Building Information Retrieval and Data Mining systems

In practice

Overview

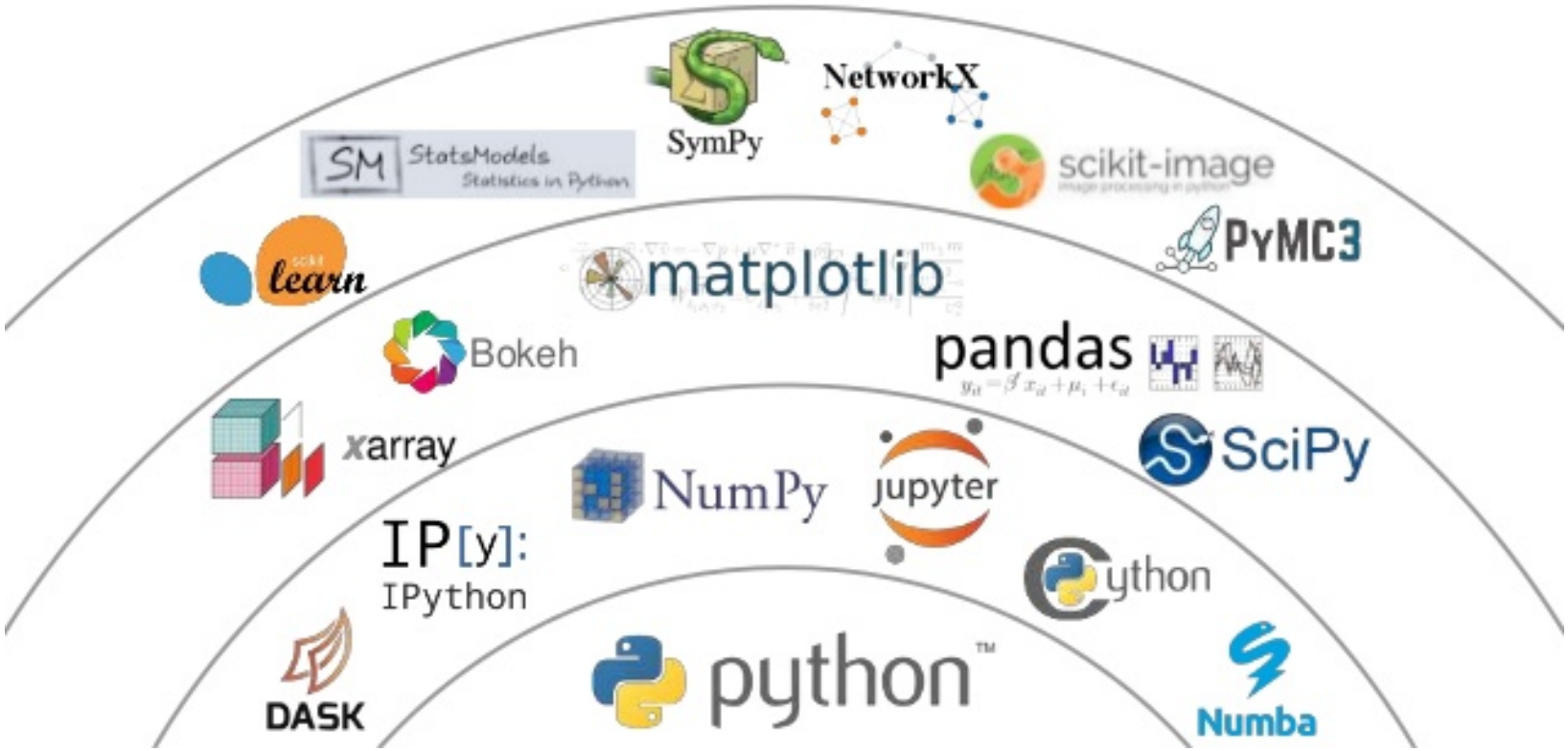
- Getting started: Python ecosystem
- Data preprocessing
- Machine Learning systems
  - Classification:
  - Regression
  - Evaluation
- Deep Learning
- Next:
  - Data and experiment management with OpenML
  - Hands-on exercises

Why Python?

- Many data-heavy applications are now developed in Python
- Highly readable, less complexity, fast prototyping
- Easy to offload number crunching to underlying C/Fortran/...
- Easy to install and import many rich libraries
  - numpy: efficient data structures
  - scipy: fast numerical recipes
  - matplotlib: high-quality graphs
  - scikit-learn: machine learning algorithms
  - tensorflow: neural networks
  - ...

Python's Scientific Stack

Jake Vanderplas PyCon 2017 Keynote



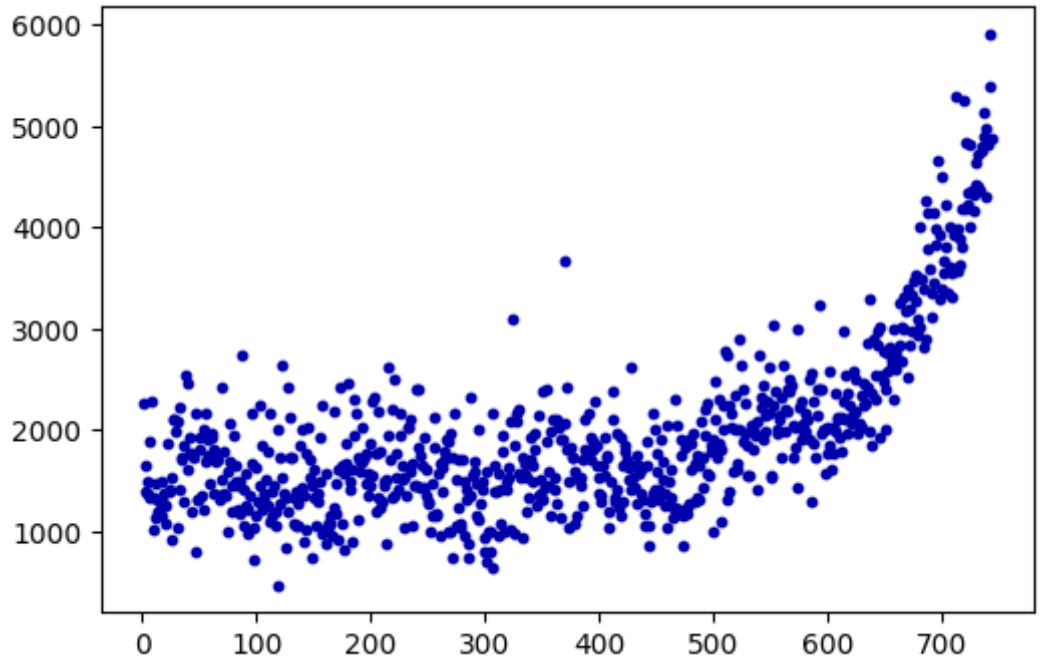
Numpy, Scipy, Matplotlib

- We'll illustrate these with a practical example
- Many good tutorials online
  - Jake VanderPlas' book and notebooks (https://github.com/jakevdp/PythonDataScienceHandbook)
  - J.R. Johansson's notebooks (https://github.com/jrjohansson/scientific-python-lectures)
  - DataCamp (https://www.datacamp.com)
  - ...

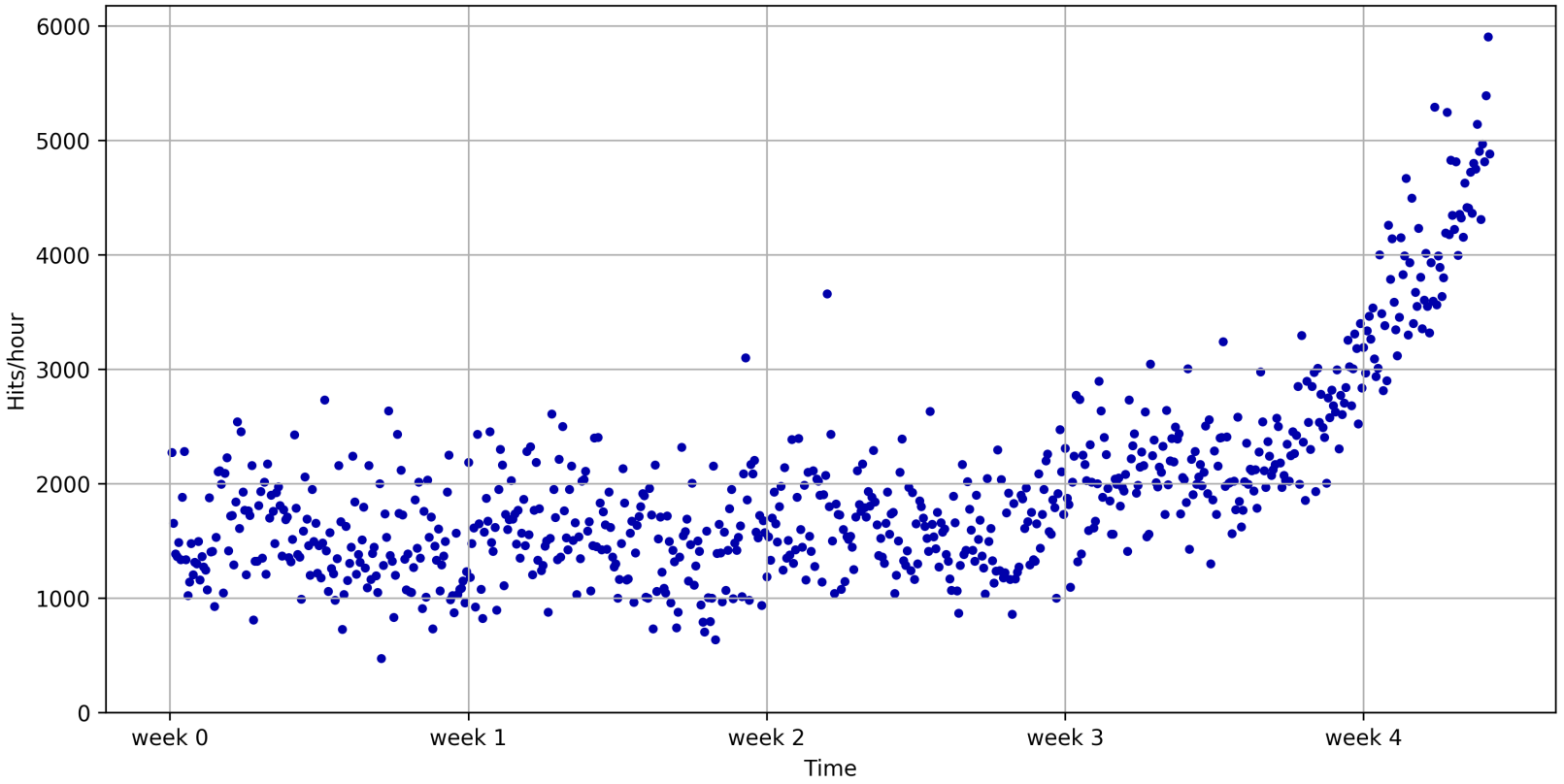
Example: Modelling web traffic

- We generate some artificial data to mimic web traffic data
  - E.g. website visits, tweets with certain hashtag,...
  - Weekly rhythm + noise + exponential increase

```
x, y = gen_web_traffic_data()
plt.scatter(x, y, s=10)
```



- With additional matplotlib layout

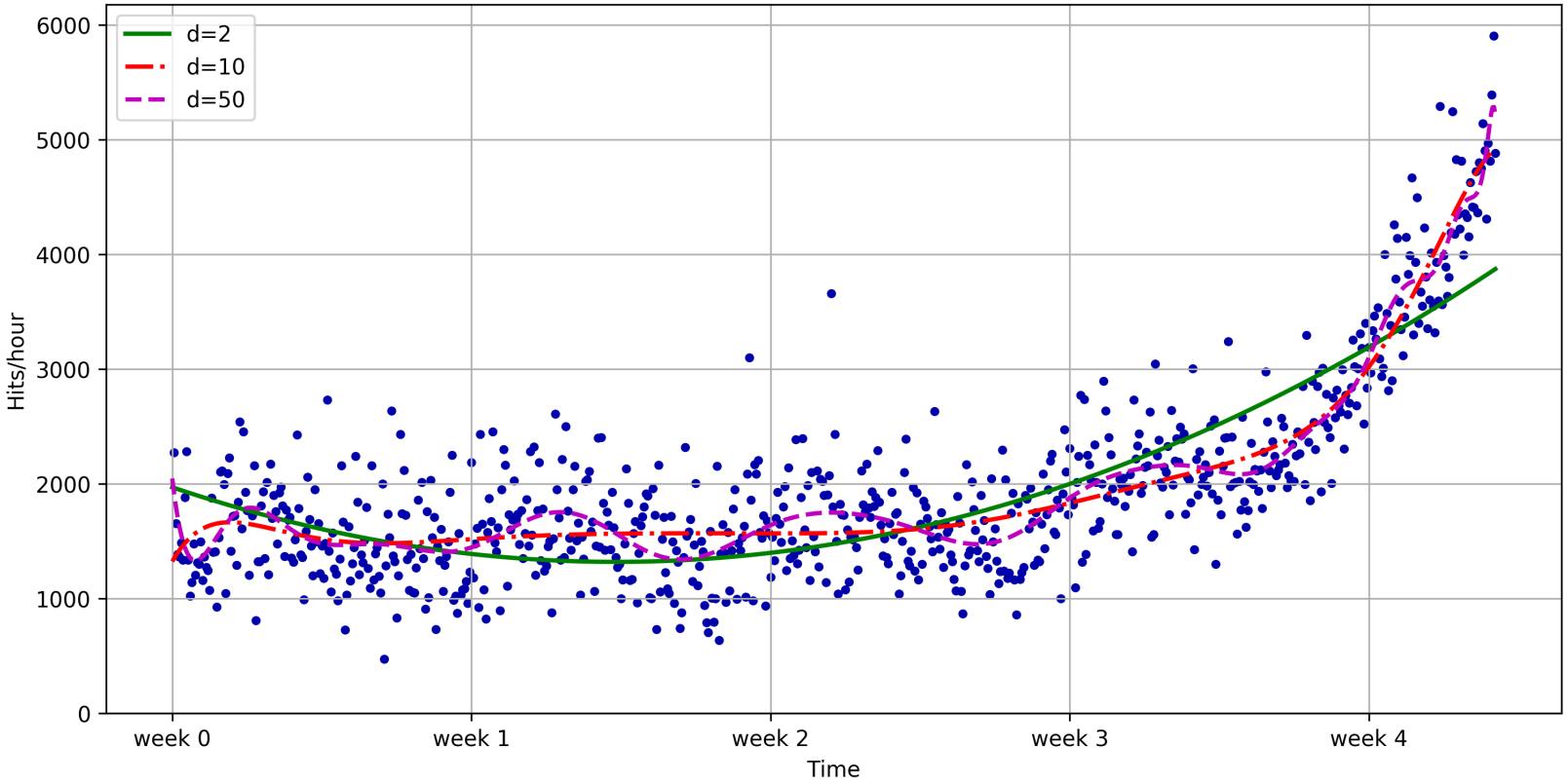


Use numpy to fit some polynomial lines

- `polyfit` fits a polynomial of degree d
- `poly1d` evaluates the function using the learned coefficients
- Plot with matplotlib

```
f2 = np.poly1d(np.polyfit(x, y, 2))
f10 = np.poly1d(np.polyfit(x, y, 10))
f50 = np.poly1d(np.polyfit(x, y, 50))

mx = np.linspace(0, x[-1], 1000)
plt.plot(mx, f2(mx))
```


d=2
d=10
d=50
6000
5000
4000
3000
2000
1000
0
Hits/hour
week 0
week 1
week 2
week 3
week 4
Time

