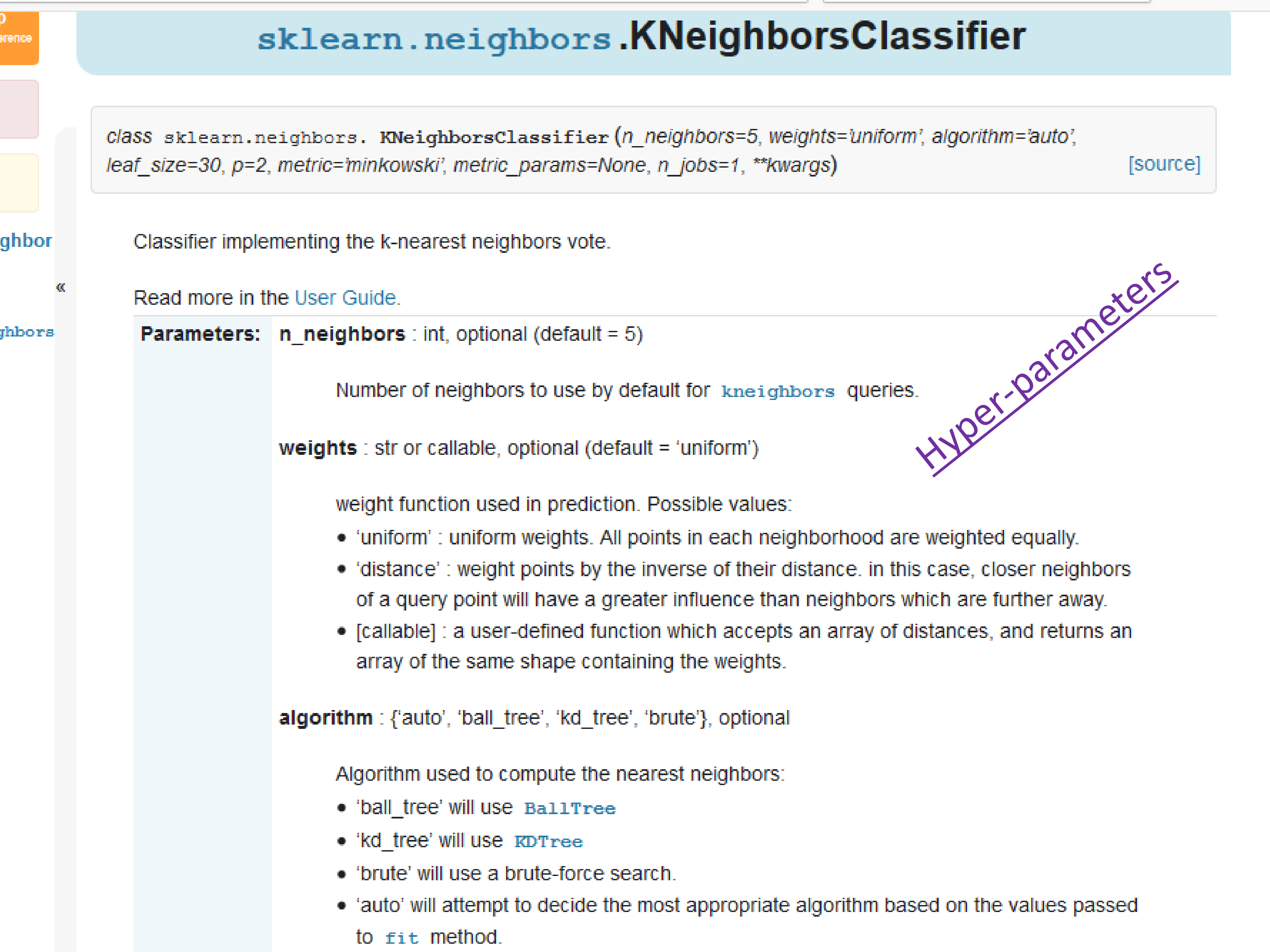
Clustering

Automatic grouping of similar objects into sets.

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

— Examples


scikit-learn.org/stable/supervised_learning.html#supervised-learning



sklearn.neighbors.KNeighborsClassifier

class `sklearn.neighbors.` **`KNeighborsClassifier`** (*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=1, **kwargs*) [source]

Classifier implementing the k-nearest neighbors vote.

