W4995 Applied Machine Learning

Trees, Forests and Boosting

02/20/17

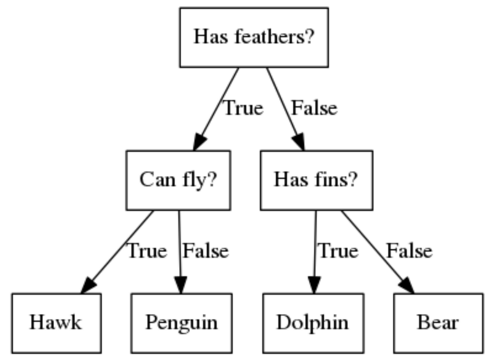
Andreas Müller

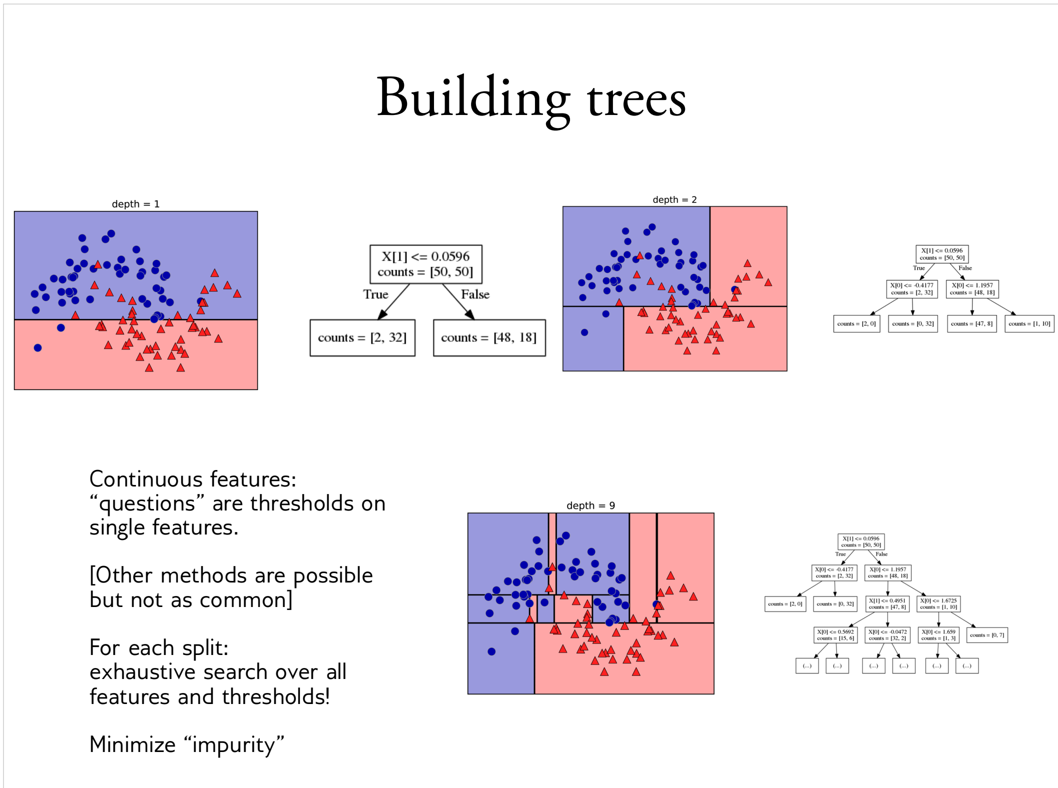
Why trees?

- Very powerful modeling method – non-linear!
- Doesn't care about scaling!
- "Interpretable"
- Basis of very powerful models!

Decision Trees for Classification

Idea: series of binary questions





The second family of models I want to talk about is decision-tree based models.

A decision tree is basically the same as a sequence of if-else branches, that ultimately lead to a decision. The point is that the question that are asked are learned, though.

A decision tree is build recursively by asking a series of questions of the form “is feature “i” greater than threshold t”. In each iteration, the question is chosen that yields the most information about the target variable. Then, the data is split according to this question, and we start again. This yields a hierarchical partitioning of the data, where each section of the partitioning becomes more and more “pure”, that is their content becomes more and more the same.

After you build the tree, you can make a prediction by checking which part of the partition a new point lies in and assigning the mean of the datapoints in this part.

Criteria (for classification)

- Gini index:

$$H_{\text{gini}}(X_m) = \sum_{k \in \mathcal{Y}} p_{mk}(1 - p_{mk})$$

- Cross-entropy:

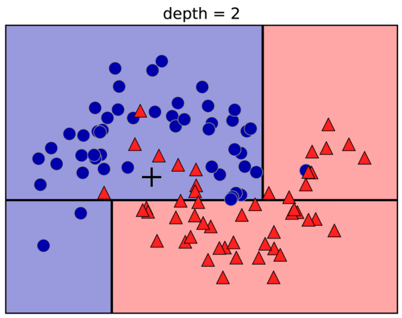
$$H_{\text{CE}}(X_m) = -\sum_{k \in \mathcal{Y}} p_{mk} \log(p_{mk})$$

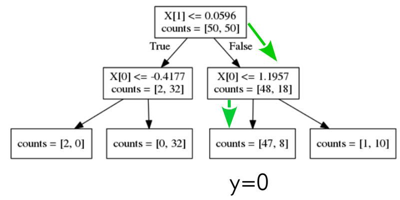
X_m observations in node m

$\mathcal{Y}$ classes

$p_{m\cdot}$ distribution over classes in node m

Prediction





Traverse tree based on feature tests
Predict most common class in leaf

Regression trees

- Impurity Criteria:
  Mean Squared Error
  Mean Absolute Error

- Prediction:
  Predict mean.

- Without regularization / pruning:
  Each leaf often contains a single point to be “pure”

Visualizing trees with sklearn

```python
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
```

```python
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, stratify=cancer.target, random_state=0)
```

```python
# tree visualization
```

```python
from sklearn.tree import DecisionTreeClassifier, export_graphviz
tree = DecisionTreeClassifier(max_depth=2)
tree.fit(X_train, y_train)
```

```python
tree_dot = export_graphviz(tree, out_file=None, feature_names=cancer.feature_names)
```

```python
print(tree_dot)
```

