Introduction to Machine Learning

Joaquin Vanschoren, Eindhoven University of Technology

Artificial Intelligence

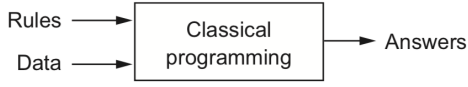
1950s: Can computers be made to 'think'?

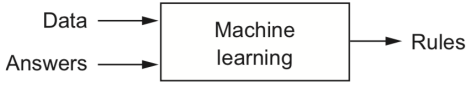
- automate intellectual tasks normally performed by humans
- encompasses learning, but also many other tasks (e.g. logic, planning,...)
- *symbolic AI*: programmed rules/algorithms for manipulating knowledge
    - Great for well-defined problems: chess, expert systems,...
    - Pervasively used today (e.g. chip design)
    - Hard for complex, fuzzy problems (e.g. images, text)

Machine Learning

Are computers capable of learning and originality? Alan Turing: Yes!

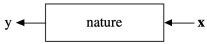
- Learn to perform a task T given experience (examples) E, always improving according to some metric M
- New programming paradigm
  - System is *trained* rather than explictly programmed
  - *Generalizes* from examples to find rules (models) to act/predict
- As more data becomes available, more ambitious problems can be tackled



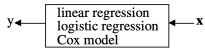


Machine learning vs Statistics

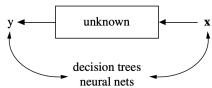
- Both aim to make predictions of natural phenomena:



- Statistics:
  - Help humans understand the world
  - Parametric: assume data is generated according to parametric model



- Machine learning:
  - Automate a task entirely (partially *replace* the human)
  - Assume that data generation process is unknown
  - Engineering-oriented, less (too little?) mathematical theory



See Breiman (2001): Statical modelling: The two cultures

Machine Learning success stories

- Search engines (e.g. Google)
- Recommender systems (e.g. Netflix)
- Automatic translation (e.g. Google Translate)
- Speech understanding (e.g. Siri, Alexa)
- Game playing (e.g. AlphaGo)
- Self-driving cars
- Personalized medicine
- Progress in all sciences: Genetics, astronomy, chemistry, neurology, physics,..

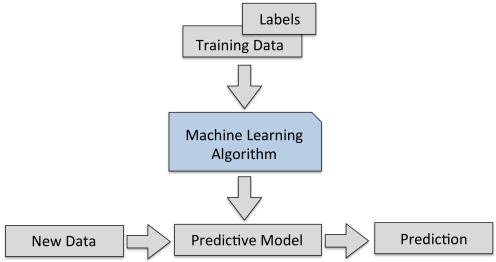
Types of machine learning

- **Supervised Learning**: learn a *model* from labeled *training data* (ground truth)
  - Given a new input X, predict the right output y
  - Given images of cats and dogs, predict whether a new image is a cat or a dog
- **Unsupervised Learning**: explore the structure of the data to extract meaningful information
  - Given inputs X, find which ones are special, similar, anomalous, ...
- **Semi-Supervised Learning**: learn a model from (few) labeled and (many) unlabeled examples
  - Unlabeled examples add information about which new examples are likely to occur
- **Reinforcement Learning**: develop an agent that improves its performance based on interactions with the environment

Note: Practical ML systems can combine many types in one system.

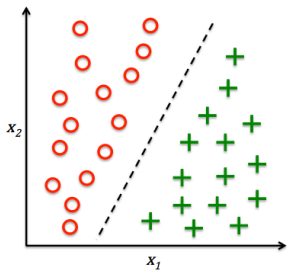
Supervised Machine Learning

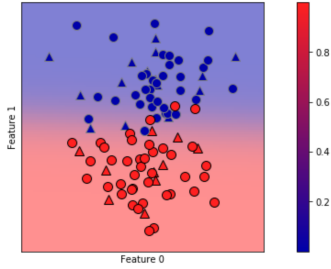
- Learn a model from labeled training data, then make predictions
- Supervised: we know the correct/desired outcome (label)
- Subtypes: *classification* (predict a class) and *regression* (predict a numeric value)
- Most supervised algorithms that we will see can do both