Read more in the User Guide.

Hyper-parameters

Parameters: **n_neighbors** : int, optional (default = 5)

Number of neighbors to use by default for `kneighbors` queries.

weights : str or callable, optional (default = 'uniform')

weight function used in prediction. Possible values:

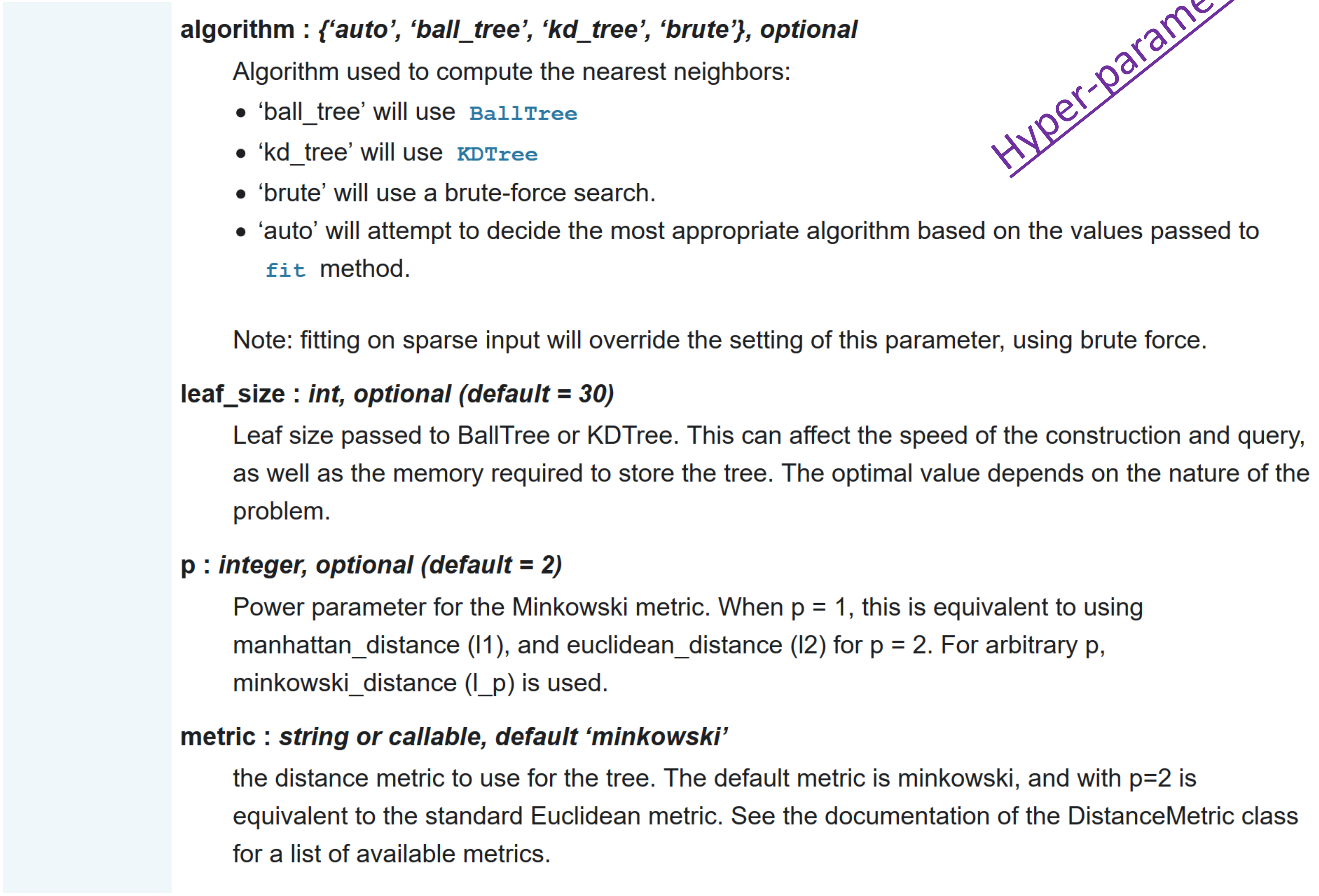
- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

algorithm : {'auto', 'ball_tree', 'kd_tree', 'brute'}, optional

Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method.