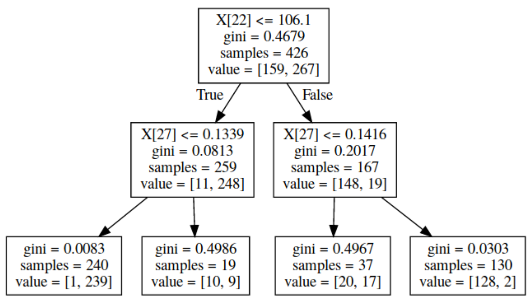
```
digraph Tree {
node [shape=box] ;
0 [label="X[22] <= 106.1\ngini = 0.4679\nsamples = 426\nvalue = [159, 267]"] ;
1 [label="X[27] <= 0.1339\ngini = 0.0813\nsamples = 259\nvalue = [11, 248]"] ;
0 -> 1 [labeldistance=2.5, labelangle=45, headlabel="True"] ;
2 [label="gini = 0.0083\nsamples = 240\nvalue = [1, 239]"] ;
1 -> 2 ;
3 [label="gini = 0.4986\nsamples = 19\nvalue = [10, 9]"] ;
1 -> 3 ;
4 [label="X[27] <= 0.1416\ngini = 0.2017\nsamples = 167\nvalue = [148, 19]"] ;
0 -> 4 [labeldistance=2.5, labelangle=-45, headlabel="False"] ;
5 [label="gini = 0.4967\nsamples = 37\nvalue = [20, 17]"] ;
4 -> 5 ;
6 [label="gini = 0.0303\nsamples = 130\nvalue = [128, 2]"] ;
4 -> 6 ;
}
```

Showing dot files in Jupyter

- First install graphvis C library:
  conda install graphviz
- Then install graphviz python library:

  pip install graphviz

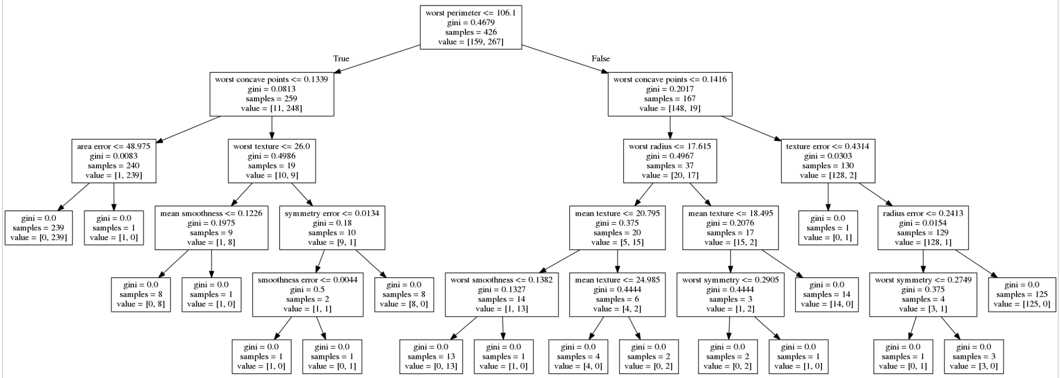
```
import graphviz
graphviz.Source(tree_dot)
```



Parameter tuning

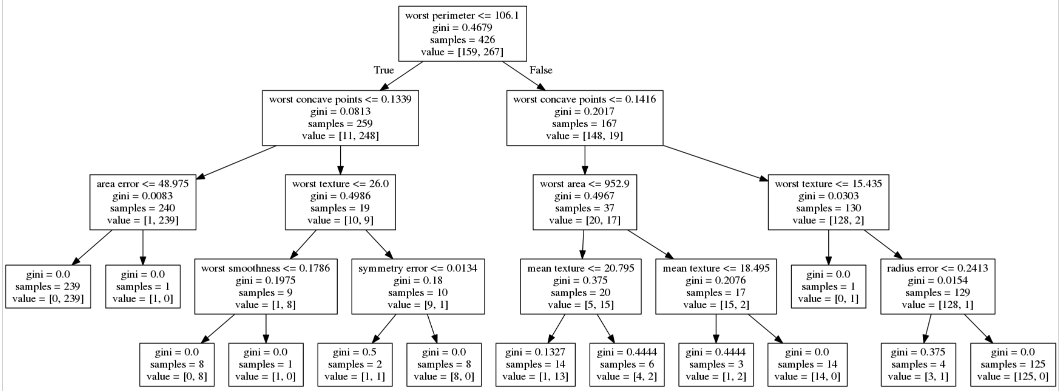
- Pre-pruning and post-pruning (not in sklearn yet)
- Limit tree size (pick one):
  max_depth
  max_leaf_nodes
  min_samples_split
  (and more)

No pruning



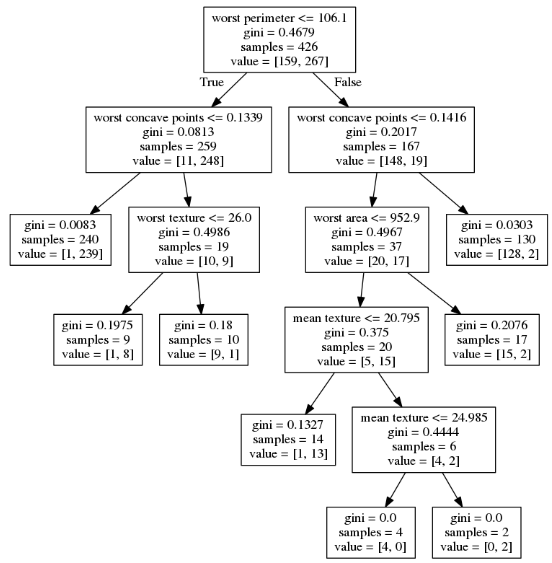
worst perimeter <= 106.1
gini = 0.4679
samples = 426
value = [159, 267]
True
False
worst concave points <= 0.1339
gini = 0.0813
samples = 259
value = [11, 248]
worst concave points <= 0.1416
gini = 0.2017
samples = 167
value = [148, 19]
area error <= 48.975
gini = 0.0083
samples = 240
value = [1, 239]
worst texture <= 26.0
gini = 0.4986
samples = 19
value = [10, 9]
worst radius <= 17.615
gini = 0.4967
samples = 37
value = [20, 17]
texture error <= 0.4314
gini = 0.0303
samples = 130
value = [128, 2]
gini = 0.0
samples = 239
value = [0, 239]
gini = 0.0
samples = 1
value = [1, 0]
mean smoothness <= 0.1226
gini = 0.1975
samples = 9
value = [1, 8]
symmetry error <= 0.0134
gini = 0.18
samples = 10
value = [9, 1]
mean texture <= 20.795
gini = 0.375
samples = 20
value = [5, 15]
mean texture <= 18.495
gini = 0.2076
samples = 17
value = [15, 2]
gini = 0.0
samples = 1
value = [0, 1]
radius error <= 0.2413
gini = 0.0154
samples = 129
value = [128, 1]
gini = 0.0
samples = 8
value = [0, 8]
gini = 0.0
samples = 1
value = [1, 0]
smoothness error <= 0.0044
gini = 0.5
samples = 2
value = [1, 1]
gini = 0.0
samples = 8
value = [8, 0]
worst smoothness <= 0.1382
gini = 0.1327
samples = 14
value = [1, 13]
mean texture <= 24.985
gini = 0.4444
samples = 6
value = [4, 2]
worst symmetry <= 0.2905
gini = 0.4444
samples = 3
value = [1, 2]
gini = 0.0
samples = 14
value = [14, 0]
worst symmetry <= 0.2749
gini = 0.375
samples = 4
value = [3, 1]
gini = 0.0
samples = 125
value = [125, 0]
gini = 0.0
samples = 1
value = [1, 0]
gini = 0.0
samples = 1
value = [0, 1]
gini = 0.0
samples = 13
value = [0, 13]
gini = 0.0
samples = 1
value = [1, 0]
gini = 0.0
samples = 4
value = [4, 0]
gini = 0.0
samples = 2
value = [0, 2]
gini = 0.0
samples = 2
value = [0, 2]
gini = 0.0
samples = 1
value = [1, 0]
gini = 0.0
samples = 1
value = [0, 1]
gini = 0.0
samples = 3
value = [3, 0]