Classification

- Predict a *class label* (category), discrete and unordered
  - Can be *binary* (e.g. spam/not spam) or *multi-class* (e.g. letter recognition)
  - Many classifiers can return a *confidence* per class
- The predictions of the model yield a *decision boundary* separating the classes



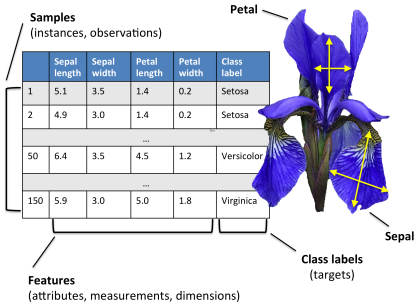


Example: Flower classification

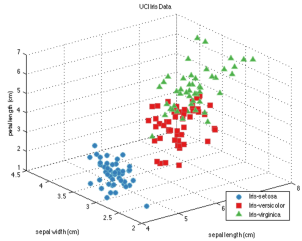
Classify types of Iris flowers (setosa, versicolor, or virginica)

Representation: input features and labels

- Dataset can have any number of input features (variables)
  - Every example is a point in a (possibly high-dimensional) space

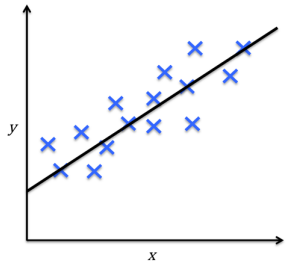


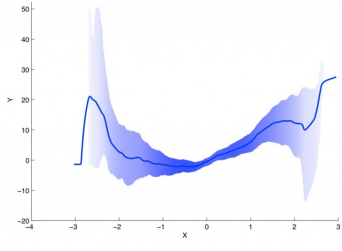
	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica



Regression

- Predict a continuous value, e.g. temperature
  - Target variable is numeric
  - Some algorithms can return a *confidence interval*
- Find the relationship between predictors and the target.
  - E.g. relationship between hours studied and final grade



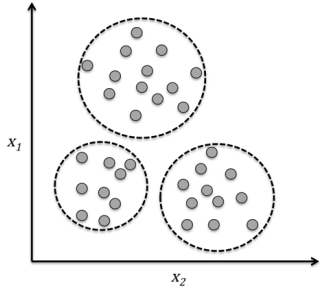


Unsupervised Machine Learning

- Unlabeled data, or data with unknown structure
- Explore the structure of the data to extract information
- Many types, we'll just discuss two.

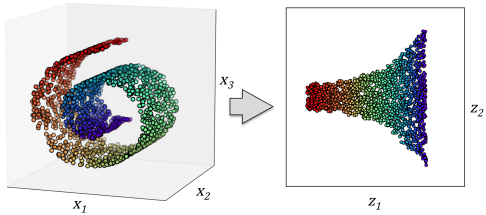
Clustering

- Organize information into meaningful subgroups (clusters)
- Objects in cluster share certain degree of similarity (and dissimilarity to other clusters)
- Example: distinguish different types of customers



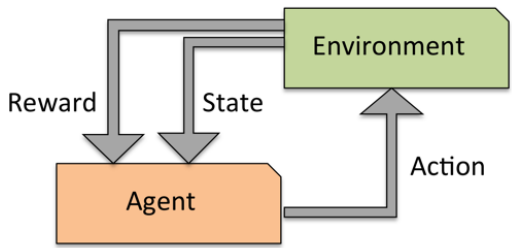
Dimensionality reduction

- Data can be very high-dimensional and difficult to understand, learn from, store,...
- Dimensionality reduction can compress the data into fewer dimensions, while retaining most of the information
- Contrary to feature selection, the new features lose their (original) meaning
- The new representation can be a lot easier to model (and visualize)



Reinforcement learning

- Develop an agent that improves its performance based on interactions with the environment
    - Example: games like Chess, Go,...
- Search a (large) space of actions and states
- *Reward function* defines how well a (series of) actions works
- Learn a series of actions (policy) that maximizes reward through exploration