Hyper-parameters



algorithm : *{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional*

Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size : *int, optional (default = 30)*

Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p : *integer, optional (default = 2)*

Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric : *string or callable, default 'minkowski'*

the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics.

A few important notes about Scikit Learn

- The following are some important requirements that you should keep in mind when working with Scikit learn.

1. Features and classes/target values are **separate** objects (data structures)
2. Features and classes should be **numerical**
3. Features and classes should be **NumPy** arrays
4. Features and classes should have a **specific shape**
    - Features should be 2D (Columns correspond to numbers of features and rows are number of data instances)
    - Class array should be one dimensional with same number of instances as there are data instances in the features array

Using Datasets

- Scikit-learn comes with a number of standard example datasets. These are broken into toy datasets and real-world datasets.

- Toy datasets include the iris dataset and digits datasets for classification and the Boston house prices dataset for regression.
- Real-world datasets include Olivetti faces dataset, newsgroups, California housing dataset, etc.

- These datasets are **dictionary-like** objects holding at least two items:
  - A NumPy array of shape *n_samples* * *n_features* with the key ***data***
  - A NumPy array of length *n_samples*, containing the class values, with key ***target***

Example

```
from sklearn import datasets

iris = datasets.load_iris()

print (iris.data)

print (iris.target)

print ( iris.data[:, [2,3]] )

print ( iris.data.shape )
print (iris.target.shape)
```

Load iris dataset into a dataset object

Accesses the data stored in the dataset object (2D numpy array)

Accesses the class associated with each data item

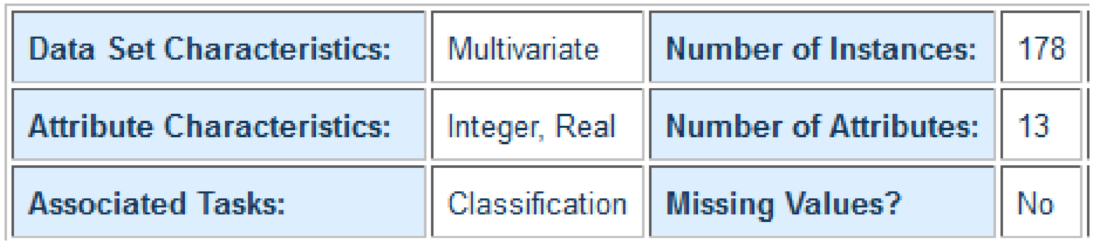
Accesses all rows of the dataset but just columns with index 2 and 3

Outputs the dimensions of the data (150, 4) and labels (150,)

Incorporating External Data

In the following example, we will read in the wine dataset into our program and set it up so that we have a data and target array. The attribute list appears on the right. The class we are trying to predict is alcohol

The wine dataset we are using can be found here



Data Set Characteristics:	Multivariate	**Number of Instances:**	178
Attribute Characteristics:	Integer, Real	**Number of Attributes:**	13
Associated Tasks:	Classification	**Missing Values?**	No

- 1) Alcohol
- 2) Malic acid
- 3) Ash
- 4) Alcalinity of ash
- 5) Magnesium
- 6) Total phenols
- 7) Flavanoids
- 8) Nonflavanoid phenols
- 9) Proanthocyanins
- 10)Color intensity
- 11)Hue
- 12)OD280/OD315 of diluted wines
- 13)Proline

Incorporating External Data

```
import numpy as np

dataset = np.genfromtxt("wine.data", delimiter=",")


target = dataset[:, 0]

data = dataset[:, 1:13]
```