max_depth=4

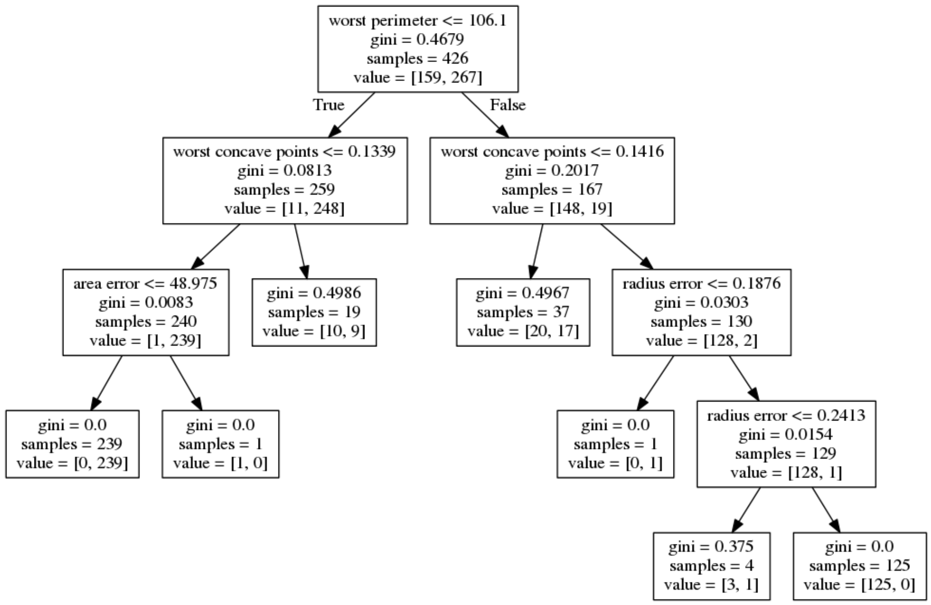


worst perimeter <= 106.1
gini = 0.4679
samples = 426
value = [159, 267]
True
False
worst concave points <= 0.1339
gini = 0.0813
samples = 259
value = [11, 248]
worst concave points <= 0.1416
gini = 0.2017
samples = 167
value = [148, 19]
area error <= 48.975
gini = 0.0083
samples = 240
value = [1, 239]
worst texture <= 26.0
gini = 0.4986
samples = 19
value = [10, 9]
worst area <= 952.9
gini = 0.4967
samples = 37
value = [20, 17]
worst texture <= 15.435
gini = 0.0303
samples = 130
value = [128, 2]
gini = 0.0
samples = 239
value = [0, 239]
gini = 0.0
samples = 1
value = [1, 0]
worst smoothness <= 0.1786
gini = 0.1975
samples = 9
value = [1, 8]
symmetry error <= 0.0134
gini = 0.18
samples = 10
value = [9, 1]
mean texture <= 20.795
gini = 0.375
samples = 20
value = [5, 15]
mean texture <= 18.495
gini = 0.2076
samples = 17
value = [15, 2]
gini = 0.0
samples = 1
value = [0, 1]
radius error <= 0.2413
gini = 0.0154
samples = 129
value = [128, 1]
gini = 0.0
samples = 8
value = [0, 8]
gini = 0.0
samples = 1
value = [1, 0]
gini = 0.5
samples = 2
value = [1, 1]
gini = 0.0
samples = 8
value = [8, 0]
gini = 0.1327
samples = 14
value = [1, 13]
gini = 0.4444
samples = 6
value = [4, 2]
gini = 0.4444
samples = 3
value = [1, 2]
gini = 0.0
samples = 14
value = [14, 0]
gini = 0.375
samples = 4
value = [3, 1]
gini = 0.0
samples = 125
value = [125, 0]

