Introduction to Machine Learning

In practice

Joaquin Vanschoren, Eindhoven Univeristy of Technology

Overview

- How to do machine learning in Python (scikit-learn)
  - Loading data
  - Training and evaluating models
  - Tuning models to your data
- Worldwind tour of algorithms
  - Working principles
  - Main hyperparameters to tune
  - Benefits and drawbacks
- Next:
  - Data and experiment management with OpenML
  - Hands-on exercises

scikit-learn

One of the most prominent Python libraries for machine learning:

- 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
- scikit-learn works well with numpy, scipy, pandas, matplotlib,...

Algorithms

See the Reference (http://scikit-learn.org/dev/modules/classes.html)

Supervised learning:

- Linear models (Ridge, Lasso, Elastic Net, ...)
- Support Vector Machines
- Tree-based methods (Classification/Regression Trees, Random Forests,...)
- Nearest neighbors
- Neural networks
- Gaussian Processes
- Feature selection

Unsupervised learning:

- Clustering (KMeans, ...)
- Matrix Decomposition (PCA, ...)
- Manifold Learning (Embeddings)
- Density estimation
- Outlier detection

Model selection and evaluation:

- Cross-validation
- Grid-search
- Lots of metrics

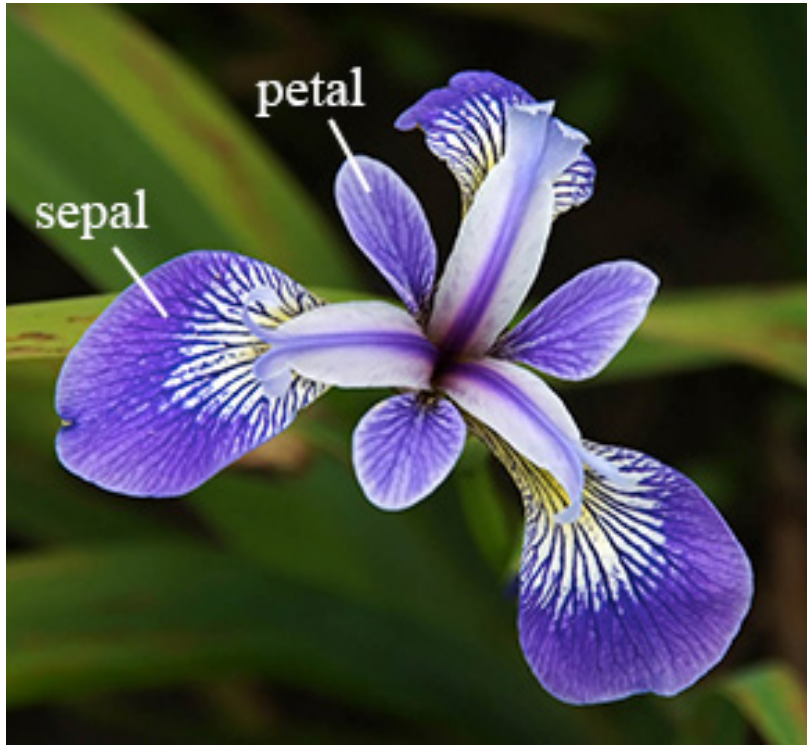
Data import

Multiple options:

- A few toy datasets are included in `sklearn.datasets`
- You can import data files (CSV) with `pandas` or `numpy`
- You can import 1000s of machine learning datasets from OpenML

Example: classification

Classify types of Iris flowers (setosa, versicolor, or virginica) based on the flower sepals and petal leave sizes.



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

```
from sklearn.datasets import load_iris
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 `lists`, 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']))
```

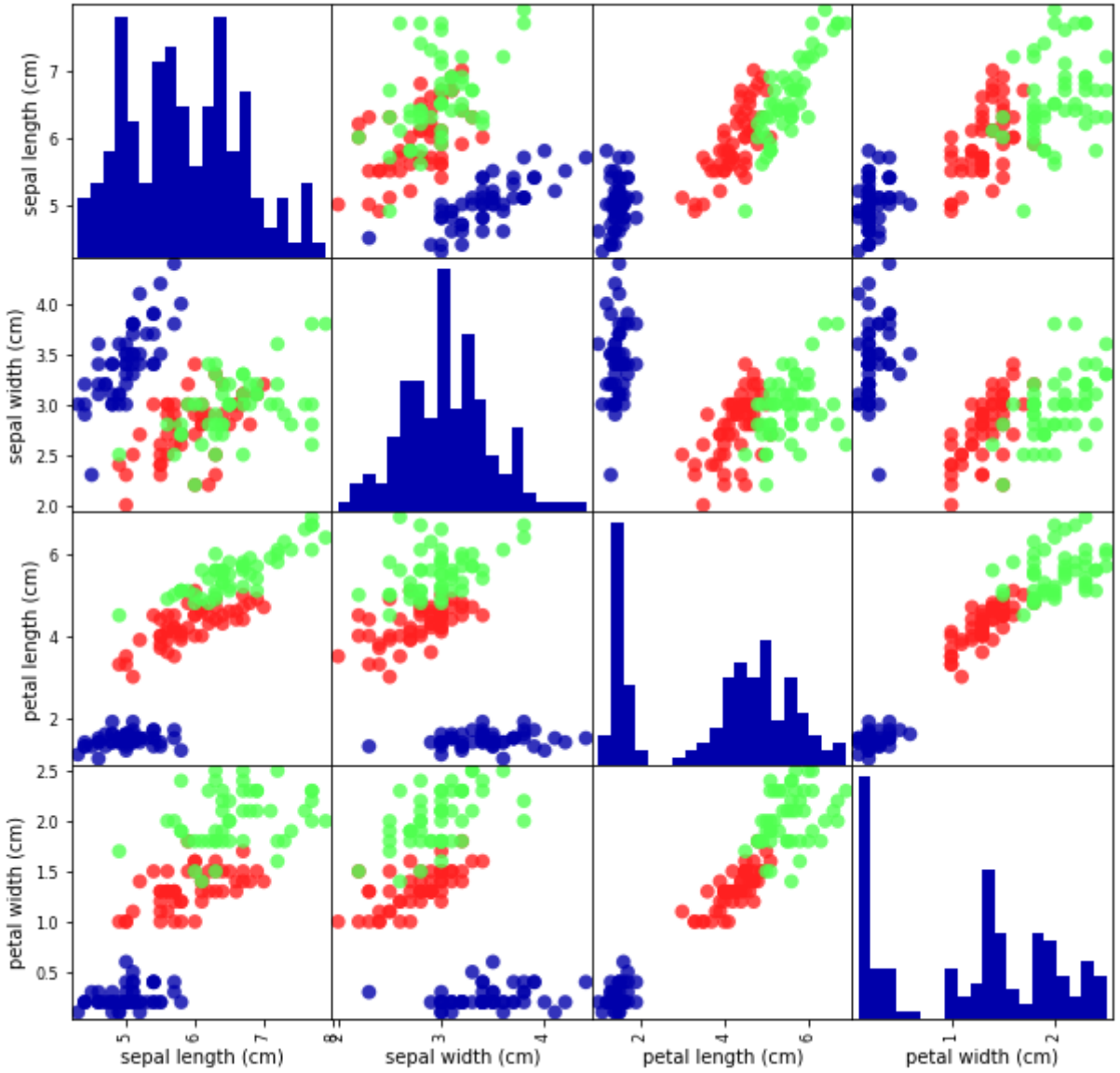
```
Target names: ['setosa' 'versicolor' 'virginica']
Targets:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 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]
```

Looking at your data

We can use a library called `pandas` to easily visualize our data. Note how several features allow to cleanly split the classes.

```
# Build a DataFrame with training examples and feature names
iris_df = pd.DataFrame(iris_dataset['data'],
                       columns=iris_dataset.feature_names)

# scatter matrix from the dataframe, color by class
sm = pd.scatter_matrix(iris_df, c=iris_dataset['target'], figsize=(10, 1
0),
                  marker='o', hist_kwds={'bins': 20}, s=60,
                  alpha=.8, cmap=mglearn.cm3)
```


sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)

Building your first model

All scikitlearn classifiers 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?

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]
```

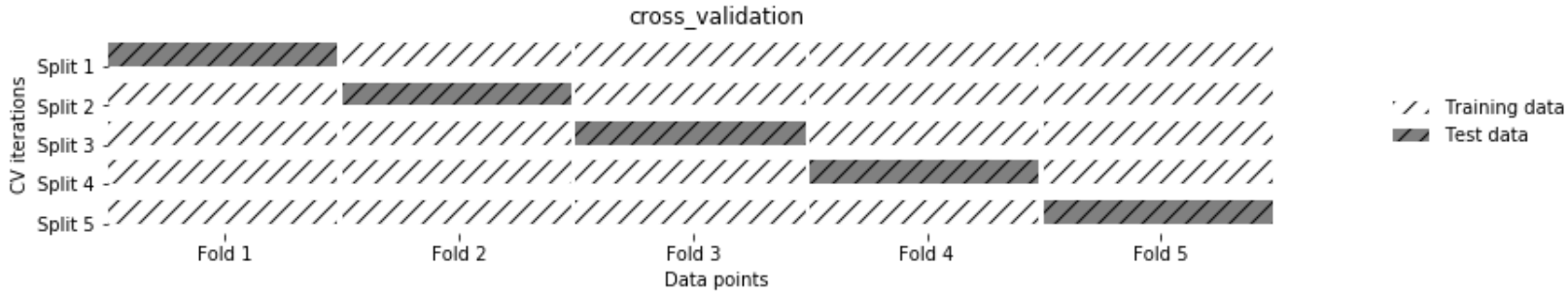
The `score` function computes the percentage of correct predictions

```
knn.score(X_test, y_test)
```

```
Score: 0.97
```

Cross-validation

- More stable, thorough way to estimate generalization performance
- *k-fold cross-validation* (CV): split (randomized) data into *k* equal-sized parts, called *folds*
  - First, fold 1 is the test set, and folds 2-5 comprise the training set
  - Then, fold 2 is the test set, folds 1,3,4,5 comprise the training set
  - Compute *k* evaluation scores, aggregate afterwards (e.g. take the mean)



Cross-validation in scikit-learn

- `cross_val_score` function with learner, training data, labels
- Returns list of all scores
  - Does 3-fold CV by default, can be changed via `cv` hyperparameter
  - Default scoring measures are accuracy (classification) or R^2 (regression)
- Even though models are built internally, they are not returned

```
logreg = LogisticRegression()
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
print("Cross-validation scores: {}".format(scores))
print("Average cross-validation score: {:.2f}".format(scores.mean()))
print("Variance in cross-validation score: {:.4f}".format(np.var(scores
)))
```