Learning = Representation + evaluation + optimization

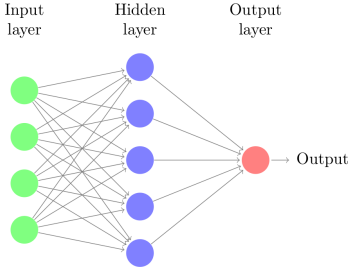
All machine learning algorithms consist of 3 components:

- **Representation**: A model must be represented in a formal language that the computer can handle
    - Defines the 'concepts' it can learn, the *hypothesis space*
    - E.g. a decision tree, neural network, set of annotated data points
- **Evaluation**: An *internal* way to choose one hypothesis over the other
    - Objective function, scoring function, loss function
    - E.g. Difference between correct output and predictions
- **Optimization**: An *efficient* way to search the hypothesis space
    - Start from simple hypothesis, extend (relax) if it doesn't fit the data
    - Defines speed of learning, number of optima,...
    - E.g. Gradient descent

A powerful/flexible model is only useful if it can also be optimized efficiently

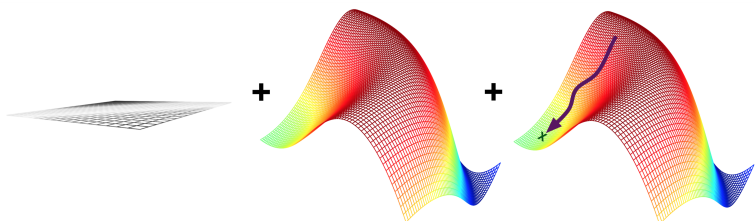
Example: neural networks

- Representation: (layered) neural network
  - Each connection has a *weight* (a.k.a. model parameters)
  - Each node receives the weighted input values and emits a new value
- The *hypothesis space* consists of the set of all weights
- The architecture, type of neurons, etc. are fixed
  - We call these *hyper-parameters* (set by user, fixed during training)
  - They can also be learned (in an outer loop)



Example: neural networks

- Representation: For illustration, consider the space of 2 model parameters
- Evaluation: A *loss function* computes, for each set of parameters, how good the predictions are
    - *Estimated* on a set of training data with the 'correct' predictions
    - We can't see the full surface, only evaluate specific sets of parameters
- Optimization: Find the optimal set of parameters
    - Usually a type of *search* in the hypothesis space
    - Given a few initial evaluations, predict which parameters may be better



Generalization, Overfitting and Underfitting

- We *hope* that the model can *generalize* from the training data: make accurate predictions on unseen data.
- We can never be sure, only hope that we make the right assumptions.
    - We typically assume that new data will be similar to previous data
    - *Inductive bias*: assumptions that we put into the algorithm (everything except the training data itself)

Example: Dating

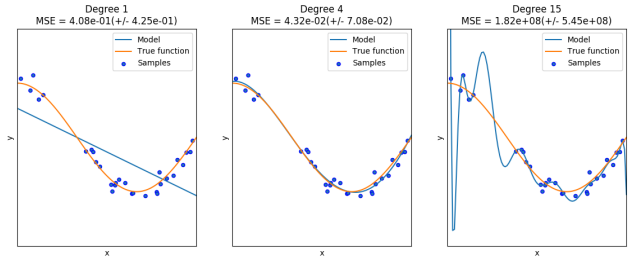
Nr	Day of Week	Type of Date	Weather	TV Tonight	Date?
1	Weekday	Dinner	Warm	Bad	No
2	Weekend	Club	Warm	Bad	Yes
3	Weekend	Club	Warm	Bad	Yes
4	Weekend	Club	Cold	Good	No
Now	Weekend	Club	Cold	Bad	?

- Can you find a simple rule that works? Is one better than others?
- What can we assume about the future? Nothing?
- What if there is noise / errors?
- What if there are factor you don't know about?