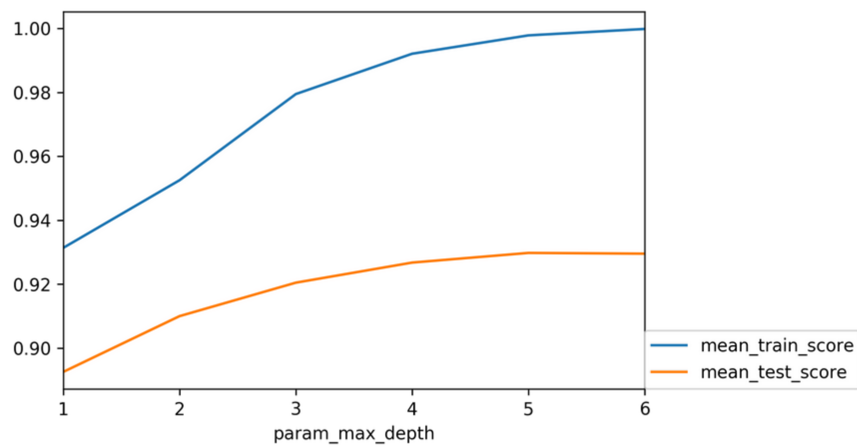
max_leaf_nodes=8



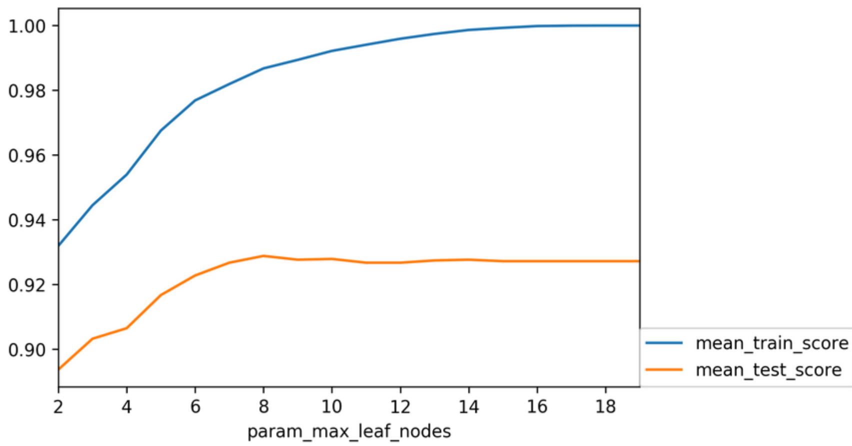
min_samples_split=50

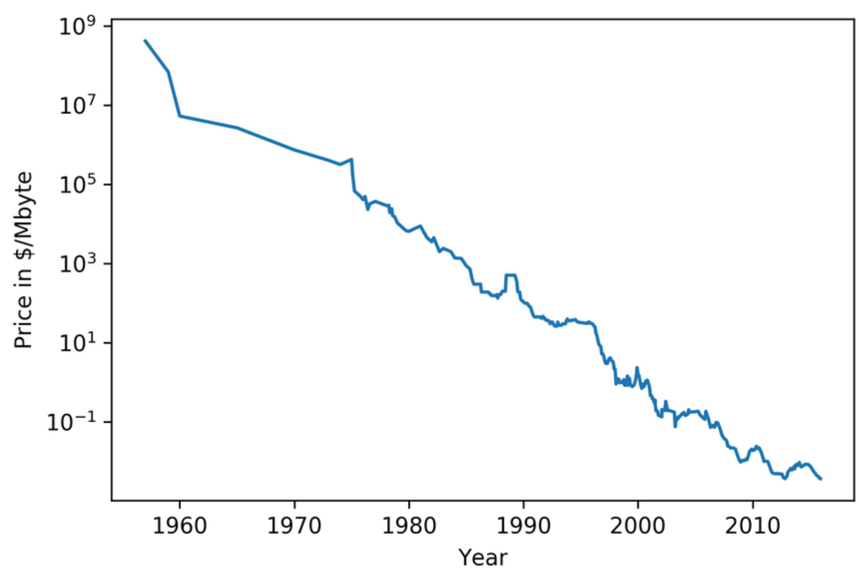


```python
from sklearn.model_selection import GridSearchCV
param_grid = {'max_depth':range(1, 7)}
grid = GridSearchCV(DecisionTreeClassifier(random_state=0), param_grid=param_grid, cv=10)
grid.fit(X_train, y_train)
```

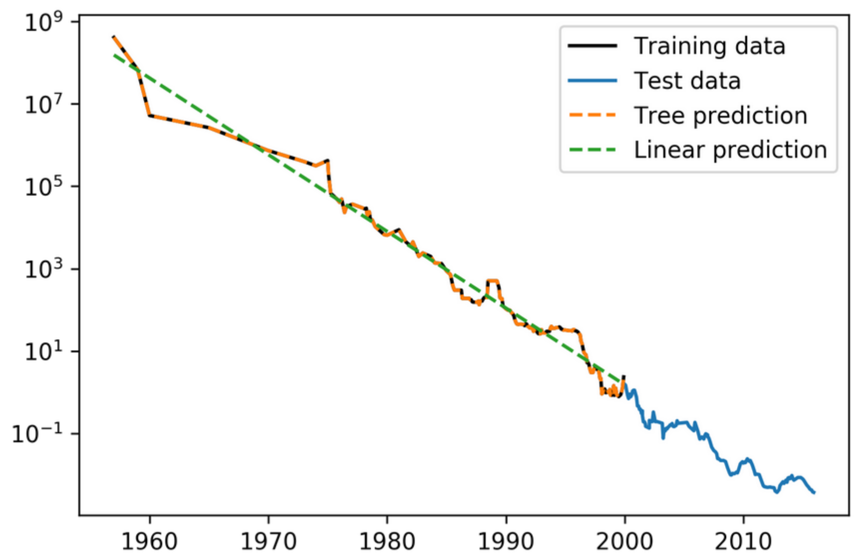


```python
from sklearn.model_selection import GridSearchCV
param_grid = {'max_leaf_nodes':range(2, 20)}
grid = GridSearchCV(DecisionTreeClassifier(random_state=0), param_grid=param_grid, cv=10)
grid.fit(X_train, y_train)
```



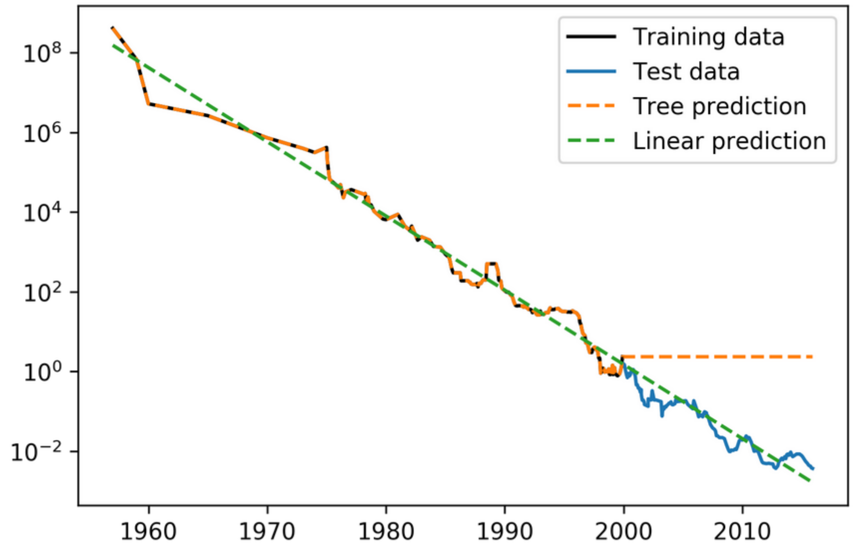


10^9
10^7
10^5
10^3
10^1
10^{-1}
Price in $/Mbyte
1960
1970
1980
1990
2000
2010
Year



Training data
Test data
Tree prediction
Linear prediction
10^9
10^7
10^5
10^3
10^1
10^{-1}
1960
1970
1980
1990
2000
2010

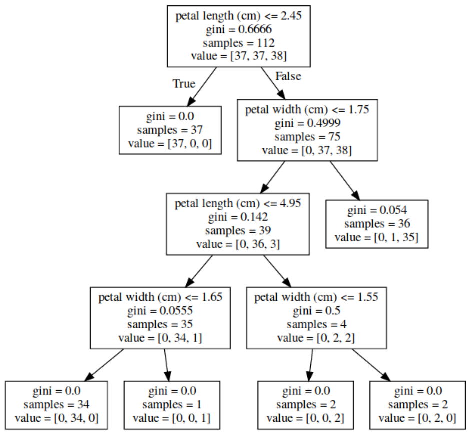
Extrapolation



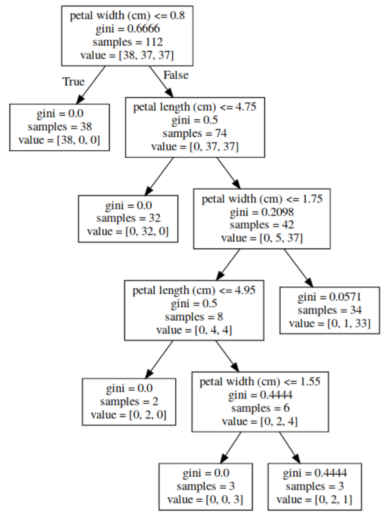
Would be the same for nearest neighbor regression!