Notice here we extract the first column as our class

Here we extract all remaining columns and use as our data

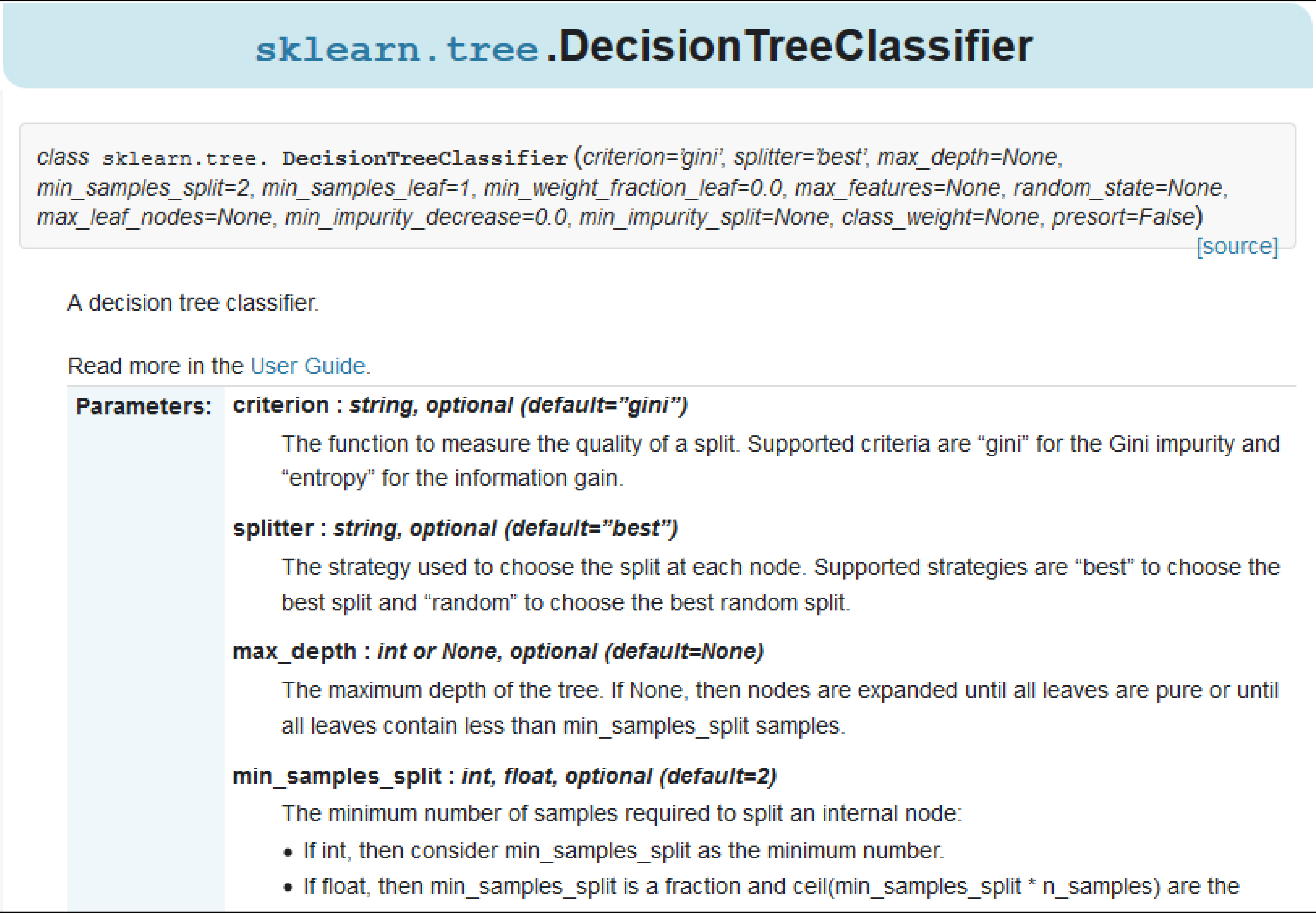
Using a Decision Tree Classifier in SciKit Learn

- Decision trees are a family of supervised learning methods (estimators) used for classification and regression.

- The class we will be using is **sklearn.tree.DecisionTreeClassifier**

- The DecisionTreeClassifier uses a fit method to train and build the tree.
- The fit method takes as input **two** arrays:
  - An array X of size **[n_samples, n_features]** holding the training samples,
  - An array Y of integer values, size **[n_samples]**, holding the class labels for the training samples.

- The **DecisionTreeClassifer**, as you can see from the API, takes many more arguments.



sklearn.tree.DecisionTreeClassifier

class `sklearn.tree.` **`DecisionTreeClassifier`** (*criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, class_weight=None, presort=False*)

[source]

A decision tree classifier.

Read more in the User Guide.