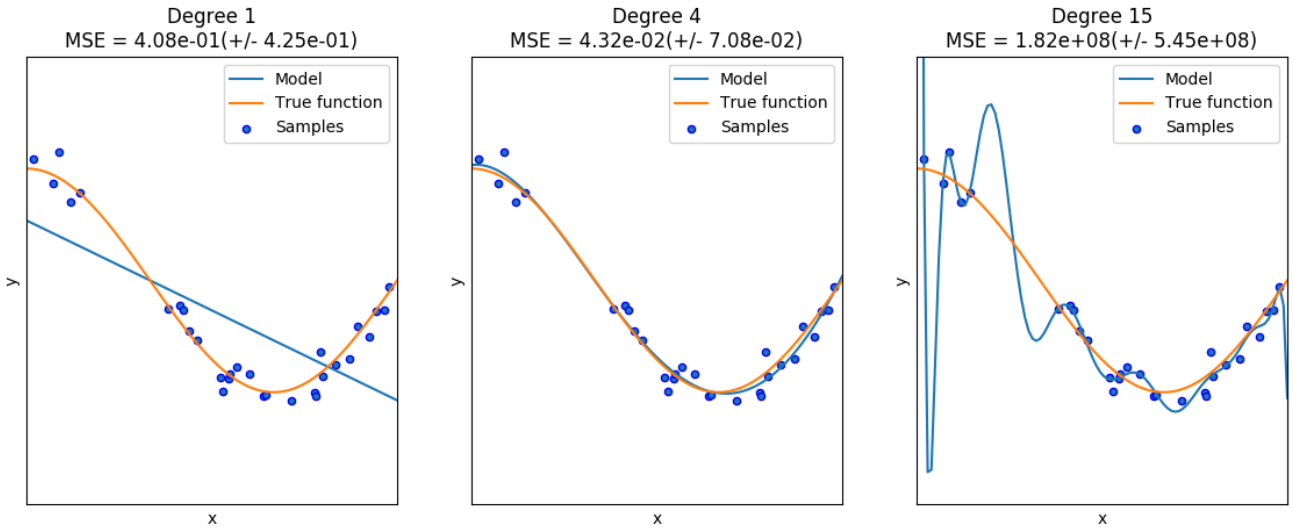
Evaluate

- Using root mean squared error: $\sqrt{\sum_i (f(x_i) - y_i)^2}$
- The degree of the polynomial needs to be tuned to the data

Generalization, Overfitting and Underfitting

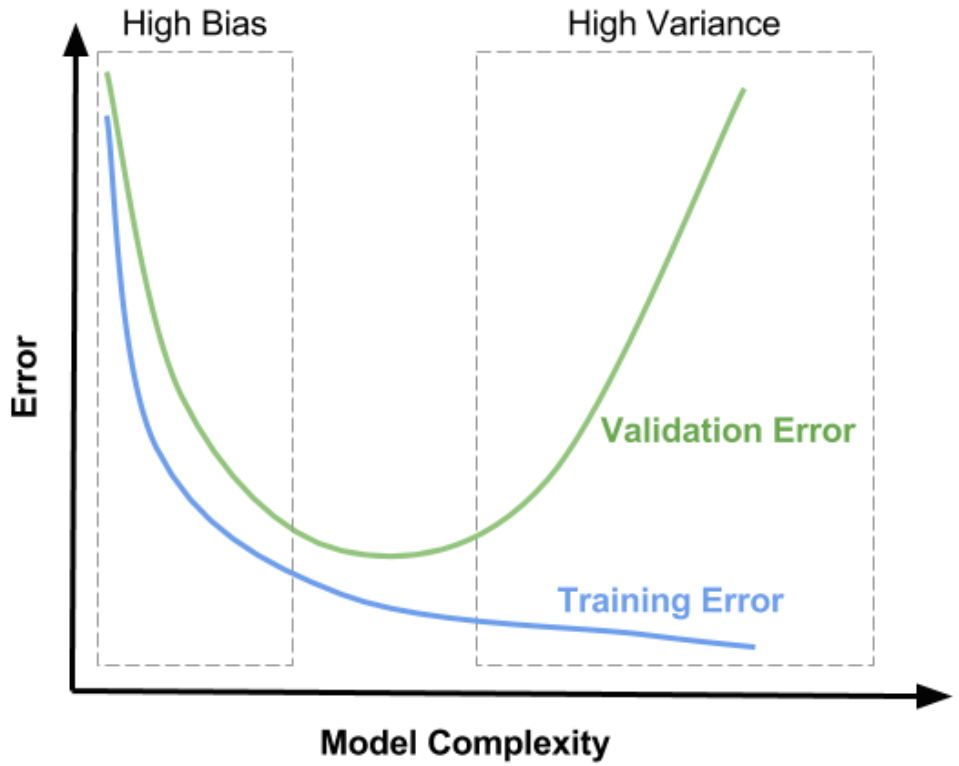
- We **hope** that the model can *generalize* from the training to future data: make accurate predictions on unseen data
- It's easy to build a complex model that is 100% accurate on the training data, but very bad on the test data
- Overfitting: building a model that is *too complex for the amount of 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.

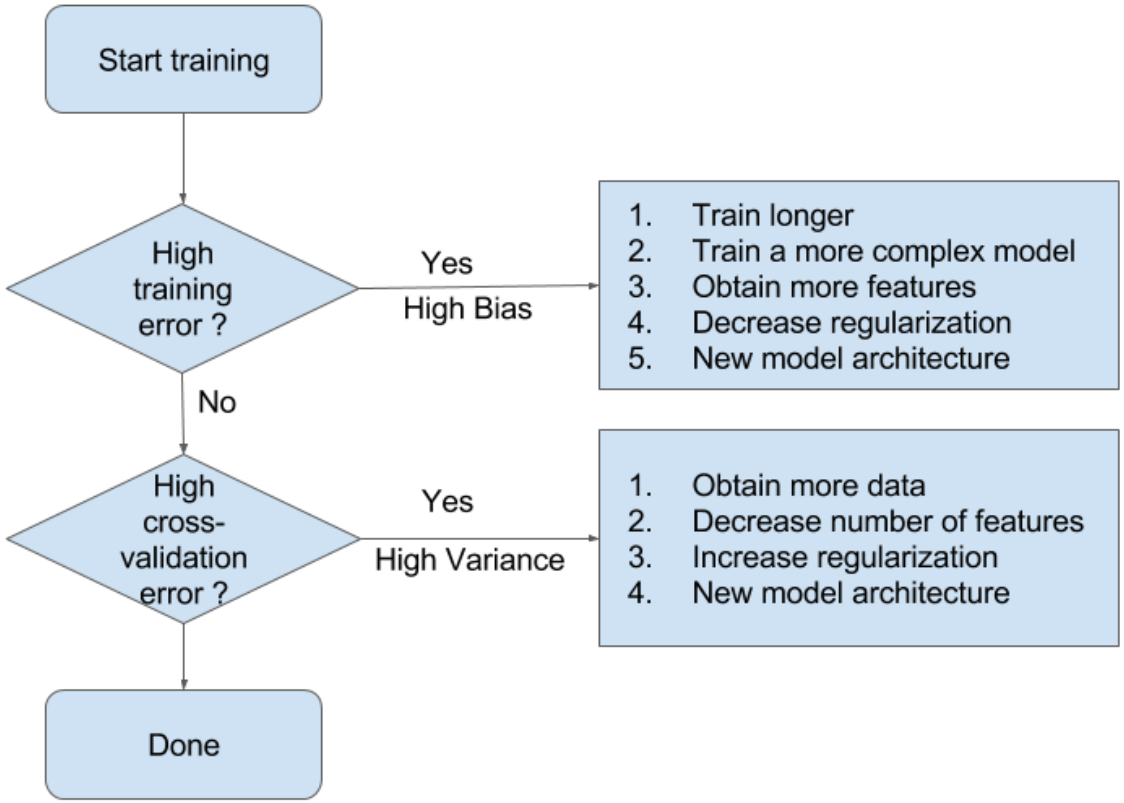


Under and Overfitting

- Split the data in training and test sets (holdout, cross-validation,...) and select models based on test sets
- Underfitting:
  - High bias: data points are always predicted wrong
  - Bad on training and test sets
- Overfitting:
  - High variance: predictions change depending on training samples
  - Good on training data, bad on test data



Bias-Variance Flowchart (Andrew Ng, Coursera)



scikit-learn

One of the most prominent Python libraries for machine learning:

- Contains many state-of-the-art machine learning algorithms
- Wide range of evaluation measures and techniques
- Offers comprehensive documentation (http://scikit-learn.org/stable/documentation) about each algorithm
- Widely used, and a wealth of tutorials (http://scikit-learn.org/stable/user_guide.html) and code snippets are available
- Works well with numpy, scipy, pandas, matplotlib,...

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`
- Import 1000s of datasets via `sklearn.datasets.fetch_openml`
- You can import data files (CSV) with `pandas` or `numpy`

```
from sklearn.datasets import load_iris, fetch_openml
iris_data = load_iris()
dating_data = fetch_openml(name="SpeedDating")
```

Building models

All scikitlearn *estimators* follow the same interface

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

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

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

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

Training and testing data

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

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

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

Fitting a model

The first model we'll build is a k-Nearest Neighbor classifier.
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[8]:
```
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)
class_name = iris_data['target_names'][prediction]
```

```
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]
```

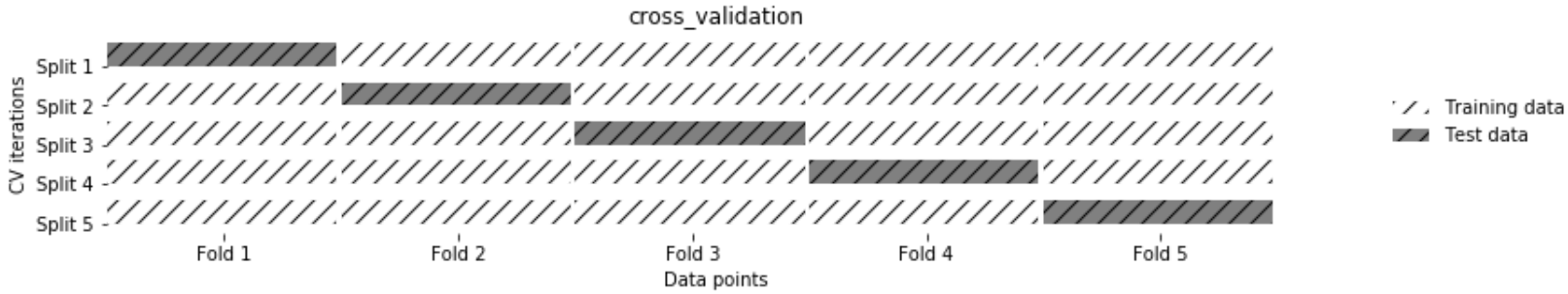
The `score` function computes the percentage of correct predictions

```
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

```
knn = KNeighborsClassifier(n_neighbors=1)
scores = cross_val_score(knn, 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: [0.98  0.922 1.   ]
Average cross-validation score: 0.97
Variance in cross-validation score: 0.0011
```

More variants

- 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

Avoid data leakage

- Simply taking the best performing model based on cross-validation performance yields optimistic results
- We've already used the test data to evaluate each model!
- Hence, we don't have an independent test set to evaluate these hyperparameter settings
  - Information 'leaks' from test set into the final model
- Solution: Set aside part of the training data to evaluate the hyperparameter settings
  - Select best model on validation set
  - Rebuild the model on the training+validation set
  - Evaluate optimal model on the test set

Model selection and Hyperparameter tuning

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

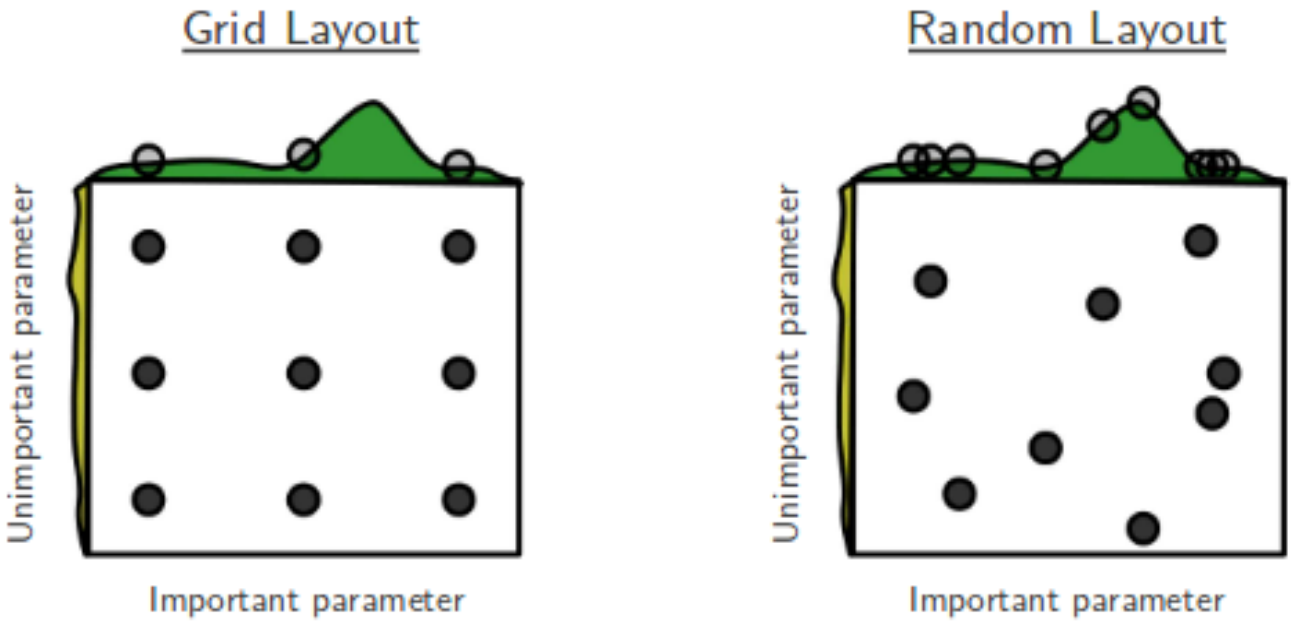
We can basically use any optimization technique to optimize hyperparameters:

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

More advanced techniques:

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

Grid vs Random Search



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 combinations 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_
```