Instability

```
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, stratify=iris.target, random_state=0)
tree = DecisionTreeClassifier(max_leaf_nodes=6).fit(X_train, y_train)
tree_dot = export_graphviz(tree, out_file=None, feature_names=iris.feature_names)
graphviz.Source(tree_dot)
```

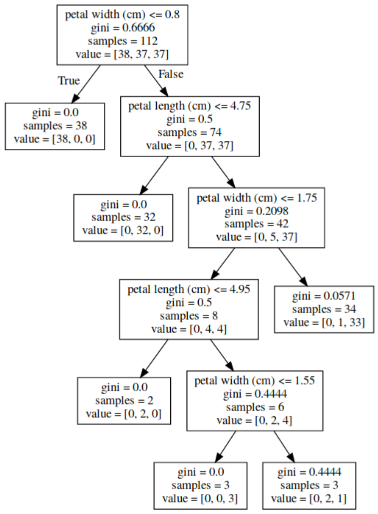



```
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, stratify=iris.target, random_state=1)
tree = DecisionTreeClassifier(max_leaf_nodes=6).fit(X_train, y_train)
tree_dot = export_graphviz(tree, out_file=None, feature_names=iris.feature_names)
graphviz.Source(tree_dot)
```



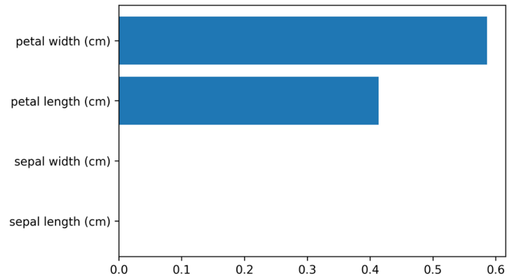
Feature importance

```
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, stratify=iris.target, random_state=1)
tree = DecisionTreeClassifier(max_leaf_nodes=6).fit(X_train, y_train)
tree_dot = export_graphviz(tree, out_file=None, feature_names=iris.feature_names)
graphviz.Source(tree_dot)
```




```
tree.feature_importances_
```

```
array([ 0.   ,  0.   ,  0.414,  0.586])
```



Unstable tree → unstable feature importances.

Might take one or multiple from a group of correlated features.

Categorical Data

- Can split on categorical data directly
- Intuitive way to split: split in two subsets
- 2 ^ n_values many possibilities
- Possible to do in linear time exactly for gini index and binary classification.
- Heuristics done in practice for multi-class.
- Not in sklearn release version :(

Predicting probabilities

- Fraction of class in leaf.
- Without pruning: Always 100% certain!
- Even with pruning might be too certain.

Conditional Inference Trees

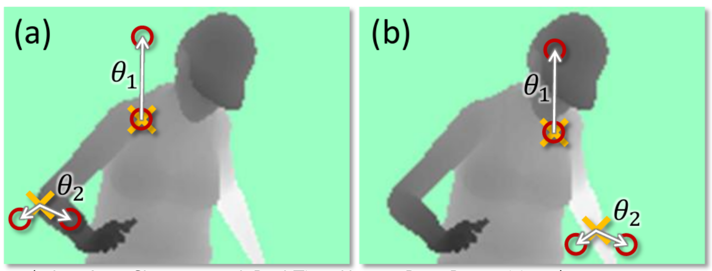
- Select "best" split with correcting for multiple-hypothesis testing.
- More "fair" to categorical variables.
- Only in R so far (party)

Relation to Nearest Neighbors

- Predict average of neighbors – either by k, by epsilon ball or by leaf.
- Trees are much faster to predict.
- Both can't extrapolate

Different Splitting Methods

- Could use anything as split candidate!
- Linear models used if extrapolation is needed.
- Computer vision: pixel comparisons
- Kinect (first generation): depth comparison



(taken from Shotton et. al. Real-Time Human Pose Recognition ..)

Ensemble Models

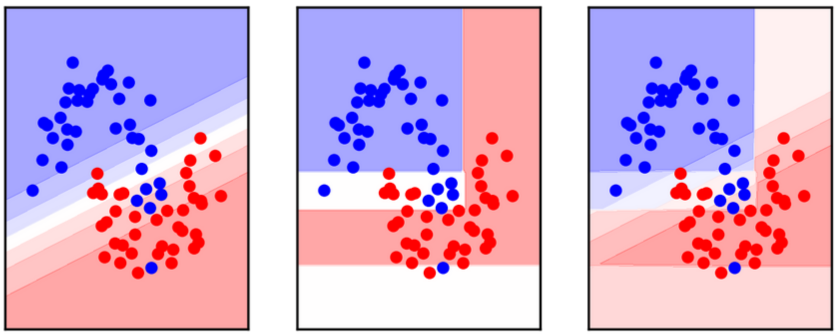
Poor man's ensembles

- Build different models
- Average the result
- Owen Zhang (long time kaggle 1st): build XGBoosting models with different random seeds.
- More models are better – if they are not correlated.
- Also works with neural networks
- You can average any models as long as they provide calibrated ("good") probabilities.
- Scikit-learn: VotingClassifier
  hard and soft voting

VotingClassifier

```
voting = VotingClassifier([('logreg', LogisticRegression(C=100)),
                           ('tree', DecisionTreeClassifier(max_depth=3, random_state=0))],
                         voting='soft')
voting.fit(X_train, y_train)
lr, tree = voting.estimators_
print(("{:.2f} " * 3).format(voting.score(X_test, y_test),
                             lr.score(X_test, y_test), tree.score(X_test, y_test)))
```