Parameters:

criterion : *string, optional (default="gini")*

The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

splitter : *string, optional (default="best")*

The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth : *int or None, optional (default=None)*

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split : *int, float, optional (default=2)*

The minimum number of samples required to split an internal node:

- If int, then consider min_samples_split as the minimum number.
- If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the

Using a Decision Tree

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class:

-- Iris Setosa
-- Iris Versicolour
-- Iris Virginica

```
from sklearn import tree
from sklearn import datasets


iris = datasets.load_iris()

# creates a new decision tree object classifier
clf = tree.DecisionTreeClassifier()

# train the classifier pass it the training data and classes
clf.fit(iris.data, iris.target)

# predict the class for an unseen example
print (clf.predict( [[4.5, 2.9, 3.0, 2.0]]  ))
```

When we run this code it will predict the output for this unseen instance as being 2, which corresponds to virginica

Using kNN on Iris Dataset

- In the next example we will apply k-nearest neighbour to the iris dataset.

```
from sklearn import datasets
from sklearn import neighbors

iris = datasets.load_iris()


knn = neighbors.KNeighborsClassifier(n_neighbors = 8)

knn.fit(iris.data, iris.target)

print (knn.predict([[3, 5, 4, 2]]))
```

We can provide many different parameters to machine learning algorithms in Scikit Learn. However, most have default values allows us to get started very quickly.

Using kNN on Iris Dataset

- In the next example we will apply k-nearest neighbour to the iris dataset.

```
from sklearn import datasets
from sklearn import neighbors

iris = datasets.load_iris()


knn = neighbors.KNeighborsClassifier(n_neighbors = 5,
algorithm = "kd_tree")

knn.fit(iris.data, iris.target)

print (knn.predict([[3, 5, 4, 2]]))
```

Notice in this example we specify an additional parameter algorithm. When we call the fit function it will use the data to build a binary search tree

Basics of using Scikit Learn

- In the following slides we look at two separate scenarios for evaluation (we will cover more **<u>advanced</u>** and **<u>realistic</u>** techniques in a later):

1. There is a separate **training** and **test** dataset that can be used.

2. A **single** dataset split into a training and test data (holdout method).

Basics of using Scikit Learn

1. There is a separate **training** and **test** dataset that can be used.

f1	f2	f3	…	fn	class

Training Set

f1	f2	f3	…	fn	class

Testing Set

Assessing Accuracy

- Assuming I have separate training and test data I might have the following arrays
  - **features_train, labels_train**
  - **features_test, labels_test**

```
from sklearn import metrics
from sklearn import tree

# creates a new decision tree object classifier
clf = tree.DecisionTreeClassifier()

# trains the classifier pass it the training data and classes
clf = clf.fit(features_train, labels_train)

# predict the class for an unseen example
results= clf.predict(features_test)

print (   metrics.accuracy_score(results, labels_test)   )
```

Assessing Accuracy

- Assuming I have separate training and test data I might have the following arrays
  - features_train, labels_train
  - features_test, labels_test

