

## Combining OR and Data Science

**Summer Term 2022**

### 7. Algorithm Selection

**J-Prof. Dr. Michael Römer, Till Porrmann**  
**Decision Analytics Group | Bielefeld University**

## Combining OR and Data Science: The Two Parts of the Course

In **part I**, we dealt with **combining OR and DS for decision-making under uncertainty**:

- using DS for supporting **modeling** uncertainty in OR approaches (e.g. stochastic programming)

In **part II**, we dealt with **combining OR and DS for selecting and configuring (OR) algorithms**:

- using DS for supporting / improving the **solution** process of OR models

## Part II: Algorithm Selection and Configuration

For many complex combinatorial optimization problems, we have

- many different algorithms that can be used to solve them and
- these algorithms exhibit many configurable parameters

In general,

- there is not a single algorithm that work best for each problem (instance)
- for a given algorithm, there is not a single parameter configuration that is best for each problem (instance)

In this part of the course, we learn how to use Machine Learning approaches for

- selecting a good algorithm
- finding a good parameter configuration

for a given problem (instance) based on problem / instance features

## This Week: Algorithm Selection

### **Introduction to Algorithm Selection**

- Introduction and motivation
- Introducing a case study from a former algorithm selection competition
- Analyzing the data and motivating instance-specific algorithm selection

### **Algorithm Selection using Unsupervised Machine Learning**

- Using features for algorithm selection
- Short review: Unsupervised learning
- Cluster-and-select: Using unsupervised learning for feature-based algorithm selection

### **Evaluating Algorithm Selection Approaches Using Cross Validation**

#### **Algorithm Selection using Classification: Predicting the Best Approach**

- brief review of ML approaches for classification
- classification for algorithm selection

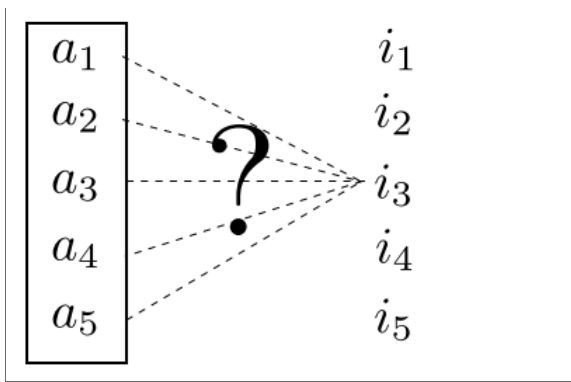
#### **Algorithm Selection using Regression: Predicting the Algorithm Performance**

- brief review of ML approaches for regression
- predicting algorithm runtime and performance using regression
- selecting algorithms based on these predictions

## This Week: Algorithm Selection

- Introduction to Algorithm Selection
- Algorithm Selection using Unsupervised Machine Learning (Clustering)
- Evaluating Algorithm Selection Approaches Using Cross Validation
- Algorithm Selection using Classification: Predicting the Best Approach
- Algorithm Selection using Regression: Predicting the Algorithm Performance

## Algorithm Selection: The Task



Given a set of algorithms  $a_1, \dots, a_n$  for a hard optimization problem, which one will solve a given problem instance, e.g.  $i_3$ , the fastest?

**In general, there is no free lunch:**

- there is no algorithm dominating all others, but:
- for different problems and different problem instances, different algorithms may be best

...and the difference between the runtime of the algorithms is often practically significant and sometimes huge

# Introducing our Algorithm Selection Case Study

## Case Study Example: MaxSat Data Set

### Algorithm Selection Library (ASLib)

- a library of data sets for algorithm selection research
- data sets for different problem classes, each set includes:
  - instance information, in particular instance features
  - runtime information for each instance and various algorithms
- we will use this data set to illustrate how to select the best algorithm!

..we somewhat pretend that we have to solve similar problems in the future and that we can use the given data to "train" our selection approach

### MaxSAT data set from ASLib

- MaxSAT (Maximum Satisfiability) is a problem class in computer science that can be used e.g. for configuration problems
  - there are various algorithms for the MaxSAT problem
- our data set has **601 instances** each of which was solved with **19 different algorithms**
- **algorithm performance** is measured using the **PAR10 score**:
  - each algorithm is run at most **1800 seconds**
  - in case of **timeout**, the PAR10 score is **10 \* 1800 seconds**



## A First Look at the Data

We first read the runtime data (that is the PAR10 score for each combination of instance and algorithm) from a csv file and give the columns the appropriate names:

```
In [3]: df_runs = pd.read_csv("algorithm_runs.csv", header = None)
df_runs.columns = ['instance_id', 'repetition', 'algorithm', 'PAR10', 'runstatus']
```

We then drop two columns that are not needed for our purpose. This means that the remaining columns contain:

- the instance id
- the the name of the algorithm
- the PAR10 score

```
In [4]: df_runs.drop(columns=['repetition', 'runstatus'], inplace=True, errors='ignore') # inplace=True means that the data frame itself is changed
df_runs.head()
```

Out[4]:

	instance_id	algorithm	PAR10
0	mul_8_11.wcnf	CCEHC2akms	18000.0
1	mul_8_11.wcnf	ahms-1.70	18000.0
2	mul_8_11.wcnf	LMHS-2016	18000.0
3	mul_8_11.wcnf	Optiriss6	18000.0
4	mul_8_11.wcnf	WPM3-2015-co	18000.0

# Inspecting the MaxSat Data Set

- we now extract the instances and the algorithms from the data set

```
In [5]: instances = df_runs["instance_id"].unique()
algorithms = df_runs["algorithm"].unique()
print(f" We have {len(instances)} instances and {len(algorithms)} algorithms")
```

We have 601 instances and 19 algorithms

- let us compute a data frame with the best algorithm run per instance

```
In [6]: min_row_idx = df_runs.groupby("instance_id")["PAR10"].idxmin() # We group all entries with the same instance_id and use ["PAR10"].idxmin() to return the row id with the lowest PAR10 value
df_best_runs = df_runs.iloc[min_row_idx] # We only select those rows that contain the minimum PAR10 value for each instance (iloc takes the row id as input)
df_best_runs.head()
```

Out[6]:

	instance_id	algorithm	PAR10
8179	10tree110p.wcnf	mscg2015a	1.37
8654	10tree115p.wcnf	mscg2015a	0.90
8198	10tree120p.wcnf	mscg2015a	4.88
8672	10tree125p.wcnf	mscg2015b	3.20
8216	10tree130p.wcnf	mscg2015b	25.09

How many algorithms performed best on at least one instance?

```
In [7]: len(df_best_runs["algorithm"].unique())
```

Out[7]:

16

## Setting up a "Virtual" Experiment: Splitting the Data

To simulate an algorithm selection experiment, we split the data as follows:

- 2/3 of the instances (and their algorithm runs) are used as "training data" used for training a selection approach / policy
- the remaining 1/3 of the instances are used for evaluation purposes, that is to see how good the selection approach works on "unknown" instances
- we can use the `train_test_split` function from scikit-learn to split the set of instances accordingly:

```
In [8]:  
from sklearn.model_selection import train_test_split  
train_instances, test_instances = train_test_split(instances, test_size=0.33, random_state=11)
```

- now, we use these instance sets to split the two dataframes containing the runtime data:

```
In [9]:  
def get_data_train_test(df_data, train_instances, test_instances):  
    return df_data[df_data["instance_id"].isin(train_instances)], df_data[df_data["instance_id"].isin(test_instances)]  
  
df_runs_train, df_runs_test = get_data_train_test(df_runs, train_instances, test_instances)  
df_best_runs_train, df_best_runs_test = get_data_train_test(df_best_runs, train_instances, test_instances)
```

# Selecting a Single "Best" Algorithm for All Instances?

Let us start simple: What if we were only allowed to choose a single algorithm that we're going to apply to the test set.

We will consider two variants:

- selecting the algorithm that was the best most often in the training data

```
In [10]: def get_algorithm_with_most_best_results(df_best_runs_train):  
  
df_number_times_best = df_best_runs_train.groupby("algorithm")["instance_id"].size().reset_index()  
  
return df_number_times_best.iloc[df_number_times_best["instance_id"].idxmax()]["algorithm"]  
get_algorithm_with_most_best_results(df_best_runs_train)
```

Out[10]:

'mscg2015a'

- selecting the algorithm that with the best average performance in the training data

```
In [11]: def get_algorithm_with_best_average_performance(df_runs_train):  
  
average_par10_train = df_runs_train.groupby("algorithm")["PAR10"].mean().reset_index()  
  
min_row_idx = average_par10_train["PAR10"].idxmin() # We use ["PAR10"].idxmin() to return the row id with the lowest PAR10 value  
return average_par10_train.iloc[min_row_idx]["algorithm"]  
get_algorithm_with_best_average_performance(df_runs_train)
```

Out[11]:

'WPM3-2015-co'

## Selecting a Single "Best" Algorithm: Results of the Evaluation

To see which approach performs better, let us compute the **average performance** on the **test instances**:  
*Note: In a realistic setting, we would perform the evaluation runs with unknown instances. Here, we have all the runtime data available from "historical" runs*

- performance of the most-often-best from the training data

```
In [12]:
df_runs_test_selected_algorithm_most_best = df_runs_test[df_runs_test['algorithm']==get_algorithm_with_most_best_results(df_best_runs_train)]
df_runs_test_selected_algorithm_most_best['PAR10'].mean()
```

Out[12]:

2953.7805025125626

- selecting the algorithm with the best average performance on the training data

```
In [13]:
df_runs_test_selected_algorithm_best_avg_performance = df_runs_test[df_runs_test['algorithm']==get_algorithm_with_best_average_performance(df_runs_train)]
df_runs_test_selected_algorithm_best_avg_performance['PAR10'].mean()
```

Out[13]:

2177.582462311558

## Putting the Results into Perspective: Two Benchmark Values

To see how good our selection was, let us consider two bounds / benchmark values:

- the average performance of all algorithms on the test set (we should be better than that)

```
In [14]: performance_average_test = df_runs_test["PAR10"].mean()  
performance_average_test
```

Out[14]:

7274.185416556466

- the average performance if we would select the best algorithm for each instance in the test set (we assume that we have an "oracle" telling us the best approach) (we cannot do better than that)

```
In [15]: performance_oracle_test = df_best_runs_test["PAR10"].mean()  
performance_oracle_test
```

Out[15]:

1400.7546231155777

## Collecting all Results in One Data Frame

- Let us now collect all results so far in a single data frame
- we will continue growing this data frame during this meeting!

```
In [16]:
df_results = pd.DataFrame(columns=['perf_test'])
df_results.loc['average_performance_test_set'] = performance_average_test
df_results.loc['oracle'] = performance_oracle_test
df_results.loc['single_best(best_avg)'] = df_runs_test_selected_algorithm_best_avg_performance['PARI0'].mean()
df_results.loc['single_best(most_often_best)'] = df_runs_test_selected_algorithm_most_best['PARI0'].mean()

df_results.style.set_table_attributes('style="font-size: 30px"')
df_results
```

Out[16]:

	perf_test
average_performance_test_set	7274.185417
oracle	1400.754623
single_best(best_avg)	2177.582462
single_best(most_often_best)	2953.780503

**Observe:** *These results depend on the random train-test split of the instances. To reproduce the results, use the same random\_state as used for the split in this notebook!*

## Exercises:

- See whether changing the train-test-split (by changing the parameter "random\_state" in the train-test split function) affects the selection of the best algorithms as well as the overall results (the original value in the uploaded notebook was 11)
- Which algorithm is most often best for the test instances? Is it the same as the one being best most often for the training set?



# Instance Features for Algorithm Selection

Key Idea: Use Instance Features for Selecting the Best Algorithm

Can't we do better? Yes, by making a "tailored" selection that considers instance properties!

**Assumption:**

Instances "prefer" different solvers/algorithms due to the variations in their structure

**Idea** Represent the structure of instances through a **feature vector** that we can use for **machine learning**

**Goal:** Use Machine Learning to support the selection of algorithms based on the instance features

## Examples for Instance Features: Traveling Salesperson Problem

Consider the **Traveling Salesperson Problem** (TSP):

### **Example Features:**

- Number of nodes
- Statistics about the distance matrix (mean, std. dev., etc.)
- Features describing the node distribution
  - Statistics about the node degree of the spanning tree (incoming, outgoing)
  - ..

Pihera, J., & Musliu, N. (2014). Application of Machine Learning to Algorithm Selection for TSP.

## Features for Broader Problem Classes

Consider a broader problem class: **Mixed Integer Programming** (MIP)

**Example features:**

- Number of variables, Number of constraints
- Number of nonzeros
- Percentage of integer, binary and continuous variables
- Number and percentage of certain constraint types
- Features from the starting of the solution process, e.g. cuts of certain types
- ..