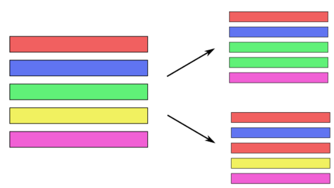
```
0.88 0.84 0.80
```



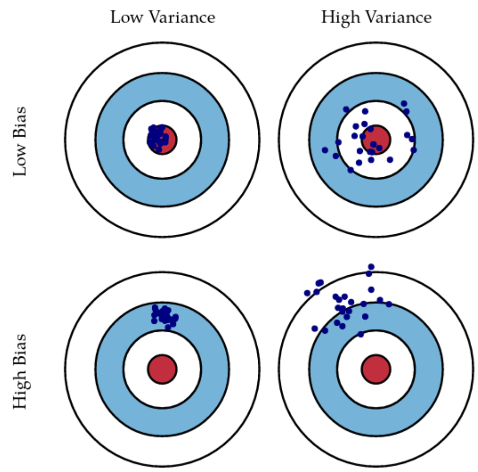
Bagging (Bootstrap AGGregation)

- Generic way to build “slightly different” models
- Draw bootstrap samples from dataset

  (as many as there are in the dataset, with repetition)
- Implemented in BaggingClassifier, BaggingRegressor



Bias and Variance



http://scott.fortmann-roe.com/docs/BiasVariance.html

Bias and Variance in Ensembles

- Breiman showed that generalization depends on strength of the individual classifiers and (inversely) on their correlation
- Uncorrelating them might help, even at the expense of strength

So decision trees are a great idea, but unfortunately they don’t work that well in practice. However, there is a modification of the algorithm that works very well, called random forest.

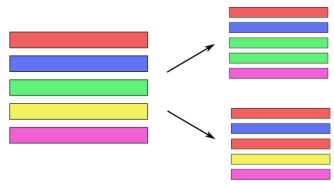
The idea behind random forest is that we build many decision trees, but we inject some randomness into each tree, so that they are all different.

Then, to make a prediction, we look at the prediction of all the decision trees and take the average.

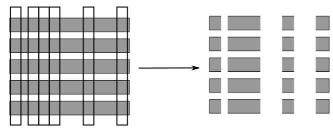
Randomize in two ways

- For each **tree**:

  Pick bootstrap sample of data



- For each **split**:
  Pick random sample of features



- More tree are always better

Tuning Random Forests

- Main parameter: max_features
  - around sqrt(n_features) for classification
  - Around n_features for regression

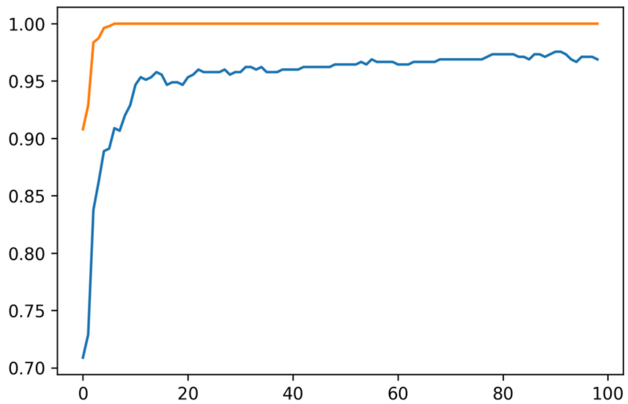
- n_estimators > 100
- Prepruning might help, definitely helps with model size!
- max_depth, max_leaf_nodes, min_samples_split again

Extremely Randomized Trees

- More randomness!
- Randomly draw threshold for each feature!
- Doesn't use bootstrap
- Faster because no sorting / searching
- Can have smoother boundaries

Warm-Starts

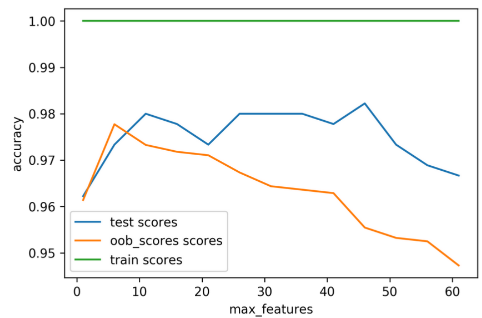
```
train_scores = []
test_scores = []

rf = RandomForestClassifier(warm_start=True)
for n_estimators in range(1, 100, 5):
    rf.n_estimators = n_estimators
    rf.fit(X_train, y_train)
    train_scores.append(rf.score(X_train, y_train))
    test_scores.append(rf.score(X_test, y_test))
```



Out-of-bag estimates

- Each tree only uses ~66% of data
- Can evaluate it on the rest!
- Make predictions for out-of-bag, average, score.
- Each prediction is an average over different subset of trees

```
feature_range = range(1, 64, 5)
for max_features in feature_range:
    rf = RandomForestClassifier(max_features=max_features, oob_score=True, n_estimators=200, random_state=0)
    rf.fit(X_train, y_train)
    train_scores.append(rf.score(X_train, y_train))
    test_scores.append(rf.score(X_test, y_test))
    oob_scores.append(rf.oob_score_)
```



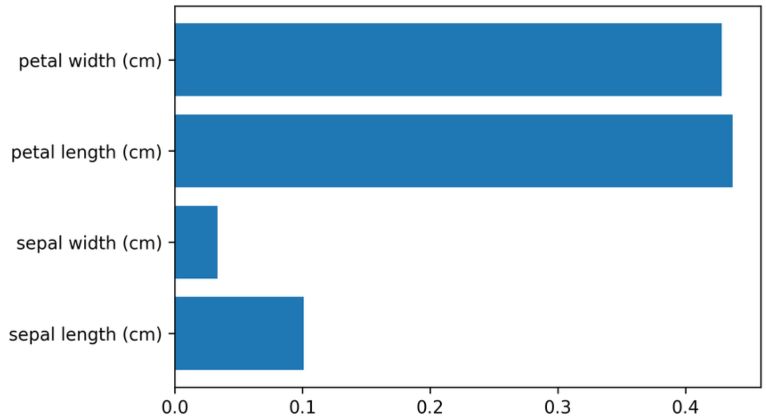
Variable Importance

```
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, stratify=iris.target, random_state=1)
rf = RandomForestClassifier(n_estimators=100).fit(X_train, y_train)
```

```
rf.feature_importances_
```

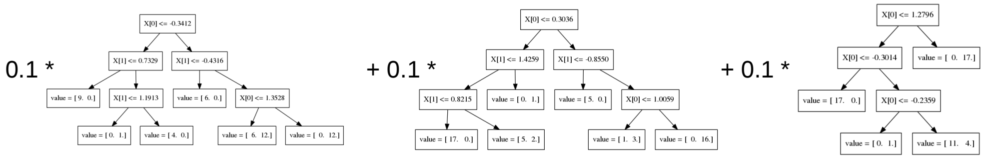
```
array([ 0.101,  0.034,  0.437,  0.428])
```

```
plt.barh(range(4), rf.feature_importances_)
plt.yticks(range(4), iris.feature_names);
```