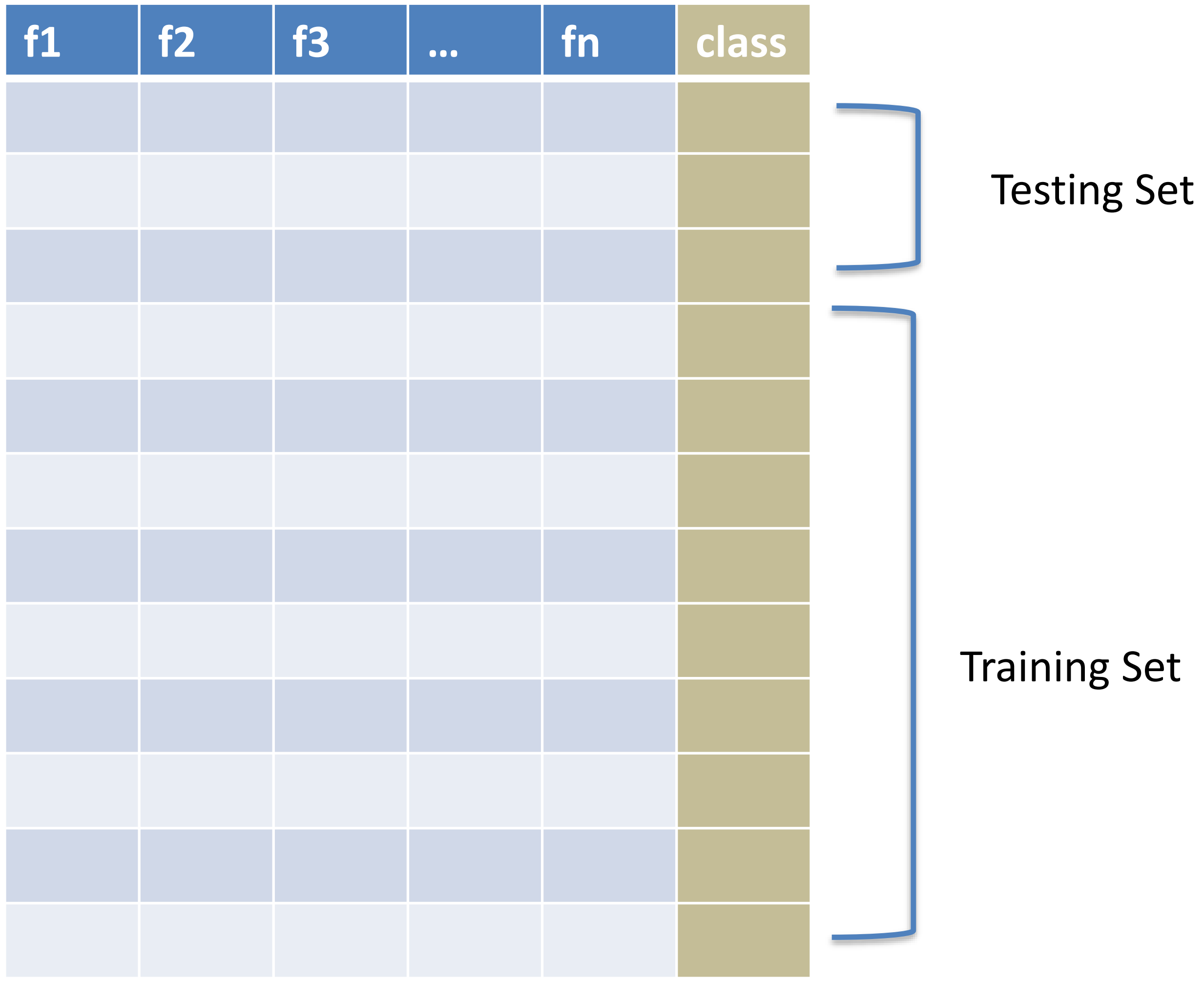
```
from sklearn import metrics
from sklearn import tree

# creates a new decision tree object classifier
clf = tree.DecisionTreeClassifier()

# trains the classifier pass it the training data and classes
clf = clf.fit(features_train, labels_train)

# predict the class for an unseen example
results= clf.predict(features_test)

print (   metrics.accuracy_score(results, labels_test)   )
```

Notice in this example when we call **predict** we are passing it a 2D NumPy array.

The **accuracy_score** function (available in the metrics module) will count the number of classes we correctly predicated and express that as a **percentage** of the total number of test data instances.

Basics of using Scikit Learn

1. In this scenario we have a single dataset, which we then split into training and test data.



We split a single dataset into test and train data we select the rows **randomly**.

Assessing Accuracy (Splitting Training Data)

- In scikit-learn a random split into training and test sets can be quickly computed with the *train_test_split* helper function.
- As arguments we pass it the **original data** and **target** as well as the **percentage of the original data** we want for the training data. We also pass it a random seed.

```
import numpy as np
from sklearn import model_selection
from sklearn import tree

dataset = np.genfromtxt("trainingData.csv", delimiter=',')
print (dataset.shape)

featureData = dataset[:,:-1]
classData = dataset[:, -1]


train_features, test_features, train_labels, test_labels = model_selection.train_test_split(
featureData, classData, test_size=0.2, random_state=0)

print (train_features.shape, train_labels.shape)
print (test_features.shape, test_labels.shape)
```

```
(4000, 11)

(3200, 10) (3200,)
(800, 10) (800,)
```

```
import numpy as np
from sklearn import model_selection
from sklearn import tree
from sklearn import metrics

dataset = np.genfromtxt("trainingData.csv", delimiter=',')
featureData = dataset[:,:-1]
classData = dataset[:, -1]

train_features, test_features, train_labels, test_labels = model_selection.train_test_split(
featureData, classData, test_size=0.2, random_state=0)

clf = tree.DecisionTreeClassifier()
clf = clf.fit(train_features, train_labels)

# predict the class for an unseen example
results= clf.predict(test_features)
print  (metrics.accuracy_score(results, test_labels))

# we can replace the above two lines with the line below as
# a shortcut
print (clf.score(test_features, test_labels))
```