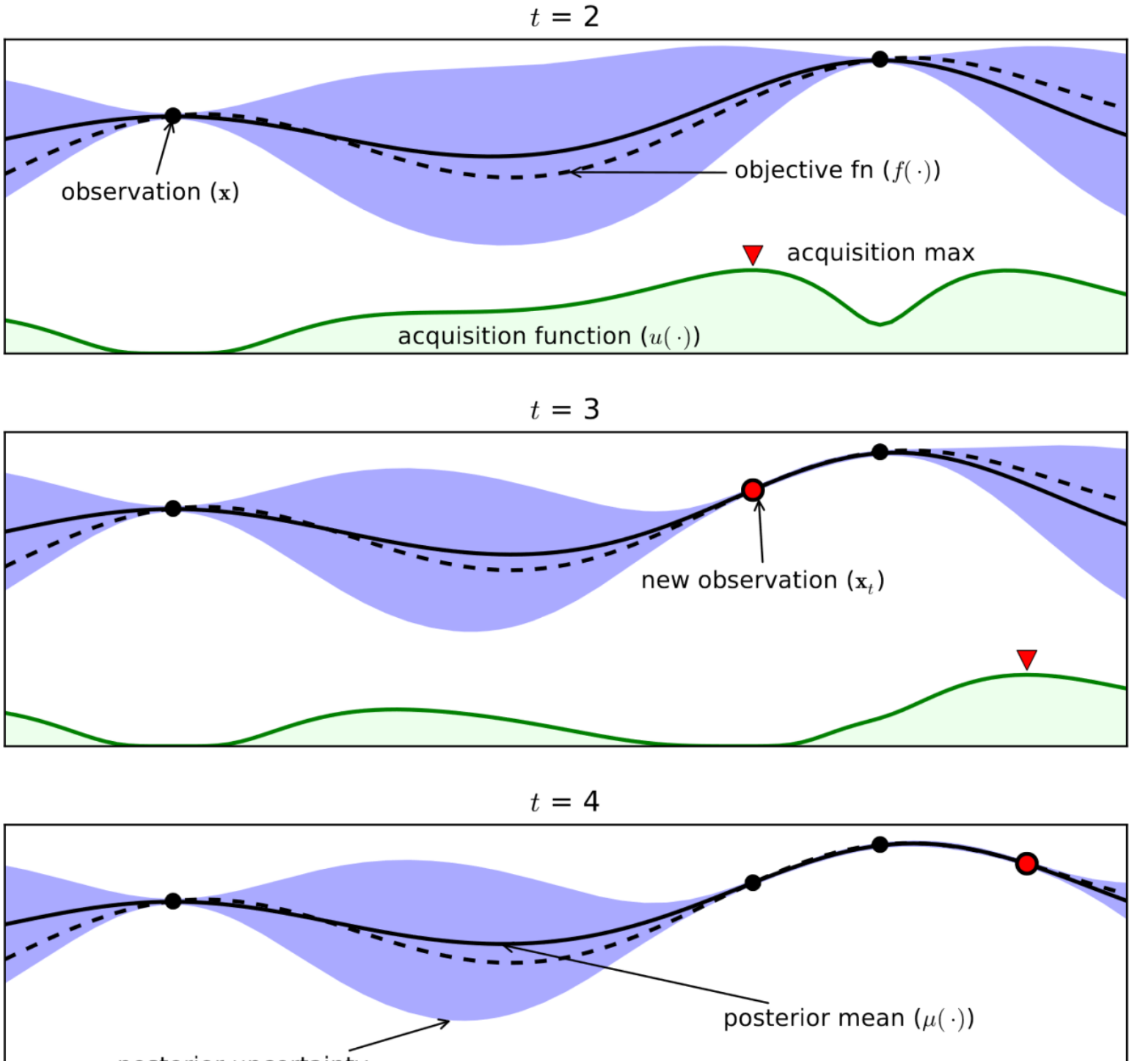
```
Out[18]: SVC(C=6.109237791897481, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.04723626633903414,
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```

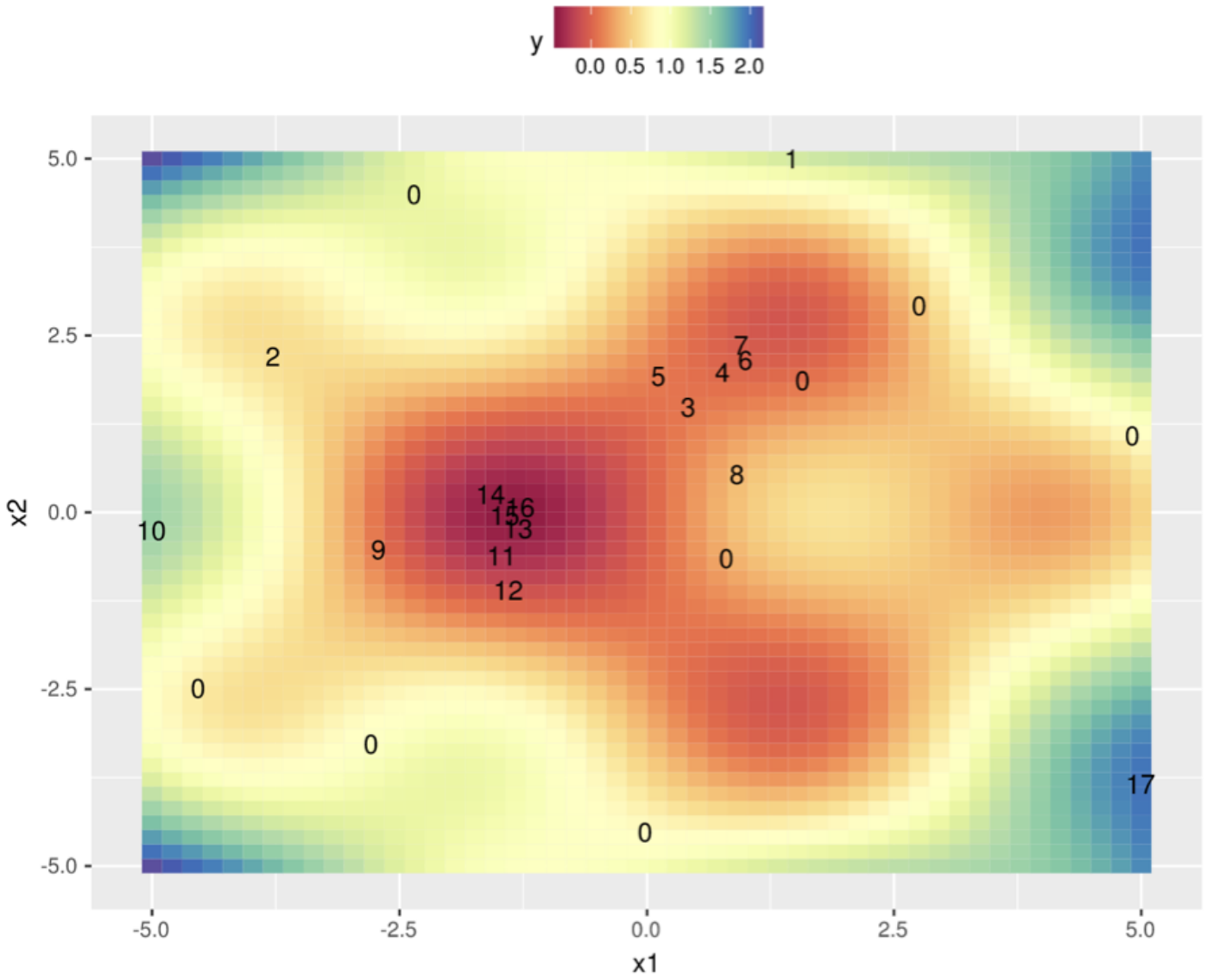
Bayesian optimization

- The incremental updates you can do with Bayesian models allow a more effective way to optimize functions
  - E.g. to optimize the hyperparameter settings of a machine learning algorithm/pipeline
- After a number of random search iterations we know more about the performance of hyperparameter settings on the given dataset
- We can use this data to train a model, and predict which other hyperparameter values might be useful
  - More generally, this is called model-based optimization
  - This model is called a *surrogate model*
- This is often a probabilistic (e.g. Bayesian) model that predicts confidence intervals for all hyperparameter settings
- We use the predictions of this model to choose the next point to evaluate
- With every new evaluation, we update the surrogate model and repeat

Example (see figure):

- Consider only 1 continuous hyperparameter (X-axis)
  - You can also do this for many more hyperparameters
- Y-axis shows cross-validation performance
- Evaluate a number of random hyperparameter settings (black dots)
  - Sometimes an initialization design is used
- Train a model, and predict the expected performance of other (unseen) hyperparameter values
  - Mean value (black line) and distribution (blue band)
- An *acquisition function* (green line) trades off maximal expected performace and maximal uncertainty
  - Exploitation vs exploration
- Optimal value of the asquisition function is the next hyperparameter setting to be evaluated
- Repeat a fixed number of times, or until time budget runs out


t = 2
observation ($\mathbf{x}$)
objective fn ($f(\cdot)$)
acquisition max
acquisition function ($u(\cdot)$)
t = 3
new observation ($\mathbf{x}_t$)
t = 4
posterior mean ($\mu(\cdot)$)


y
0.0 0.5 1.0 1.5 2.0
5.0
2.5
0.0
-2.5
-5.0
x2
-5.0
-2.5
0.0
2.5
5.0
x1
1
0
0
2
7
6
5
4
0
3
0
8
14
16
15
13
10
9
11
0
12
0
0
17
0

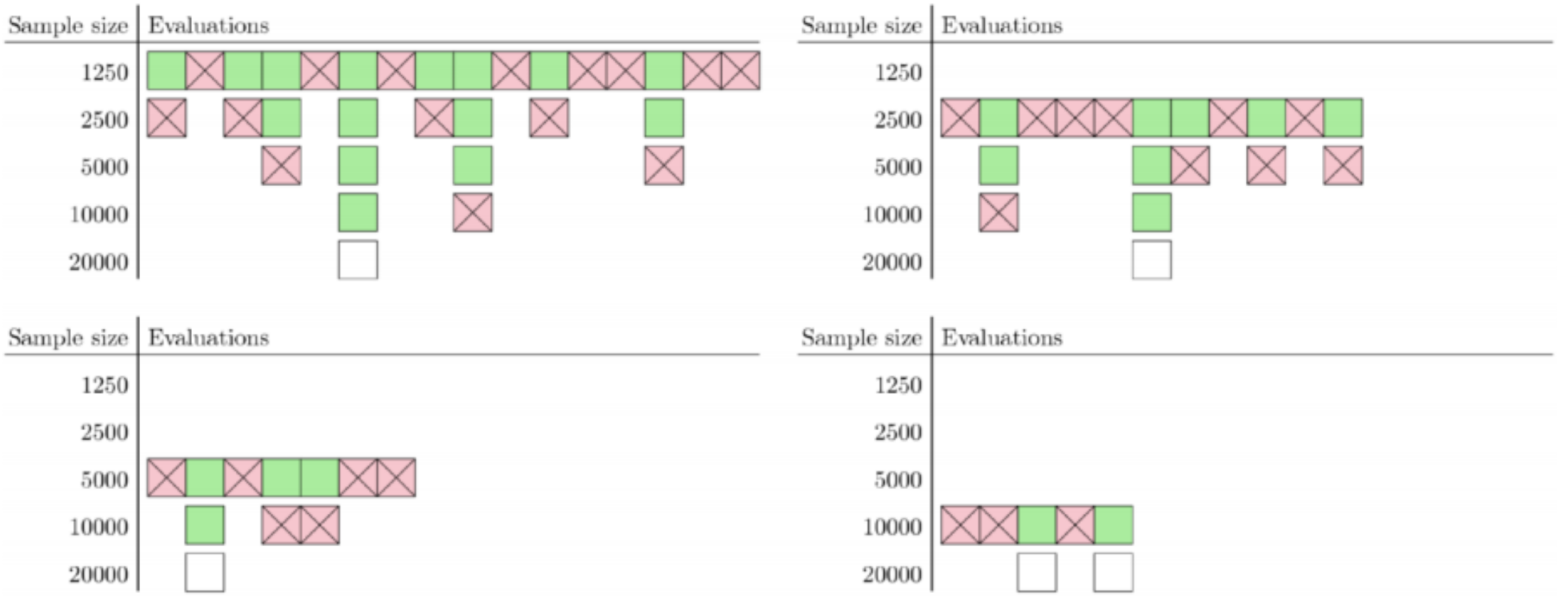
Auto-sklearn

- Different libraries exist for Bayesian optimization
- E.g. auto-sklearn
  - Drop-in sklearn classifier
  - Also optimizes pipelines (e.g. feature selection)
  - Uses OpenML to find good models on similar datasets
  - Lacks Windows support

```
automl = autosklearn.classification.AutoSklearnClassifier(
    time_left_for_this_task=60, # sec., for entire process
    per_run_time_limit=15, # sec., for each model
    ml_memory_limit=1024, # MB, memory limit
)
automl.fit(X_train, y_train)
```

Bandit-based techniques

- Multi-arm bandits: Find best of *n* choices ('arms'), each try ('pull') gives only partial information
- Successive halving for model selection:
  - Test every model initially on small part of the data
  - Train 50% best models (green) on twice the amount of data, repeat
- Avoid premature removal by restarting a random subset on a larger initial training set



Hyperband

- Hyperparameter tuning with bandits (Hyperband)
- Different implementations exist, including scikit-learn drop-ins (https://github.com/thuijskens/scikit-hyperband)
- At the moment still experimental (e.g. no preprocessing support)

Pipelines

- Many learning algorithms are greatly affected by *how* you represent the training data
- Examples: Scaling, numeric/categorical values, missing values, feature selection/construction
- We typically need chain together different algorithms
  - Many *preprocessing* steps
  - Possibly many models
- This is called a *pipeline* (or *workflow*)
- The best way to represent data depends not only on the semantics of the data, but also on the kind of model you are using.

Example: Speed dating data

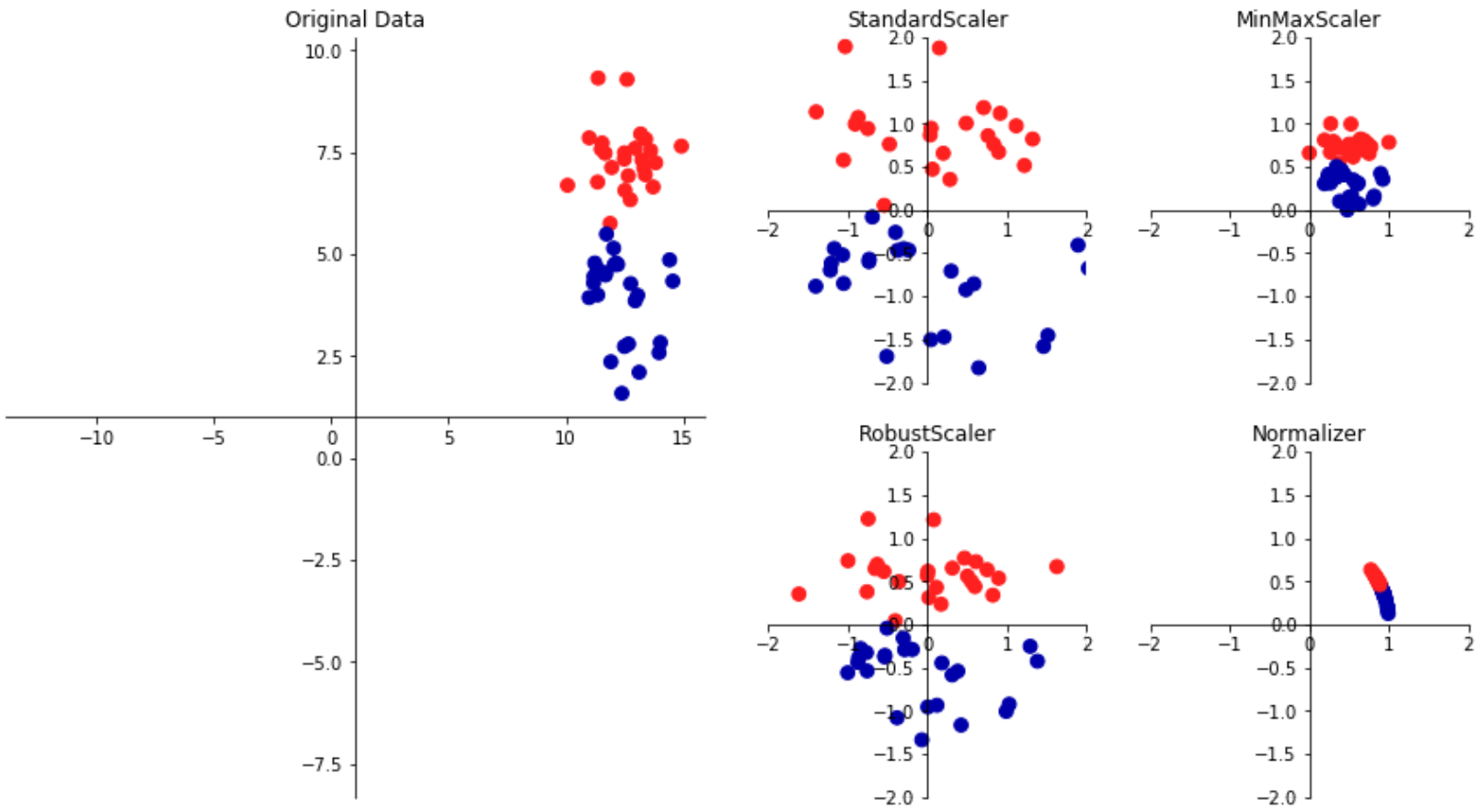
- Data collected from speed dating events
- See https://www.openml.org/d/40536 (https://www.openml.org/d/40536)
- Could also be collected from dating website or app
- Real-world data:
  - Different numeric scales
  - Missing values
  - Likely irrelevant features
  - Different types: Numeric, categorical,...
  - Input errors (e.g. 'lawyer' vs 'Lawyer')