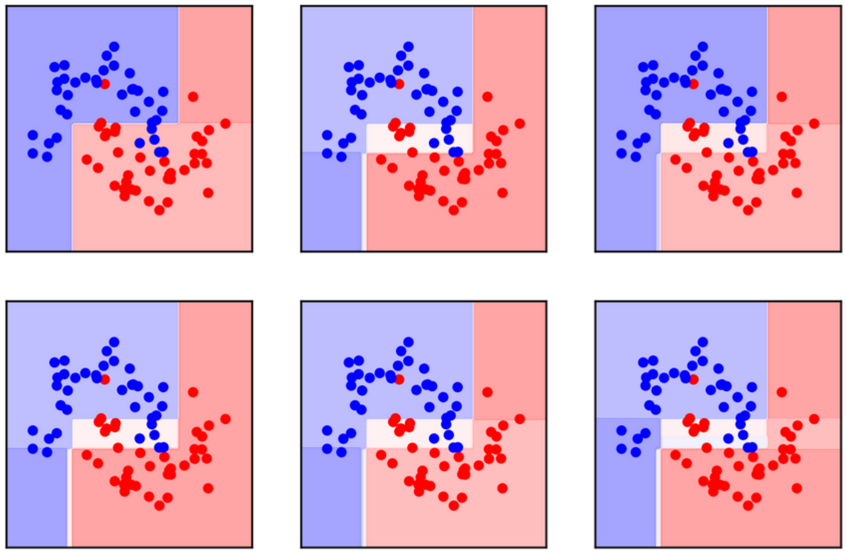
Gradient Boosting

Gradient Boosting Algorithm



- Iteratively add regression trees to model
- Use log loss for classification
- Discount update by learning rate

GradientBostingClassifier(max_depth=2)



Gradient Boosting

- Many shallow trees
- learning_rate ↔ n_estimators
- Slower to train than RF (serial), but much faster to predict
- Small model size
- Uses one-vs-rest for multi-class!

Tuning Gradient Boosting

- Pick n_estimators, tune learning rate
- Can also tune max_features
- Typically strong pruning via max_depth

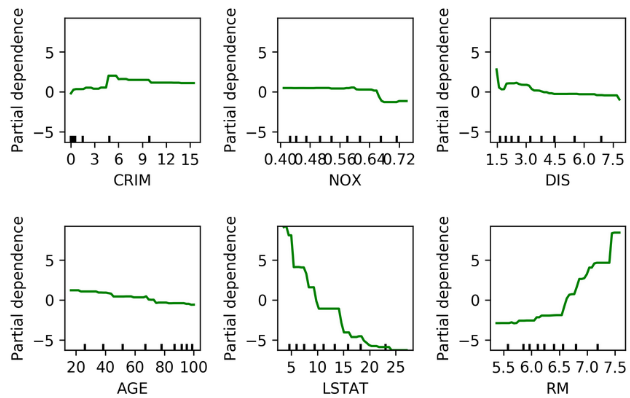
Partial Dependence Plots

- Marginal dependence of prediction on one or two features

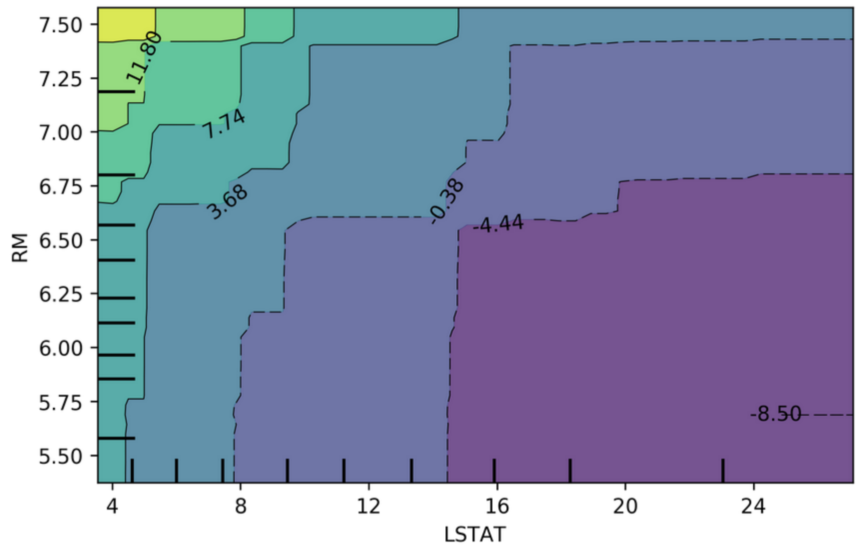
```
boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(
    boston.data, boston.target, random_state=0)

gbrt = GradientBoostingRegressor().fit(X_train, y_train)
gbrt.score(X_test, y_test)
```

```
0.81962024379538989
```

```
from sklearn.ensemble.partial_dependence import plot_partial_dependence
fig, axs = plot_partial_dependence(gbrt, X_train, np.argsort(gbrt.feature_importances_)[-6:],
                                   feature_names=boston.feature_names,
                                   n_jobs=3, grid_resolution=50)
plt.tight_layout()
```

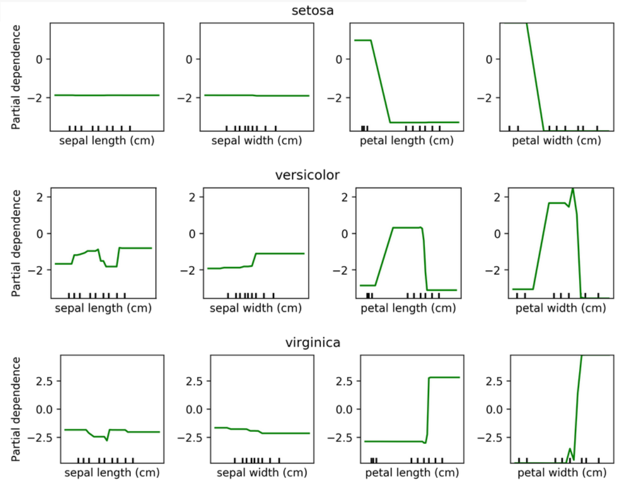


```
fig, axs = plot_partial_dependence(gbrt, X_train, [np.argsort(gbrt.feature_importances_)[-2:]],
                                   feature_names=boston.feature_names,
                                   n_jobs=3, grid_resolution=50)
```



Partial Dependence for Classification

```
from sklearn.ensemble.partial_dependence import plot_partial_dependence
for i in range(3):
    fig, axs = plot_partial_dependence(gbrt, X_train, range(4), n_cols=4,
                                       feature_names=iris.feature_names, grid_resolution=50, label=i,
                                       figsize=(8, 2))
    fig.suptitle(iris.target_names[i])
    for ax in axs: ax.set_xticks(())

    plt.tight_layout()
```



XGBoost

- Efficient implementation of gradient boosting
- Improvements on original algorithm
- https://arxiv.org/abs/1603.02754
- Adds l1 and l2 penalty on leaf-weights
- Fast approximate split finding
- Can pip-install
- Scikit-learn compatible interface

Boosting in General

- "Meta-algorithm" to create strong learners from weak learners.
- AdaBoost, GentleBoost, …
- Trees or stumps work best
- Gradient Boosting often the best of the bunch
- Many specialized algorithms (ranking etc)

When to use tree-based models

- Model non-linear relationships
- Single tree: very interpretable (if small)
- Random forests very robust, good benchmark
- Gradient boosting often best performance with careful tuning
- Doesn't care about scaling, no need for feature engineering!