Georges, A. et al. (2018): Feature-Based Algorithm Selection for Mixed Integer Programming

## Instance Features for the MaxSAT Data Set

- the MaxSAT data set comes with a file with 37 features per instance
- the semantics of the features are unknown to us (there names are f\_1, f\_2, etc.)
- the values are normalized to take values between 0 and 1

```
In [17]:
df_instance_features = pd.read_csv("feature_values.csv", header = None)
df_instance_features.columns = ['instance_id', 'repetition'] + [f'f_{i}' for i in range(df_instance_features.shape[1] - 2)]
df_instance_features.drop(columns='repetition',inplace=True,errors='ignore')
df_instance_features.head()
```

Out[17]:

	instance_id	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	...	f_27	f_28	
0	cnf.13.p.9.wcnf	87124.0	393887.0	0.01198	1.0	0.0	1.0	1.0	0.22119	0.00003	...	1.0	0.03386	0.7819
1	cnf.19.p.10.wcnf	132880.0	600905.0	0.01198	1.0	0.0	1.0	1.0	0.22113	0.00002	...	1.0	0.03374	0.7819
2	cnf.12.t.9.wcnf	74960.0	341160.0	0.01219	1.0	0.0	1.0	1.0	0.21972	0.00003	...	1.0	0.03670	0.7805
3	cnf.8.p.9.wcnf	51700.0	233615.0	0.01199	1.0	0.0	1.0	1.0	0.22130	0.00004	...	1.0	0.03404	0.7818
4	cnf.14.p.9.wcnf	88600.0	400577.0	0.01198	1.0	0.0	1.0	1.0	0.22118	0.00003	...	1.0	0.03386	0.7819

We also split the features data set:

```
In [18]:
df_instance_features_train, df_instance_features_test = get_data_train_test(df_instance_features, train_instances, test_instances)
```

## 2. Clustering Instances for Algorithm Selection

## (Unsupervised) Clustering for Algorithm Selection

**Question:** How can we use feature information to support the (instance-specific) selection of algorithms?

**A first idea: Cluster-and-Select**

- **Cluster** similar instances in the training data based on feature information
- **Select** the best algorithm, for each cluster e.g. the one with lowest average PAR10 value in the training set

**For an "unseen" instance**, we can then:

- determine the cluster it belongs to
- choose the algorithm for that was selected for the cluster in the training step

## Machine Learning: Unsupervised vs Supervised Learning

### **Unsupervised Learning**

- given a vector  $\mathbf{x}_i$  input data features for each observation / instance  $i$
- learn something interesting such as relations or clusters → this video

### **Supervised Learning:**

- given a vector  $\mathbf{x}_i$  input data features
- and a label  $y_i$  for each  $i$
- learn the to predict  $y_j$  for an unlabeled feature vector  $\mathbf{x}_j$  → next week

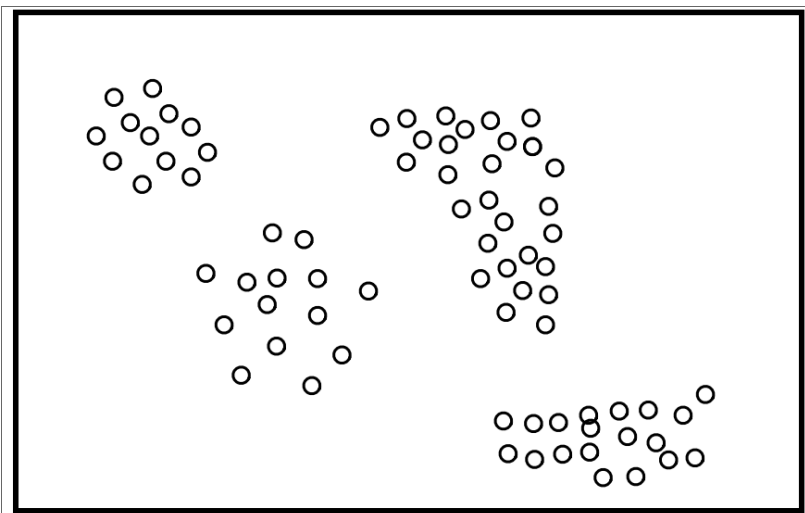


# How Does Clustering Work?

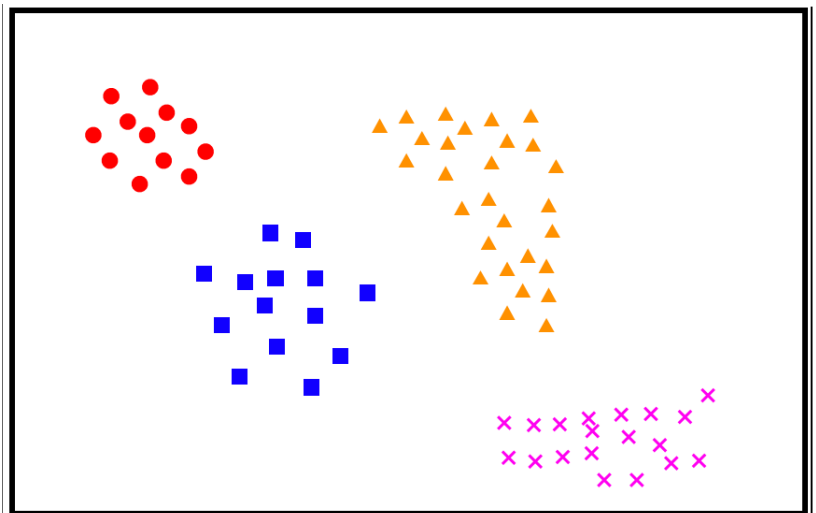
## Basic Idea

- Group data points that are "similar"

Raw Data Points



Clustered Data Points

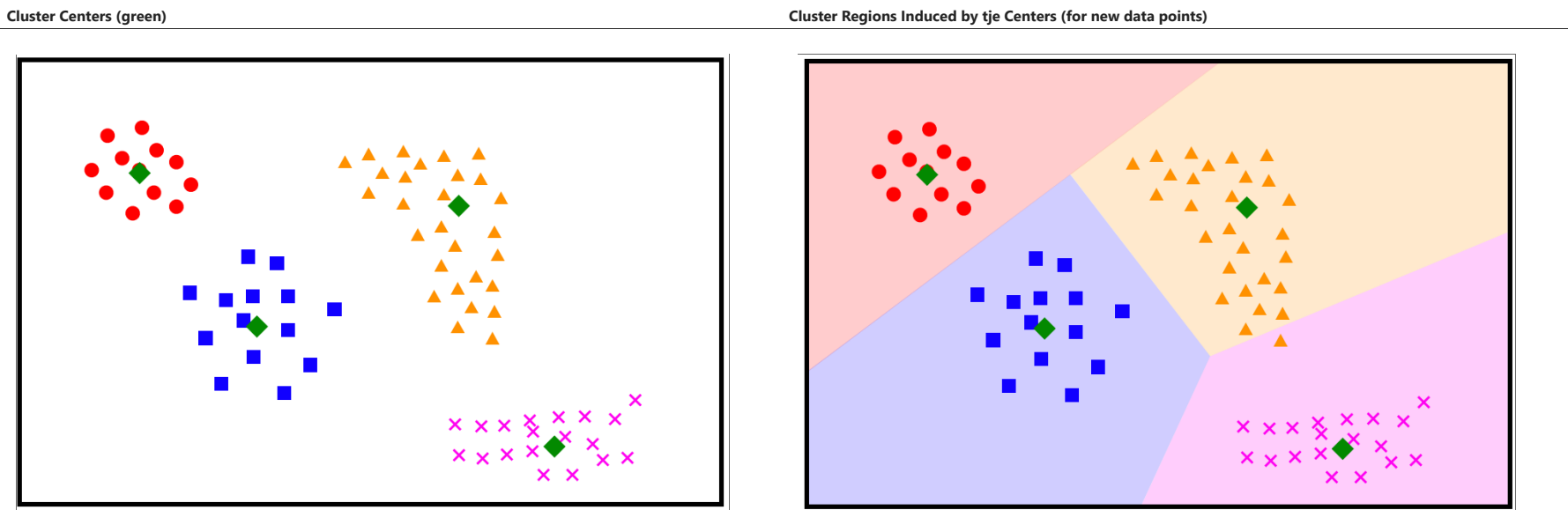


# How Does Clustering Work?: $k$ -Means Clustering

$k$ -Means clustering is one of the best-known clustering approaches

**Key Idea:**

- Form  $k$  **clusters** using some distance metric, for example the Euclidean distance
- Find an optimal set of  $k$  cluster **centers**



## $k$ -Means Optimization Problem

The problem of finding the best  $k$  cluster centers can be stated as an optimization problem:

$$\arg \min_{\mathbf{S}} \sum_{i=1}^k \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \mu_i\|^2$$

where:

- $\mathbf{S}$  is a vector of  $k$  sets  $S_1, \dots, S_k$
- $\mathbf{x}$  is the input data that is clustered into sets  $S_1, \dots, S_k$
- $\mu_i$  is the mean of the points in  $S_i$

Unfortunately, this problem is very difficult to solve to optimality.

- ML software usually uses iterative heuristic algorithms (not discussed here)
- To predict the cluster of a new / unknown data point, we simply pick the cluster with the closest distance from the cluster mean to the new data point

# (Unsupervised) Clustering for Algorithm Selection

Let us see clustering in action for the **cluster-and-select** approach sketched above.

Step 1: **Cluster** similar instances in the training data based on feature information

```
In [19]:
from sklearn.cluster import KMeans

clu = KMeans(n_clusters=10, random_state=1) # the clustering algorithm / model that we will use. Here, you may also take another one.

prd = clu.fit_predict(df_instance_features_train.loc[:, "f_0":]) # Create clusters based on training instances and predict the cluster for each of the training instances

df_clustered_instances_train = df_instance_features_train.assign(cluster=prd)[['instance_id', 'cluster']] # create a new dataframe that contains the instances and their assigned cluster as columns
df_clustered_instances_train.head()
```

Out[19]:

	instance_id	cluster
1	cnf.19.p.10.wcnf	6
4	cnf.14.p.9.wcnf	6
5	cnf.20.p.10.wcnf	6
8	cnf.15.p.9.wcnf	6
12	cnf.11.p.9.wcnf	9

# (Unsupervised) Clustering for Algorithm Selection

Step 2: **Select** the best algorithm, for each cluster e.g.the one with lowest average PAR10 value in the training set.

```
In [20]:
    ## add the cluster to the runs data set
df_runs_train_with_clusters = pd.merge(df_runs_train, df_clustered_instances_train, left_on='instance_id', right_on='instance_id')

##compute the average performance of each algorithm per cluster
df_average_performance_clusters = df_runs_train_with_clusters.groupby(['cluster','algorithm'])["PAR10"].mean().reset_index()

##select the algorithm with the best average performance
df_algorithm_for_cluster_best_avg_performance = df_average_performance_clusters.iloc[df_average_performance_clusters.groupby('cluster')['PAR10'].idxmin()]

## only use the two columns
df_algorithm_for_cluster_best_avg_performance = df_algorithm_for_cluster_best_avg_performance[['cluster','algorithm']]

df_algorithm_for_cluster_best_avg_performance
```

Out[20]:

	cluster	algorithm
14	0	maxhs-b
23	1	Open-WBO15
55	2	mscg2015a
65	3	QMaxSAT16UC
91	4	maxino16-c10
99	5	Open-WBO15
132	6	mscg2015b
149	7	maxino16-dis
158	8	Optiriss6
182	9	WPM3-2015-co

# (Unsupervised) Clustering for Algorithm Selection

For an "unseen" instance, we can then:

- determine the cluster it belongs to

```
In [21]: # predict the clusters for the test instances
prd = clu.predict(df_instance_features_test.loc[:, "f_0":])

# Add the assigned cluster to df_instances as a new column
df_clustered_instances_test = df_instance_features_test.assign(cluster=prd[['instance_id', 'cluster']])

df_clustered_instances_test.head()
```

Out[21]:

	instance_id	cluster
0	cnf.13.p.9.wcnf	6
2	cnf.12.t.9.wcnf	6
3	cnf.8.p.9.wcnf	9
6	cnf.13.p.8.wcnf	6
7	cnf.17.d.10.wcnf	6

... and choose the best algorithm for that was selected for the cluster in the training step

```
In [22]: df_selected_algorithm_test_instances = pd.merge(df_clustered_instances_test, df_algorithm_for_cluster_best_avg_performance, left_on='cluster', right_on='cluster')

df_selected_algorithm_test_instances.head()
```

Out[22]:

	instance_id	cluster	algorithm
0	cnf.13.p.9.wcnf	6	mscg2015b
1	cnf.12.t.9.wcnf	6	mscg2015b
2	cnf.13.p.8.wcnf	6	mscg2015b
3	cnf.17.d.10.wcnf	6	mscg2015b
4	cnf.20.d.9.wcnf	6	mscg2015b

# How Well Does this Selection Perform?

Let us now evaluate this clustering-based selection on the test data set:

- (we write a little helper function that we can re-use later)

```
In [23]: def evaluate_selected_algorithms(df_runs_test, df_selected_algorithms):
df_runs_with_selected_algorithms = pd.merge(df_runs_test, df_selected_algorithms, left_on=['instance_id', 'algorithm'], right_on=['instance_id', 'algorithm'])
return df_runs_with_selected_algorithms['PAR10'].mean()

evaluate_selected_algorithms(df_runs_test, df_selected_algorithm_test_instances)
```

Out[23]:

1995.3791457286434

- and compare the performance to our previous selection and to the benchmark values

```
In [24]: df_results.loc['cluster_and_select'] = evaluate_selected_algorithms(df_runs_test, df_selected_algorithm_test_instances)
df_results
```

Out[24]:

	perf_test
average_performance_test_set	7274.185417
oracle	1400.754623
single_best(best_avg)	2177.582462
single_best(most_often_best)	2953.780503
cluster_and_select	1995.379146

## Key Results so far

We considered two main approaches:

- selecting a single best approach and
- an unsupervised learning approach ("cluster-and-select") and evaluated it using a single train/test split.