Overfitting and Underfitting

- It's easy to build a complex model that is 100% accurate on the training data, but very bad on new data
- Overfitting: building a model that is *too complex for the amount of data* that we have
    - You model peculiarities in your training data (noise, biases,...)
    - Solve by making model simpler (regularization), or getting more data
    - **Most algorithms have hyperparameters that allow regularization**
- Underfitting: building a model that is *too simple given the complexity of the data*
    - Use a more complex model
- There are techniques for detecting overfitting (e.g. bias-variance analysis). More about that later
- You can build *ensembles* of many models to overcome both underfitting and overfitting

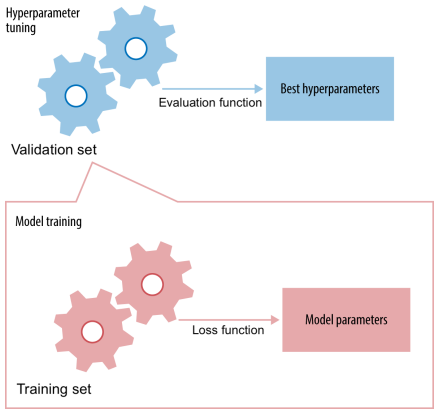
- There is often a sweet spot that you need to find by optimizing the choice of algorithms and hyperparameters, or using more data.
- Example: regression using polynomial functions



Model selection

- Next to the (internal) loss function, we need an (external) evaluation function
    - Feedback signal: are we actually learning the right thing?
    - Are we under/overfitting?
    - More freely chosen to fit the application. Loss functions have constraints (e.g. differentiable)
    - Needed to choose between algorithms (or different hyper-parameter settings)

- Data needs to be split into *training* and *test* sets
  - Optimize model parameters on the training set, evaluate on independent test set
  - To optimize hyperparameters as well, set aside part of training set as a *validation* set

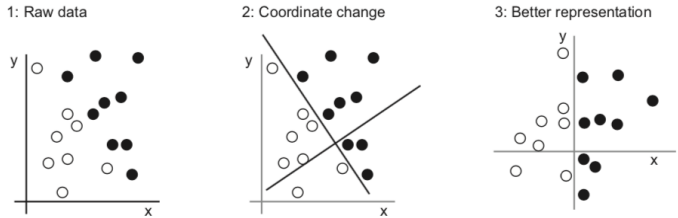


Only generalization counts!

- Never evaluate your final models on the training data, except for:
    - Tracking whether the optimizer converges (learning curves)
    - Detecting under/overfitting:
        - Low training and test score: underfitting
        - High training score, low test score: overfitting
- Always keep a completely independent test set
- Avoid data leakage:
    - Never optimize hyperparameter settings on the test data
    - Never choose preprocessing techniques based on the test data
- On small datasets, use multiple train-test splits to avoid bias
    - E.g. Use cross-validation (see later)

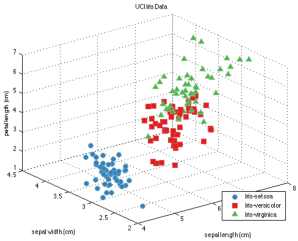
Data (problem) representation

- Algorithm needs to correctly transform the inputs to the right outputs
- A lot depends on how we present the data to the algorithm
  - Transform the data to a more useful representation (a.k.a. *encoding* or *embedding*)
  - Can be done end-to-end (e.g. deep learning) or by first 'preprocessing' the data



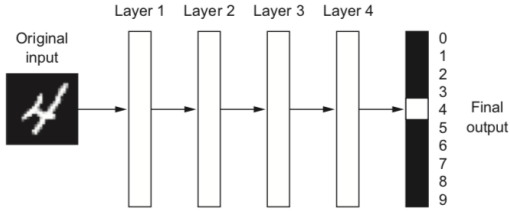
Feature engineering

- Most machine learning techniques require humans to build a good representation of the data
    - Sometimes data is naturally structured (e.g. medical tests)
- Nothing beats domain knowledge (when available) to get a good representation
    - E.g. Iris data: leaf length/width separate the classes well
- Feature engineering is often necessary to get the best results
    - Feature selection, dimensionality reduction, scaling, ...