The slide shows the full program, where we take in the training data. We split into a training and test set. We then assess its accuracy

```
0.78875
0.78875
```

Using a Regression in Scikit Learn

- In the following slide we apply a k nearest neighbour regression algorithm to a regression dataset.
- Specifically we apply the KNeighborsRegressor algorithm to this dataset.

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.datasets.samples_generator import make_regression

data, target = make_regression(n_samples=10000, n_features=20, noise = 0.2,
random_state = 10)

train_features, test_features, train_labels, test_labels = model_selection.train_test_split(
data, target, test_size=0.2)

reg = KNeighborsRegressor()
reg = reg.fit(train_features, train_labels)

results= reg.predict(test_features)
print (metrics.r2_score(test_labels,results))

# As before we can replace the above lines
# with all call to the score function.
print (reg.score(test_features, test_labels))
```

Above we generate a sample dataset using make_regression. This is often really useful if you want to explore certain functionality in Scikit as you can generate a dataset and give them specific characteristics. (Note there is also a make_classification function)

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.datasets.samples_generator import make_regression

data, target = make_regression(n_samples=10000, n_features=20, noise = 0.2,
random_state = 10)

train_features, test_features, train_labels, test_labels = model_selection.train_test_split(
data, target, test_size=0.2)

reg = KNeighborsRegressor()
reg = reg.fit(train_features, train_labels)

results= reg.predict(test_features)
print (metrics.r2_score(test_labels,results))

print (reg.score(test_features, test_labels))
```

Notice, we again use the score method (as we did with classification). However, the default metric in the score value for regression models is r^2. Alternatively we can call the r2_score function from the metrics module. If you look at the metrics module it provides a much broad range of metrics (more on this later)

Classification metrics

See the Classification metrics section of the user guide for further details.

`metrics.accuracy_score` (y_true, y_pred[, …])	Accuracy classification score.
`metrics.auc` (x, y[, reorder])	Compute Area Under the Curve (AUC) using the trapezoidal rule
`metrics.average_precision_score` (y_true, y_score)	Compute average precision (AP) from prediction scores
`metrics.balanced_accuracy_score` (y_true, y_pred)	Compute the balanced accuracy
`metrics.brier_score_loss` (y_true, y_prob[, …])	Compute the Brier score.
`metrics.classification_report` (y_true, y_pred)	Build a text report showing the main classification metrics
`metrics.cohen_kappa_score` (y1, y2[, labels, …])	Cohen's kappa: a statistic that measures inter-annotator agreement.
`metrics.confusion_matrix` (y_true, y_pred[, …])	Compute confusion matrix to evaluate the accuracy of a classification
`metrics.f1_score` (y_true, y_pred[, labels, …])	Compute the F1 score, also known as balanced F-score or F-measure
`metrics.fbeta_score` (y_true, y_pred, beta[, …])	Compute the F-beta score
`metrics.hamming_loss` (y_true, y_pred[, …])	Compute the average Hamming loss.
`metrics.hinge_loss` (y_true, pred_decision[, …])	Average hinge loss (non-regularized)
`metrics.jaccard_score` (y_true, y_pred[, …])	Jaccard similarity coefficient score
`metrics.log_loss` (y_true, y_pred[, eps, …])	Log loss, aka logistic loss or cross-entropy loss.
`metrics.matthews_corrcoef` (y_true, y_pred[, …])	Compute the Matthews correlation coefficient (MCC)
`metrics.multilabel_confusion_matrix` (y_true, …)	Compute a confusion matrix for each class or sample
`metrics.precision_recall_curve` (y_true, …)	Compute precision-recall pairs for different probability thresholds
`metrics.precision_recall_fscore_support` (…)	Compute precision, recall, F-measure and support for each class
`metrics.precision_score` (y_true, y_pred[, …])	Compute the precision
`metrics.recall_score` (y_true, y_pred[, …])	Compute the recall
`metrics.roc_auc_score` (y_true, y_score[, …])	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.