In addition, we computed two benchmark values (for the same split):

- average performance of all algorithms on the test set
- best possible performance assuming that we have an "oracle" always selecting the best algorithm for each instance:



## Exercise: Play with the Clustering Methods

We more or less arbitrarily chose  $k$ -Means and its parameter (e.g.  $k = 10$ ).  
Experiment with the clustering approach and see how this affects the results:

- change the number  $k$  of clusters used for  $k$ -Means
- change the clustering method. For an overview of possible clustring approaches in scikit-learn, see:
  - <https://scikit-learn.org/stable/modules/clustering.html> (overview of the different methods)
  - <https://scikit-learn.org/stable/modules/classes.html#module-sklearn.cluster> (functions and paramters to call)

Note that in general, you can reuse the the code above and only replace the `KMeans` in the following lines:

- `from sklearn.cluster import KMeans`
- `clu = KMeans(n_clusters=10, random_state=1)`

with other methods, e.g. with `FeatureAgglomeration`. Only note that only for some of the approaches, the number of clusters has to be specified in advance.

```
In [25]: df_results
```

Out[25]:

	perf_test
average_performance_test_set	7274.185417
oracle	1400.754623
single_best(best_avg)	2177.582462
single_best(most_often_best)	2953.780503
cluster_and_select	1995.379146

### 3. Cross Validation

How valid are our results?

**How valid are these selection results?**

- do the results of evaluating a selection using a single train/test split generalize?
- is the clustering-based approach superior to the single best approach in general or only for the given train/test combination?

**Exercise:** Try a different train/test split (by changing the `random_state` parameter in the `train_test_split` function) and compare the result!

Testing different train/test splits is the central idea of **cross validation**:

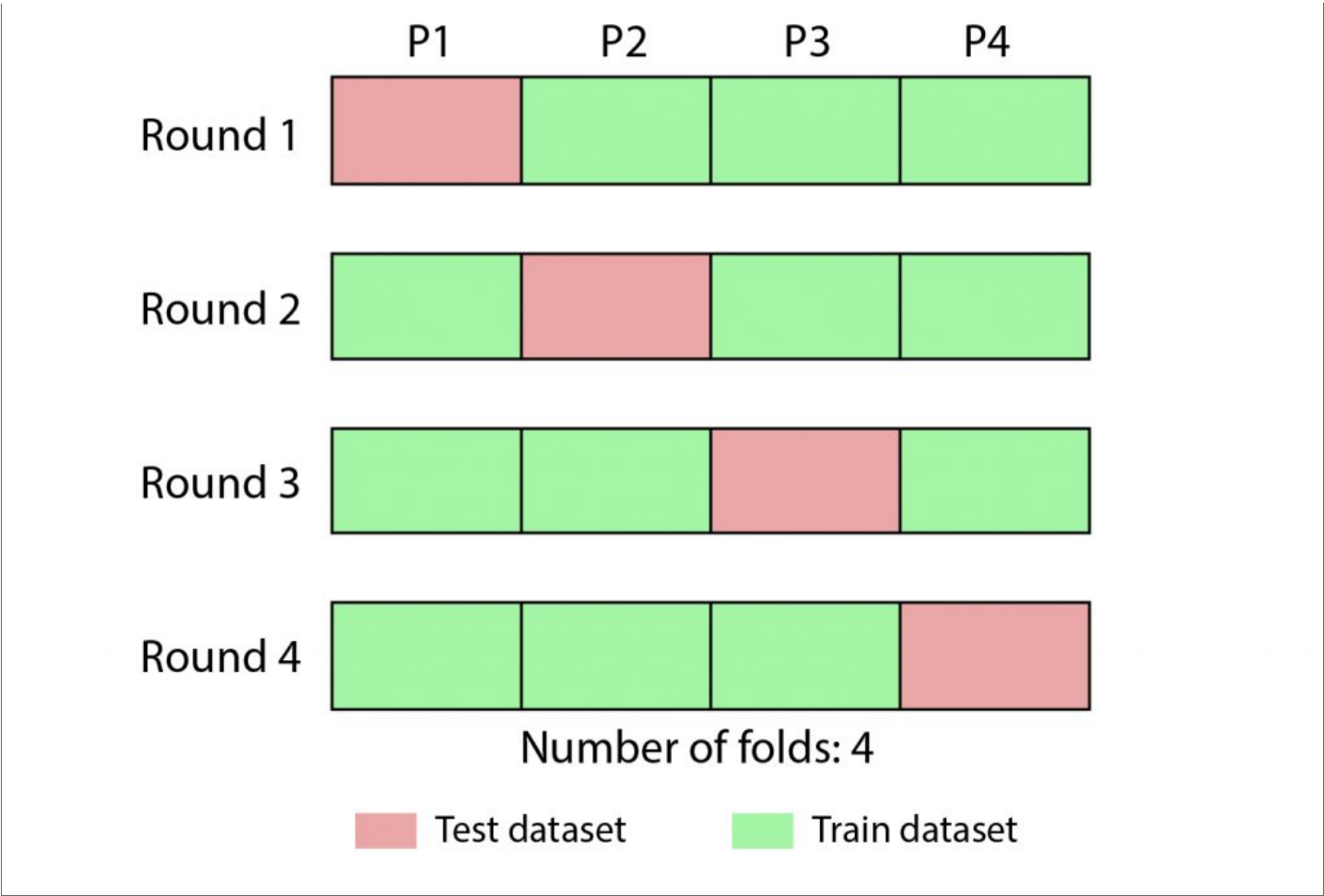
- try different splits (partitions) into train and test data
- and use the average performance of all splits (on all test sets) as performance estimation

.. we will consider cross validation here!

$k$ -fold Cross Validation

**Key Idea:** Partition the data set into  $k$  folds (subsets of equal size)

- in each of the  $k$  rounds 1 fold is used for testing and the rest of the data for training
- model is re-trained in each round
- average performance is used to compare different approaches



## Applying Cross Validation to Algorithm Selection: Key Steps

To evaluate an algorithm selection approach using cross validation:

**For each** of the train-test-splits do the following steps:

1. **train** the selection approach based on training data
2. **select** algorithms for the test set according to the approach
3. **evaluate** average performance on test data and **store** it in the results array

Finally, take the **average of the results array** to obtain a performance estimate from cross validation

## Illustration: Cross Validation for the Single-Best-Approach to Algorithm Selection

- note: scikit-learn provides functionality for different types of cross validation
- here, we use KFold cross validation

```
In [26]: from sklearn.model_selection import KFold

kf = KFold(n_splits=6, shuffle=True, random_state=11) # we use the parameter "shuffle=True" to avoid selecting the subsets according to the order of the instances in the data set
results = []
for train, test in kf.split.instances):
    # split the runs data
    df_runs_train_cv, df_runs_test_cv = get_data_train_test(df_runs, instances[train], instances[test])

    # step 1.: train the selection approach - here: select the best algorithm in the training set
    algo = get_algorithm_with_best_average_performance(df_runs_train_cv)

    # step 2.: select the algorithm
    df_runs_test_selected_algorithm = df_runs_test_cv[df_runs_test_cv['algorithm']==algo]

    # step 3: evaluate on the test set
    results.append(df_runs_test_selected_algorithm['PAR10'].mean())

print(f"Using cross validation, we obtain the following expected performance: {np.mean(results):.02f}")
```

Using cross validation, we obtain the following expected performance: 2461.44

## Making the Implementation of Cross Validation More Generic

**How can we avoid to write the full loop multiple times for different selection approaches??**

- we can try using a function `evaluate_using_cross_validation`

**The problem: The three steps in the inner loop are very different for each approach**

- possible solution: we give `evaluate_using_cross_validation` a **function**  
`evaluate_train_test_split` as a parameter

The resulting function then looks as follows:

```
In [27]: def evaluate_using_cross_validation(evaluate_approach_train_test_split, kf):
results = []
for train, test in kf.split.instances):
    results.append(evaluate_approach_train_test_split(instances[train], instances[test]))
return np.mean(results)
```

To apply that function for the evaluation for a concrete algorithm selection approach, we need to implement an evaluation function that takes the training and the test instances!

# An Evaluation Function for the Single-Best Algorithm Selection Approach

As an example, let's start with the function `evaluate_train_test_split_single_best`:

```
In [28]: def evaluate_train_test_split_single_best (train_instances, test_instances):  
  
# get data  
df_runs_train, df_runs_test = get_data_train_test(df_runs, train_instances, test_instances)  
  
# step 1.: train the selection approach - here: select the best algorithm in the training set  
algo = get_algorithm_with_best_average_performance(df_runs_train)  
  
# step 2.: select  
df_runs_test_selected_algorithm = df_runs_test[df_runs_test['algorithm']==algo]  
  
# step 3: evaluate on the test set  
return df_runs_test_selected_algorithm['PAR10'].mean()
```

Let's try it out (compare to the results we got above):

```
In [29]: evaluate_using_cross_validation(evaluate_train_test_split_single_best, kf)
```

Out[29]:

2461.442754950495



# Evaluation Functions for the Benchmarks

An implementation for the "oracle" benchmark:

```
In [30]:
def evaluate_train_test_split_oracle(train_instances, test_instances):
    df_best_runs_train, df_best_runs_test = get_data_train_test(df_best_runs, train_instances, test_instances)

    return df_best_runs_test['PAR10'].mean()

evaluate_using_cross_validation(evaluate_train_test_split_oracle,kf)
```

Out[30]:

1383.3625869636965

An implementation for the "average" benchmark:

```
In [31]:
def evaluate_train_test_split_average(train_instances, test_instances):
    df_runs_train, df_runs_test = get_data_train_test(df_runs, train_instances, test_instances)

    return df_runs_test['PAR10'].mean()

evaluate_using_cross_validation(evaluate_train_test_split_average,kf)
```

Out[31]:

7161.04845006948

# Cluster-and-Select

- here, we write an evaluation function for the cluster-and-select approach

```
In [32]:
    clustering_model = KMeans(n_clusters=10, random_state=0)

def evaluate_train_test_split_cluster_and_select(train_instances, test_instances):
    df_runs_train, df_runs_test = get_data_train_test(df_runs, train_instances, test_instances)
    df_instance_features_train, df_instance_features_test = get_data_train_test(df_instance_features, train_instances, test_instances)

    # step 1.: train the selection approach -
    prd = clustering_model.fit_predict(df_instance_features_train.loc[:, "f_0":]) # Create clusters based on training instances and predict the cluster for each of the training instances
    df_clustered_instances_train = df_instance_features_train.assign(cluster=prd)[['instance_id','cluster']] # create a new dataframe that contains the instances and their assigned cluster as columns
    df_runs_train_with_clusters = pd.merge(df_runs_train, df_clustered_instances_train, left_on='instance_id', right_on='instance_id') #augment the training set
    df_average_performance_clusters = df_runs_train_with_clusters.groupby(['cluster','algorithm'])["PAR10"].mean().reset_index()
    df_algorithm_for_cluster_best_avg_performance = df_average_performance_clusters.iloc[df_average_performance_clusters.groupby('cluster')['PAR10'].idxmin()]
    df_algorithm_for_cluster_best_avg_performance = df_algorithm_for_cluster_best_avg_performance[['cluster','algorithm']] #only use certain columns

    # step 2.: Select an algorithm for each test set instance
    prd = clustering_model.predict(df_instance_features_test.loc[:, "f_0":]) # Assign test instances to the created clusters
    df_clustered_instances_test = df_instance_features_test.assign(cluster=prd)[['instance_id','cluster']] # Add the assigned cluster to df_instances as a new column and select only the instance and cluster columns
    df_selected_algorithm_test_instances = pd.merge(df_clustered_instances_test, df_algorithm_for_cluster_best_avg_performance, left_on='cluster', right_on='cluster')

    # step 3.: Evaluate the selection
    return evaluate_selected_algorithms(df_runs_test, df_selected_algorithm_test_instances)

evaluate_using_cross_validation(evaluate_train_test_split_cluster_and_select,kf)
```

Out[32]:

2306.603374092409

## All Results in one Data Frame

Now, we can put all results in a single data frame along with the original results from last week that were based on a single train-test split:

```
In [33]:
df_results.loc['oracle','cross_val_perf'] = evaluate_using_cross_validation(evaluate_train_test_split_oracle,kf)
df_results.loc['average_performance_test_set','cross_val_perf'] = evaluate_using_cross_validation(evaluate_train_test_split_average,kf)
df_results.loc['single_best(best_avg)','cross_val_perf'] = evaluate_using_cross_validation(evaluate_train_test_split_single_best,kf)
df_results.loc['cluster_and_select','cross_val_perf'] = evaluate_using_cross_validation(evaluate_train_test_split_cluster_and_select,kf)
df_results
```

Out[33]:

	perf_test	cross_val_perf
average_performance_test_set	7274.185417	7161.048450
oracle	1400.754623	1383.362587
single_best(best_avg)	2177.582462	2461.442755
single_best(most_often_best)	2953.780503	NaN
cluster_and_select	1995.379146	2306.603374

### 3. Classification for Algorithm Selection

## Machine Learning: Unsupervised vs Supervised Learning

### **Unsupervised Learning**

- given a vector  $\boldsymbol{x}_i$  input data features for each observation / instance  $i$
- learn something interesting such as relations or clusters

### **Supervised Learning:**

- given a vector  $\boldsymbol{x}_i$  input data features
- and a label  $y_i$  for each  $i$
- learn the to predict  $y_j$  for an unlabeled feature vector  $\boldsymbol{x}_j$

## Supervised Learning: Classification vs Regression

The most important types of supervised learning are:

- **classification**: predict which category or type (labels are categorical values or classes)
  - our use in algorithm selection: predict the best algorithm
- **regression**: predict how much or how many (labels are numbers)
  - our use in algorithm selection: predict the algorithm performance

Classification

Goal

Given a set of *examples* with some given (categorical) *outcomes*, we wish to learn a *model* to predict an outcome given a new example

(Classical) **Example:** Categorizing Iris species from flower petal information (e.g. size, length, color,...)



## Classification for Algorithm Configuration

One (admittedly naive) approach for using classification for algorithm configuration:

**Use instance feature information for predicting the best algorithm**

- for training, we need the best algorithm for each instance in the test data set
- then, we can use the trained classifier for predicting the best algorithm

**In this meeting, we will consider three different classification approaches:**

- $k$ -nearest neighbor
- decision trees
- random forests



## Predicting the Best Algorithm for the MaxSAT Data Set

- the MaxSAT data set comes with a file with 37 features per instance
- the semantics of the features are unknown to us (there names are f\_1, f\_2, etc.)
- the values are normalized to take values between 0 and 1

For training the prediction of the best approach, we need a data frame with **one row per instance** (in the training set) containing

- the instance features
- the best algorithm

We first create the full data frame for the best runs (including both performance and feature information)

```
In [34]: df_best_runs = pd.merge(df_best_runs, df_instance_features, left_on='instance_id', right_on='instance_id')
df_best_runs.head()
```

Out[34]:

	instance_id	algorithm	PAR10	f_0	f_1	f_2	f_3	f_4	f_5	f_6	...	f_27	f_28	f_29
0	10tree110p.wcnf	mscg2015a	1.37	6642.0	25589.0	0.00821	1.0	0.0	1.0	1.0	...	1.0	0.00993	0.67185
1	10tree115p.wcnf	mscg2015a	0.90	4564.0	16586.0	0.01266	1.0	0.0	1.0	1.0	...	1.0	0.01266	0.65833
2	10tree120p.wcnf	mscg2015a	4.88	6708.0	25809.0	0.00814	1.0	0.0	1.0	1.0	...	1.0	0.00984	0.67635
3	10tree125p.wcnf	mscg2015b	3.20	4604.0	16706.0	0.01257	1.0	0.0	1.0	1.0	...	1.0	0.01257	0.66198
4	10tree130p.wcnf	mscg2015b	25.09	6744.0	25929.0	0.00810	1.0	0.0	1.0	1.0	...	1.0	0.00980	0.67878

5 rows × 40 columns

- we then split the data frame into a train and a test part

```
In [35]: df_best_runs_train, df_best_runs_test = get_data_train_test(df_best_runs, train_instances, test_instances)
```






## $k$ -Nearest Neighbor Classification

### Key Idea:

- store all (or most) of the the labeled training data points
- when encountering a new data point, compute the **distance** to the labeled data points
- determine the  $k$  closest data points (the neighbors)
- select the label that occurs most often among these neighbors

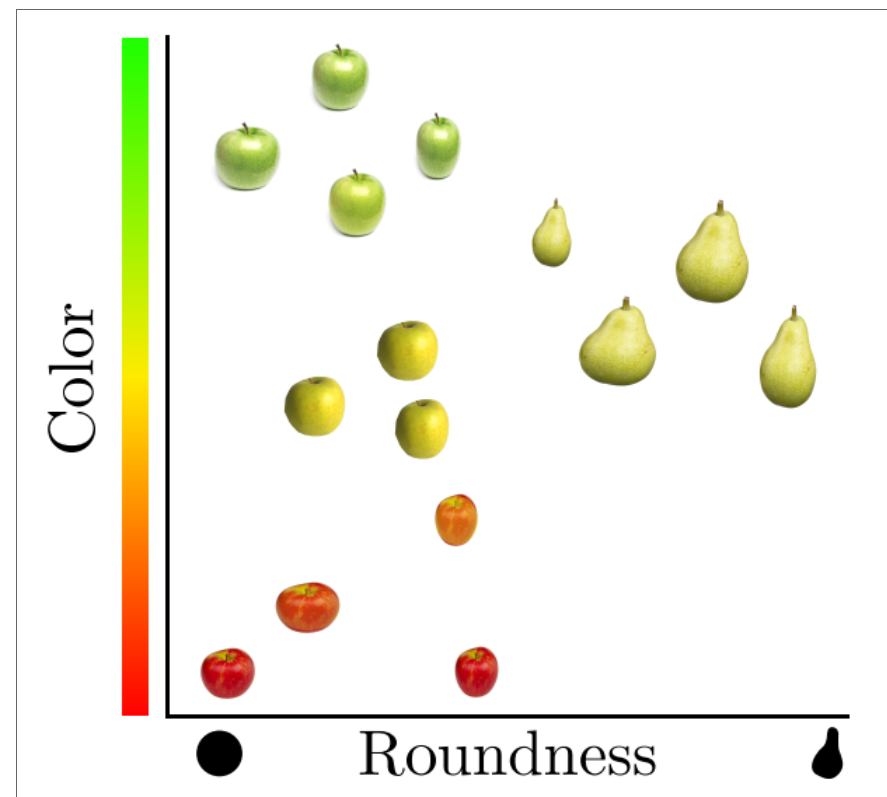
$k$ -Nearest Neighbor: Illustration

Is it a pear or an apple?

Color	Roundness	Picture	Label
Red	1		Apple
Green	0.8		Apple
Yellow	0.3		Pear
Yellow	0.9		Apple
Brown	0.9		???

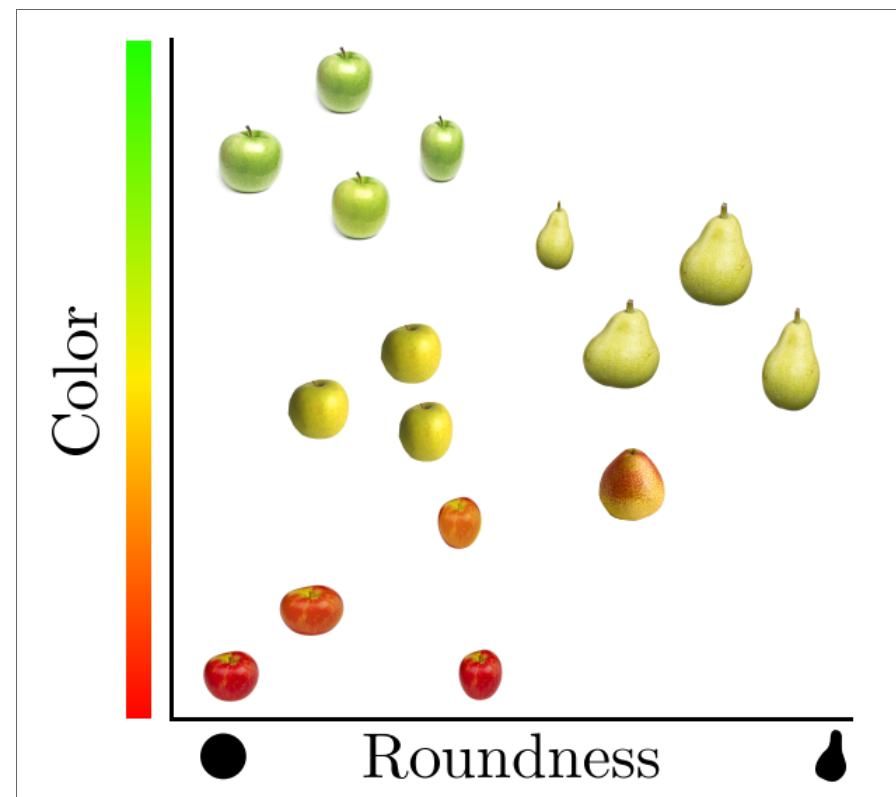
## $k$ -Nearest Neighbor: Illustration

## Is it a pear or an apple?

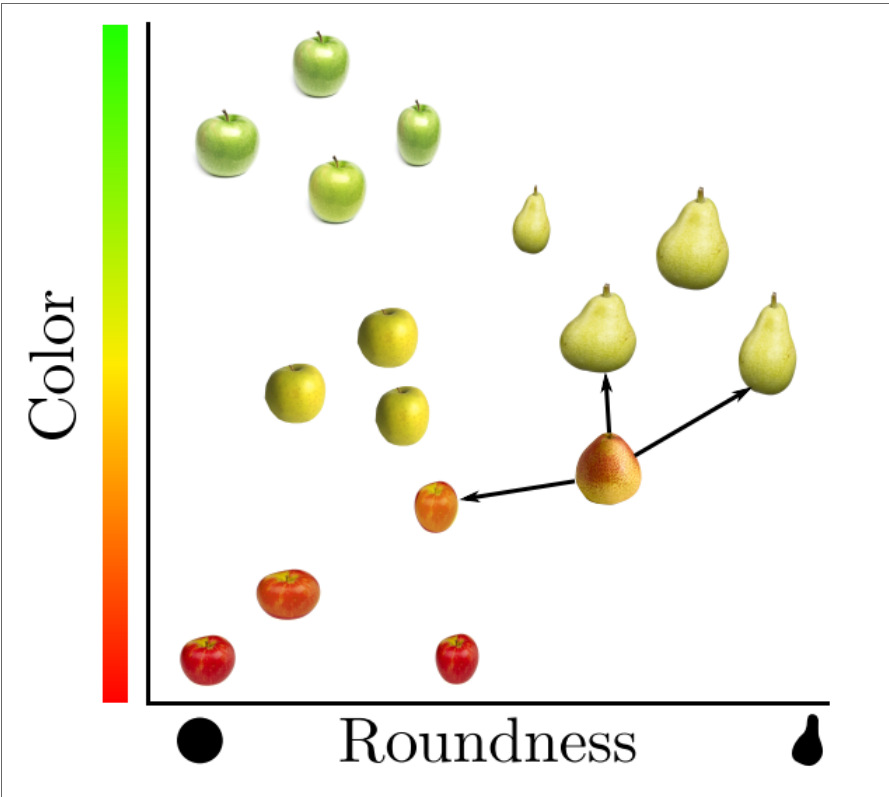


## $k$ -Nearest Neighbor: Illustration

## Is it a pear or an apple?



Is it a pear or an apple?



It is a pear!

## $k$ -Nearest Neighbor Classification for Algorithm Selection

### Let's try on our case study data:

First, we use the  $k$ -Nearest Neighbor Classifier from scikit-learn and train it:

```
In [36]:
from sklearn.neighbors import KNeighborsClassifier

clf = KNeighborsClassifier()

clf.fit(df_best_runs_train.loc[:, "f_0:"], df_best_runs_train["algorithm"]) # df_best_runs_train.loc[:, "f_0:"] is the range of the data frame that contains the feature value

prd = clf.predict(df_best_runs_test.loc[:, "f_0:"])
```

- a quick assessment of the accuracy is not overly promising:
  - (but remember that we have 19 different classes!)

```
In [37]:
accuracy_score(df_best_runs_test["algorithm"], prd)
```

Out[37]:

0.37185929648241206

k-Nearest Neighbor Classification for Algorithm Selection

Let us now predict an algorithm for the test instances:

```
In [38]: prd = clf.predict(df_instance_features_test.loc[:, "f_0:"])
```

```
In [39]: df_selected_algorithm_test_instances = df_instance_features_test.filter(['instance_id'])
```

```
In [40]: df_selected_algorithm_test_instances['algorithm']=prd
```

```
In [41]: print(evaluate_selected_algorithms(df_runs_test, df_selected_algorithm_test_instances))
```

3038.2383919597987

.. and add it to the results

```
In [42]: df_results.loc['KNeighborsClassifier', 'perf_test'] = evaluate_selected_algorithms(df_runs_test, df_selected_algorithm_test_instances)
df_results
```

Out[42]:

	perf_test	cross_val_perf
average_performance_test_set	7274.185417	7161.048450
oracle	1400.754623	1383.362587
single_best(best_avg)	2177.582462	2461.442755
single_best(most_often_best)	2953.780503	NaN
cluster_and_select	1995.379146	2306.603374
KNeighborsClassifier	3038.238392	NaN



Exercise: Evaluate the Performance for the KNN-Classifier with Cross Validation!