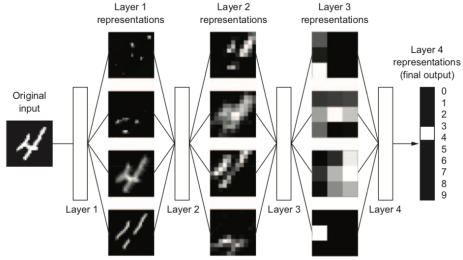
Learning data transformations end-to-end

- For unstructured data (e.g. images, text), it's hard to extract good features
- Deep learning: learn your own representation (embedding) of the data
    - Through multiple layers of representation (e.g. layers of neurons)
    - Each layer transforms the data a bit, based on what reduces the error



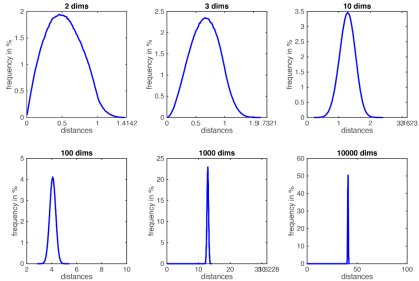
Example: digit classification

- Input pixels go in, each layer transforms them to an increasingly informative representation for the given task
- Often less intuitive for humans



Curse of dimensionality

- Intuition fails in high dimensions:
  - Randomly sample points in an n-dimensional space (e.g. a unit square)
  - The more dimensions you have, the more sparse the space becomes
  - Distances between any two points will become almost identical



Practical consequences

- For every dimension (feature) you add, you need exponentially more data to avoid sparseness
- Affects any algorithm that is based on distances (e.g. kNN, SVM, kernel-based methods, tree-based methods,...)
- Blessing of non-uniformity: on many applications, the data lives in a very small subspace
- You can drastically improve performance by selecting features or using lower-dimensional data representations

More data beats a cleverer algorithm

- More data reduces the chance of overfitting
- Less sparse data reduces the curse of dimensionality
- *Non-parametric* models: number of model parameters grows with the amount of data
    - Tree-based techniques, k-Nearest neighbors, SVM,...
    - They can learn any model given sufficient data (but can get stuck in local minima)
- *Parametric* (fixed size) models: fixed number of model parameters
    - Linear models, Neural networks,...
    - Can be given a huge number of parameters to benefit from more data
    - Deep learning models can have millions of weights, learn almost any function.
- The bottleneck is moving from data to compute/scalability

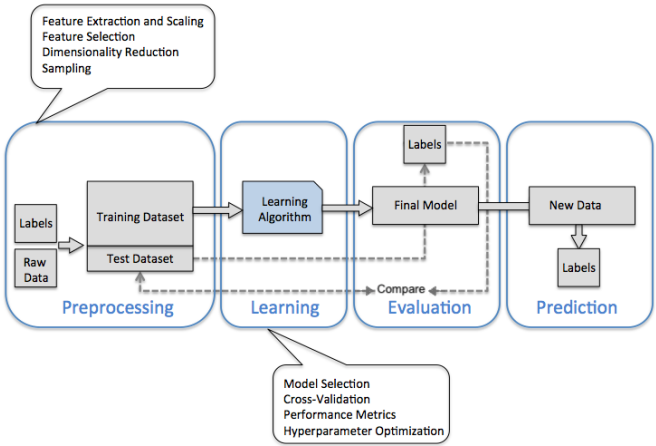
Building machine learning systems

A typical machine learning system has multiple components:

- Preprocessing: Raw data is rarely ideal for learning
    - Feature scaling: bring values in same range
    - Encoding: make categorical features numeric
    - Discretization: make numeric features categorical
    - Label imbalance correction (e.g. downsampling)
    - Feature selection: remove uninteresting/correlated features
    - Dimensionality reduction can also make data easier to learn
    - Using pre-learned embeddings (e.g. word-to-vector, image-to-vector)

- Learning and evaluation
    - Every algorithm has its own biases
    - No single algorithm is always best
    - *Model selection* compares and selects the best models
        - Different algorithms, different hyperparameter settings
    - Split data in training, validation, and test sets
- Prediction
    - Final optimized model can be used for prediction
    - Expected performance is performance measured on *independent* test set

- Together they form a *workflow* of *pipeline*
- You need to optimize pipelines continuously
    - *Concept drift*: the phenomenon you are modelling can change over time
    - *Feedback*: your model's predictions may change future data



In Practice

- Let's build a simple model for our Iris dataset
- We'll use **scikit-learn**
  - Contains many state-of-the-art machine learning algorithms
  - Offers comprehensive documentation (http://scikit-learn.org/stable/documentation) about each algorithm
  - Widely used, and a wealth of tutorials (http://scikit-learn.org/stable/user_guide.html) and code snippets are available
  - Works well with numpy, scipy, pandas, matplotlib,...

Data import

Multiple options:

- A few toy datasets are included in `sklearn.datasets`
- Import 1000s of machine learning datasets from OpenML (http://www.openml.org) with `sklearn.datasets.fetch_openml`
- You can import data files (CSV) with `pandas` or `numpy`

Iris is included in scikitlearn, we can just load it.
This will return a `Bunch` object (similar to a `dict`)

```
iris_dataset = load_iris()
print("Keys of iris_dataset: {}".format(iris_dataset.keys()))
print(iris_dataset['DESCR'][:193] + "\n...")
```

```
Keys of iris_dataset: dict_keys(['data', 'target', 'target_names', 'DESC
R', 'feature_names', 'filename'])
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, pre
...
```

The targets (classes) and features are stored as `list`s, the data as an `ndarray`

```
print("Targets: {}".format(iris_dataset['target_names']))
print("Features: {}".format(iris_dataset['feature_names']))
print("Shape of data: {}".format(iris_dataset['data'].shape))
print("First 5 rows:\n{}".format(iris_dataset['data'][:5]))
```

```
Targets: ['setosa' 'versicolor' 'virginica']
Features: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
'petal width (cm)']
Shape of data: (150, 4)
First 5 rows:
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
```

The targets are stored separately as an `ndarray`, with indices pointing to the features

```
print("Target names: {}".format(iris_dataset['target_names']))
print("Targets:\n{}".format(iris_dataset['target']))
```

```
Target names: ['setosa' 'versicolor' 'virginica']
Targets:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 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 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Building (fitting) models

All scikit-learn estimators follow the same interface

```
class SupervisedEstimator(...):
    def __init__(self, hyperparam, ...):

    def fit(self, X, y):    # Fit/model the training data
        ...                 # given data X and targets y
        return self

    def predict(self, X):   # Make predictions
        ...                 # on unseen data X
        return y_pred

    def score(self, X, y): # Predict and compare to true
        ...                 # labels y
        return score
```

Training and testing data

To evaluate our classifier, we need to test it on unseen data.
`train_test_split` : splits data randomly in 75% training and 25% test data.

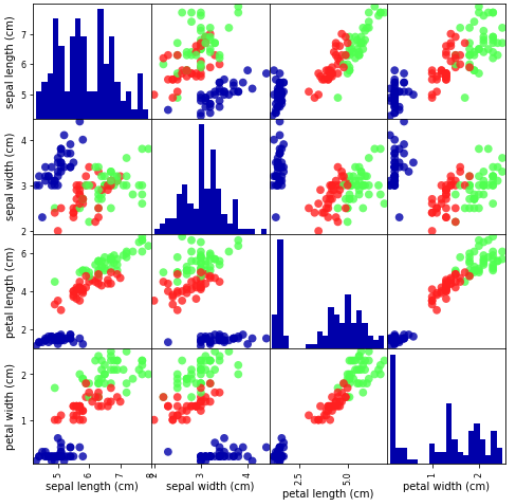
```
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)
```

```
X_train shape: (112, 4)
y_train shape: (112,)
X_test shape: (38, 4)
y_test shape: (38,)
```

Note: there are several problems with this approach that we will discuss later:

- Why 75%? Are there better ways to split?
- What if one random split yields different models than another?
- What if all examples of one class all end up in the training/test set?

Looking at your data (with pandas)



Fitting a model

The first model we'll build is called k-Nearest Neighbor, or kNN. More about that soon.
kNN is included in `sklearn.neighbors`, so let's build our first model

```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
```

Out[7]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=1, p=2,
           weights='uniform')
```

Making predictions

Let's create a new example and ask the kNN model to classify it

```
X_new = np.array([[5, 2.9, 1, 0.2]])
prediction = knn.predict(X_new)
```

```
Prediction: [0]
Predicted target name: ['setosa']
```

Evaluating the model

Feeding all test examples to the model yields all predictions

```
y_pred = knn.predict(X_test)
```

```
Test set predictions:
 [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1
0
 2]
```

We can now just count what percentage was correct, or use `score`

```
print("Score: {:.2f}".format(np.mean(y_pred == y_test)))
print("Score: {:.2f}".format(knn.score(X_test, y_test) ))
```

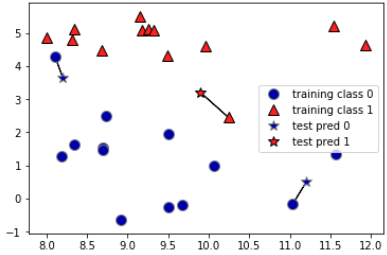
```
Score: 0.97
Score: 0.97
```

k-Nearest Neighbor

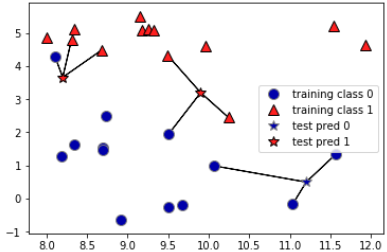
- Building the model consists only of storing the training dataset.
- To make a prediction, the algorithm finds the *k* closest data points in the training dataset
    - Classification: predict the most frequent class of the k neighbors
    - Regression: predict the average of the values of the k neighbors
    - Both can be weighted by the distance to each neighbor
- Main hyper-parameters:
    - Number of neighbors (k). Acts as a regularizer.
    - Choice of distance function (e.g. Euclidean)
    - Weighting scheme (uniform, distance,...)
- Model:
    - Representation: Store training examples (e.g. in KD-tree)
    - Typical loss functions:
        - Classification: Accuracy (Zero-One Loss)
        - Regression: Root mean squared error
    - Optimization: None (no model parameters to tune)

k-Nearest Neighbor Classification

for k=1: return the class of the nearest neighbor



for k>1: do a vote and return the majority (or a confidence value for each class)



Let's build a kNN model for this dataset (called 'Forge')

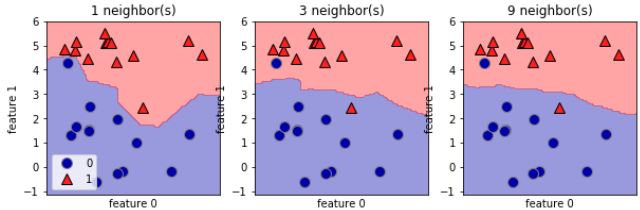
```
X, y = mglearn.datasets.make_forge()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0
)
clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(X_train, y_train)
```

Out[13]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=3, p=2,
           weights='uniform')
```