```
Cross-validation scores: [0.961 0.922 0.958]
Average cross-validation score: 0.95
Variance in cross-validation score: 0.0003
```

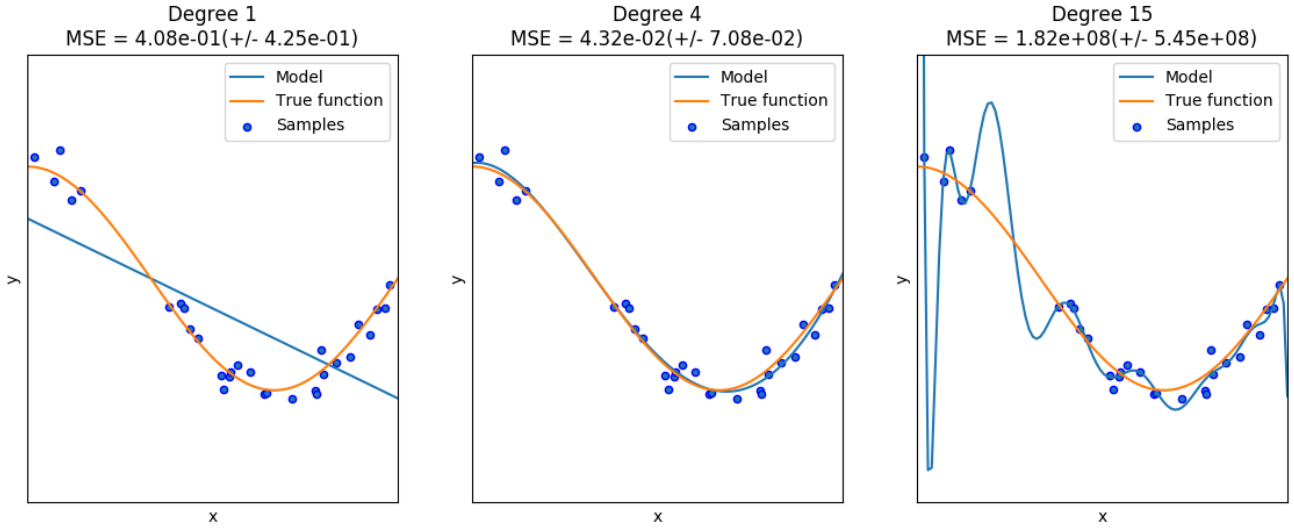
More variants

- See lecture 03 - Model Selection
- Stratified cross-validation: for inbalanced datasets
- Leave-one-out cross-validation: for very small datasets
- Shuffle-Split cross-validation: whenever you need to shuffle the data first
- Repeated cross-validation: more trustworthy, but more expensive
- Cross-validation with groups: Whenever your data contains non-independent datapoints, e.g. data points from the same patient
- Bootstrapping: sampling with replacement, for extracting statistical properties

Generalization, Overfitting and Underfitting

- We **hope** that the model can *generalize* from the training to the test data: make accurate predictions on unseen data
- It's easy to build a complex model that is 100% accurate on the training data, but very bad on the test data
- Overfitting: building a model that is *too complex for the amount of data* that we have
  - You model peculiarities in your data (noise, biases,...)
  - Solve by making model simpler (regularization), or getting more data
- Underfitting: building a model that is *too simple given the complexity of the data*
  - Use a more complex model

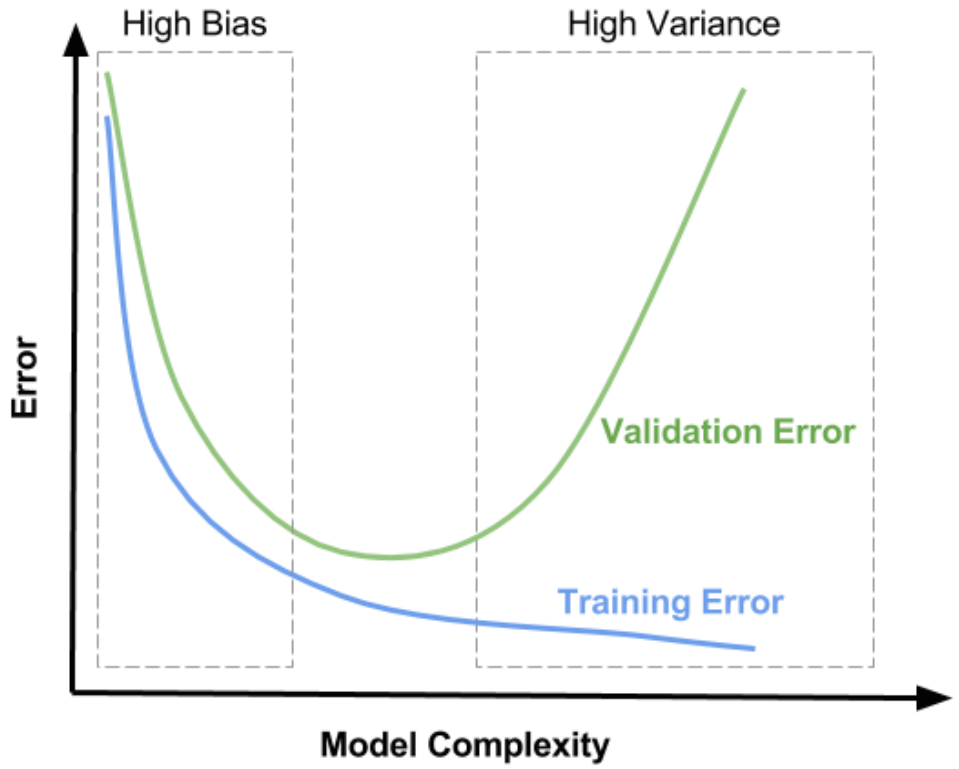
There is often a sweet spot that you need to find by optimizing the choice of algorithms and hyperparameters, or using more data.



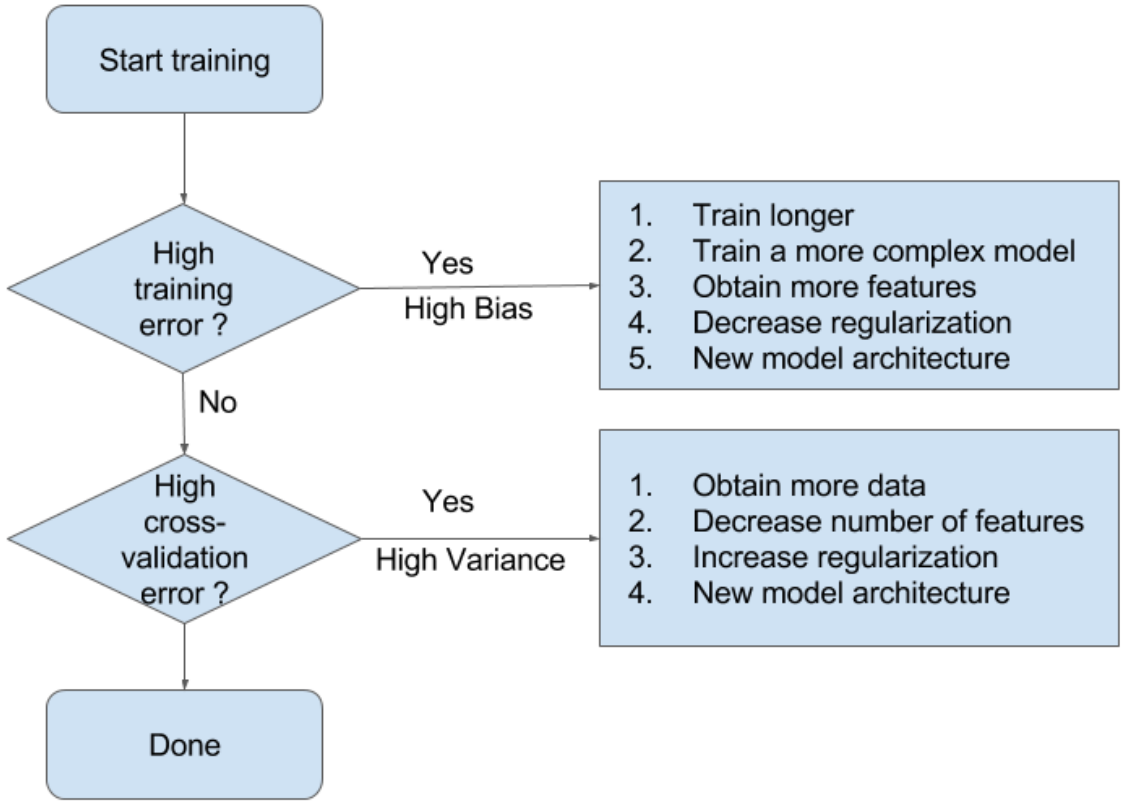
Bias-variance and overfitting

A learning algorithm makes 2 types of errors:

- Bias: data points are always predicted wrong because the model is underfitting
- Variance: predictions fluctuate a lot because the model is overfitting



Bias-Variance Flowchart (Andrew Ng, Coursera)



Hyperparameter tuning

- Most algorithms have parameters (hyperparameters) that control model complexity
- Now that we know how to evaluate models, we can improve them by `tuning` their hyperparameters for your data

We can basically use any optimization technique to optimize hyperparameters:

- **Grid search**
- **Random search**

More advanced techniques:

- Local search
- Racing algorithms
- Bayesian optimization
- Multi-armed bandits
- Genetic algorithms

Grid Search

- For each hyperparameter, create a list of interesting/possible values
  - E.g. For kNN: k in [1,3,5,7,9,11,33,55,77,99]
  - E.g. For SVM: C and gamma in $[10^{-10}..10^{10}]$
- Evaluate all possible combination of hyperparameter values
  - E.g. using cross-validation
- Split the training data into a training and validation set
- Select the hyperparameter values yielding the best results on the validation set

Grid search in scikit-learn

- Create a parameter grid as a dictionary
  - Keys are parameter names
  - Values are lists of hyperparameter values

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
print("Parameter grid:\n{}".format(param_grid))
```

```
Parameter grid:
{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 1
00]}
```

- `GridSearchCV`: like a classifier that uses CV to automatically optimize its hyperparameters internally
  - Input: (untrained) model, parameter grid, CV procedure
  - Output: optimized model on given training data
  - Should only have access to training data

```
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

```
Out[15]: GridSearchCV(cv=5, error_score='raise-deprecating',
             estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
           kernel='rbf', max_iter=-1, probability=False, random_state=None,
           shrinking=True, tol=0.001, verbose=False),
             fit_params=None, iid='warn', n_jobs=None,
             param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001,
         0.01, 0.1, 1, 10, 100]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
             scoring=None, verbose=0)
```

The optimized test score and hyperparameters can easily be retrieved:

```
grid_search.score(X_test, y_test)
grid_search.best_params_
grid_search.best_score_
grid_search.best_estimator_
```

```
Test set score: 0.97
Best parameters: {'C': 100, 'gamma': 0.01}
Best cross-validation score: 0.97
Best estimator:
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

Nested cross-validation

- Note that we are still using a single split to create the outer test set
- We can also use cross-validation here
- Nested cross-validation:
  - Outer loop: split data in training and test sets
  - Inner loop: run grid search, splitting the training data into train and validation sets
- Result is a just a list of scores
  - There will be multiple optimized models and hyperparameter settings (not returned)
- To apply on future data, we need to train `GridSearchCV` on all data again

```
scores = cross_val_score(GridSearchCV(SVC(), param_grid, cv=5),
                         iris.data, iris.target, cv=5)
```

```
Cross-validation scores:  [0.967 1.    0.967 0.967 1.   ]
Mean cross-validation score:  0.9800000000000001
```

Random Search

- Grid Search has a few downsides:
  - Optimizing many hyperparameters creates a combinatorial explosion
  - You have to predefine a grid, hence you may jump over optimal values
- Random Search:
  - Picks `n_iter` random parameter values
  - Scales better, you control the number of iterations
  - Often works better in practice, too
    - not all hyperparameters interact strongly
    - you don't need to explore all combinations

- Executing random search in scikit-learn:
  - `RandomizedSearchCV` works like `GridSearchCV`
  - Has `n_iter` parameter for the number of iterations
  - Search grid can use distributions instead of fixed lists

```
param_grid = {'C': expon(scale=100),
              'gamma': expon(scale=.1)}
random_search = RandomizedSearchCV(SVC(), param_distributions=param_grid
,
                                   n_iter=20)
random_search.fit(X_train, y_train)
random_search.best_estimator_
```

```
Out[18]: SVC(C=6.109237791897481, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.04723626633903414,
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```

Learning algorithms

In all supervised algorithms that we will discuss, we'll cover:

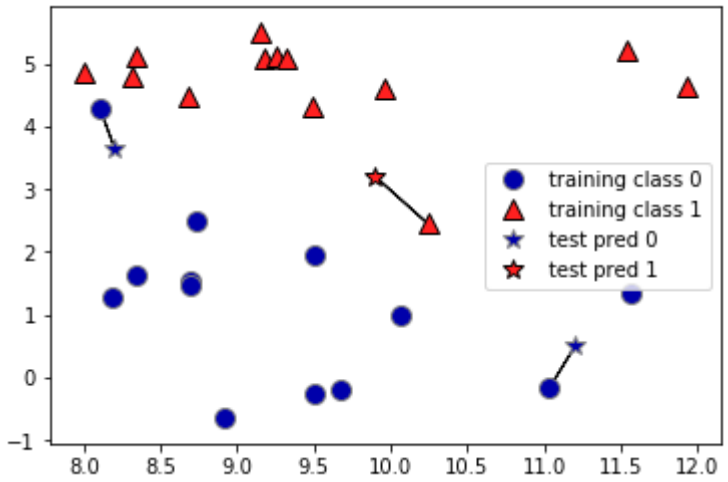
- How do they work
- How to control complexity
- Hyperparameters (user-controlled parameters)
- Strengths and weaknesses

k-Nearest Neighbor

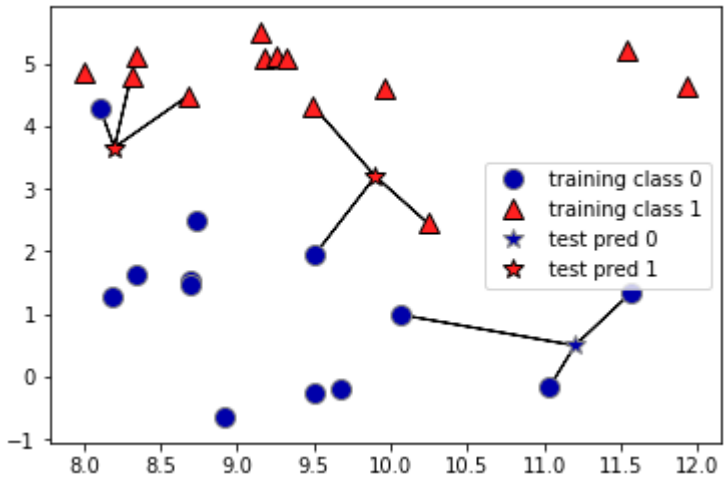
- Building the model consists only of storing the training dataset.
- To make a prediction, the algorithm finds the *k* closest data points in the training dataset

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')

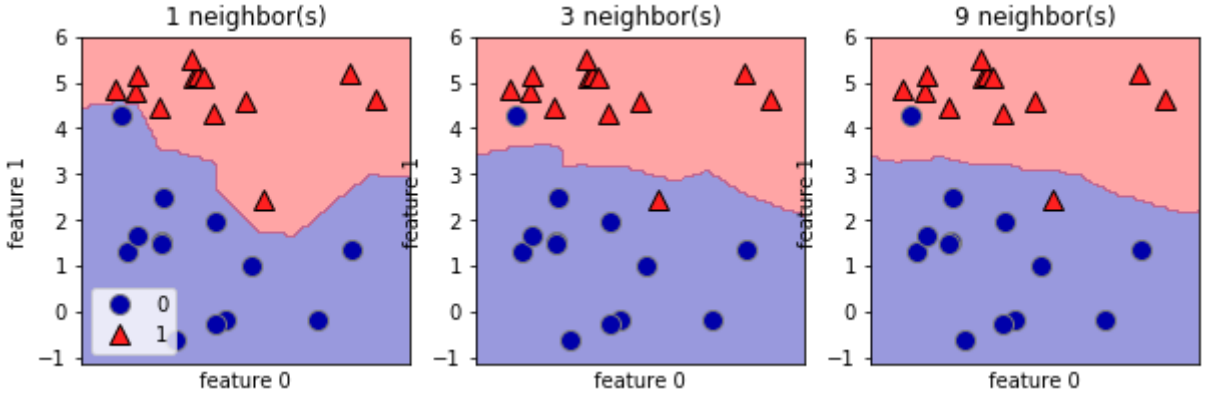
```
X, y = mglearn.datasets.make_forge()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0
)
clf = KNeighborsClassifier(n_neighbors=19)
clf.fit(X_train, y_train)
```

Out[21]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=19, p=2,
           weights='uniform')
```

```
Test set accuracy: 0.43
```