- write an evaluation function `evaluate_train_test_split_classification_knn` using the code above
- check if the results from that function with the single train test split are the same as we computed above
- perform a cross validation evaluation using that function
- add the results to the missing cell in the results data frame

In [43]:

Out[43]:

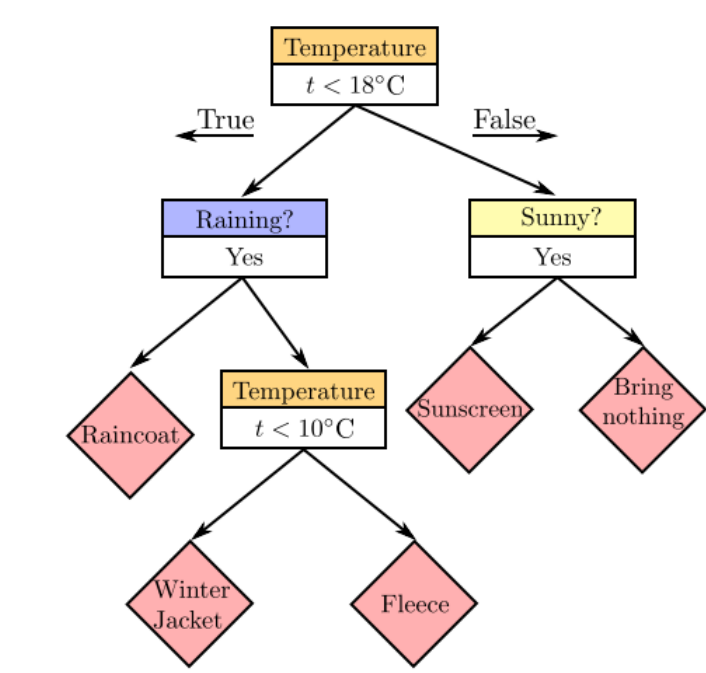
3463.225983828383

# Classification with Decision Trees

Let us now check different classification approaches

**Basic Idea:**

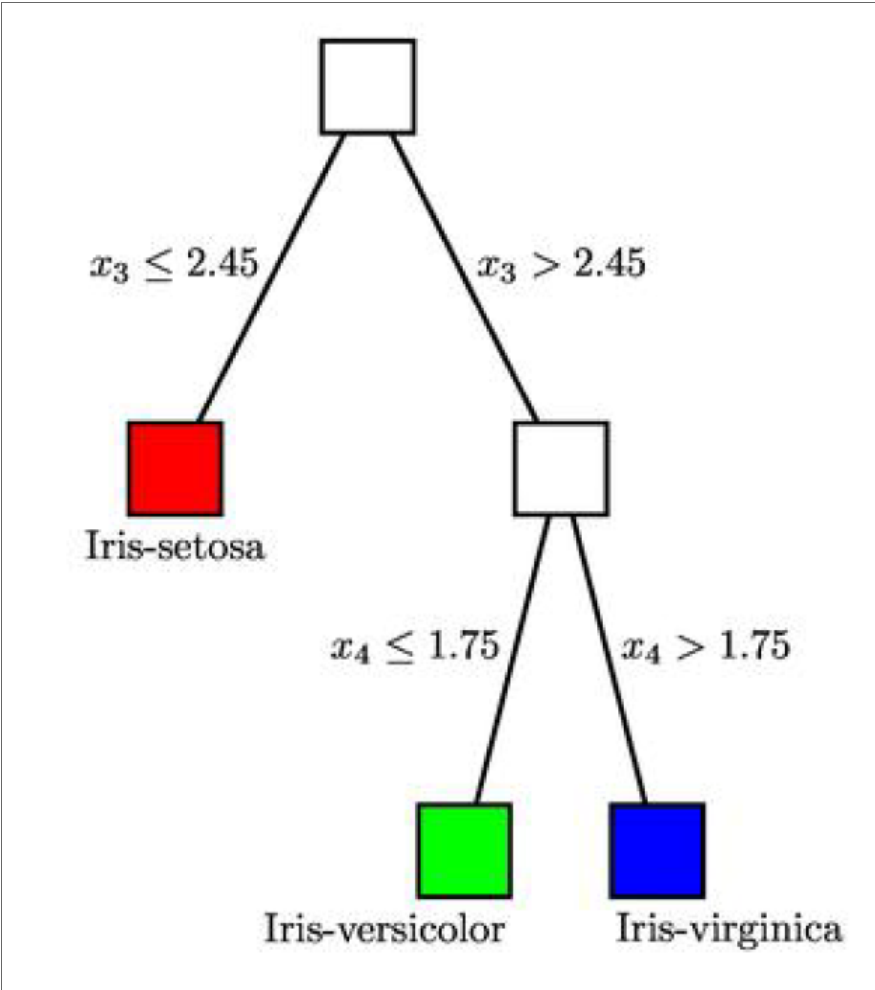
Create a tree of if-statements to determine the label of a training example.  
In general, decision trees look as follows:



# Classification with Decision Trees

**Basic Idea:**

Create a tree of if-statements to determine the label of a training example.  
A (partial) decision tree for the iris classification example may look as follows:



## Using a Decision Tree Classifier in our Case Study

- we use a decision tree from scikit-learn
- and replace a single line in the evaluate\_train\_test\_split function:
  - `clf = DecisionTreeClassifier(max_depth=10, random_state=0)`
- otherwise, the function is identical to the one used for the  $k$ -nearest neighbour approach

```
In [44]:
from sklearn.tree import DecisionTreeClassifier

def evaluate_train_test_split_classification_decision_tree(train_instances, test_instances):

    #prepare the data
    df_instance_features_train, df_instance_features_test = get_data_train_test(df_instance_features, train_instances, test_instances)
    df_runs_train, df_runs_test = get_data_train_test(df_runs, train_instances, test_instances)
    df_best_runs_train, df_best_runs_test = get_data_train_test(df_best_runs, train_instances, test_instances)

    # step 1: Training
    clf = DecisionTreeClassifier(max_depth=10, random_state=0)

    clf.fit(df_best_runs_train.loc[:, "f_0:"], df_best_runs_train["algorithm"])

    # step 2: Selection
    prd = clf.predict(df_instance_features_test.loc[:, "f_0:"])

    df_selected_algorithm_test_instances = df_instance_features_test.filter(['instance_id'])

    df_selected_algorithm_test_instances['algorithm']=prd

    return evaluate_selected_algorithms(df_runs_test, df_selected_algorithm_test_instances)
```

## Results from using the Decision Tree Classifier

- we evaluate the algorithm selection based on predicting the best with the decision tree classifier for the single train-test split and using cross validation
- and directly add the results to our results data frame:

```
In [45]:
df_results.loc['DecisionTreeClassifier', 'perf_test'] = evaluate_train_test_split_classification_decision_tree(train_instances, test_instances)
df_results.loc['DecisionTreeClassifier', 'cross_val_perf'] = evaluate_using_cross_validation(evaluate_train_test_split_classification_decision_tree, kf)
df_results
```

Out[45]:

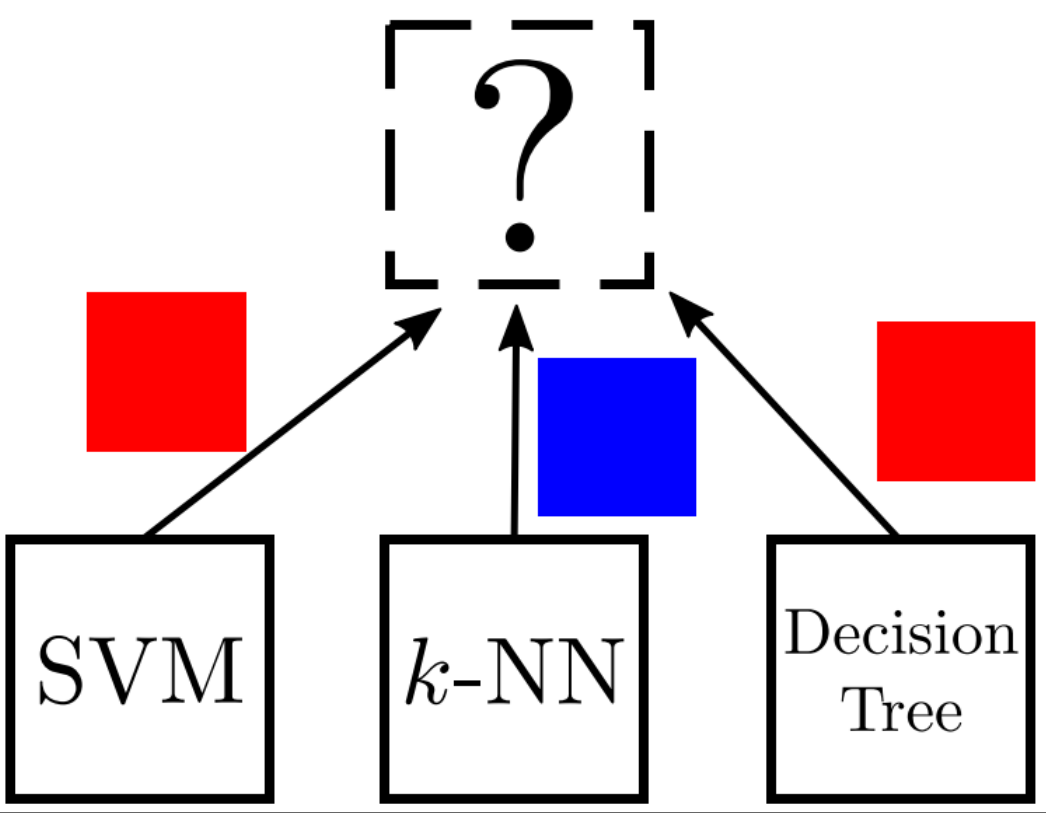
	perf_test	cross_val_perf
average_performance_test_set	7274.185417	7161.048450
oracle	1400.754623	1383.362587
single_best(best_avg)	2177.582462	2461.442755
single_best(most_often_best)	2953.780503	NaN
cluster_and_select	1995.379146	2306.603374
KNeighborsClassifier	3038.238392	NaN
DecisionTreeClassifier	2860.636683	2589.799244

..turns out that the decision tree performs much better than the  $k$ -NN classifier, but still the clustering-and-select approach is better.

# Ensemble Methods

Ensemble Methods

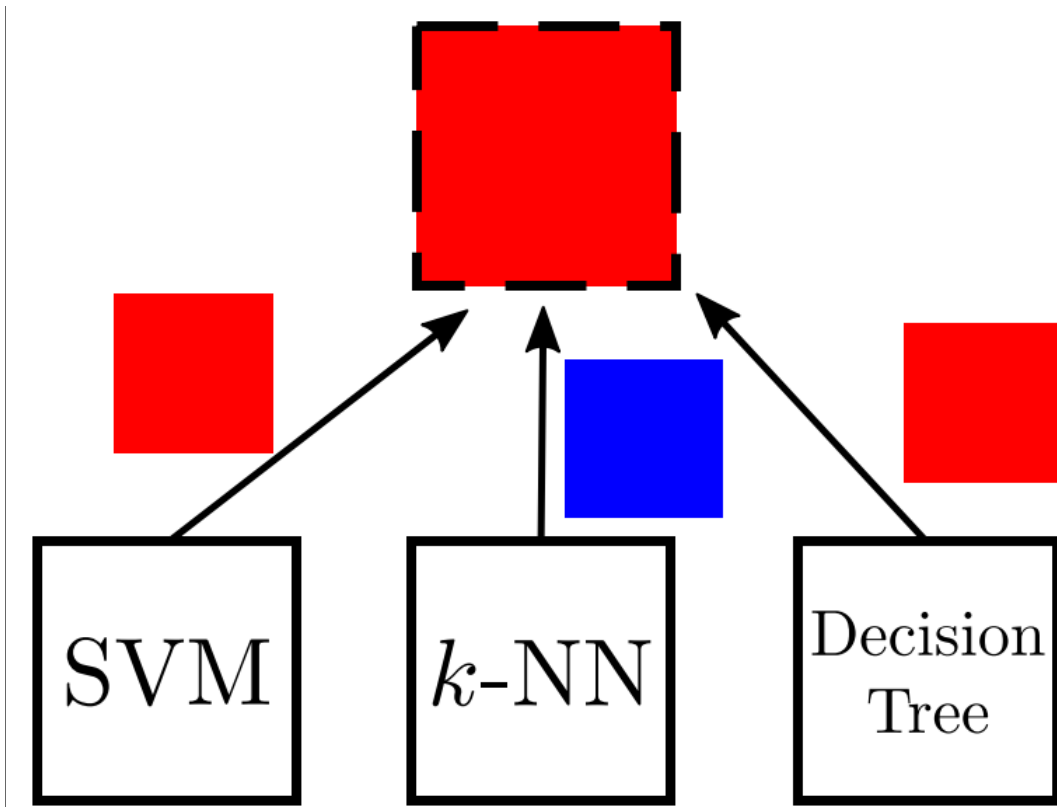
**Idea:** If a single learned algorithm is good at making predictions, can we perform even better by simultaneously using multiple algorithms?