```
dating_data = fetch_openml("SpeedDating")
```

Scaling

When the features have different scales (their values range between very different minimum and maximum values), one feature will overpower the others. Several scaling techniques are available to solve this:

- `StandardScaler` rescales all features to mean=0 and variance=1
  - Does not ensure and min/max value
- `RobustScaler` uses the median and quartiles
  - Median m: half of the values < m, half > m
  - Lower Quartile lq: 1/4 of values < lq
  - Upper Quartile uq: 1/4 of values > uq
  - Ignores *outliers*, brings all features to same scale
- `MinMaxScaler` brings all feature values between 0 and 1
- `Normalizer` scales data such that the feature vector has Euclidean length 1
  - Projects data to the unit circle
  - Used when only the direction/angle of the data matters


Original Data
StandardScaler
MinMaxScaler
RobustScaler
Normalizer

Applying scaling transformations

- Lets apply a scaling transformation *manually*, then use it to train a learning algorithm
- First, split the data in training and test set
- Next, we `fit` the preprocessor on the **training data**
  - This computes the necessary transformation parameters
  - For `MinMaxScaler`, these are the min/max values for every feature
- After fitting, we can `transform` the training and test data

```
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Feature Selection

It can be a good idea to reduce the number of features to only the most useful ones:

- Simpler models that generalize better
- Help algorithms that are sensitive to the number of features (e.g. kNN).

Use it when:

- You expect some inputs to be uninformative, and your model does not select features internally (as tree-based models do)
- You need to speed up prediction without loosing much accuracy
- You want a more interpretable model (with fewer variables)

Univariate feature selection

We want to keep the features for which there is statistically significant relationship between it and the target. These test consider each feature individually (they are univariate), and are completely independent of the model that you might want to apply afterwards.

In scikit-learn we have two options:

- `SelectKBest` will only keep the *k* features with the lowest p values.
- `SelectPercentile` selects a fixed percentage of features.

Retrieve the selected features with `get_support()`

We can use different tests to measure how informative a feature is:

`f_regression`: For numeric targets. Measures the performance of a linear regression model trained on only one feature.

`f_classif`: For categorical targets. Measures the *F-statistic* from one-way Analysis of Variance (ANOVA), or the proportion of total variance explained by one feature.

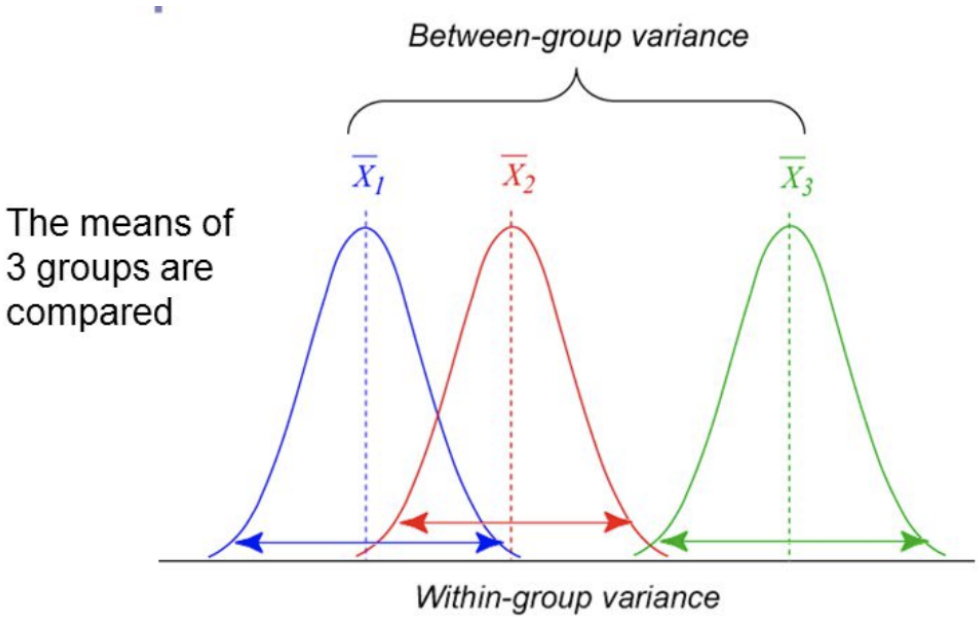
`chi2`: For categorical features and targets. Performs the chi-square statistic. Similar results as F-statistic, but less sensitive to nonlinear relationships.

For both the F-statistic and χ^2, we actually obtain the p-value under the F- and χ^2 distribution, respectively.

F-statistic = variation between sample means / mean variation within the samples (higher is better)

X_i: all samples with class i.

Better is samples means are far apart and variation within samples is small.



Model-based Feature Selection

Model-based feature selection uses a supervised machine learning model to judge the importance of each feature, and keeps only the most important ones. They consider all features together, and are thus able to capture interactions: a feature may be more (or less) informative in combination with others.

The supervised model that is used for feature selection doesn't need to be the same model that is used for the final supervised modeling, it only needs to be able to measure the (perceived) importance for each feature:

- Decision tree–based models return a `feature_importances_` attribute
- Linear models return coefficients, whose absolute values also reflect feature importance

In scikit-learn, we can do this using `SelectFromModel`. It requires a model and a threshold. Threshold='median' means that the median observed feature importance will be the threshold, which will remove 50% of the features.

```
select = SelectFromModel(
    RandomForestClassifier(n_estimators=100, random_state=42),
    threshold="median")
```

Iterative feature selection

Instead of building a model to remove many features at once, we can also just ask it to remove the worst feature, then retrain, remove another feature, etc. This is known as *recursive feature elimination* (RFE).

```
select = RFE(RandomForestClassifier(n_estimators=100, random_state=42),
             n_features_to_select=40)
```

Vice versa, we could also ask it to iteratively add one feature at a time. This is called *forward selection*.

In both cases, we need to define beforehand how many features to select. When this is unknown, one often considers this as an additional hyperparameter of the whole process (pipeline) that needs to be optimized.

Missing value imputation

- Many sci-kit learn algorithms cannot handle missing value
- `Imputer` replaces specific values
  - `missing_values` (default 'NaN') placeholder for the missing value
  - `strategy`:
    - `mean`, replace using the mean along the axis
    - `median`, replace using the median along the axis
    - `most_frequent`, replace using the most frequent value
- Many more advanced techniques exist, but not yet in scikit-learn
  - e.g. low rank approximations (uses matrix factorization)

```
imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
imp.fit_transform(X1_train)
```

Feature encoding

- scikit-learn classifiers only handle numeric data. If your features are categorical, you need to encode them first
- `LabelEncoder` simply replaces each value with an integer value
- `OneHotEncoder` converts a feature of *n* values to *n* binary features
  - Provide `categories` as array or set to 'auto'

```
X_enc = OneHotEncoder(categories='auto').fit_transform(X)
```

- `ColumnTransformer` can apply different transformers to different features
- Transformers can be pipelines doing multiple things

```
numeric_features = ['age', 'pref_o_attractive']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['gender', 'd_d_age', 'field']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing'
)),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])
```

Building Pipelines

- In scikit-learn, a `pipeline` combines multiple processing *steps* in a single estimator
- All but the last step should be transformer (have a `transform` method)
  - The last step can be a transformer too (e.g. Scaler+PCA)
- It has a `fit`, `predict`, and `score` method, just like any other learning algorithm
- Pipelines are built as a list of steps, which are (name, algorithm) tuples
  - The name can be anything you want, but can't contain `'__'`
  - We use `'__'` to refer to the hyperparameters, e.g. `svm__C`
- Let's build, train, and score a `MinMaxScaler` + `LinearSVC` pipeline:

```
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", LinearSVC())])
pipe.fit(X_train, y_train).score(X_test, y_test)
```

```
Test score: 0.97
```

- Now with cross-validation:

```
scores = cross_val_score(pipe, cancer.data, cancer.target)
```

```
Cross-validation scores: [0.984 0.953 0.979]
Average cross-validation score: 0.97
```

- We can retrieve the trained SVM by querying the right step indices

```
pipe.steps[1][1]
```

```
SVM component: LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercep
t=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
```

- Or we can use the `named_steps` dictionary

```
pipe.named_steps['svm']
```

```
SVM component: LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercep
t=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
```

- When you don't need specific names for specific steps, you can use `make_pipeline`
  - Assigns names to steps automatically

```
pipe_short = make_pipeline(MinMaxScaler(), LinearSVC(C=
100))
print("Pipeline steps:\n{}".format(pipe_short.steps))
```

```
Pipeline steps:
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))), ('linear
svc', LinearSVC(C=100, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0))]
```

Using Pipelines in Grid-searches

- We can use the pipeline as a single estimator in `cross_val_score` or `GridSearchCV`
- To define a grid, refer to the hyperparameters of the steps
  - Step `svm`, parameter `C` becomes `svm__C`

```
param_grid = {'svm__C': [0.001, 0.01, 0.1, 1, 10, 100],
              'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
pipe = pipeline.Pipeline([("scaler", MinMaxScaler()), ("svm", SVC(C=100
))])
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5)
grid.fit(X_train, y_train)
```

```
Best cross-validation accuracy: 0.97
Test set score: 0.97
Best parameters: {'svm__C': 10, 'svm__gamma': 1}
```

Learning algorithms

Recap of main classification/regression algorithms and how to use them

- How to control complexity (avoid overfitting)
- Hyperparameters (user-controlled parameters)
- Strengths and weaknesses

Regression example: Online news popularity

- Data collected from a news site (Mashable)
- Predict the number of `shares` based on article statistics
- See https://www.openml.org/d/4545 (https://www.openml.org/d/4545)
- Real-world data:
  - Different numeric scales
  - Likely irrelevant features
  - Numeric or binary features
  - No missing values

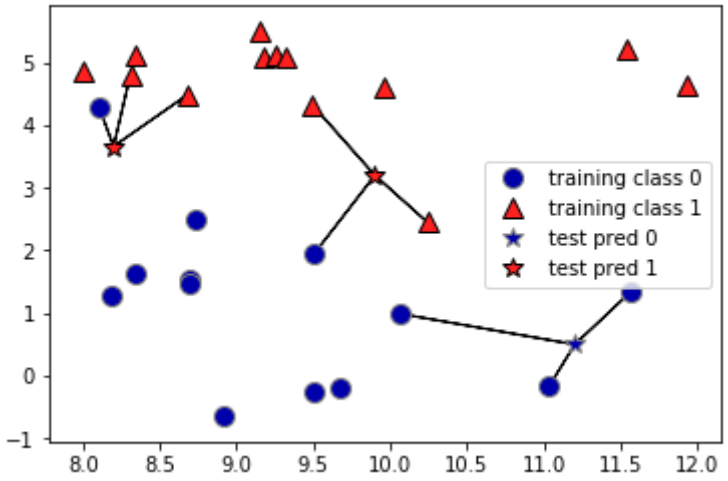
```
news_data = fetch_openml("OnlineNewsPopularity")
```

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

Find nearest neighbors, do a vote, return the majority (or a confidence value for each class)



Let's build a kNN model (on dataset 'Forge')

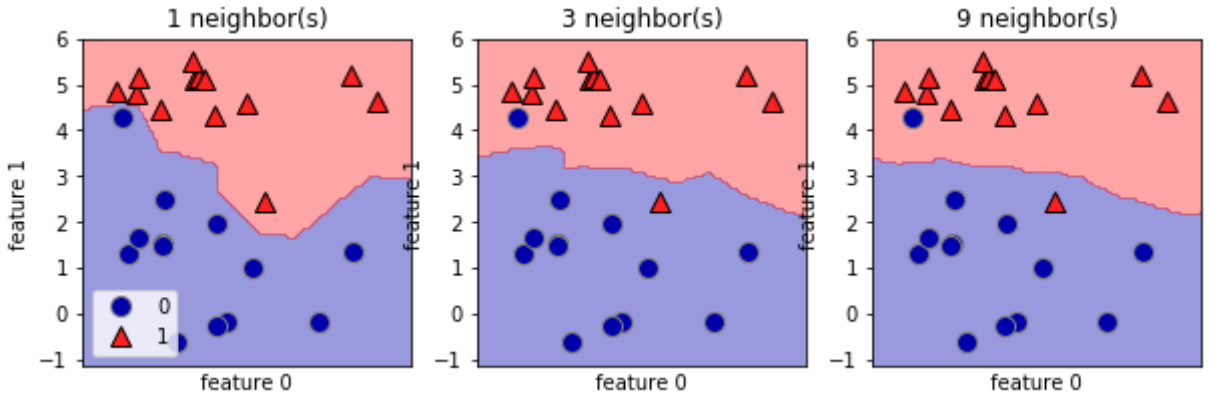
```
X, y = mglearn.datasets.make_forge()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0
)
clf = KNeighborsClassifier(n_neighbors=19)
clf.fit(X_train, y_train)
clf.score(X_test, y_test)
```

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
```

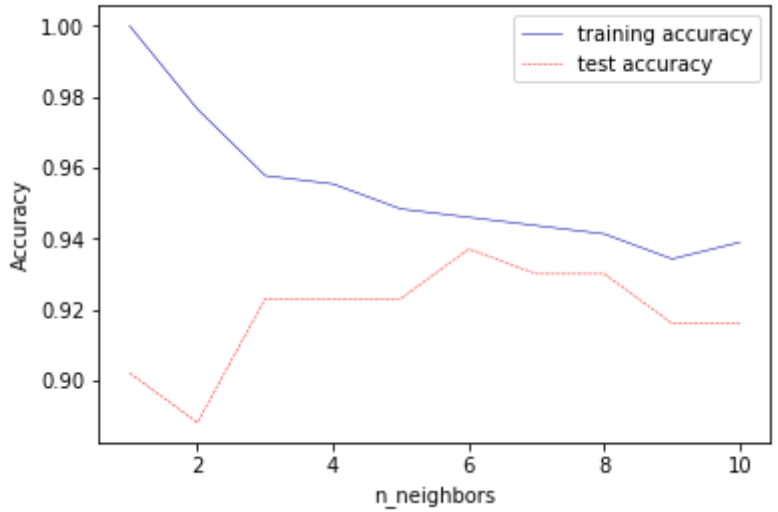
Analysis

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



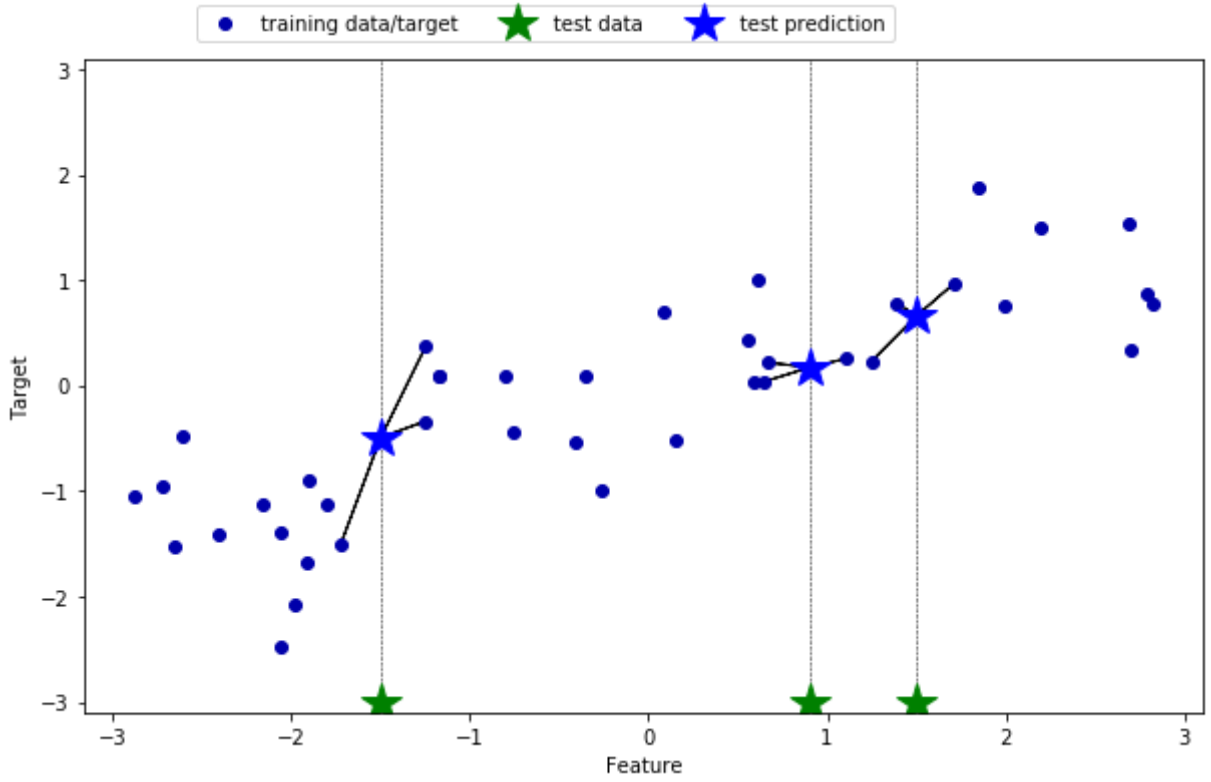
We can more directly measure the effect on the training and test error on a larger dataset (breast_cancer)

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



k-Neighbors Regression

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).

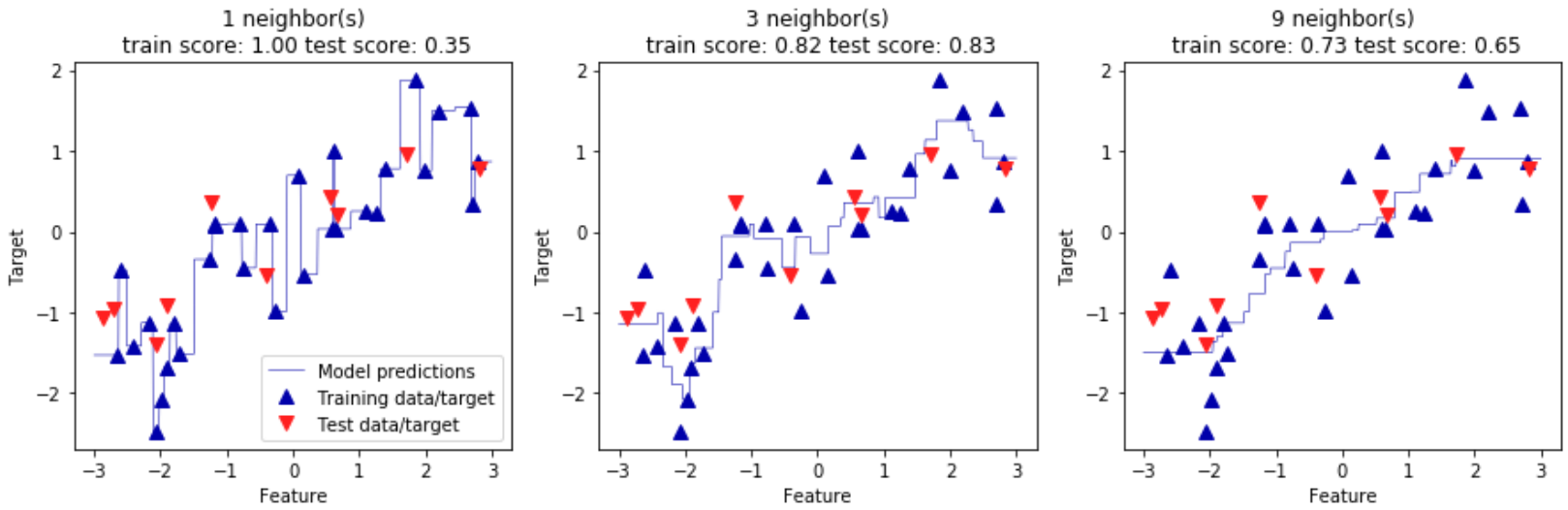
```
reg.predict(X_test)
reg.score(X_test, y_test)
```

```
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.

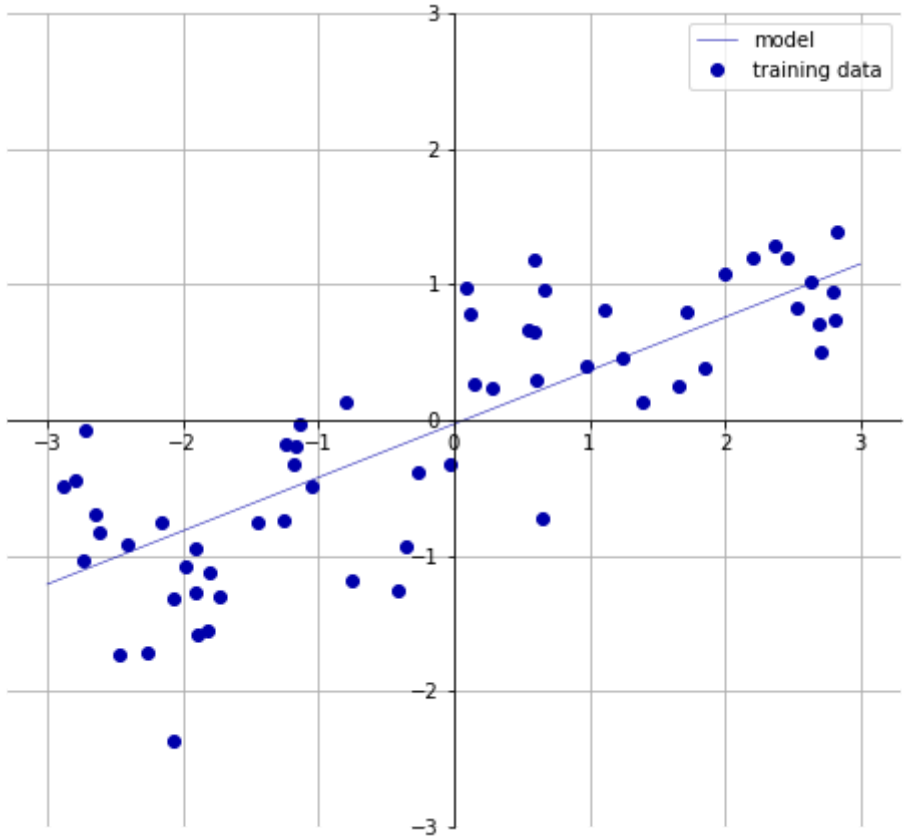
Linear models for regression

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

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

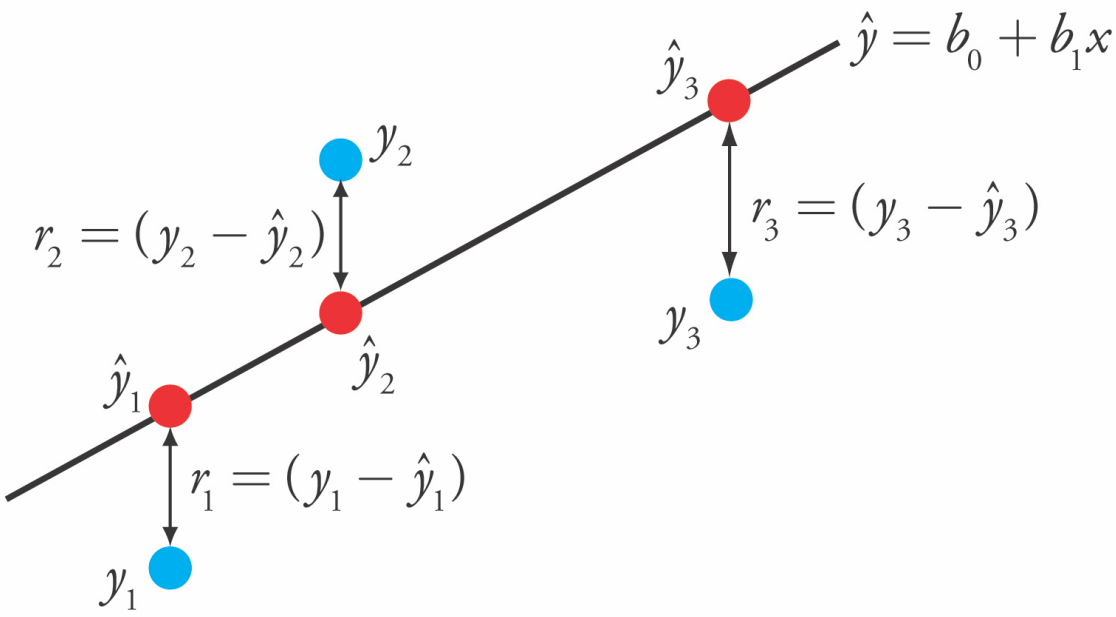
There are many different algorithms, differing in how w and b are learned from the training data.

```
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.
- It has no hyperparameters, thus model complexity cannot be controlled.



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`

- 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

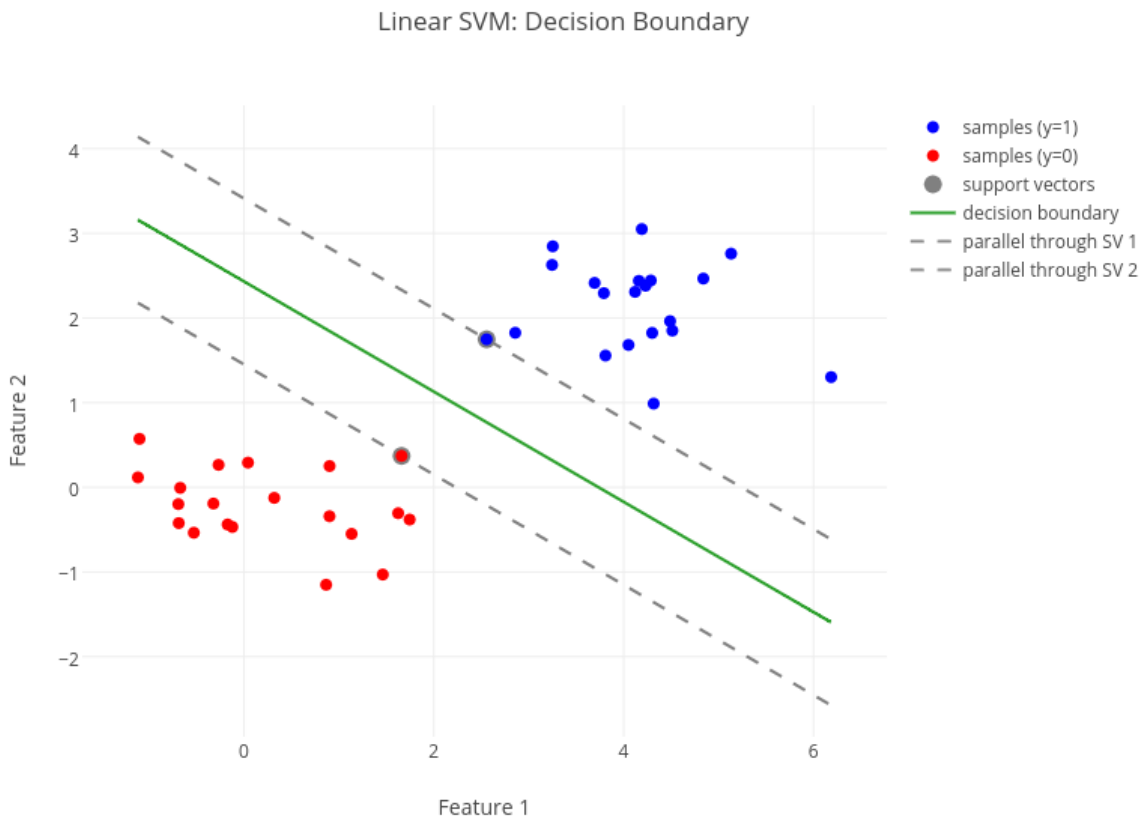
The logistic model uses the *logistic* (or *sigmoid*) function to estimate the probability that a given sample belongs to class 1:

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

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

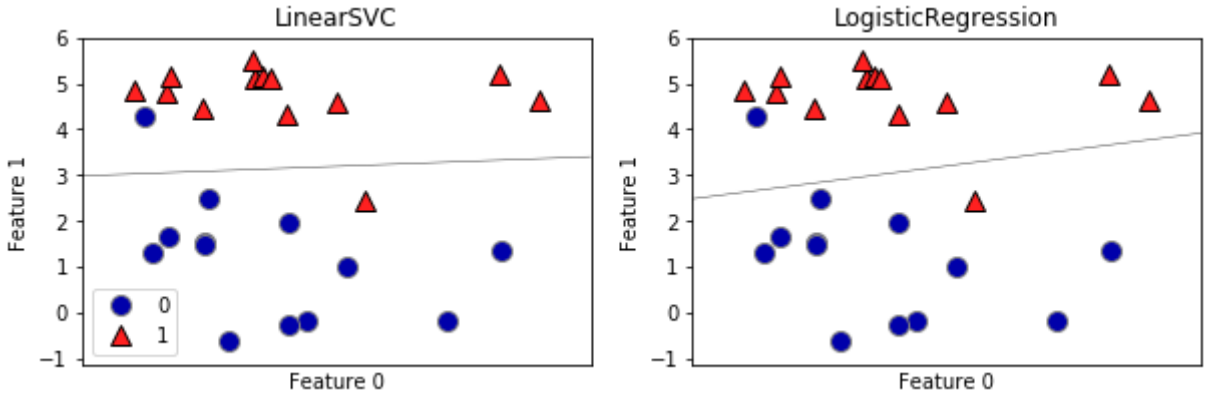
Linear Support Vector Machine

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.

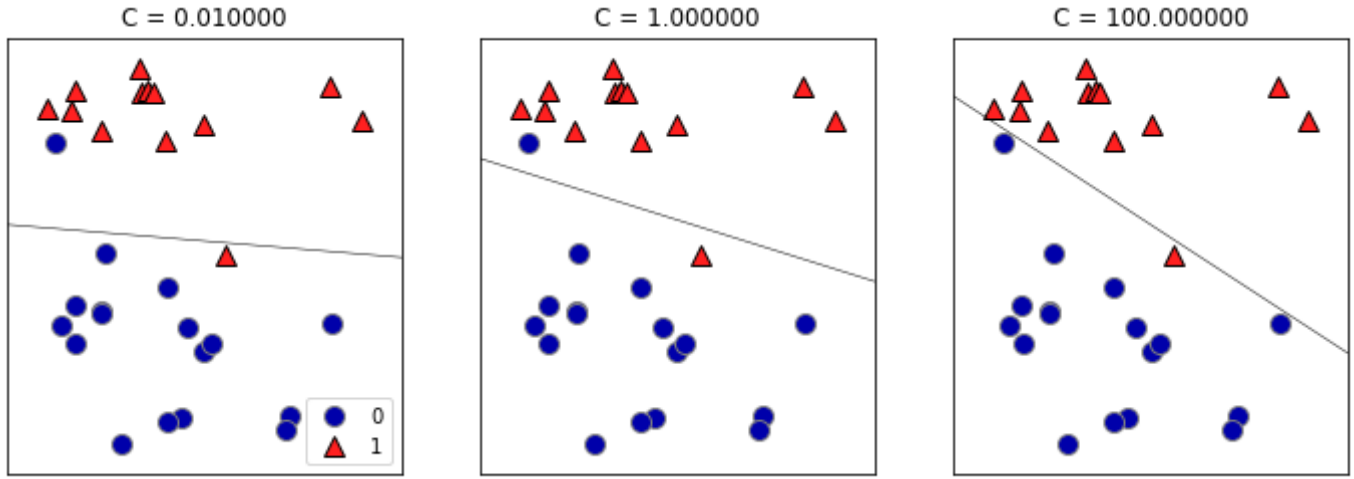
- Logistic regression can be run with `linear_model.LogisticRegression`
- Linear SVMs can be run with `svm.LinearSVC`



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

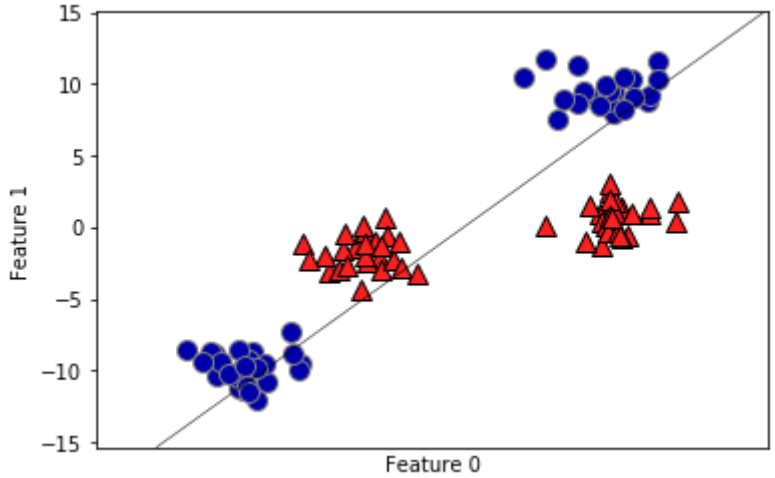
```
logreg = LogisticRegression(C=1,penalty=l2).fit(X_train,
y_train)
```



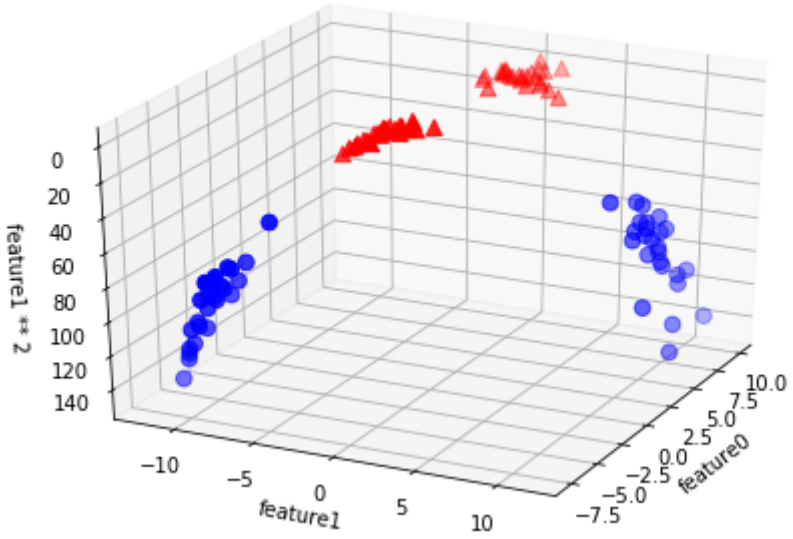
Kernelized Support Vector Machines

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

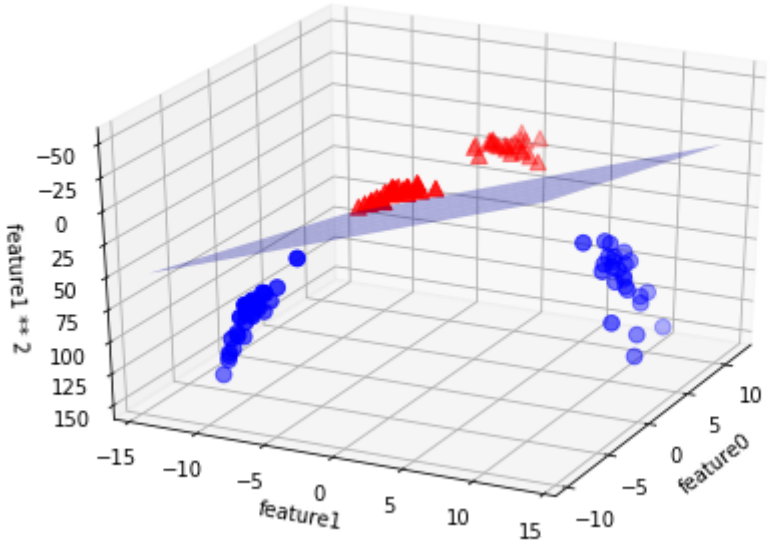
Our linear model doesn't fit the data well



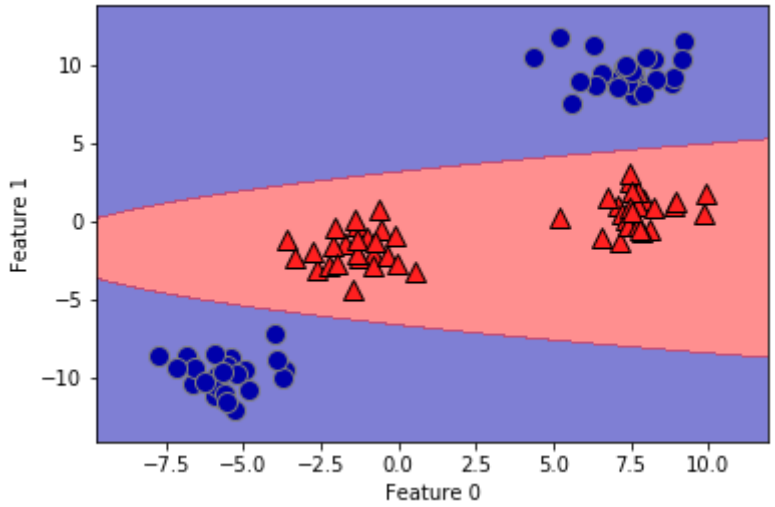
We can add a new feature by taking the squares of feature1 values



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:

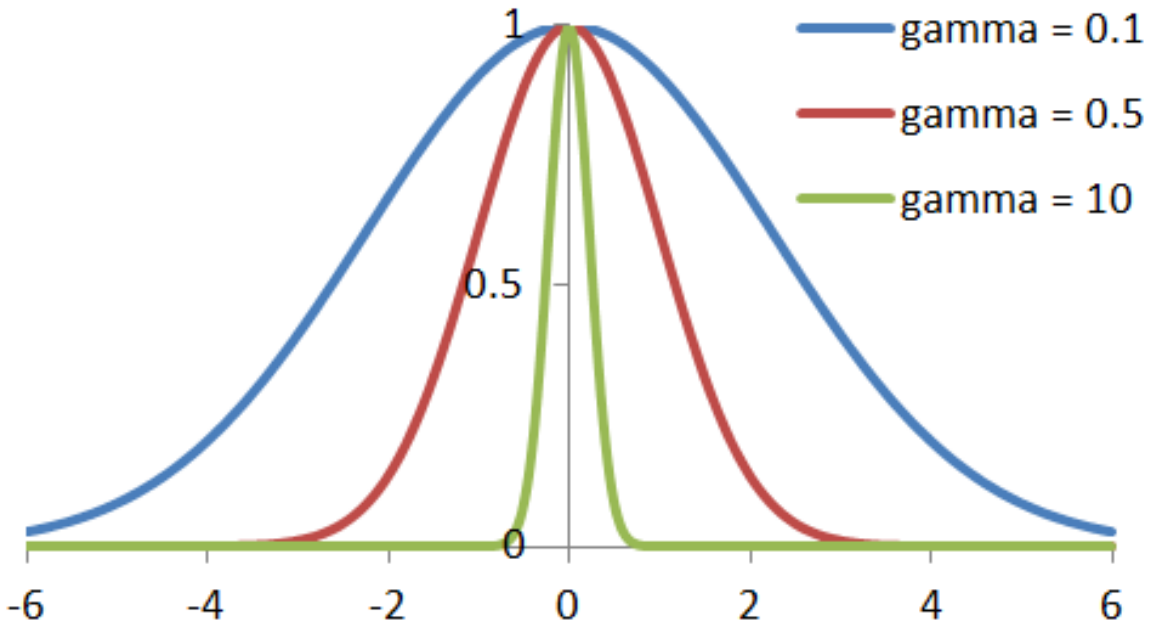
- Symmetry: $k(x_1, x_2) = k(x_2, x_1) \quad \forall x_1, x_2 \in X$
- Positive definite: for each finite subset of data points $x_1, \ldots, x_n$, the kernel Gram matrix is positive semi-definite

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

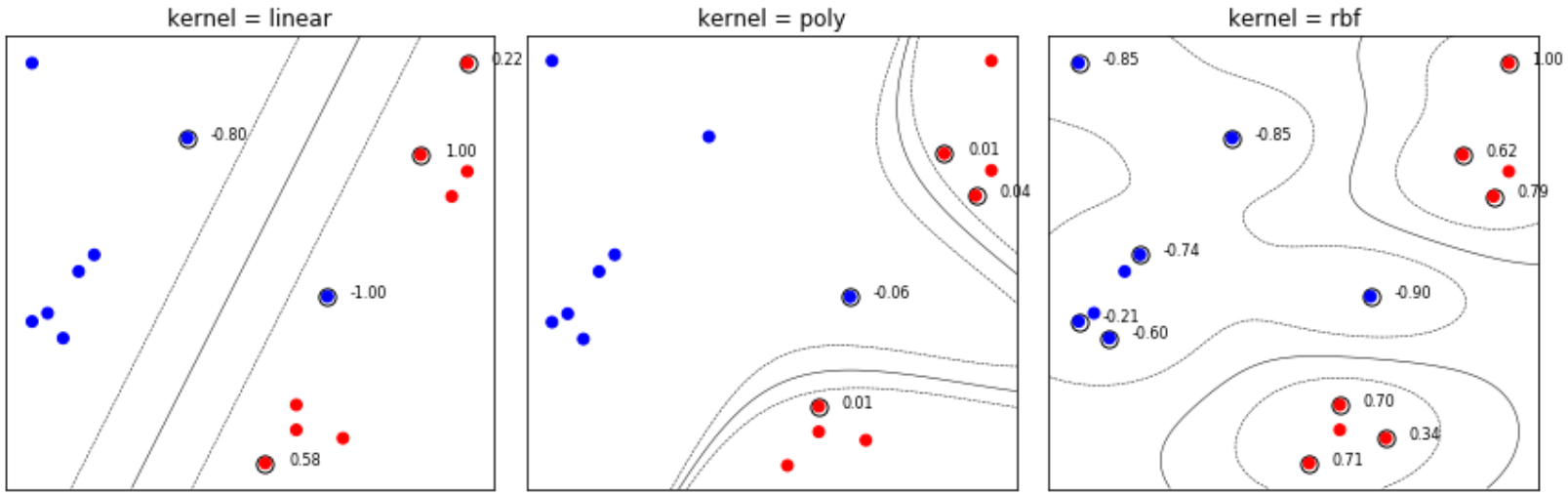
Kernels: examples

- 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 $\gamma \geq 0$

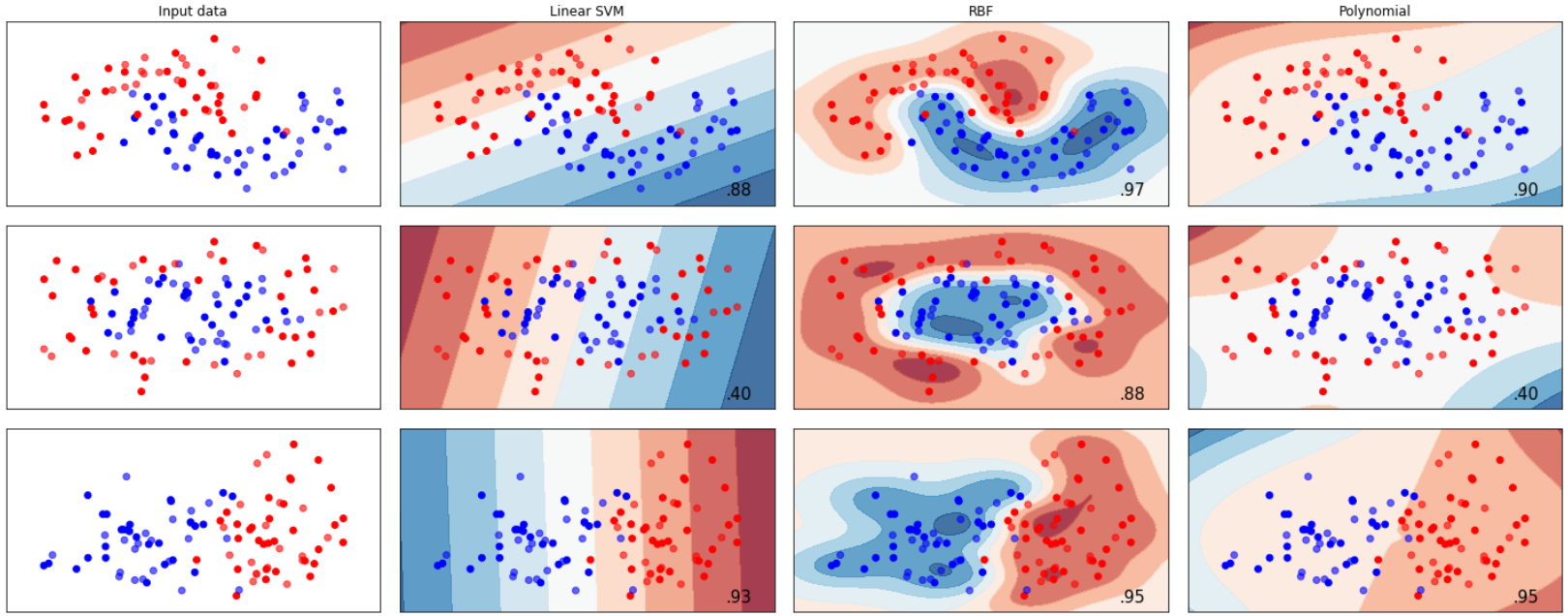


Different kernels lead to different decision boundaries, because the distance to the support vectors is measured differently.



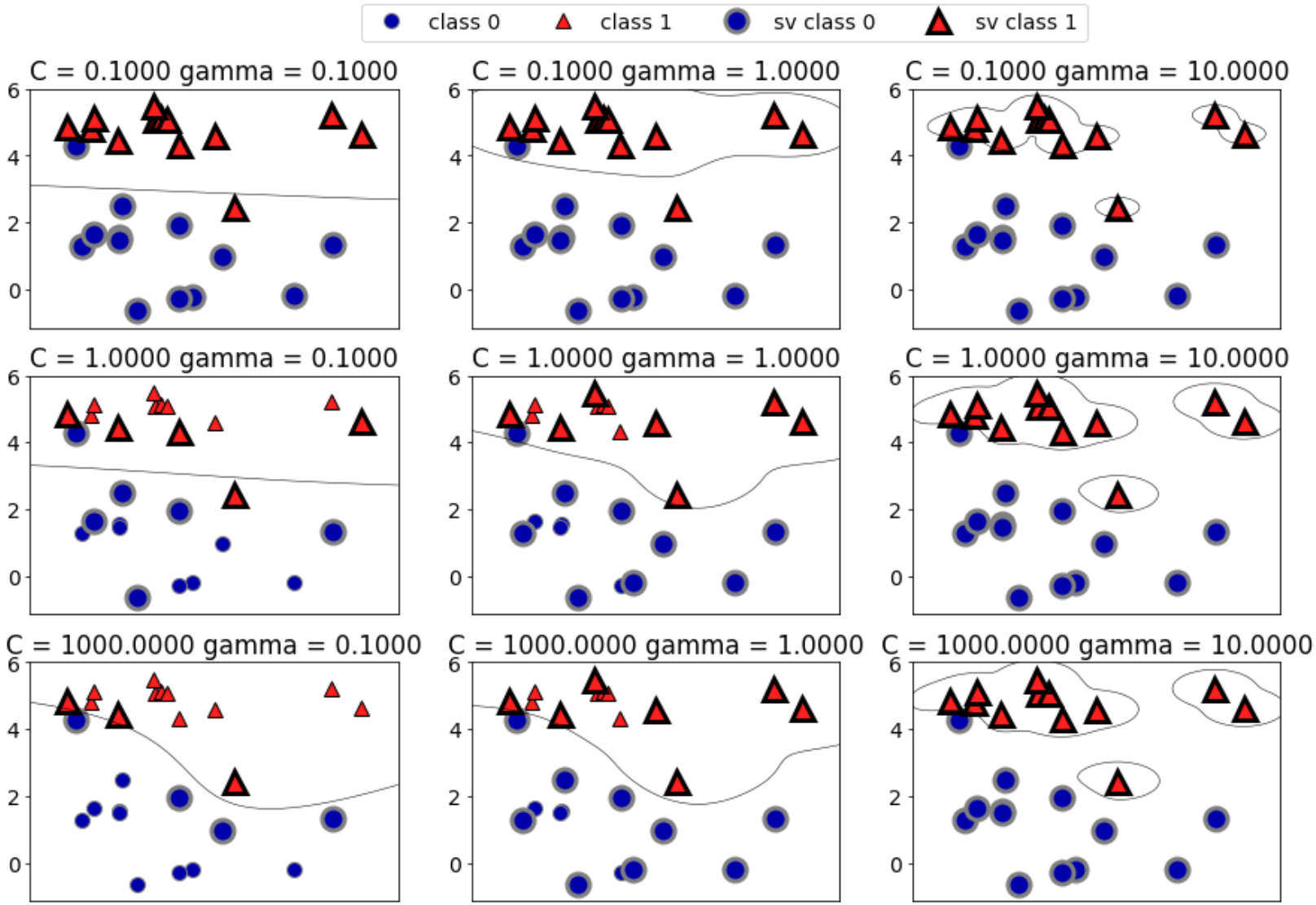
The first important hyperparameter to tune is the choice of kernel

- 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

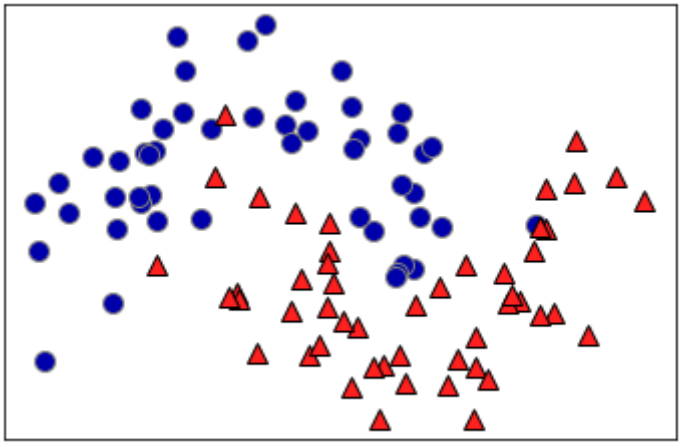
- SVMs allow complex decision boundaries, even with few features.
- Work well on both low- and high-dimensional data
- Don't scale very well to large datasets (>100000)
- Require careful preprocessing of the data and tuning of the parameters.
- SVM models are hard to inspect

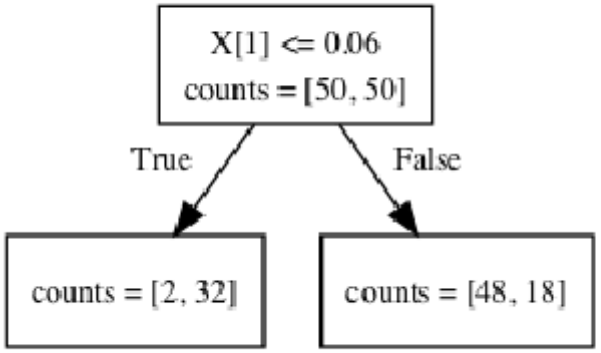
Important parameters:

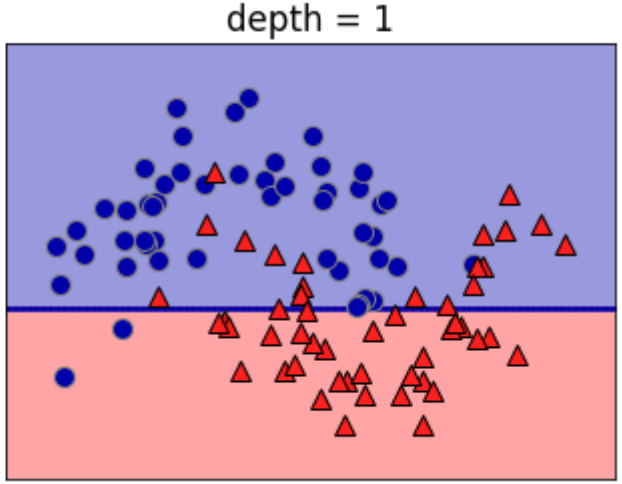
- regularization parameter *C*
- choice of the kernel and kernel-specific parameters
  - Typically string correlation with *C*

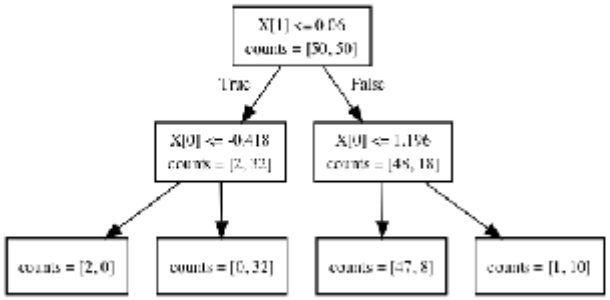
Decision Trees

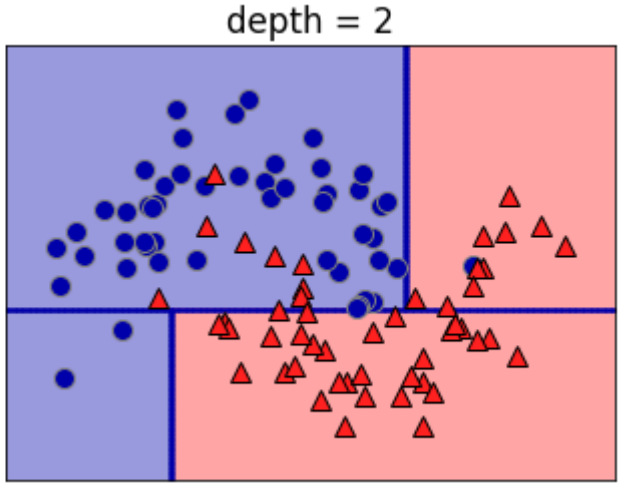
- Split the data in two (or more) parts
- Search over all possible splits and choose the one that is most *informative*
  - Many heuristics
  - E.g. *information gain*: how much does the entropy of the class labels decrease after the split (purer 'leafs')
- Repeat recursive partitioning
- In scikit-learn: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

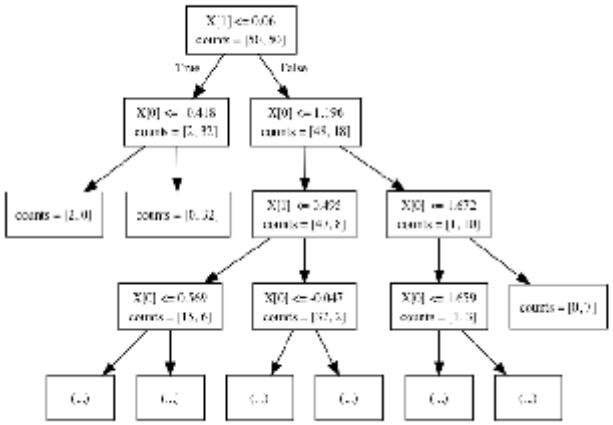


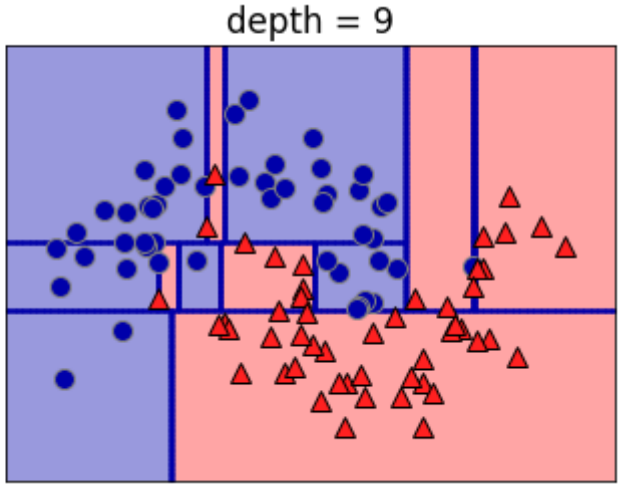

X[1] <= 0.06
counts = [50, 50]
True
False
counts = [2, 32]
counts = [48, 18]


depth = 1


X[1] <= 0.06
counts = [50, 50]
True
False
X[0] <= 0.418
counts = [2, 32]
X[0] <= 1.156
counts = [48, 18]
counts = [2, 0]
counts = [0, 32]
counts = [47, 8]
counts = [1, 10]


depth = 2




depth = 9

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 Tree Regression
max_depth=2
max_depth=5
data
target
data
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
1
2
3
4
5

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:

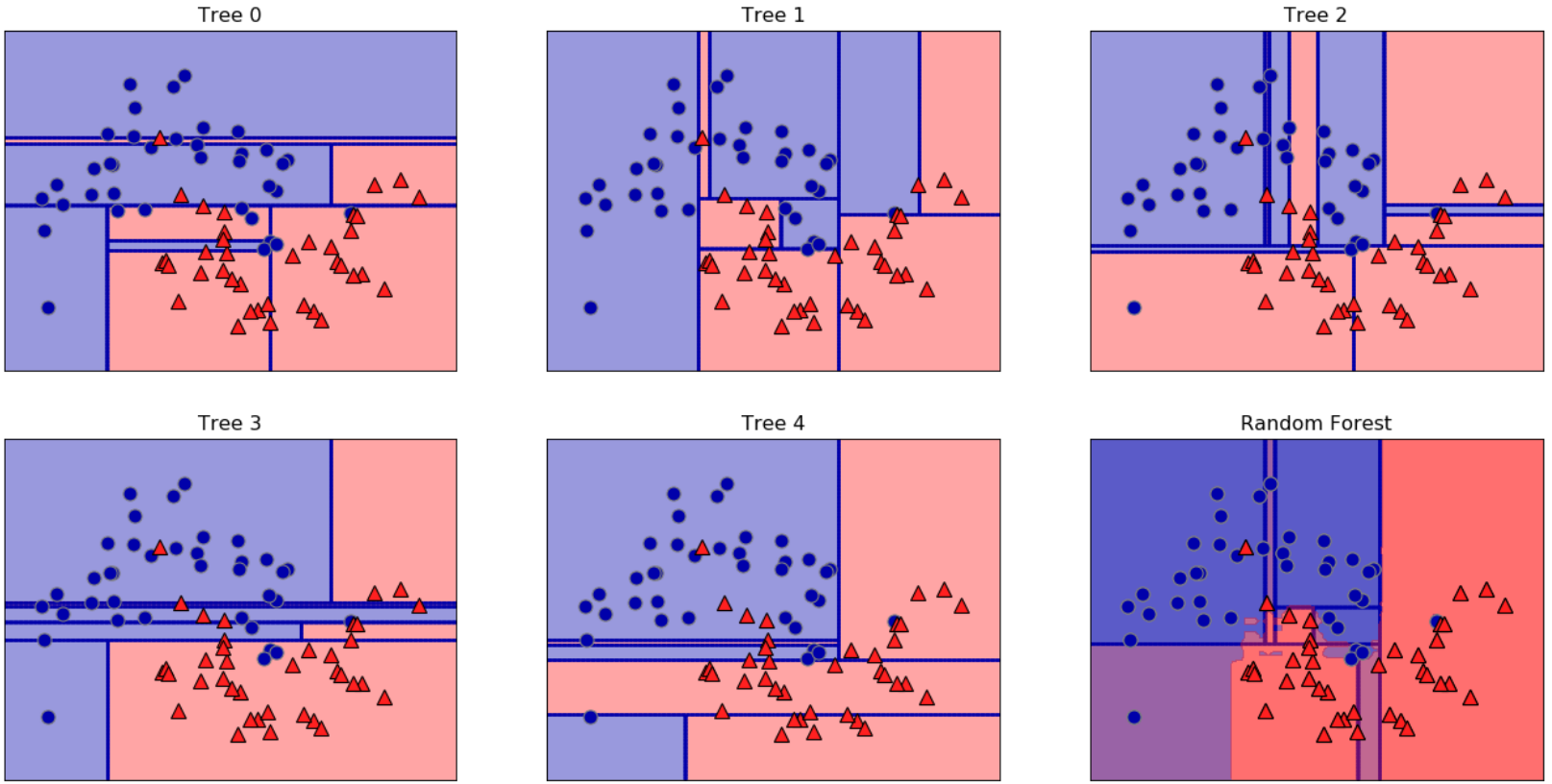
- **RandomForests**: Build randomized trees on random samples of the data
- **Gradient boosting machines**: Build trees iteratively, giving higher weights to the points misclassified by previous trees

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

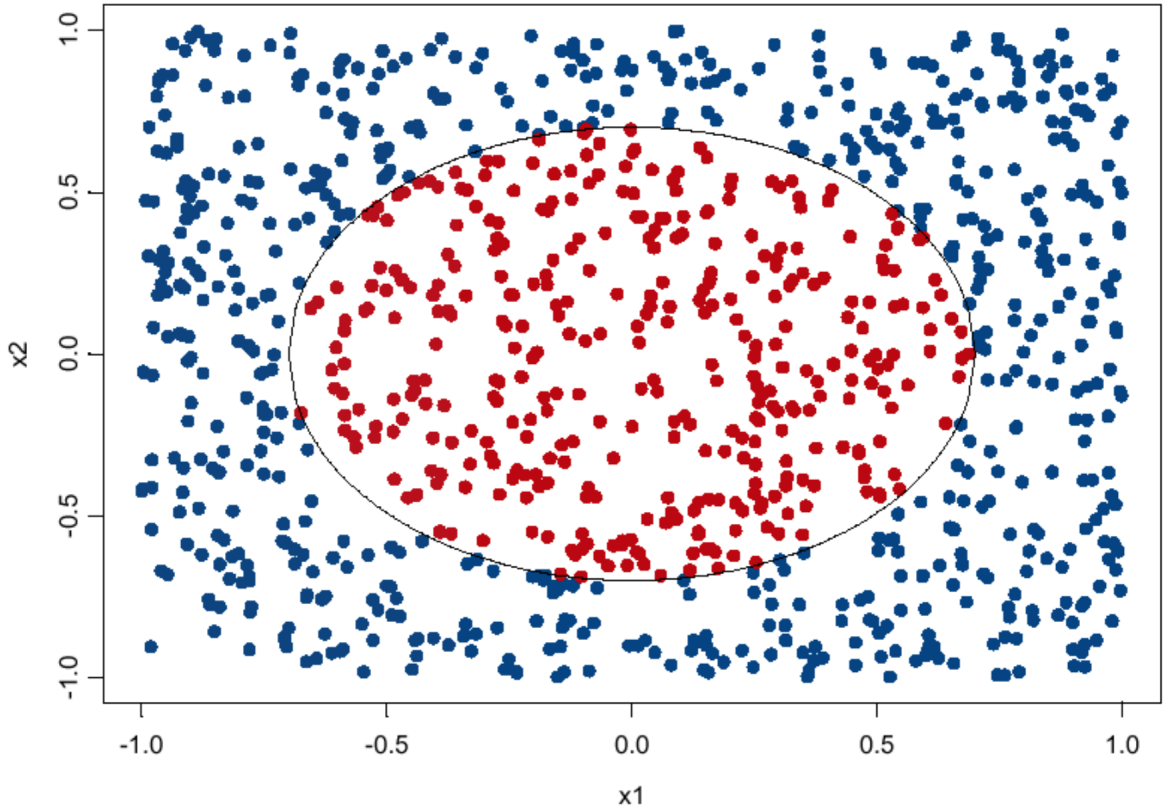

Tree 0
Tree 1
Tree 2
Tree 3
Tree 4
Random Forest

Gradient Boosting

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

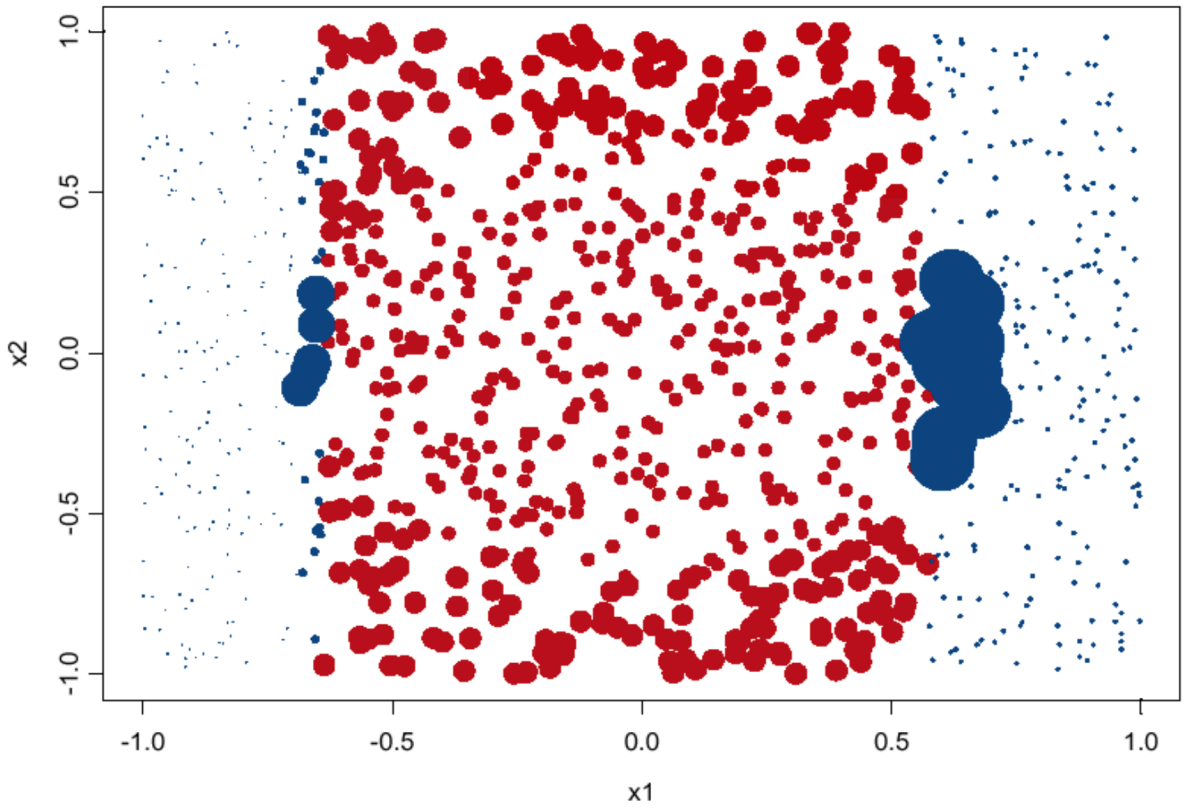
- Use strong pre-pruning to build very shallow trees
  - Default `max_depth`=3
- Iteratively build new trees by increasing weights of points that were badly predicted
- Example of *additive modelling*: each tree depends on the outcome of previous trees
- Optimization: find optimal weights for all data 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:

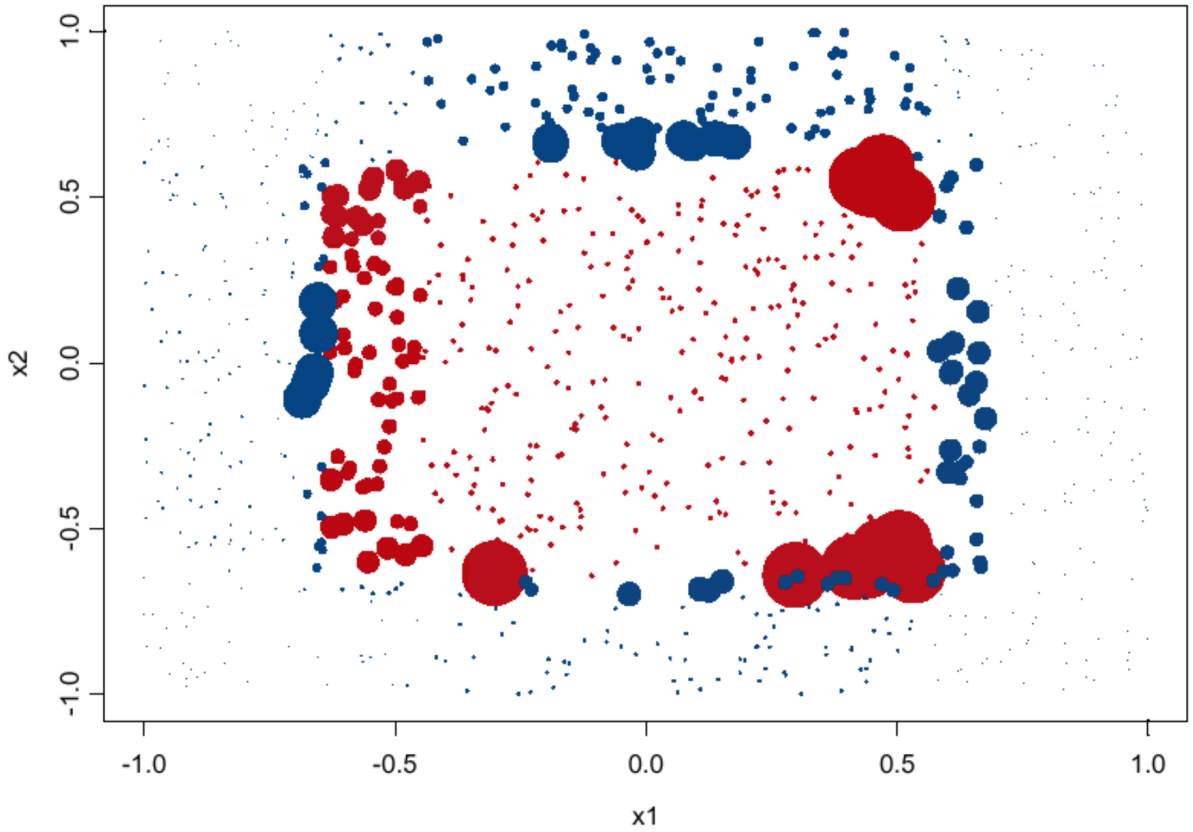


After 1 iteration

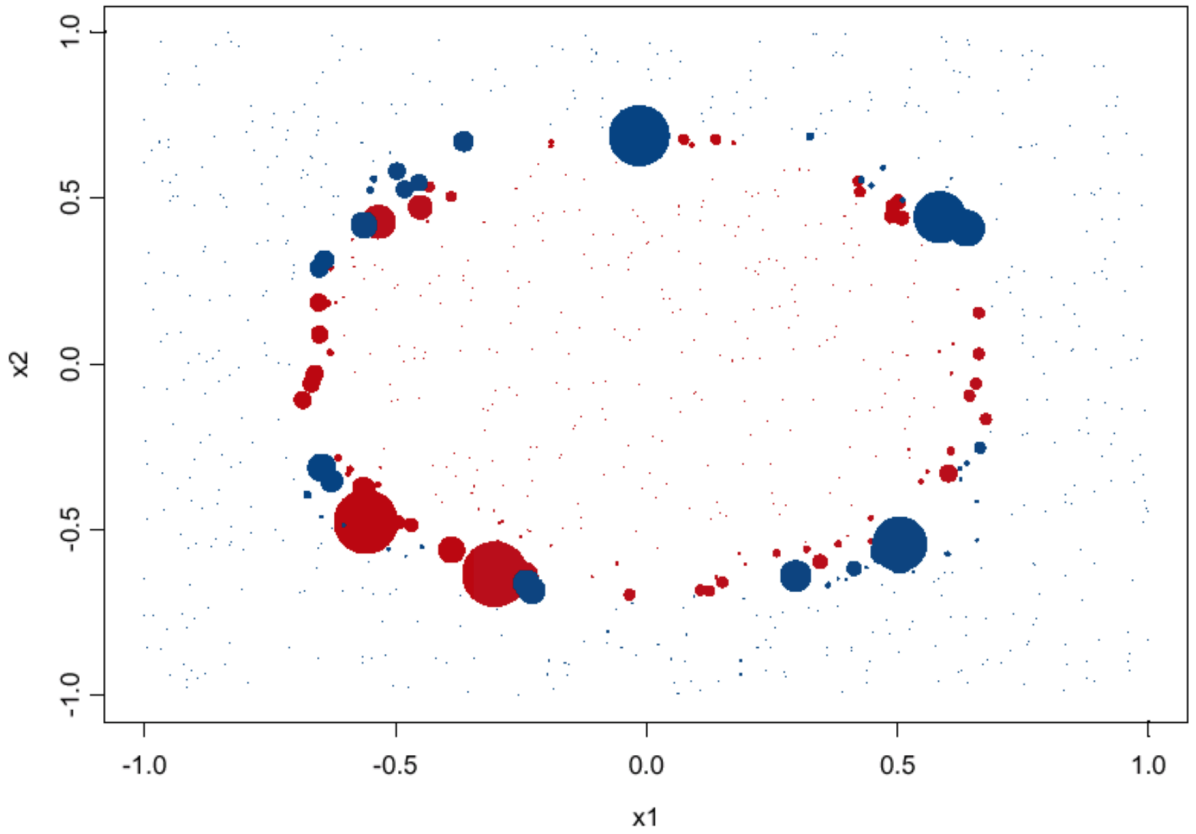
- The simple decision tree divides space
- Misclassified points get higher weight (larger dots)

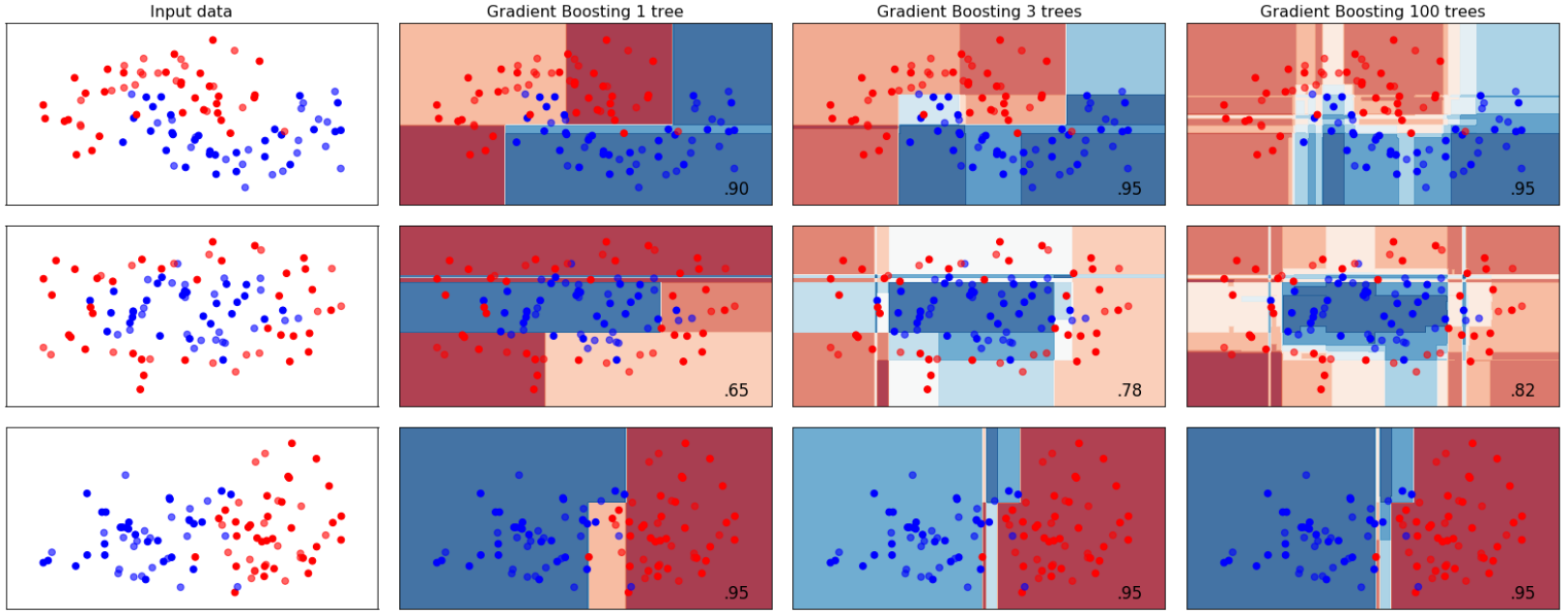


After 3 iterations



After 20 iterations




Input data
Gradient Boosting 1 tree
Gradient Boosting 3 trees
Gradient Boosting 100 trees
.90
.95
.95
.65
.78
.82
.95
.95
.95

Many more algorithms:

- Probabilistic techniques
  - Naive Bayes
  - Bayesian Networks
  - Gaussian Processes
- Graphical models
  - Hidden Markov models
  - ...
- Neural Networks
  - Next