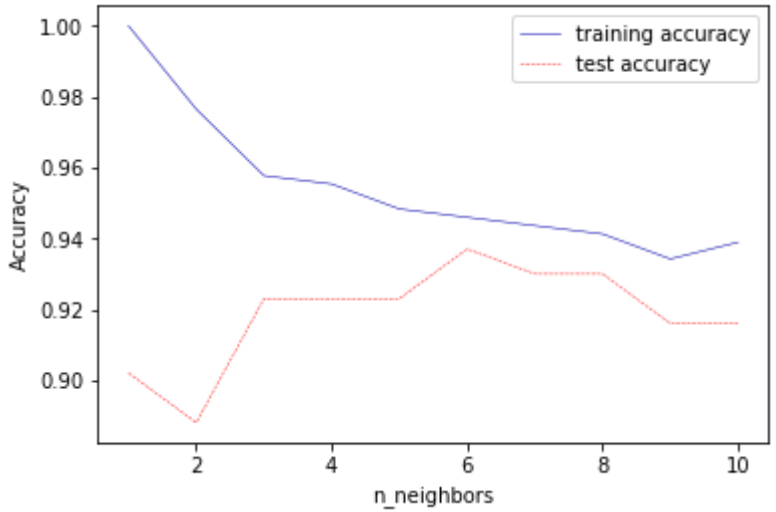
Analysis

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



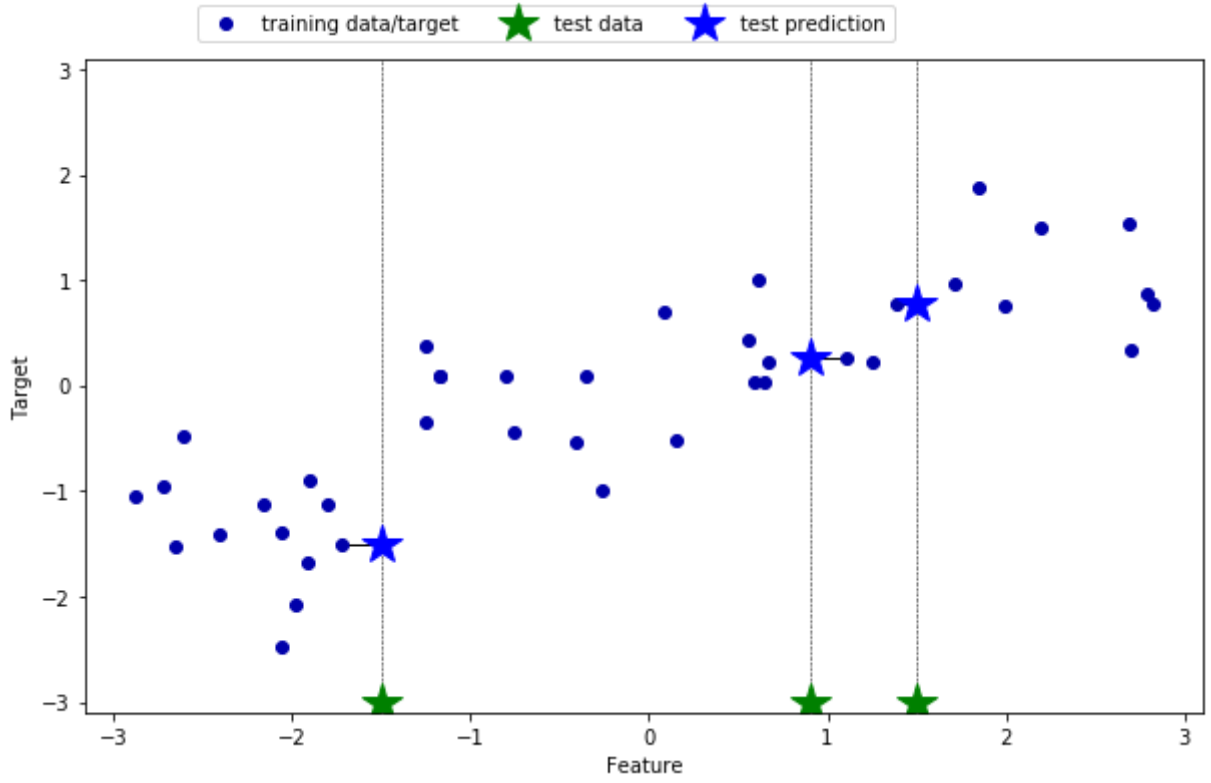
We can more directly measure the effect on the training and test error on a larger dataset (breast_cancer)

- It first overfits, then underfits
- Tune the number of neighbors to your dataset to find the sweet spot

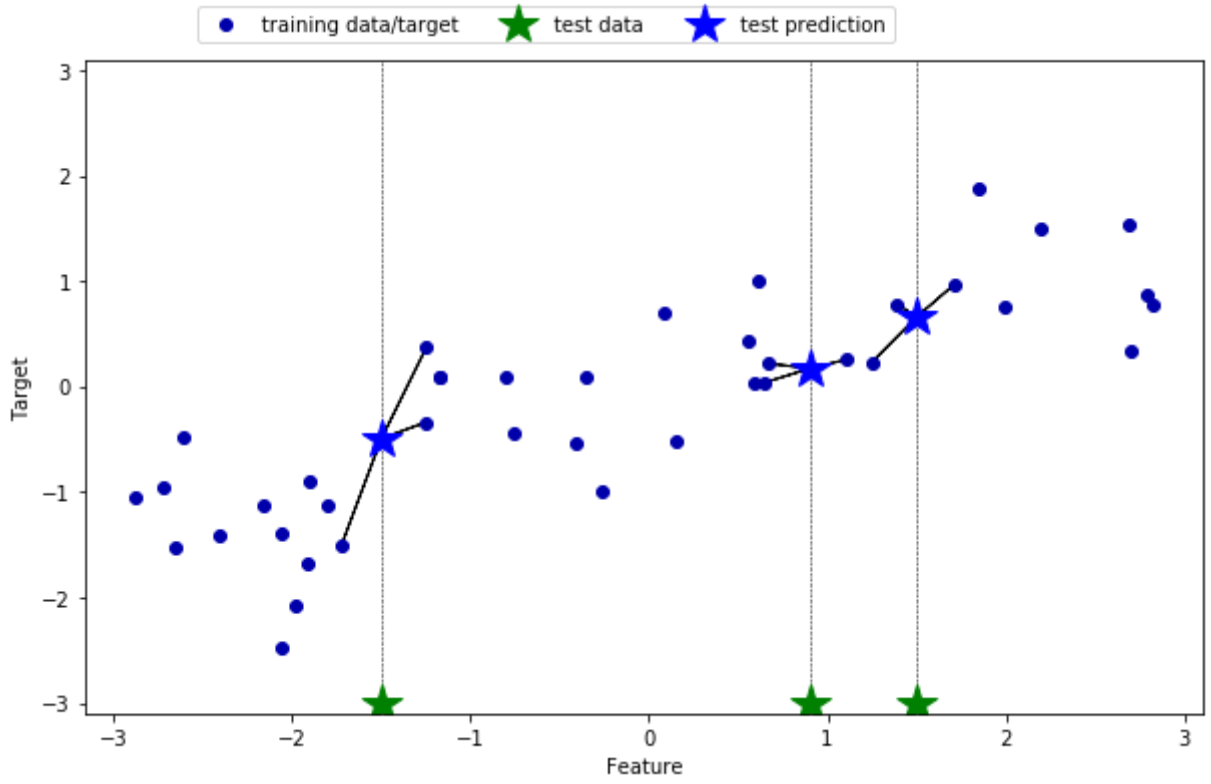


k-Neighbors Regression

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[27]:

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

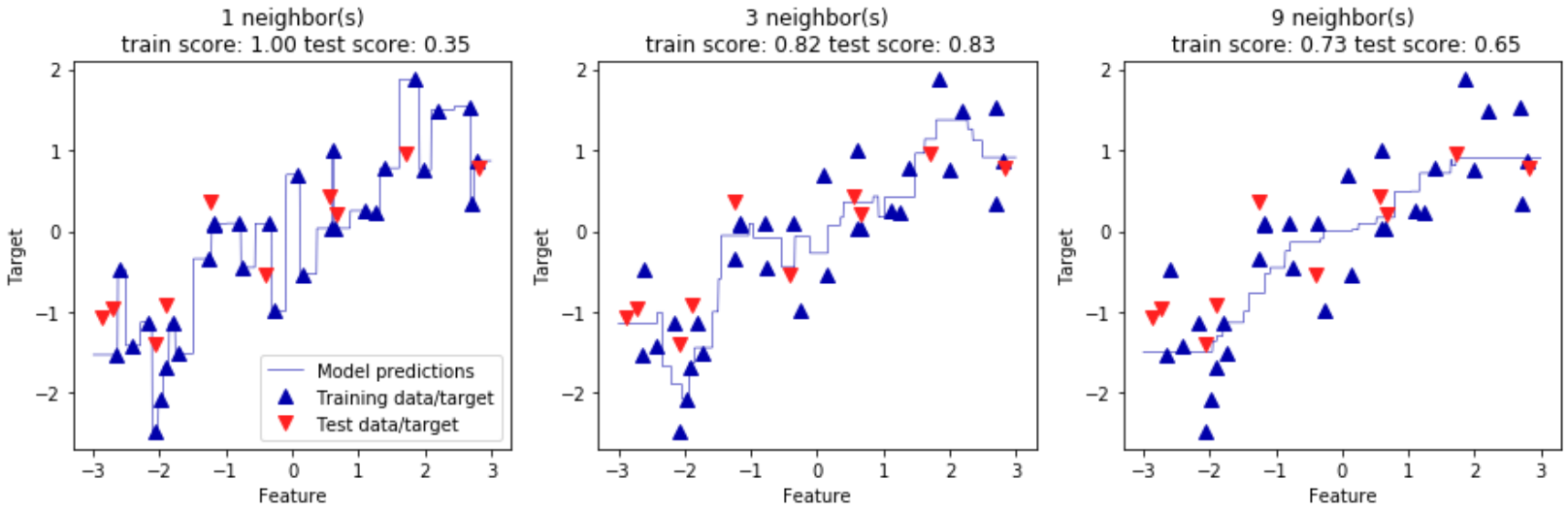
The default scoring function for regression models is R^2. It measures how much of the data variability is explained by the model, relative to just predicting the mean. Usually between 0 and 1 (<0 means you predictions are worse than just predicting the mean).

```
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

- There are two important hyperparameters:
  - n_neighbors: the number of neighbors used
  - metric: the distance measure used
    - Default is Minkowski (generalized Euclidean) distance.
- 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)

Linear models

Linear models make a prediction using a linear function of the input features. Can be very powerful for or datasets with many features.

If you have more features than training data points, any target y can be perfectly modeled (on the training set) as a linear function.

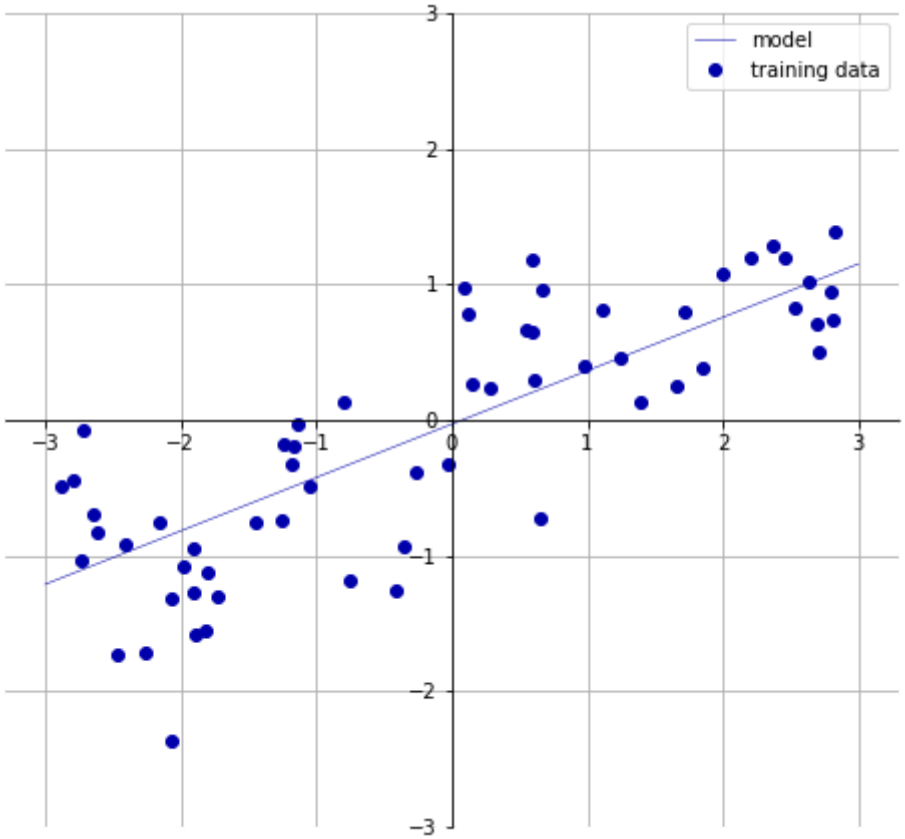
Linear models for regression

Prediction formula for input features x. w_i and b are the *model parameters* that need to be learned.

$$\hat{y} = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p + b$$

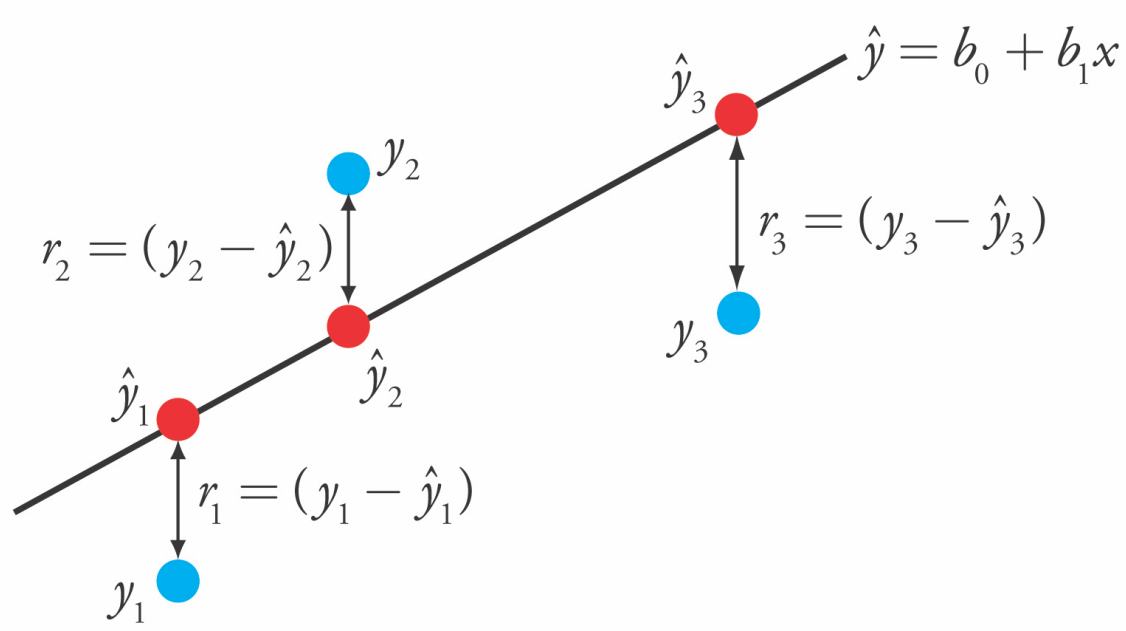
There are many different algorithms, differing in how w and b are learned from the training data.

```
w[0]: 0.393906  b: -0.031804
```



Linear Regression aka Ordinary Least Squares

- Finds the parameters w and b that minimize the *mean squared error* between predictions and the true regression targets, y, on the training set.
  - MSE: Sum of the squared differences between the predictions and the true values.
- Convex optimization problem with unique closed-form solution (if you have more data points than model parameters w)
- It has no hyperparameters, thus model complexity cannot be controlled.
  - Some other algorithms do: Ridge regression and Lasso


$\hat{y} = b_0 + b_1 x$
$\hat{y}_3$
y_2
$r_2 = (y_2 - \hat{y}_2)$
$r_3 = (y_3 - \hat{y}_3)$
y_3
$\hat{y}_1$
$\hat{y}_2$
$r_1 = (y_1 - \hat{y}_1)$
y_1

Linear regression can be found in `sklearn.linear_model`. We'll evaluate it on the Boston Housing dataset.

```
X, y = mglearn.datasets.load_extended_boston()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0
)
lr = LinearRegression().fit(X_train, y_train)
```

Has many large coefficients: sign of overfitting!

```
Weights (coefficients): [ -412.711   -52.243  -131.899   -12.004   -15.511
    28.716    54.704
   -49.535    26.582    37.062   -11.828   -18.058   -19.525    12.203
  2980.781  1500.843   114.187   -16.97     40.961   -24.264    57.616
  1278.121 -2239.869   222.825    -2.182    42.996   -13.398   -19.389
    -2.575   -81.013     9.66      4.914    -0.812    -7.647    33.784
   -11.446    68.508   -17.375    42.813     1.14     -0.773    56.826
    14.288    53.955   -32.171    19.271   -13.885    60.634   -12.315
   -12.004   -17.724   -33.987     7.09     -9.225    17.198   -12.772
   -11.973    57.387   -17.533     4.101    29.367   -17.661    78.405
   -31.91     48.175   -39.534     5.23     21.998    25.648   -49.998
    29.146     8.943   -71.66    -22.815     8.407    -5.379     1.201
    -5.209    41.145   -37.825    -2.672   -25.522   -33.398    46.227
   -24.151   -17.753   -13.972   -23.552    36.835   -94.689   144.303
   -15.116   -14.951   -28.773   -31.767    24.955   -18.438     3.651
     1.731    35.362    11.955     0.677     2.735    30.372]
```

Ridge regression

- Same formula as linear regression
- Adds a penalty term to the least squares sum : $\alpha \sum_i w_i^2$
- Requires that the coefficients (w) are close to zero.
  - Each feature should have as little effect on the outcome as possible
- Regularization: explicitly restrict a model to avoid overfitting.
- Type of L2 regularization: prefers many small weights
  - L1 regularization prefers sparsity: many weights to be 0, others large

`Ridge` can also be found in `sklearn.linear_model`.

```
ridge = Ridge().fit(X_train, y_train)
```

```
Training set score: 0.89
Test set score: 0.75
```

Test set score is higher and training set score lower: less overfitting!

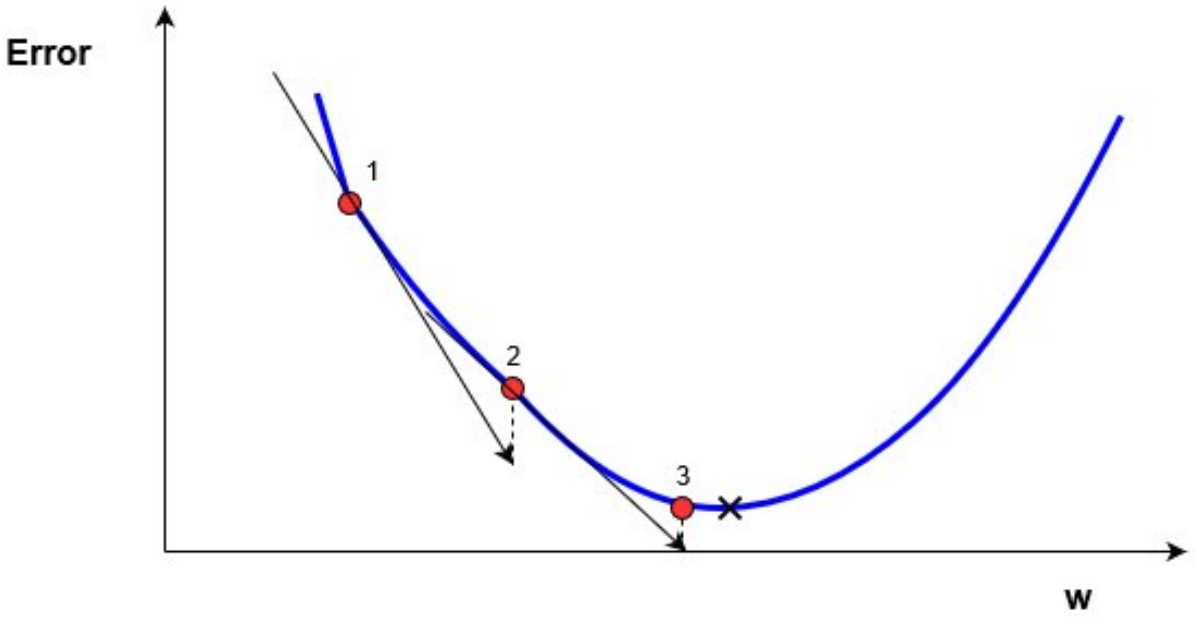
The strength of the regularization can be controlled with the `alpha` parameter. Default is 1.0.

- Increasing alpha forces coefficients to move more toward zero (more regularization)
- Decreasing alpha allows the coefficients to be less restricted (less regularization)
- Optimize alpha for your dataset

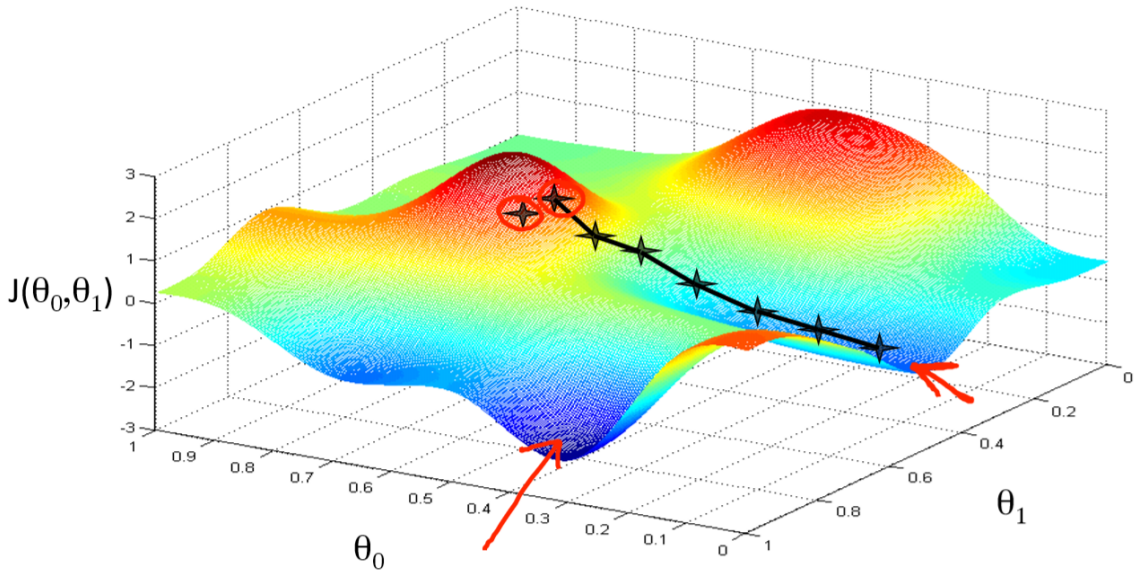
Lasso

- Another form of regularization
- Adds a penalty term to the least squares sum : $\alpha \sum_i |w_i|$
- Prefers coefficients to be exactly zero (L1 regularization).
- Some features are entirely ignored by the model: automatic feature selection.
- Same parameter `alpha` to control the strength of regularization.
- Weights are optimized using gradient descent
- New parameter `max_iter`: the maximum number of *gradient descent* iterations
    - Should be higher for small values of `alpha`

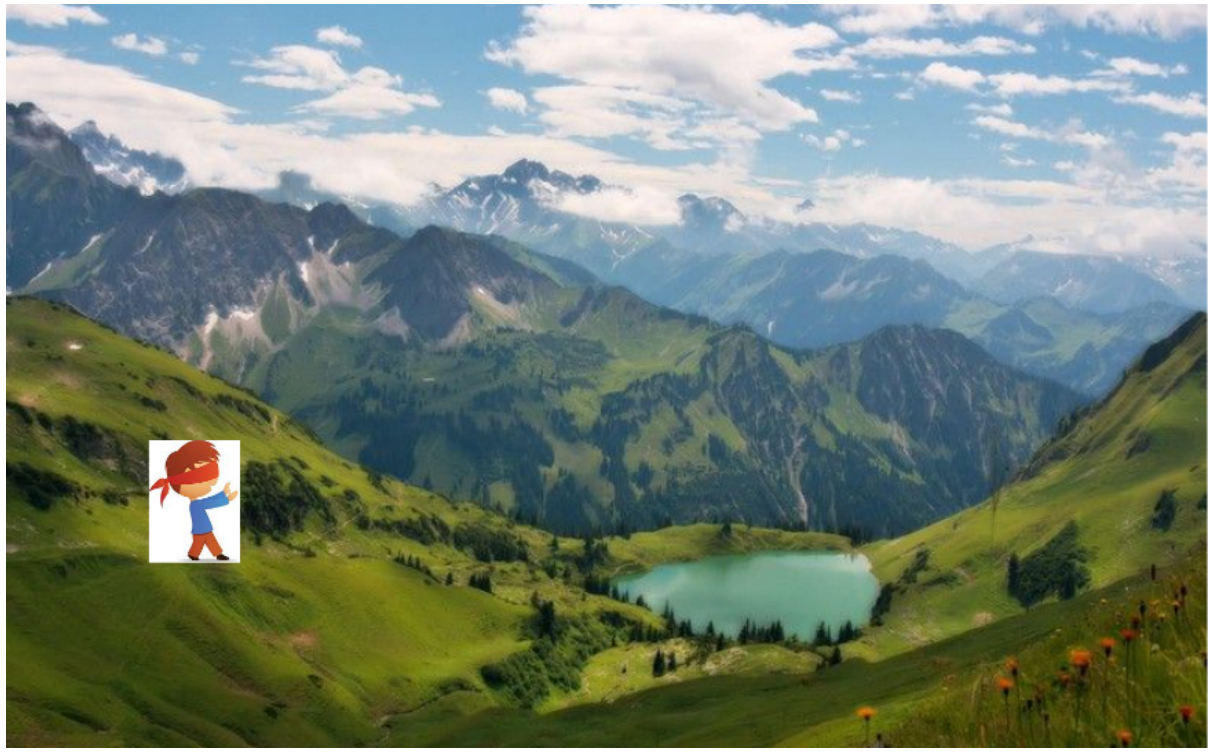
Optimization: Gradient Descent



Gradient Descent



Gradient Descent



- Run Lasso using the `Lasso` estimator
- Tune the alpha (and max_iter) to your dataset

```
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)
```

```
alpha=0.01, max_iter=100000
Training set score: 0.90
Test set score: 0.77
Number of features used: 33
```

Linear models for Classification

Aims to find a (hyper)plane that separates the examples of each class.
For binary classification (2 classes), we aim to fit the following function:

$$\hat{y} = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p + b > 0$$

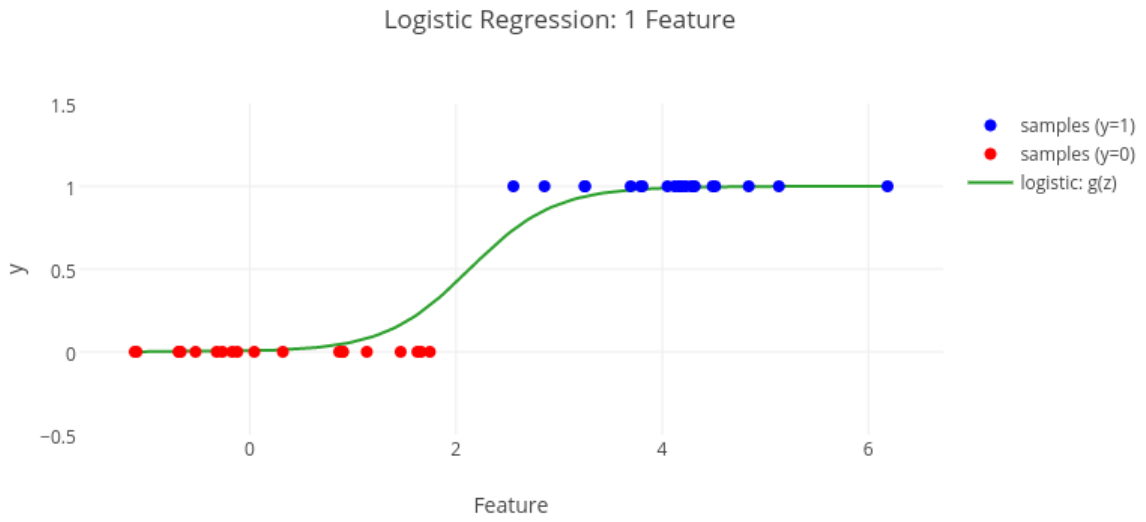
When $\hat{y} < 0$, predict class -1, otherwise predict class +1

Logistic regression

The logistic model uses the *logistic* (or *sigmoid*) function to estimate the probability that a given sample belongs to class 1:

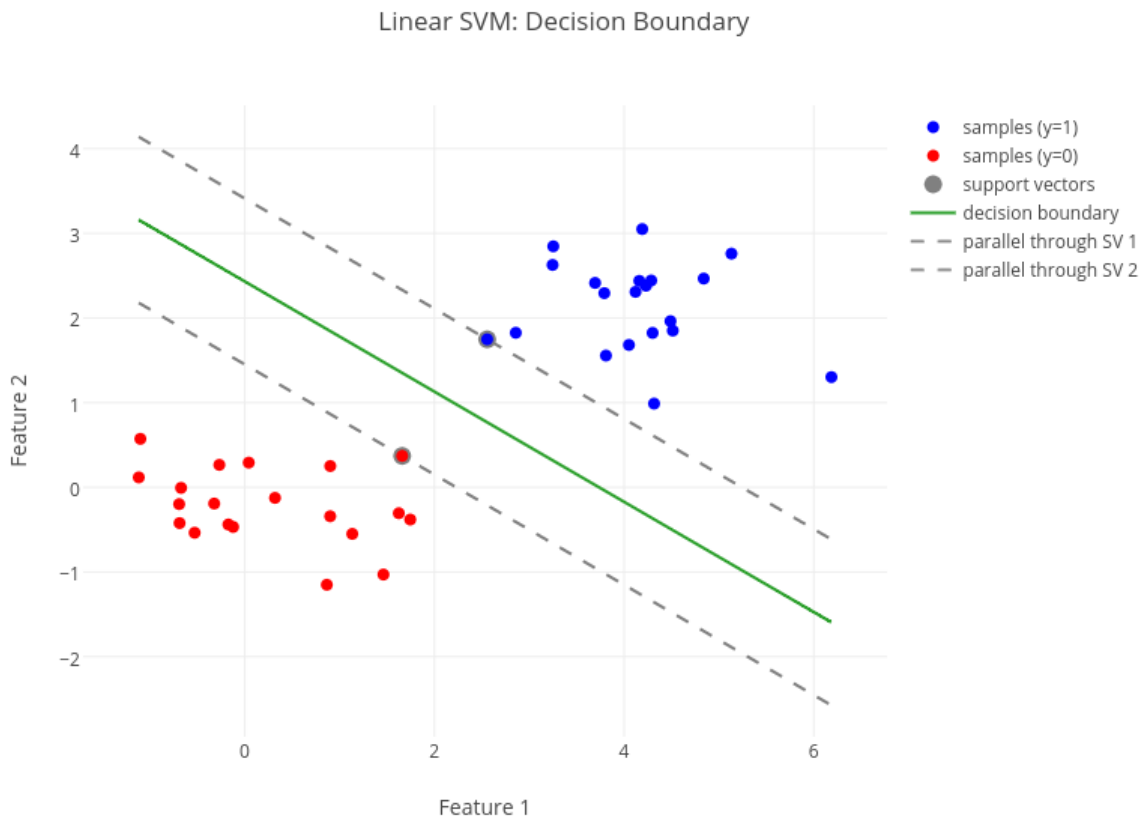
$$z = f(x) = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p$$

$$\hat{y} = Pr[1|x_1, \ldots, x_k] = g(z) = \frac{1}{1 + e^{-z}}$$



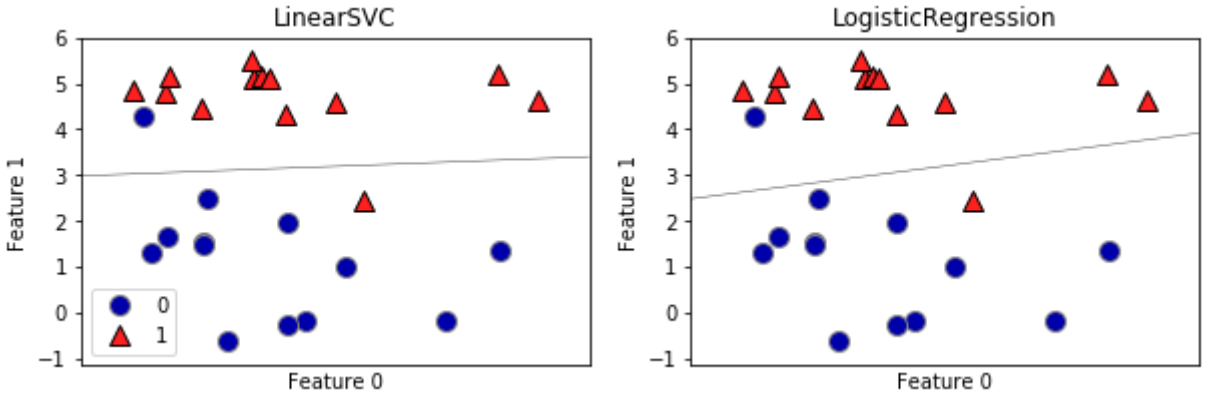
Linear Support Vector Machine

Find hyperplane maximizing the *margin* between the classes



Prediction is identical to weighted kNN: find the support vector that is nearest, according to a distance measure (kernel) and a weight for each support vector.

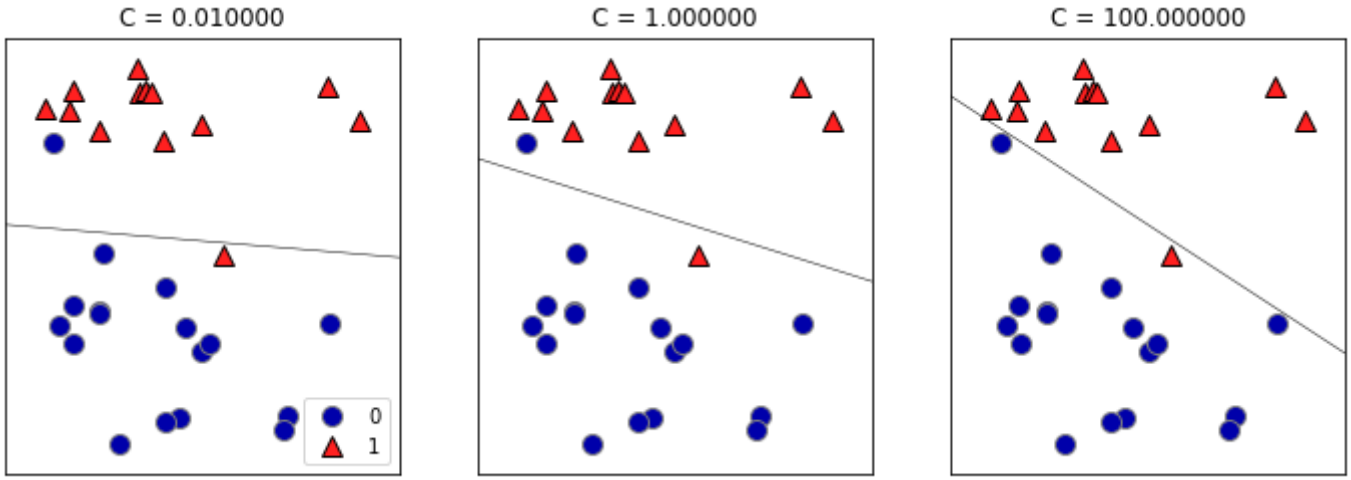
- Logistic regression can be run with `linear_model.LogisticRegression`
- Linear SVMs can be run with `svm.LinearSVC`



Both methods can be regularized:

- L2 regularization by default, L1 also possible
- *C* parameter: inverse of strength of regularization
  - higher *C*: less regularization
  - penalty for misclassifying points while keeping w_i close to 0

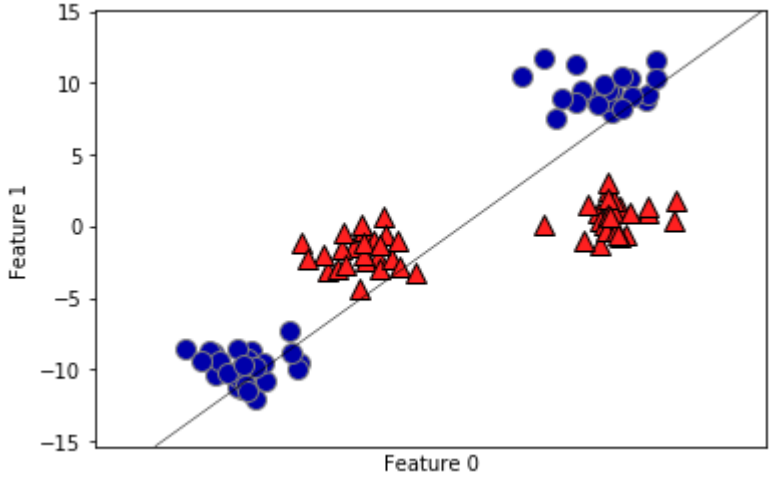
```
logreg = LogisticRegression(C=1,penalty=l2).fit(X_train,
y_train)
```



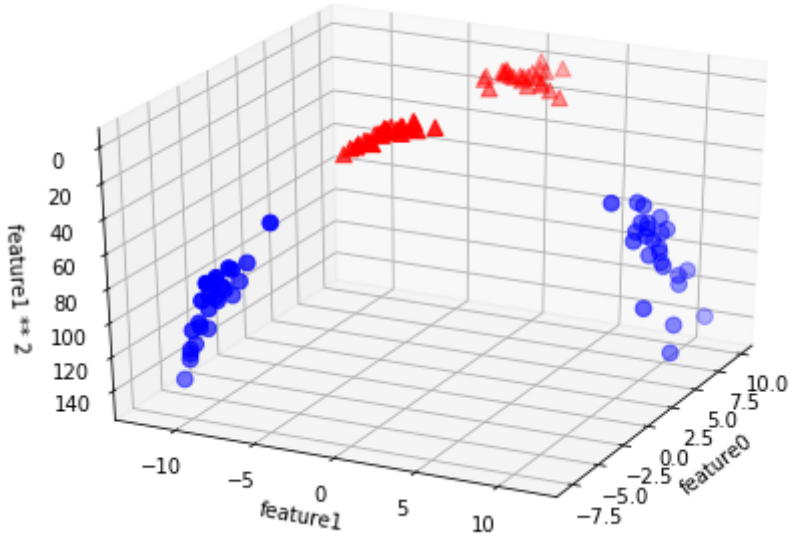
Kernelized Support Vector Machines

- Linear models work well in high dimensional spaces.
- You can *create* additional dimensions yourself.
- Let's start with an example.

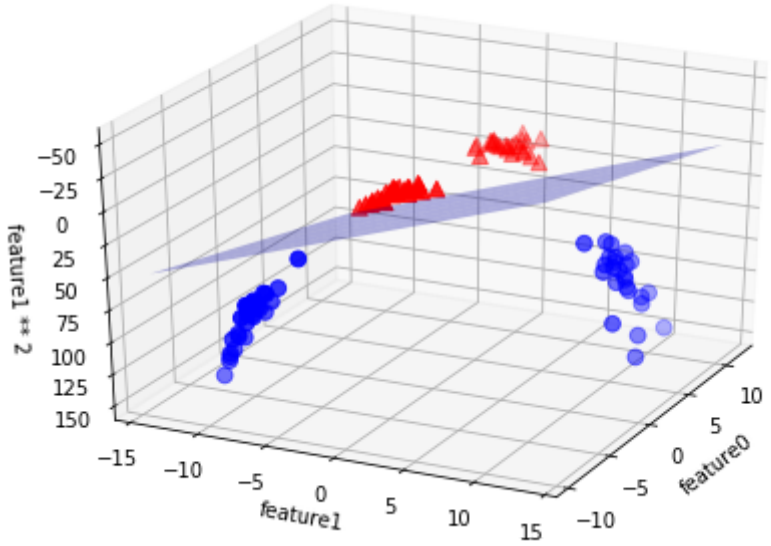
Our linear model doesn't fit the data well



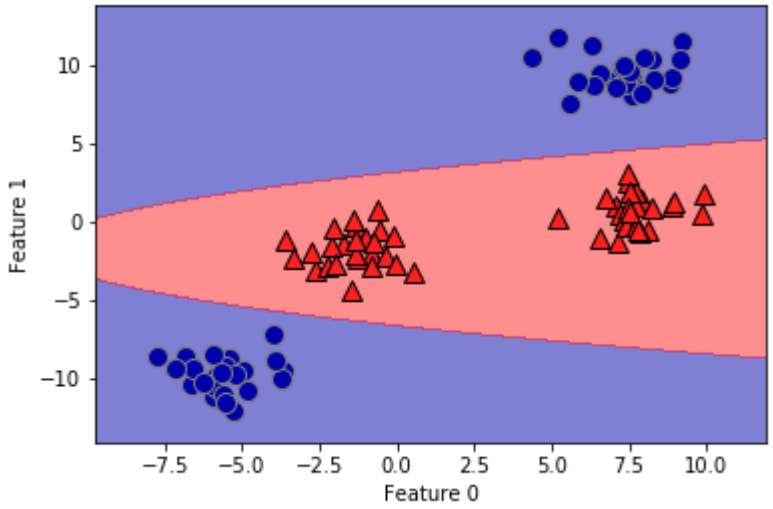
We can add a new feature by taking the squares of feature1 values



We can now fit a linear model



As a function of the original features, the linear SVM model is not actually linear anymore, but more of an ellipse



Kernels

A (Mercer) Kernel on a space X is a (similarity) function

$k : X \times X \to \mathbb{R}$

Of two arguments with the properties:

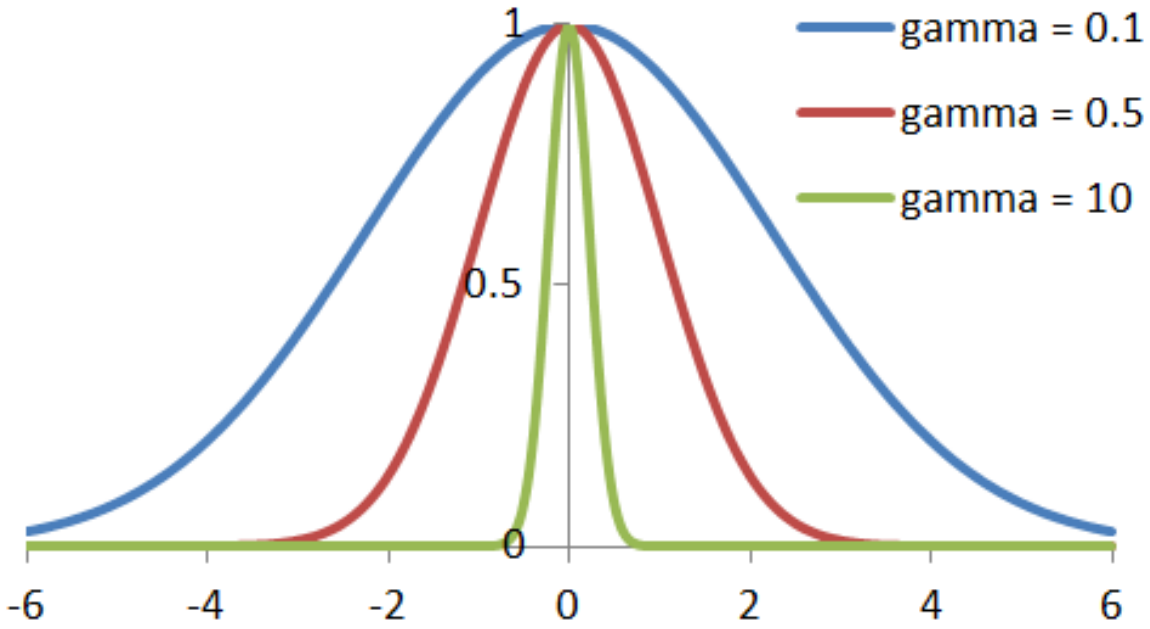
- Symmetry: $k(x_1, x_2) = k(x_2, x_1) \quad \forall x_1, x_2 \in X$
- Positive definite: for each finite subset of data points $x_1, \ldots, x_n$, the kernel Gram matrix is positive semi-definite

Kernel matrix = $K \in \mathbb{R}^{n \times n}$ with $K_{ij} = k(x_i, x_j)$

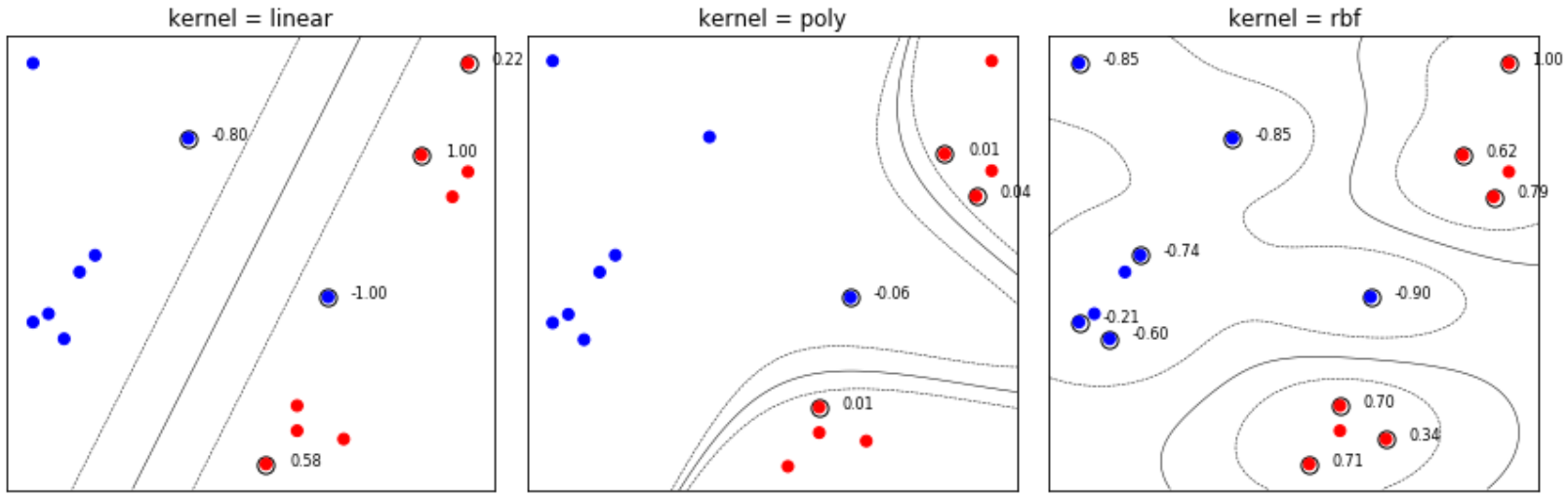
Kernels: examples

- The inner product is a kernel. The standard inner product is the **linear kernel**:
  $k(x_1, x_2) = x_1^T x_2$
- Kernels can be constructed from other kernels k_1 and k_2:
  - For $\lambda \geq 0$, $\lambda . k_1$ is a kernel
  - $k_1 + k_2$ is a kernel
  - $k_1 . k_2$ is a kernel (thus also k_1^n)
- This allows to construct the **polynomial kernel**:
  $k(x_1, x_2) = (x_1^T x_2 + b)^d$, for $b \geq 0$ and $d \in \mathbb{N}$

- The 'radial base fucntion' (or **Gaussian**) kernel is defined as:
  $k(x_1, x_2) = exp(-\gamma||x_1 - x_2||^2)$, for $\gamma \geq 0$

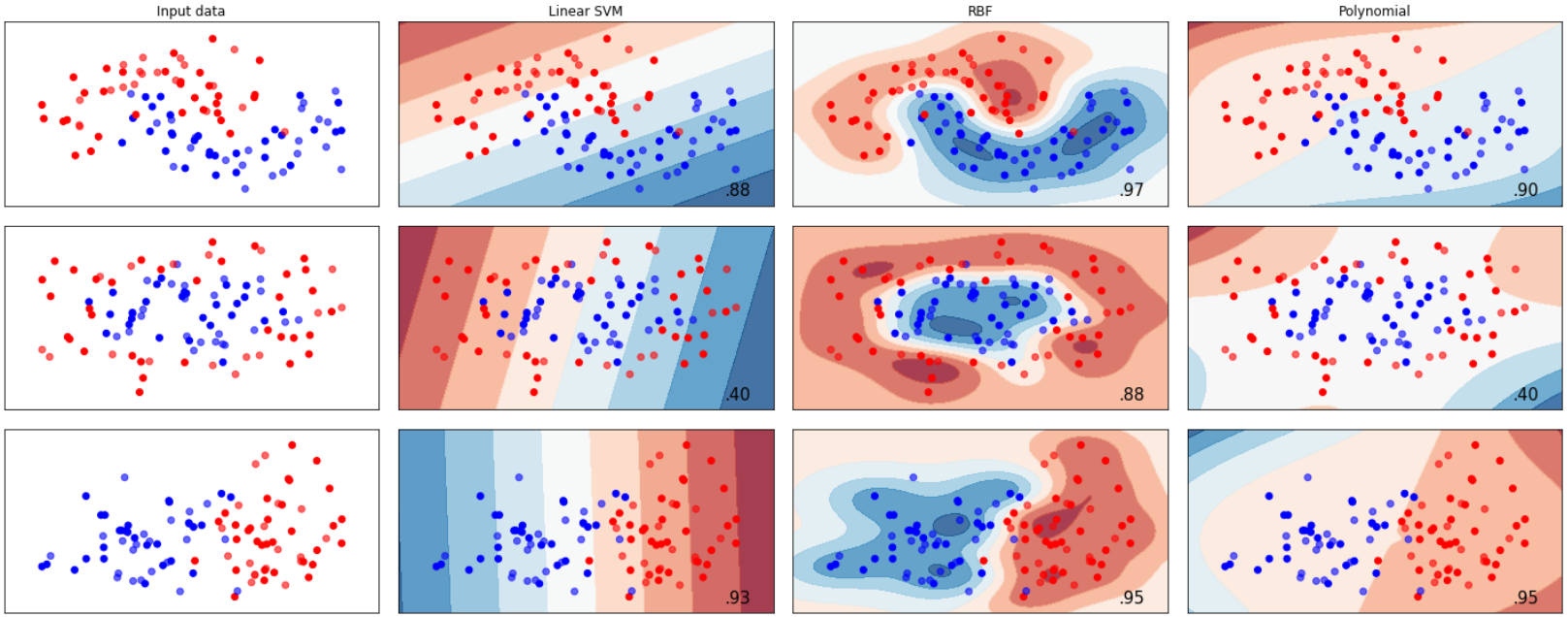


Different kernels lead to different decision boundaries, because the distance to the support vectors is measured differently.



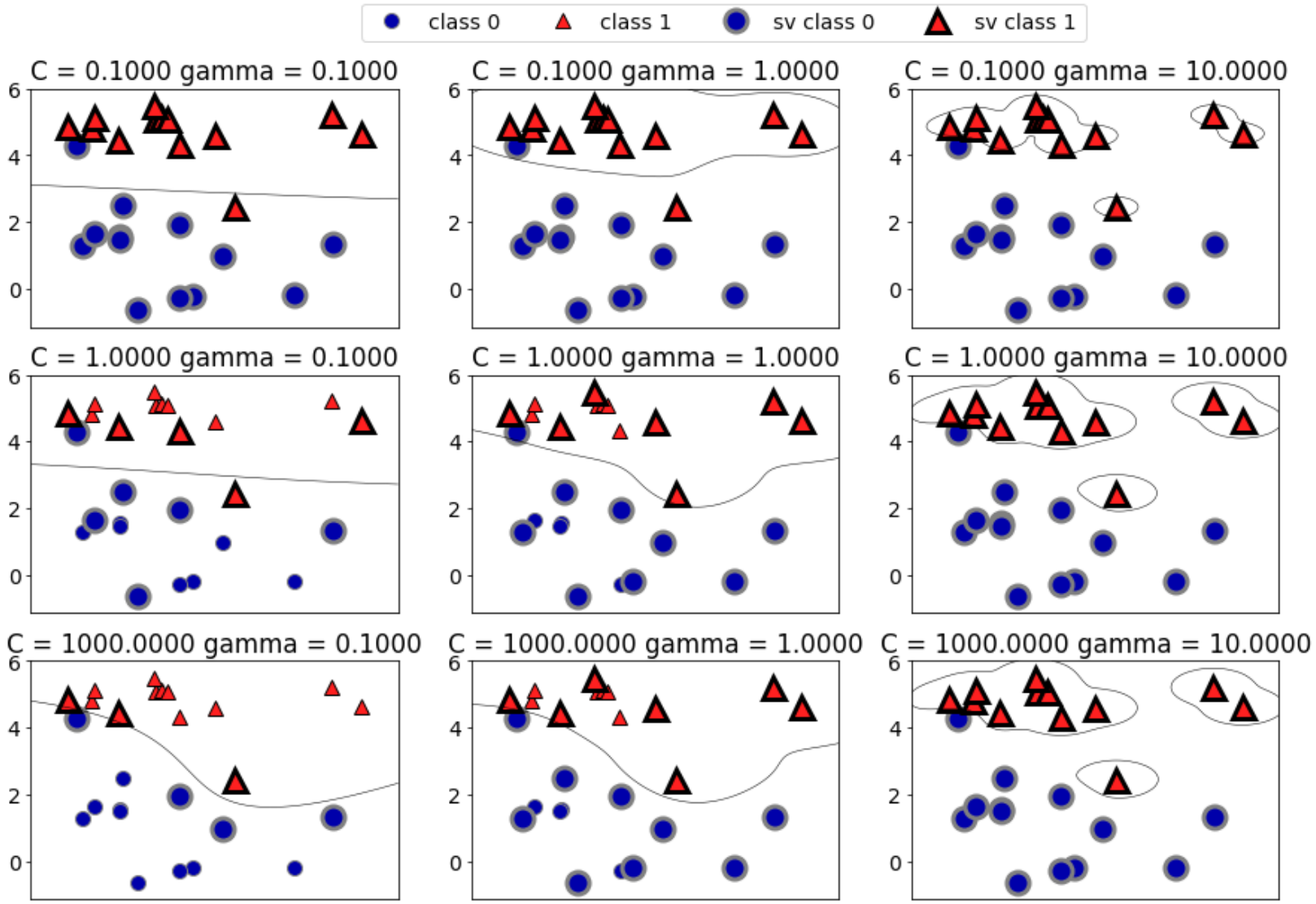
The first important hyperparameter to tune is the choice of kernel

- RBF is *usually* best


Input data
Linear SVM
RBF
Polynomial
.88
.97
.90
.40
.88
.40
.93
.95
.95

But also:

- the amount of regulatization ('C')
  - Too low: underfitting, Too high: overfitting
- the hyperparameters of the kernel itself, e.g. 'gamma'
  - Too low: underfitting, Too high: overfitting


class 0
class 1
sv class 0
sv class 1
C = 0.1000 gamma = 0.1000
C = 0.1000 gamma = 1.0000
C = 0.1000 gamma = 10.0000
C = 1.0000 gamma = 0.1000
C = 1.0000 gamma = 1.0000
C = 1.0000 gamma = 10.0000
C = 1000.0000 gamma = 0.1000
C = 1000.0000 gamma = 1.0000
C = 1000.0000 gamma = 10.0000

SVMs: Strengths, weaknesses and parameters

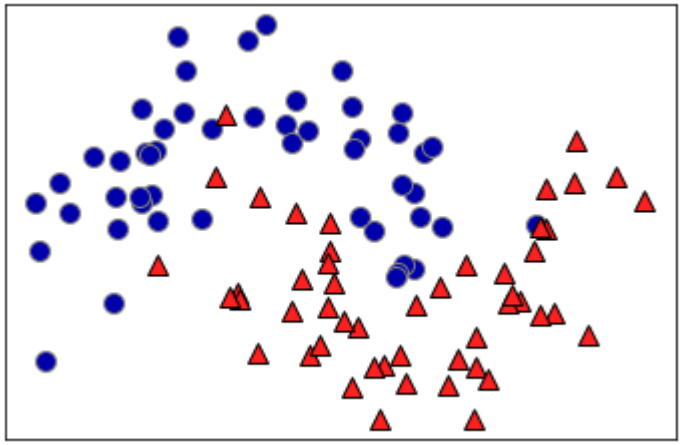
- SVMs allow complex decision boundaries, even with few features.
- Work well on both low- and high-dimensional data
- Don't scale very well to large datasets (>100000)
- Require careful preprocessing of the data and tuning of the parameters.
- SVM models are hard to inspect

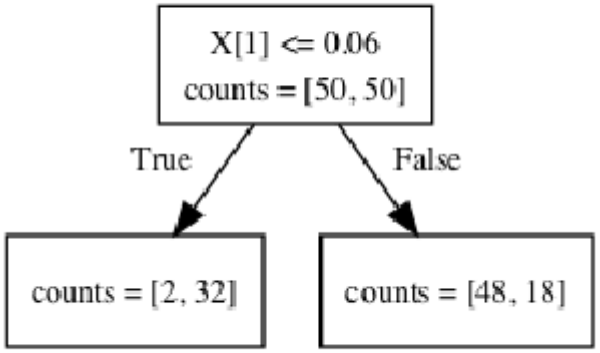
Important parameters:

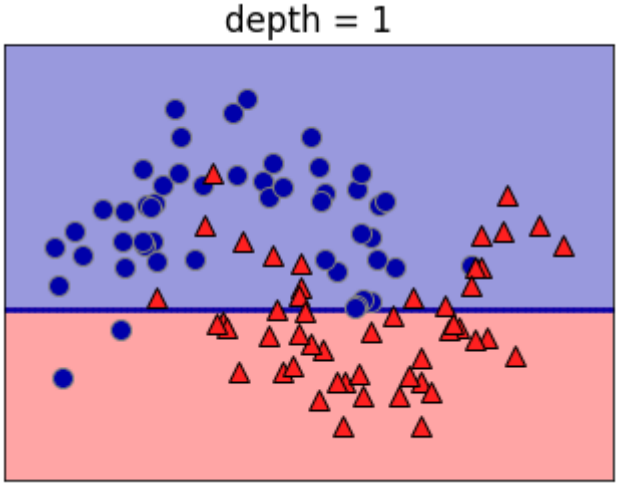
- regularization parameter *C*
- choice of the kernel and kernel-specific parameters
  - Typically string correlation with *C*

Decision Trees

- Split the data in two (or more) parts
- Search over all possible splits and choose the one that is most *informative*
  - Many heuristics
  - E.g. *information gain*: how much does the entropy of the class labels decrease after the split (purer 'leafs')
- Repeat recursive partitioning
- In scikit-learn: `tree.DecisionTreeClassifier`




X[1] <= 0.06
counts = [50, 50]
True
False
counts = [2, 32]
counts = [48, 18]


depth = 1

Overfitting: Controlling complexity of Decision Trees

Decision trees can very easily overfit the data. Regularization strategies:

- Pre-pruning: stop creation of new leafs at some point
  - Limiting the depth of the tree, or the number of leafs
    - Use lower `max_depth`, `max_leaf_nodes`
  - Requiring a minimal leaf size (number of instances)
    - Use higher `min_samples_leaf` (default=1)
- Post-pruning: build full tree, then prune (join) leafs
  - Reduced error pruning: evaluate against held-out data
  - Many other strategies exist.
  - scikit-learn supports none of them (yet)

Decision trees for regression

- Heuristic: Minimal quadratic distance
- Consider splits at every data point for every variable
- Choose splits so that predicting the average of all leaf values gives the smallest error

Decision trees: Strengths, weaknesses and parameters

- Work well with features on completely different scales, or a mix of binary and continuous features
  - Does not require normalization
- Interpretable, easily visualized
- Do not extrapolate well
- Still tend to overfit easily. Use ensembles of trees.

Ensemble learning

Ensembles are methods that combine multiple machine learning models to create more powerful models. Most popular are:

- **RandomForests**: Build randomized trees on random samples of the data
- **Gradient boosting machines**: Build trees iteratively, giving higher weights to the points misclassified by previous trees

In both cases, predictions are made by doing a vote over the members of the example.
Stacking is another technique that builds a (meta)model over the predictions of each member.

RandomForests

Reduce overfitting by averaging out individual predictions (variance reduction)
In scikit-learn: `ensemble.RandomForestClassifier`

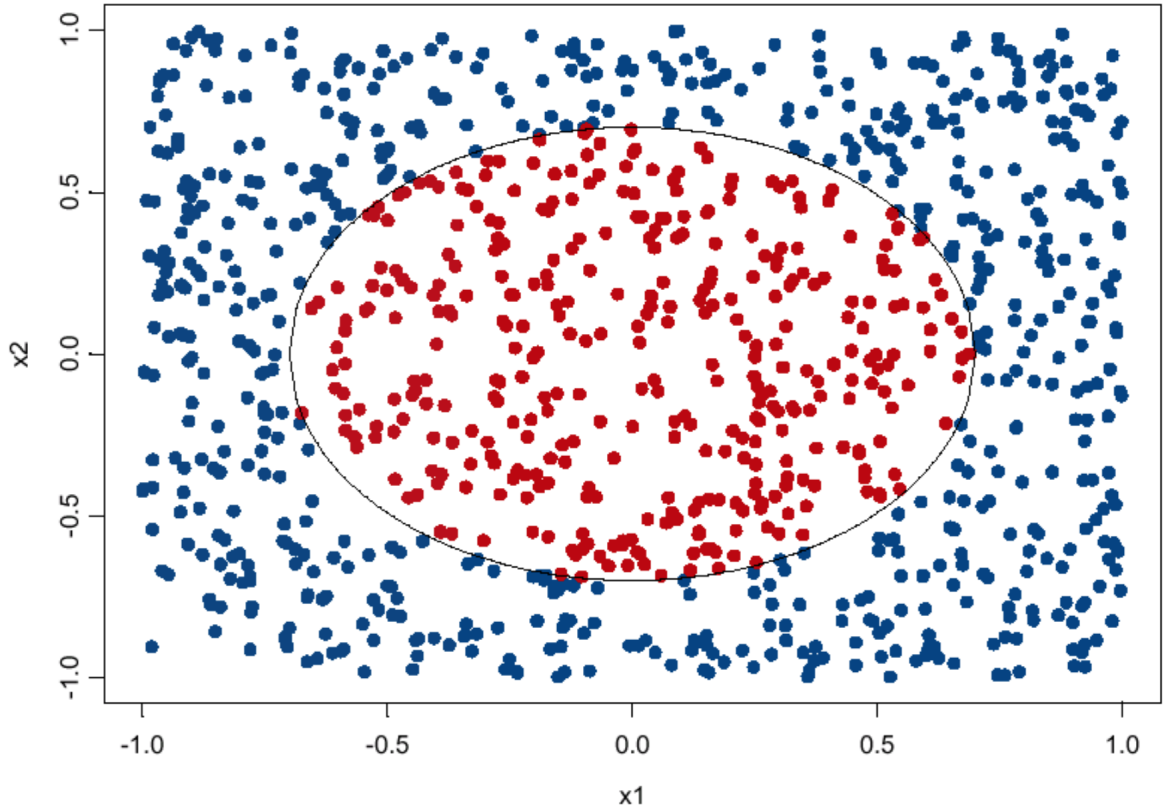
- Take a *bootstrap sample* of your data
  - Randomly sample with replacement
  - Build a tree on each bootstrap
- Repeat `n_estimators` times
  - Higher values: more trees, more smoothing
  - Make prediction by aggrating the individual tree predictions
    - a.k.a. Bootstrap aggregating (Bagging)
- RandomForest: Randomize trees by considering only a random subset of features of size `max_features` *in each node*
  - Small `max_features` yields more different trees, more smoothing
  - Default: $sqrt(n\_features)$ for classification, $log2(n\_features)$ for regression

Gradient Boosting

Instead of reducing the variance of overfitted models, reduce the bias of underfitted models
In scikit-learn: `ensemble.GradientBoostingClassifier`

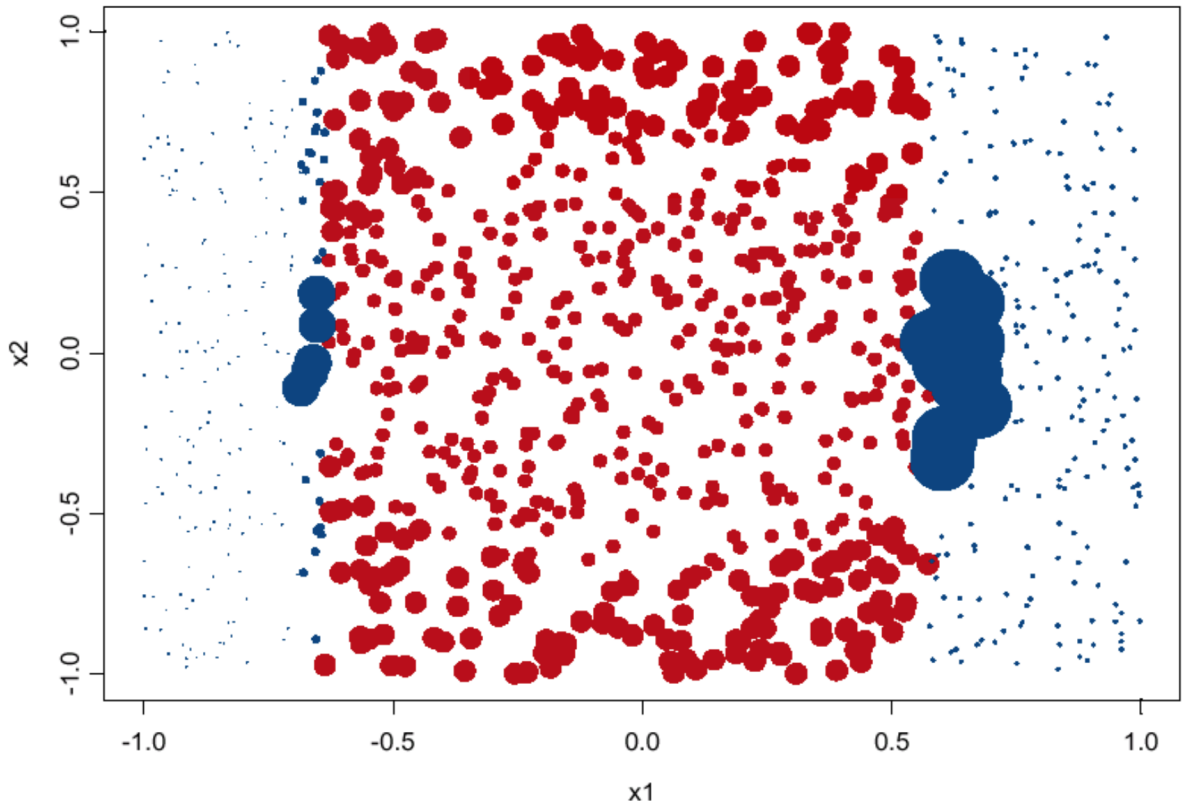
- Use strong pre-pruning to build very shallow trees
  - Default `max_depth`=3
- Iteratively build new trees by increasing weights of points that were badly predicted
- Example of *additive modelling*: each tree depends on the outcome of previous trees
- Optimization: find optimal weights for all data points
  - Gradient descent (covered later) finds optimal set of weights
  - `learning rate` controls how strongly the weights are altered in each iteration (default 0.1)
- Repeat `n_estimators` times (default 100)

Example:

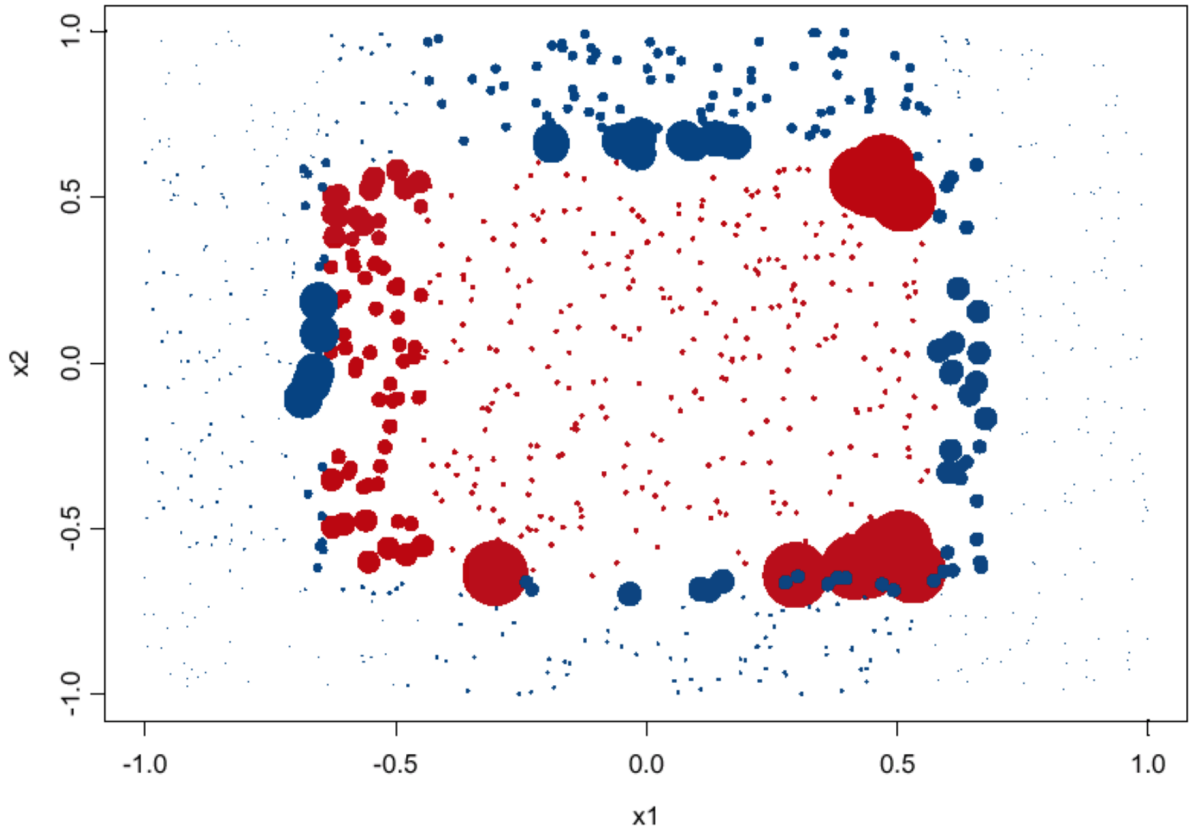


After 1 iteration

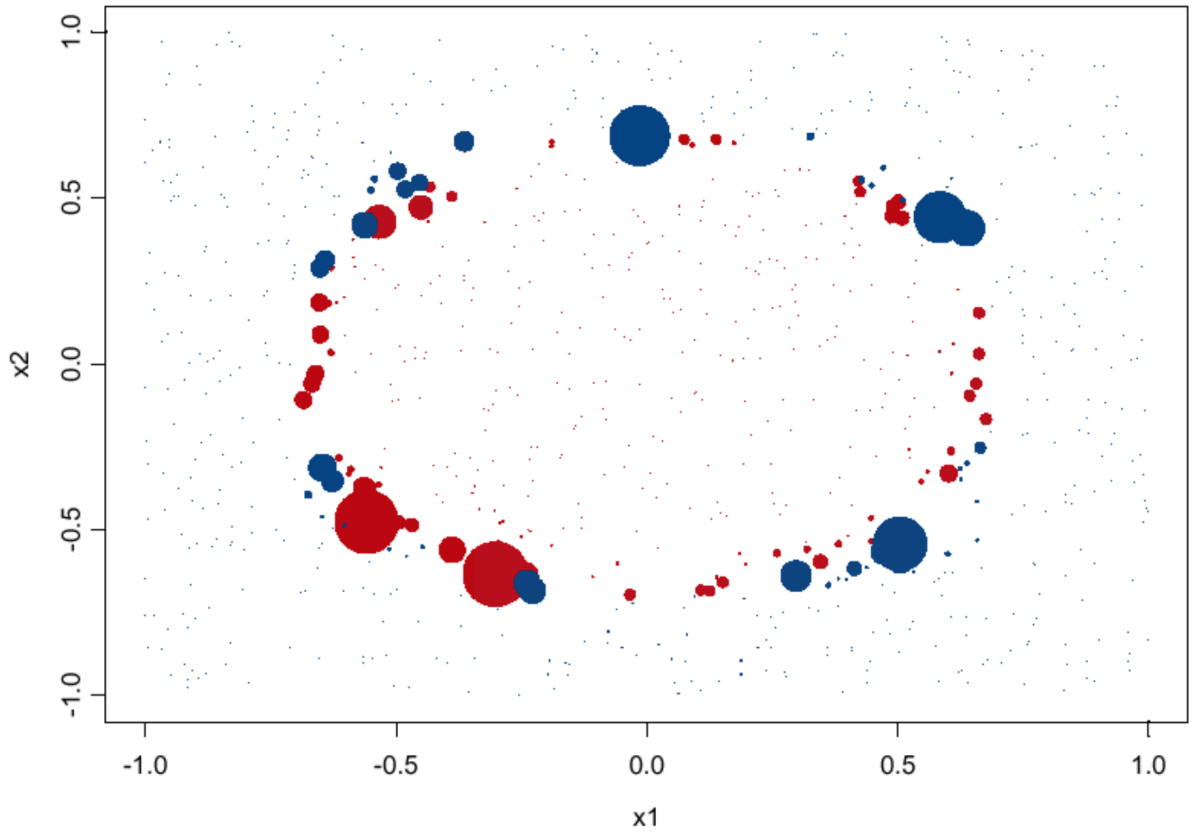
- The simple decision tree divides space
- Misclassified points get higher weight (larger dots)



After 3 iterations



After 20 iterations



Many more algorithms:

- Probabilistic techniques
  - Naive Bayes
  - Bayesian Networks
  - Gaussian Processes
- Graphical models
  - Hidden Markov models
  - ...
- Neural Networks
  - See next week