**Example:** Three different classification approaches "vote" on the correct classification of some input data

## Ensemble Methods

**Idea:** If a single learned algorithm is good at making predictions, can we perform even better by simultaneously using multiple algorithms?



**Example:** Three different classification approaches "vote" on the correct classification of some input data  
→ **The majority vote wins!**



## Ensembles with Bagging

**Some classifiers exhibit a high degree of variance:**

- small changes in the data lead to vastly different models

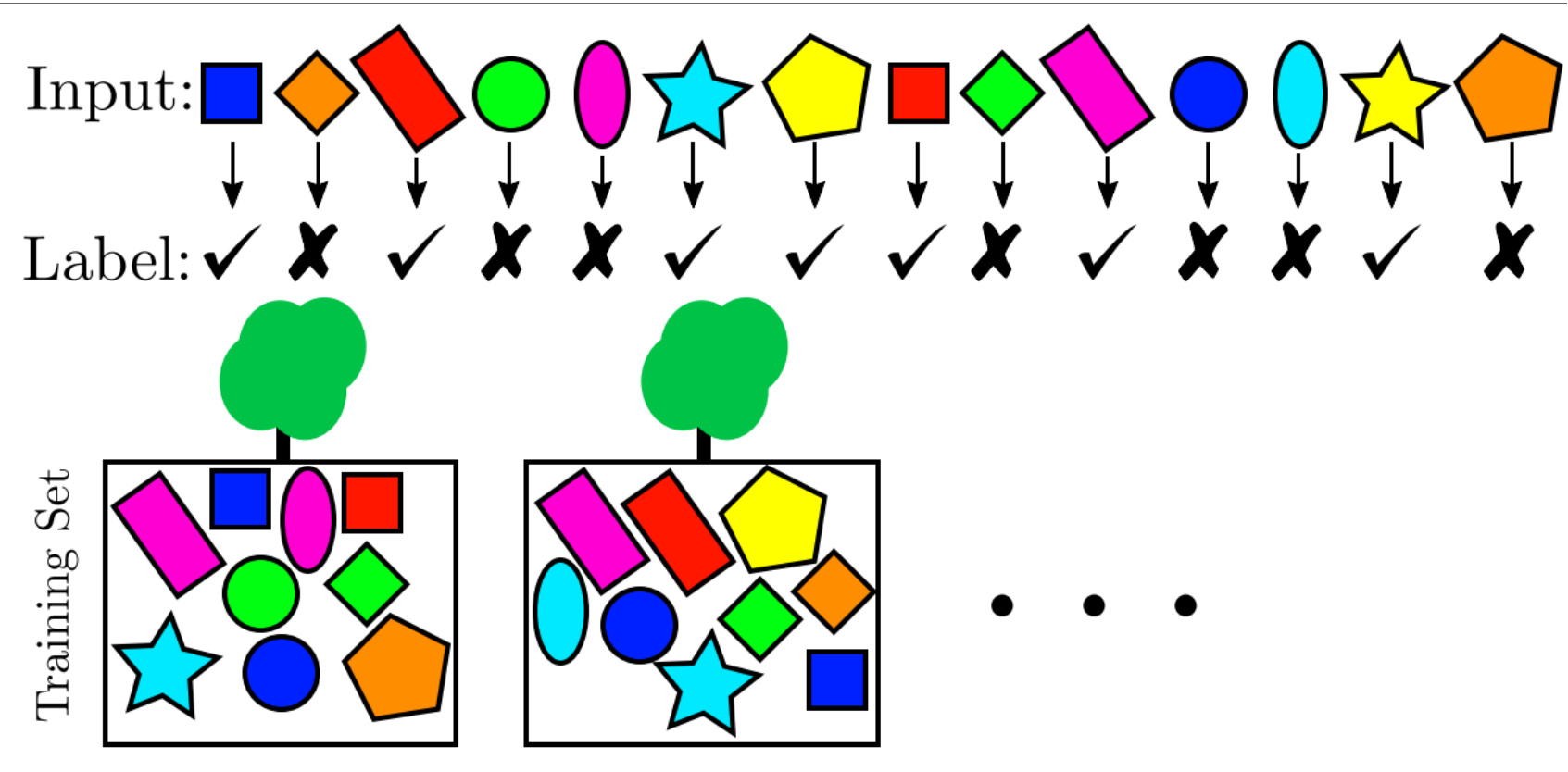
This behavior can be overcome with **Bagging: Bootstrap Aggregation**.

The idea of bagging intuitively works as follows:

- bootstrap (sample with replacement) from the training data multiple times
- train a classifier for each of the bootstrapped training sets
- aggregate the predictions from all classifiers by
  - majority voting (for classification problems)
  - averaging (for regression problems)

Random Forests: Bagging with Decision Trees

Since decision trees have a high variance, bagging can be used for creating stable predictions.



As we use multiple trees, the resuling model is called a **Random Forest**

## Using a Random Forest Classifier in our Case Study

- we use a random forest from scikit-learn
- and replace a single line in the evaluate\_train\_test\_split function:
  - RandomForestClassifier(max\_depth=10, random\_state=0)
- otherwise, the evaluation function is identical to the one used for the  $k$ -nearest neighbour approach

```
In [46]:
from sklearn.ensemble import RandomForestClassifier

def evaluate_train_test_split_classification_random_forest(train_instances, test_instances):

    #prepare the data
    df_instance_features_train, df_instance_features_test = get_data_train_test(df_instance_features, train_instances, test_instances)
    df_runs_train, df_runs_test = get_data_train_test(df_runs, train_instances, test_instances)
    df_best_runs_train, df_best_runs_test = get_data_train_test(df_best_runs, train_instances, test_instances)

    # step 1: Training
    clf = RandomForestClassifier(max_depth=10, random_state=0)

    clf.fit(df_best_runs_train.loc[:, "f_0:"], df_best_runs_train["algorithm"])

    # step 2: Selection
    prd = clf.predict(df_instance_features_test.loc[:, "f_0:"])

    df_selected_algorithm_test_instances = df_instance_features_test.filter(['instance_id'])

    df_selected_algorithm_test_instances['algorithm']=prd

    return evaluate_selected_algorithms(df_runs_test, df_selected_algorithm_test_instances)
```

## Results from using the Random Forest Classifier

- we evaluate the algorithm selection based on predicting the best with the decision tree classifier for the single train-test split and using cross validation
- and directly add the results to our results data frame:

```
In [47]:
df_results.loc['RandomForestClassifier', 'perf_test'] = evaluate_train_test_split_classification_random_forest(train_instances, test_instances)
df_results.loc['RandomForestClassifier', 'cross_val_perf'] = evaluate_using_cross_validation(evaluate_train_test_split_classification_random_forest, kf)
df_results
```

Out[47]:

	perf_test	cross_val_perf
average_performance_test_set	7274.185417	7161.048450
oracle	1400.754623	1383.362587
single_best(best_avg)	2177.582462	2461.442755
single_best(most_often_best)	2953.780503	NaN
cluster_and_select	1995.379146	2306.603374
KNeighborsClassifier	3038.238392	NaN
DecisionTreeClassifier	2860.636683	2589.799244
RandomForestClassifier	2578.188995	2343.732815

..turns out that the random forest yields the best results among the classification-based approaches, but still the clustering-and-select approach is better.

# 5. Regression for Algorithm Selection

## Supervised Learning: Classification vs Regression

The most important types of supervised learning are:

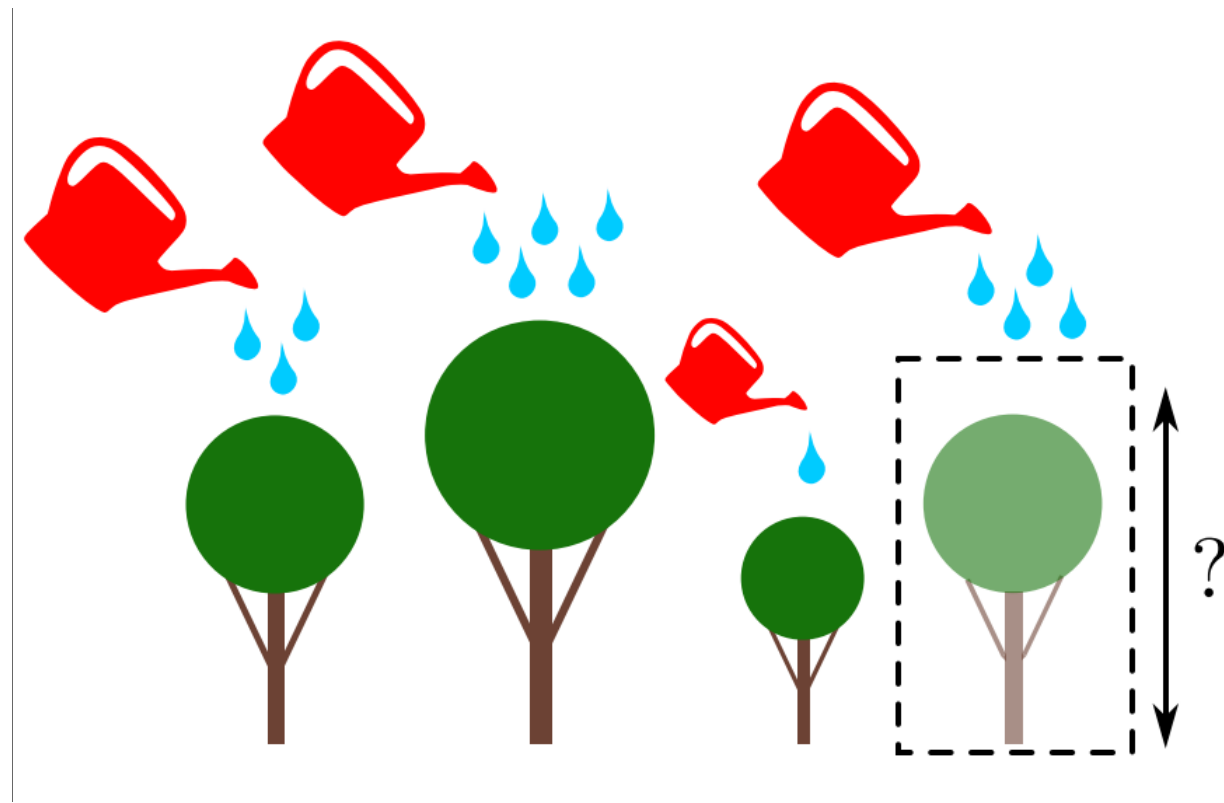
- **classification**: predict which category or type (labels are categorical values or classes)
  - our use in algorithm selection: predict the best algorithm
- **regression**: predict how much or how many (labels are numbers)
  - our use in algorithm selection: predict the algorithm performance