```
Test set accuracy: 0.86
```

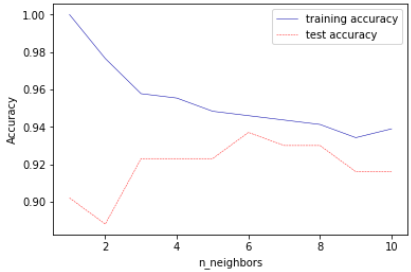
Analysis

We can plot the prediction for each possible input to see the *decision boundary*



Using few neighbors corresponds to high model complexity (left), and using many neighbors corresponds to low model complexity and smoother decision boundary (right).

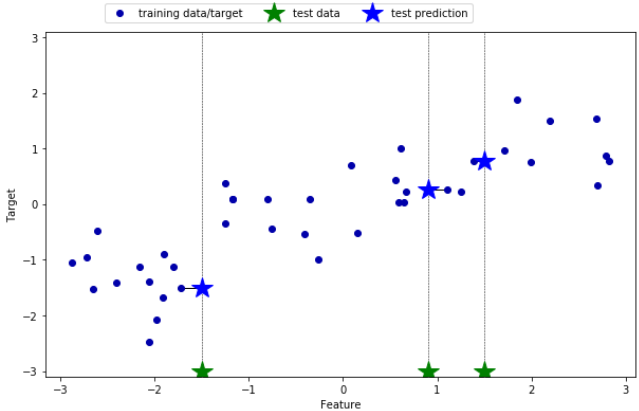
We can more directly measure the effect on the training and test error on a larger dataset (breast_cancer)



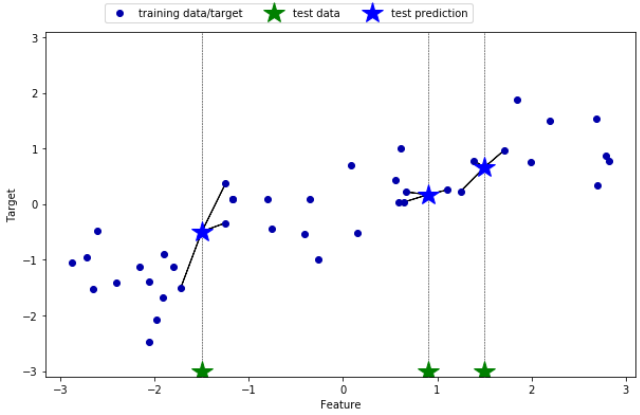
For small numbers of neighbors, the model is too complex, and overfits the training data. As more neighbors are considered, the model becomes simpler and the training accuracy drops, yet the test accuracy increases, up to a point. After about 8 neighbors, the model starts becoming too simple (underfits) and the test accuracy drops again.

k-Neighbors Regression

for k=1: return the target value of the nearest neighbor



for k>1: return the *mean* of the target values of the *k* nearest neighbors



To do regression, simply use `KNeighborsRegressor` instead

```
X, y = mglearn.datasets.make_wave(n_samples=40)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0
)
reg = KNeighborsRegressor(n_neighbors=3)
reg.fit(X_train, y_train)
```

Out[19]:
```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
          metric_params=None, n_jobs=None, n_neighbors=3, p=2,
          weights='uniform')
```

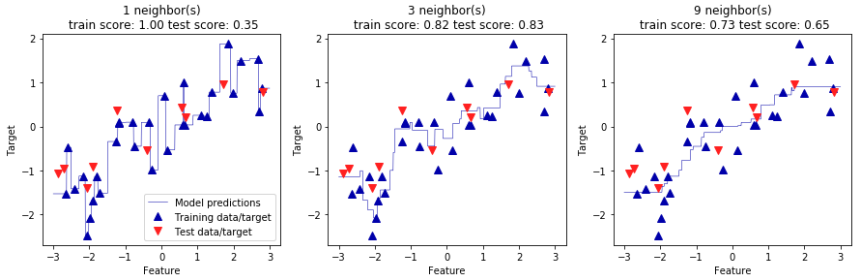
The default scoring function for regression models is R^2. It will be discussed later. the optimal value is 1. Negative values mean the predictions are worse than just predicting the mean.

```
Test set predictions:
[-0.054  0.357  1.137 -1.894 -1.139 -1.631  0.357  0.912 -0.447 -1.139]

Test set R^2: 0.83
```

Analysis

We can again output the predictions for each possible input, for different values of k.



We see that again, a small k leads to an overly complex (overfitting) model, while a larger k yields a smoother fit.

kNN: Strengths, weaknesses and parameters

- Easy to understand, works well in many settings
- Training is very fast, predicting is slow for large datasets
- Bad at high-dimensional and sparse data (curse of dimensionality)

Summary

- We've covered the main machine learning concepts
- We used scikit-learn to build a first model
- We met our first algorithm (kNN)