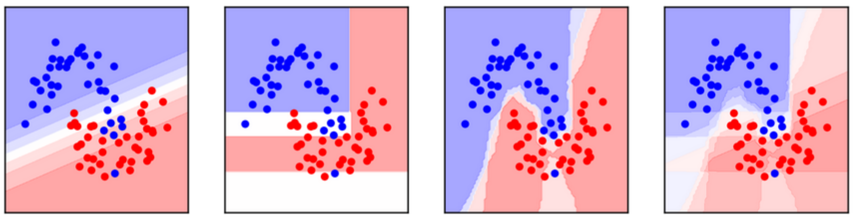
More ensembles: Stacking

Poor man's Stacking

- Build multiple models
- Train model on probabilities / scores produced

```
from sklearn.neighbors import KNeighborsClassifier

X, y = make_moons(noise=.2, random_state=18)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=0)

voting = VotingClassifier([('logreg', LogisticRegression(C=100)),
                           ('tree', DecisionTreeClassifier(max_depth=3, random_state=0)),
                           ('knn', KNeighborsClassifier(n_neighbors=3))
                          ],
                         voting='soft')
voting.fit(X_train, y_train)
lr, tree, knn = voting.estimators_
print(("{:.2f} " * 4).format(voting.score(X_test, y_test),
                             lr.score(X_test, y_test), tree.score(X_test, y_test),
                             knn.score(X_test, y_test)))
```

```
0.88 0.84 0.80 1.00
```



Poor man's Stacking

```
from sklearn.preprocessing import FunctionTransformer
# we need to reshape the result from votingclassifier.transform because
# of some annoyance in sklearn. We then keep only the probabilities of the positive classes!
reshaper = FunctionTransformer(lambda X_: np.rollaxis(X_, 1).reshape(-1, 6)[:, 1::2], validate=False)
stacking = make_pipeline(voting, reshaper,
                         LogisticRegression(C=100))
stacking.fit(X_train, y_train)
stacking.score(X_train, y_train)
```

```
0.98666666666666669
```

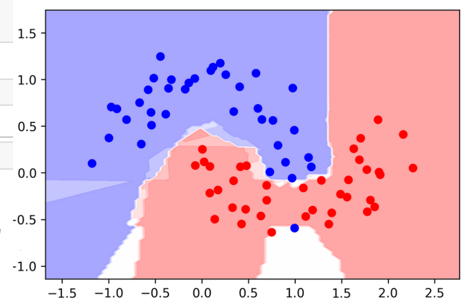
```
stacking.score(X_test, y_test)
```

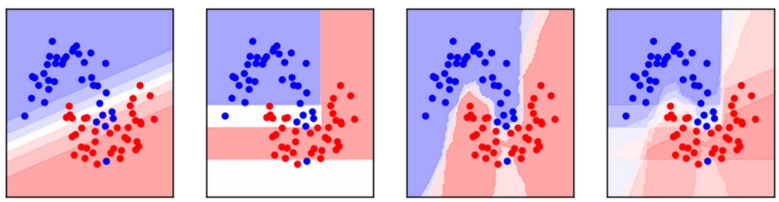
```
0.95999999999999996
```

```
stacking.named_steps['logisticregression'].coef_
```

```
array([[-2.625,  6.261,  9.501]])
```

Problem: Overfitting!

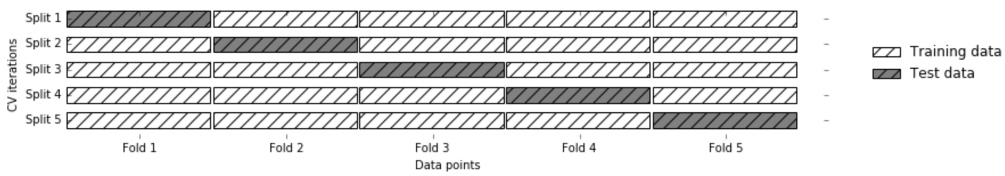




Stacking

- Use cross-validation (even LOO!) to produce probability estimates on training set.
- Train second step estimator on held-out estimates
- No overfitting of second step!
- For testing: as usual

Hold-out estimates of probabilities



- Split 1 produces probabilities for Fold 1, split2 for Fold 2 etc.
- Get a probability estimate for each data point!
- Unbiased estimates (like on the test set) for the whole training set!
- Without it: The best estimator is the one that memorized the training set.

Stacking with Scikit-learn

```
from sklearn.model_selection import cross_val_predict
first_stage = make_pipeline(voting, reshaper)
transform_cv = cross_val_predict(first_stage, X_train, y_train, cv=10, method="transform")
```

```
second_stage = LogisticRegression(C=100).fit(transform_cv, y_train)
print(second_stage.coef_)
```

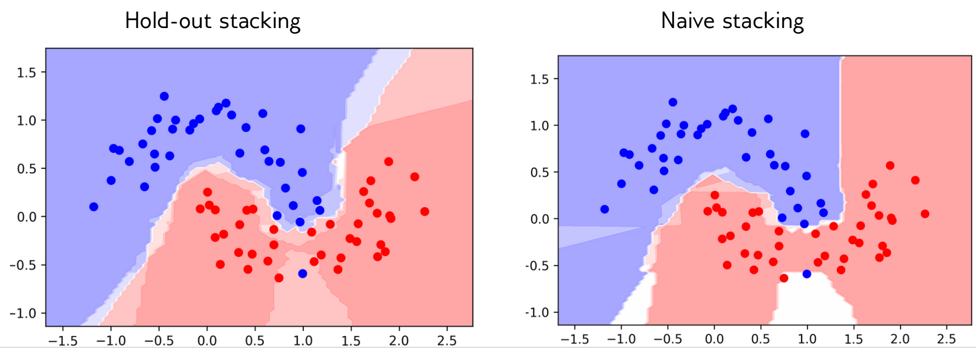
```
[[ 2.09  -1.424  7.93 ]]
```

```
second_stage.score(transform_cv, y_train)
```

```
0.9599999999999996
```

```
second_stage.score(first_stage.transform(X_test), y_test)
```

```
1.0
```



Summary