## Supervised Learning: Regression

### Goal

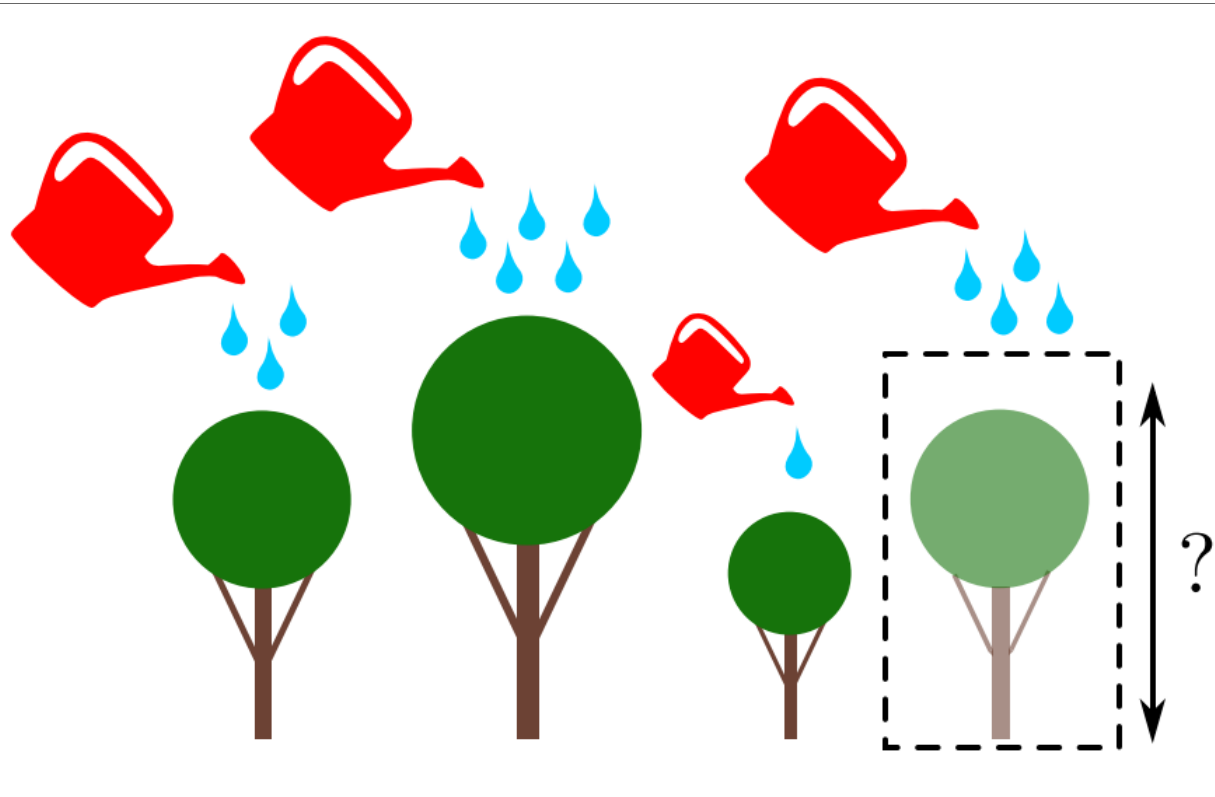
Given a set of *examples* with some given (categorical) *outcomes*, we wish to learn a *model* to predict an outcome given a new example

**Example:** If I water my trees with  $x$  liter of water in May, how tall will they grow?



Illustrating Linear Regression

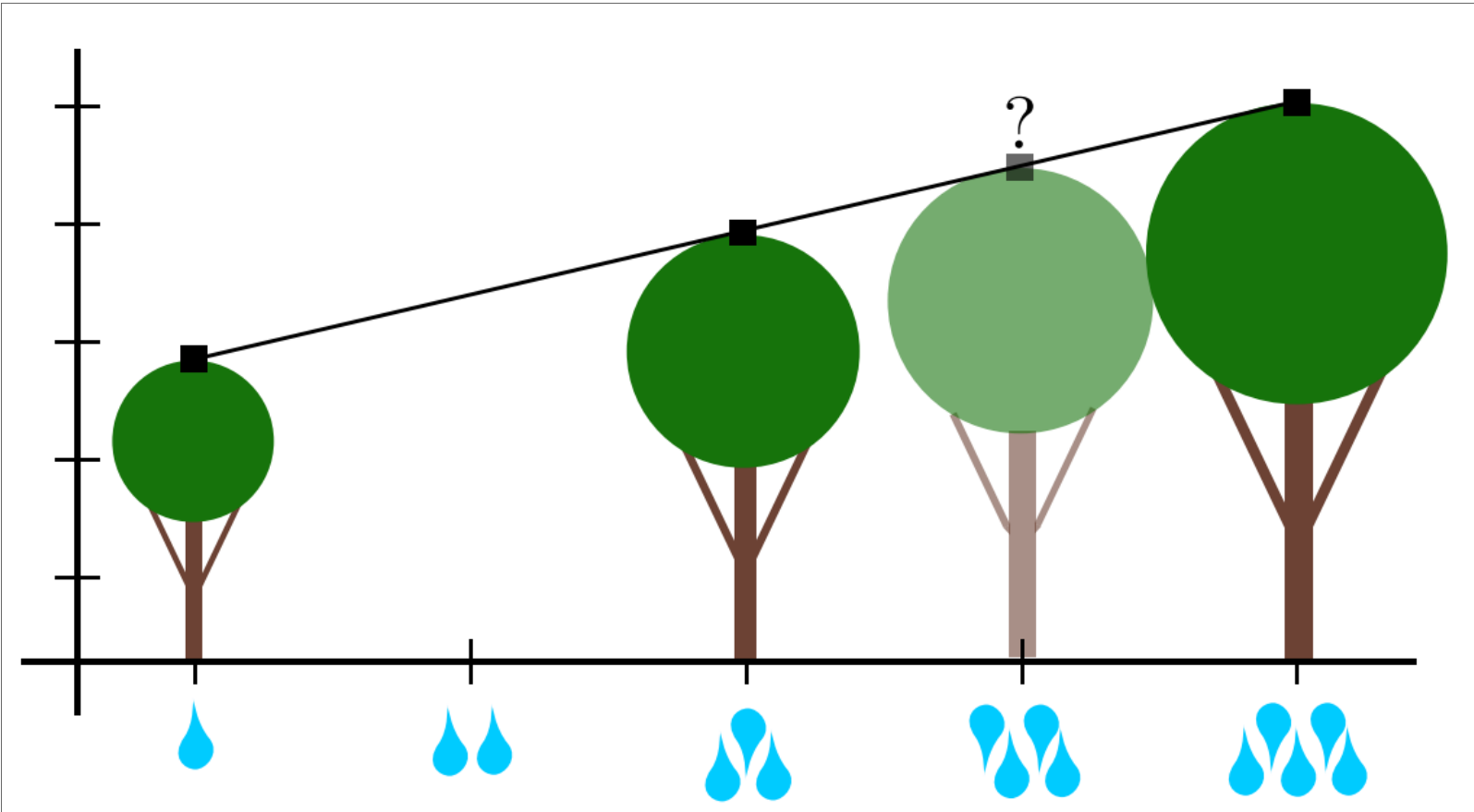
Consider our tree growth example again..





# Illustrating Linear Regression

Consider our tree growth example again..



- **Assumption:** Linear relation between water and height
- **Note:** Missing data for 2 and 4 l of water
- **Prediction:** 4 l of water yields a height between 3 and 5 metres

## Regression for Algorithm Selection

### **Key Idea: Predict Runtime and Select Algorithm**

- for each algorithm train a regression model that used instance features and PAR10 labels to predict runtime (PAR10) performance
- when encountering a new instance, chose the algorithm with the best predicted performance

### **Observe:**

- instead of fitting a single model, we need to fit **one model for each algorithm!**

## Predicting the Best Algorithm for the MaxSAT Data Set

- the MaxSAT data set comes with a file with 37 features per instance
- the semantics of the features are unknown to us (there names are f\_1, f\_2, etc.)
- the values are normalized to take values between 0 and 1

For training a regressor that predicts the PAR10 performance of an algorithm of an instance, we need

- a data frame with **one row per combination of instance and algorithm** (in the training set) containing
  - the instance
  - the instance features
  - the algorithm
  - the PAR10 value

We first create the full data frame including both performance and feature information:

```
In [48]:
df_runs = pd.merge(df_runs, df_instance_features, left_on='instance_id', right_on='instance_id')
df_runs.head()
```

Out[48]:

	instance_id	algorithm	PAR10	f_0	f_1	f_2	f_3	f_4	f_5	f_6	...	f_27	f_28	
<sup>0</sup>	mul_8_11.wcnf	CCEHC2akms	18000.0	15008.0	80288.0	0.18693	1.0	0.0	1.0	1.0	...	1.0	0.18693	0.18
<sup>1</sup>	mul_8_11.wcnf	ahms-1.70	18000.0	15008.0	80288.0	0.18693	1.0	0.0	1.0	1.0	...	1.0	0.18693	0.18
<sup>2</sup>	mul_8_11.wcnf	LMHS-2016	18000.0	15008.0	80288.0	0.18693	1.0	0.0	1.0	1.0	...	1.0	0.18693	0.18
<sup>3</sup>	mul_8_11.wcnf	Optiriss6	18000.0	15008.0	80288.0	0.18693	1.0	0.0	1.0	1.0	...	1.0	0.18693	0.18
<sup>4</sup>	mul_8_11.wcnf	WPM3-2015-co	18000.0	15008.0	80288.0	0.18693	1.0	0.0	1.0	1.0	...	1.0	0.18693	0.18

5 rows × 40 columns

Split this data frame into a train and a test set

```
In [49]:
df_runs_train, df_runs_test = get_data_train_test(df_runs, train_instances, test_instances)
```

# Using Linear Regression for Algorithm Selection: Case Study

We will use the linear regression from scikit-learn:

```
In [50]: from sklearn.linear_model import LinearRegression
```

First, we will create one prediction model per algorithm.  
For each algorithm, we:

- filter the training data set to contain only runs with this algorithm
- fit a regression model using instance features as features and PAR10 values as labels

```
In [51]: runtime_prediction_models = {} # a dictionary / map where we will store the fitted model for each algorithm

# for alg in algorithms:
for alg in algorithms: # by using tqdm, we can monitor the progress of the loop

    reg = LinearRegression()

    df_runs_train_alg = df_runs_train[df_runs_train["algorithm"] == alg] # only consider runs with the algorithm under consideration

    reg.fit(df_runs_train_alg.loc[:, "f_0:"], df_runs_train_alg["PAR10"]) # Train a model for algorithm alg to predict the PAR10 value on an instance based on the instance features
    runtime_prediction_models[alg] = reg
```

## Using Linear Regression for Algorithm Selection: Case Study

Then, we perform the selection of algorithm in the test set as follows:

- predict the performance for each algorithm in each instance in the test set
- for each instance in the test set, select the algorithm with the best predicted performance

```
In [52]:
    test_set_predictions=[]

# create a List of data frames (one per algorithm) containing the PAR10 predictions for the test instances
for alg, reg in runtime_prediction_models.items():
    df_instance_with_prediction_test = df_instance_features_test.assign(algorithm=alg, PAR10_prediction = reg.predict(df_instance_features_test.loc[:, "f_0":]))
    test_set_predictions.append(df_instance_with_prediction_test)

# combine the dataframes into a single one
df_instance_with_prediction_test = pd.concat(test_set_predictions, join="inner").reset_index(drop=True) # Concatenate the results of all alg

# for each instance, select the algorithm with the best predicted performance
row_idx_for_best_predicted_algorithms = df_instance_with_prediction_test.groupby("instance_id")["PAR10_prediction"].idxmin() # For each instance, get the row id of the entry with the Lowest PAR10 prediction
df_selected_algorithms = df_instance_with_prediction_test.iloc[row_idx_for_best_predicted_algorithms ] # We use that to filter only those rows

#df_selected_algorithms[["instance_id", "algorithm"]].head()
```

Now, let us evaluate the average performance of the selection on the test instances:

```
In [53]:
    print(evaluate_selected_algorithms(df_runs_test, df_selected_algorithms))
df_results.loc[type(reg).__name__ , 'perf_test'] = np.mean(evaluate_selected_algorithms(df_runs_test, df_selected_algorithms))
df_results
```

1991.6450753768845

Out[53]:

	perf_test	cross_val_perf
average_performance_test_set	7274.185417	7161.048450
oracle	1400.754623	1383.362587
single_best(best_avg)	2177.582462	2461.442755
single_best(most_often_best)	2953.780503	NaN
cluster_and_select	1995.379146	2306.603374
KNeighborsClassifier	3038.238392	NaN
DecisionTreeClassifier	2860.636683	2589.799244
RandomForestClassifier	2578.188995	2343.732815
LinearRegression	1991.645075	NaN

Exercise: Evaluate the Performance for the Linear Regression Approach with Cross Validation!

- write an evaluation function `evaluate_train_test_split_regression_linear` using the code above
- check if the results from that function with the single train test split are the same as we computed above
- perform a cross validation evaluation using that function
- add the results to the missing cell in the results data frame

```
In [55]:
```

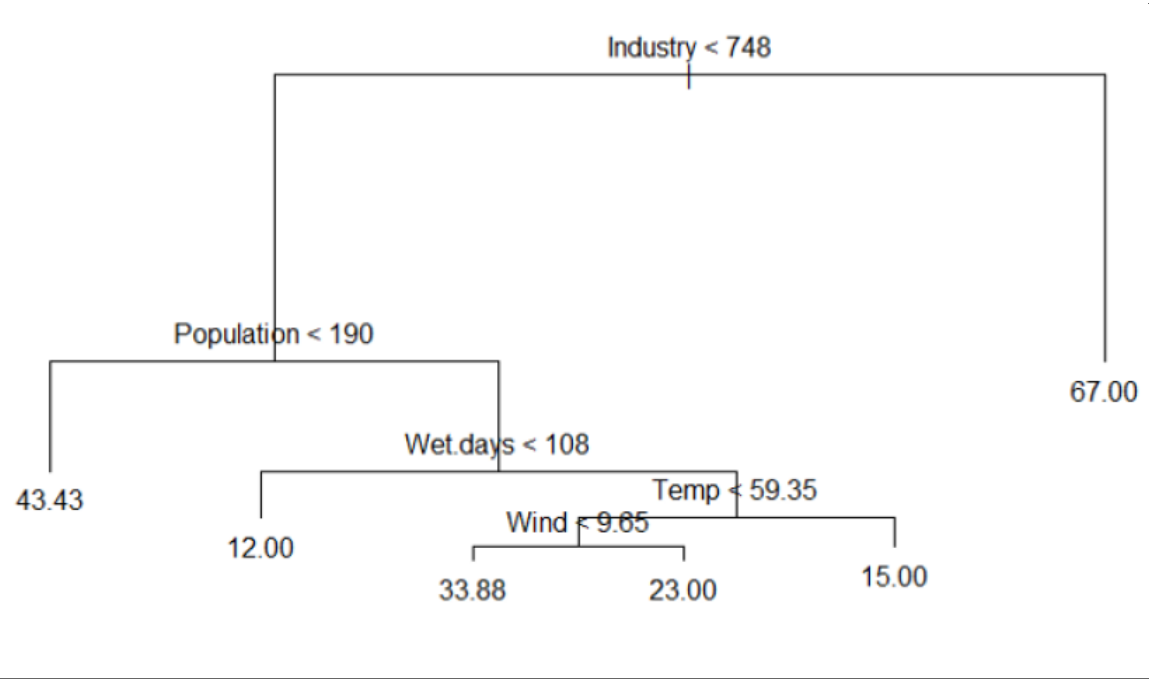
```
In [ ]:
```

# Regression Trees

## Regression Trees

Decision Trees can not only be used for classification, but also for regression:

- same idea as for classification
- instead of classes, leaves represent continous values
- example: predicting the level of pollution





# Case Study: Regression Trees for Algorithm Selection: Evaluation Function

The evaluation function looks *almost* the same as for linear regression,

- we use replace the line in which we define the regression model: `reg = DecisionTreeRegressor(max_depth=10, random_state=0)`

```
In [57]:
from sklearn.tree import DecisionTreeRegressor

def evaluate_train_test_split_regression_tree(train_instances, test_instances):
    #prepare the data
    df_instance_features_train, df_instance_features_test = get_data_train_test(df_instance_features, train_instances, test_instances)
    df_runs_train, df_runs_test = get_data_train_test(df_runs, train_instances, test_instances)

    # step 1: training
    runtime_prediction_models = {} # a dictionary / map where we will store the fitted model for each algorithm

    # for alg in algorithms:
    for alg in algorithms: # by using tqdm, we can monitor the progress of the loop

        reg = DecisionTreeRegressor(max_depth=10, random_state=0)

        df_runs_train_alg = df_runs_train[df_runs_train["algorithm"] == alg] # only consider runs with the algorithm under consideration
        reg.fit(df_runs_train_alg.loc[:, "f_0:"], df_runs_train_alg["PAR10"]) # Train a model for algorithm alg to predict the PAR10 value on an instance based on the instance features
        runtime_prediction_models[alg] = reg

    # step 2
    test_set_predictions=[]

    for alg, reg in runtime_prediction_models.items():
        df_instance_with_prediction_test = df_instance_features_test.assign(algorithm=alg, PAR10_prediction = reg.predict(df_instance_features_test.loc[:, "f_0:"]))
        test_set_predictions.append(df_instance_with_prediction_test)

    df_instance_with_prediction_test = pd.concat(test_set_predictions, join="inner").reset_index(drop=True) # Concatenate the results of all alg

    row_idx_for_best_predicted_algorithms = df_instance_with_prediction_test.groupby("instance_id")["PAR10_prediction"].idxmin() # For each instance, get the row id of the entry with the lowest PAR10 prediction

    df_selected_algorithms = df_instance_with_prediction_test.iloc[row_idx_for_best_predicted_algorithms ] # We use that to filter only those rows

    df_selected_algorithms[["instance_id", "algorithm"]]

    # step 3
    return evaluate_selected_algorithms(df_runs_test, df_selected_algorithms)
```

# Case Study: Regression Trees for Algorithm Selection: Results

```
In [58]: df_results.loc['DecisionTreeRegressor', 'perf_test'] = evaluate_train_test_split_regression_tree(train_instances, test_instances)
df_results.loc['DecisionTreeRegressor', 'cross_val_perf'] = evaluate_using_cross_validation(evaluate_train_test_split_regression_tree, kf)
df_results
```

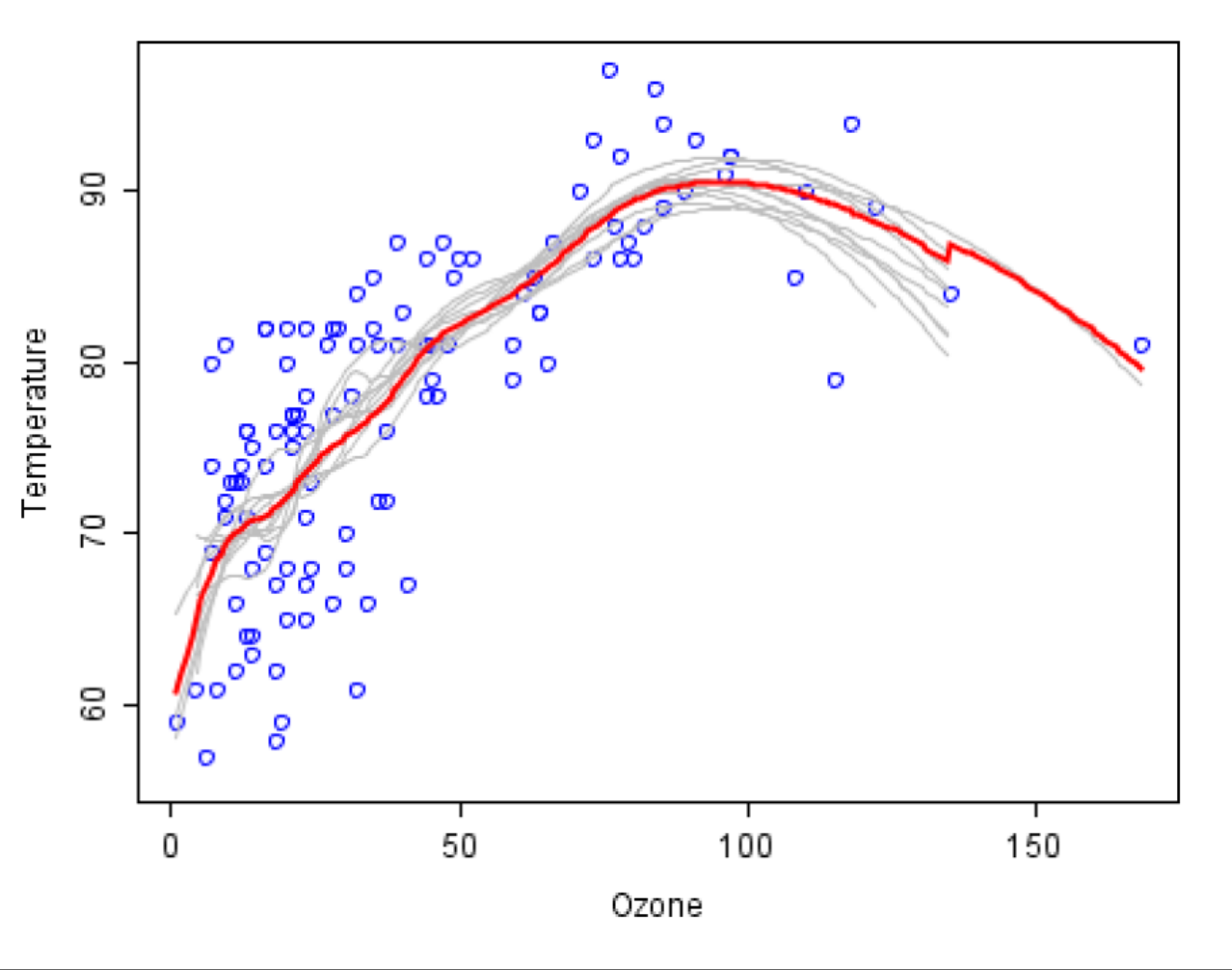
Out[58]:

	perf_test	cross_val_perf
average_performance_test_set	7274.185417	7161.048450
oracle	1400.754623	1383.362587
single_best(best_avg)	2177.582462	2461.442755
single_best(most_often_best)	2953.780503	NaN
cluster_and_select	1995.379146	2306.603374
KNeighborsClassifier	3038.238392	NaN
DecisionTreeClassifier	2860.636683	2589.799244
RandomForestClassifier	2578.188995	2343.732815
LinearRegression	1991.645075	1927.844332
DecisionTreeRegressor	2604.766884	2381.047806

..(much) worse than linear regression!

# Random Forest Regression

Bagging for Regression



- grey lines: prediction based on bootstrapped regression models
- red line: average of the grey lines (result from aggregation)

## Random Forests for Regression

### **We can also use Random Forests for Regression:**

- applying bagging to multiple regression trees
- create multiple regression trees for subset of the training data
- average the results of the trees to obtain a robust prediction

→ same idea as for random forests for classification

### **For use in algorithm selection**

- we simply use scikit-learn's `RandomForestRegressor`

# Case Study: Regression Trees for Algorithm Selection: Evaluation Function

The evaluation function looks *almost* the same as for linear regression,

- we use replace the line in which we define the regression model: `reg = RandomForestRegressor`

```
In [59]:
from sklearn.ensemble import RandomForestRegressor

def evaluate_train_test_split_regression_forest(train_instances, test_instances):
    #prepare the data
    df_instance_features_train, df_instance_features_test = get_data_train_test(df_instance_features, train_instances, test_instances)
    df_runs_train, df_runs_test = get_data_train_test(df_runs, train_instances, test_instances)

    # step 1: training
    runtime_prediction_models = {} # a dictionary / map where we will store the fitted model for each algorithm

    # for alg in algorithms:
    for alg in algorithms: # by using tqdm, we can monitor the progress of the loop

        reg = RandomForestRegressor(max_depth=10, random_state=0)

        df_runs_train_alg = df_runs_train[df_runs_train["algorithm"] == alg] # only consider runs with the algorithm under consideration
        reg.fit(df_runs_train_alg.loc[:, "f_0":], df_runs_train_alg["PAR10"]) # Train a model for algorithm alg to predict the PAR10 value on an instance based on the instance features
        runtime_prediction_models[alg] = reg

    # step 2
    test_set_predictions=[]

    for alg, reg in runtime_prediction_models.items():
        df_instance_with_prediction_test = df_instance_features_test.assign(algorithm=alg, PAR10_prediction = reg.predict(df_instance_features_test.loc[:, "f_0":]))
        test_set_predictions.append(df_instance_with_prediction_test)

    df_instance_with_prediction_test = pd.concat(test_set_predictions, join="inner").reset_index(drop=True) # Concatenate the results of all alg

    row_idx_for_best_predicted_algorithms = df_instance_with_prediction_test.groupby("instance_id")["PAR10_prediction"].idxmin() # For each instance, get the row id of the entry with the lowest PAR10 prediction

    df_selected_algorithms = df_instance_with_prediction_test.iloc[row_idx_for_best_predicted_algorithms ] # We use that to filter only those rows

    df_selected_algorithms[["instance_id", "algorithm"]]

    # step 3
    return evaluate_selected_algorithms(df_runs_test, df_selected_algorithms)
```

# Regression Forests for Algorithm Selection: Results

```
In [60]: df_results.loc['RandomForestRegressor', 'perf_test'] = evaluate_train_test_split_regression_forest(train_instances, test_instances)
df_results.loc['RandomForestRegressor', 'cross_val_perf'] = evaluate_using_cross_validation(evaluate_train_test_split_regression_forest, kf)
df_results
```

Out[60]:

	perf_test	cross_val_perf
average_performance_test_set	7274.185417	7161.048450
oracle	1400.754623	1383.362587
single_best(best_avg)	2177.582462	2461.442755
single_best(most_often_best)	2953.780503	NaN
cluster_and_select	1995.379146	2306.603374
KNeighborsClassifier	3038.238392	NaN
DecisionTreeClassifier	2860.636683	2589.799244
RandomForestClassifier	2578.188995	2343.732815
LinearRegression	1991.645075	1927.844332
DecisionTreeRegressor	2604.766884	2381.047806
RandomForestRegressor	1892.741307	1843.295538

..best performance so far!

# Conclusions

## This week, we

- started with the second part of the course
- introduced the problem of algorithm selection
- introduced a case study data set for motivating and implementing our approaches
- started using feature information for **instance-specific** algorithm selection
- discussed how to use cross validation for obtaining more robust selection results
- reviewed different approaches for **unsupervised learning** and **supervised learning** (classification an regression)
- learned how to use these approaches for algorithm selection

## Next time, we

- will learn how to use machine learning for **algorithm configuration**.

In [ ]: