Prediction Models

Zhirong Yang

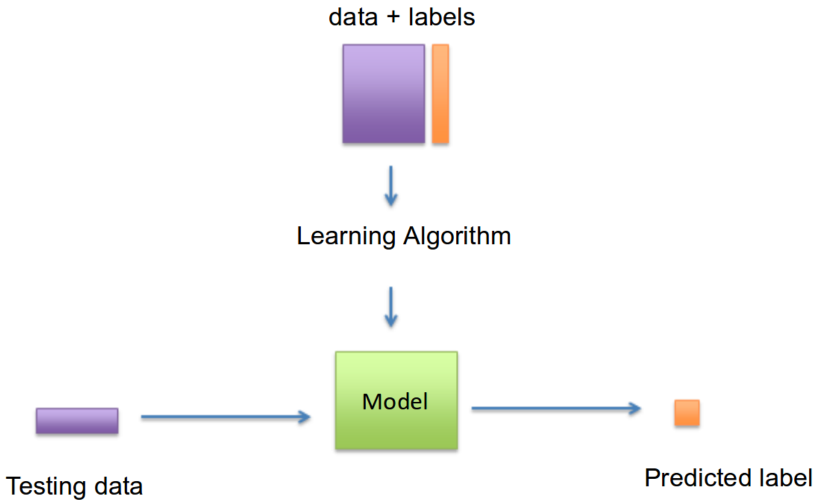
Supervised Learning

- Supervised learning is the major form for ML applications

Supervised Learning

- Supervised learning is the major form for ML applications
- Given a training set $D_{\text{tr}} = \{x_i, y_i\}_{i=1}^{N}$, $x_i \in \mathbb{X}$, $y_i \in \mathbb{Y}$
- Fit a model or function $f : \mathbb{X} \mapsto \mathbb{Y}$
- For newly coming $x^{\text{new}} \in \mathbb{X}$, predict $\hat{y}^{\text{new}} = f(x^{\text{new}})$.
- In this course we focus on $\mathbb{X} = \mathbb{R}^d$ and $\mathbb{Y} = \mathbb{R}$
- The problem is called
  - *classification*: if $\mathbb{Y}$ is categorical, e.g. face recognition
  - *regression*: if $\mathbb{Y}$ is continuous, e.g. temperature forecast

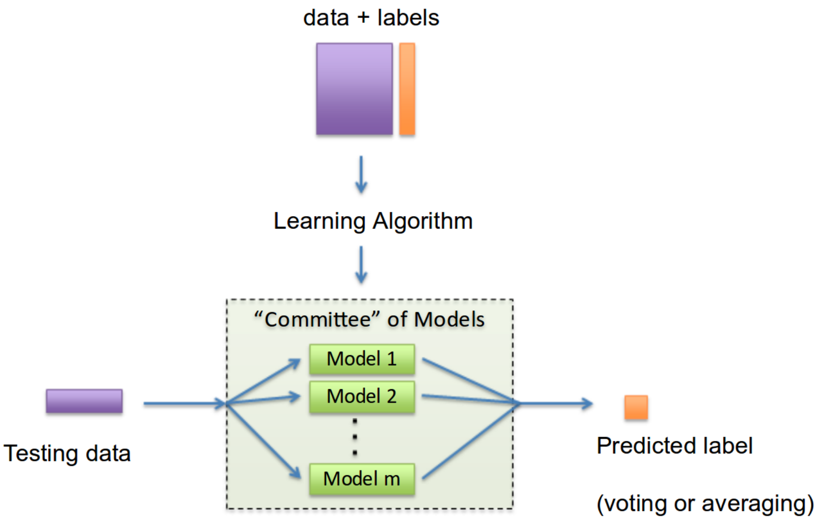
Usual supervised Learning



Ensemble Learning (EL)

- State-of-the-art
- Model ensemble basics
- AdaBoost
- Random Forest
- XGBoost
- CatBoost/LightGBM

What is ensemble learning



What is ensemble learning?

For example, given base (weak) learners $\{f_b\}_{b=1}^{B}$ and their weights w_b

- $$f(x) = \sum_{b=1}^{B} w_b f_b(x)$$
- If w_b is uniform
  - for binary classification: $f(x) = \mathrm{SIGN}(\frac{1}{B}\sum_{b=1}^{B} f_b(x))$
  - for multi-class classification: majority vote of $\{f_b(x)\}_{b=1}^{B}$

Why ensemble learning?

kaggle

- “*The proof is in the pudding*”
- Kaggle: the largest machine learning competition platform
- Provide training $\{x_i, y_i\}_{i=1}^{N}$ and testing $\{x_i\}_{i=N+1}^{N+M}$
- Compare prediction performance on $\{y_i\}_{i=N+1}^{N+M}$
- In practice, almost all Kaggle winners use ensemble learning

Why ensemble learning?

- In estimation theory (for true y and estimation $\hat{y}$),

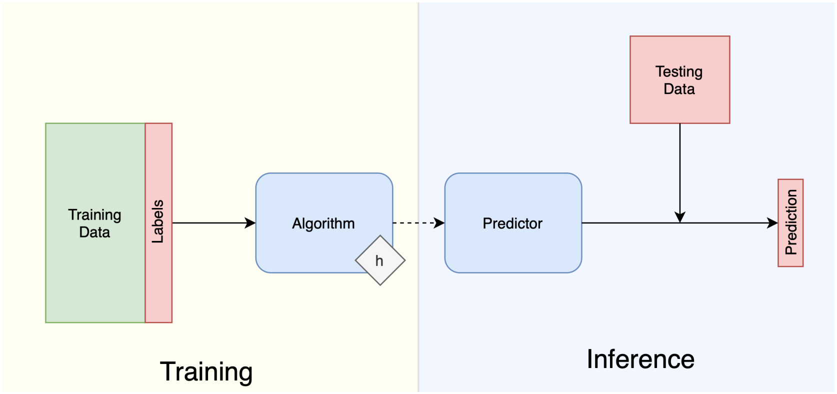
$$\mathbb{E}\left[(y-\hat{y})^2\right] = \underbrace{(y-\mathbb{E}[\hat{y}])^2}_{\mathrm{Bias}[\hat{y}]^2} + \underbrace{\mathbb{E}\left[(\mathbb{E}[\hat{y}]-\hat{y})^2\right]}_{\mathrm{Var}[\hat{y}]} + \text{constant}$$

- High bias: poor prediction on average
- High variance: not robust
  - occasionally works
  - sensitive to small perturbation of data
- Ensemble learning can
  - decrease bias, e.g. by boosting
  - decrease variance, e.g. by bagging
  - overall improvement, e.g. by stacking

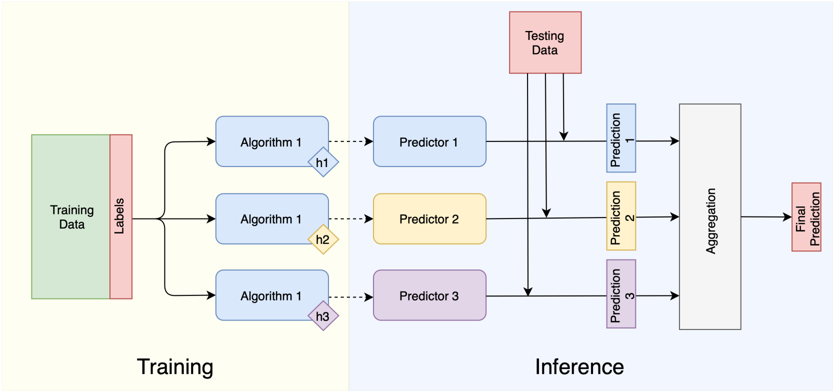
How to create different learners?

- With a combination of multiple learners, the individual performances are not very important.
- It is more important to have a colorful (diverse) collection
  - Different hyperparameters (e.g. degree of complexity)
  - Different learning algorithms
  - Different representations (feature selection/extraction)
  - Different training sets
  - Artificial noise added to the data
  - Random samples from posterior of the model parameters (instead of finding the maximum)

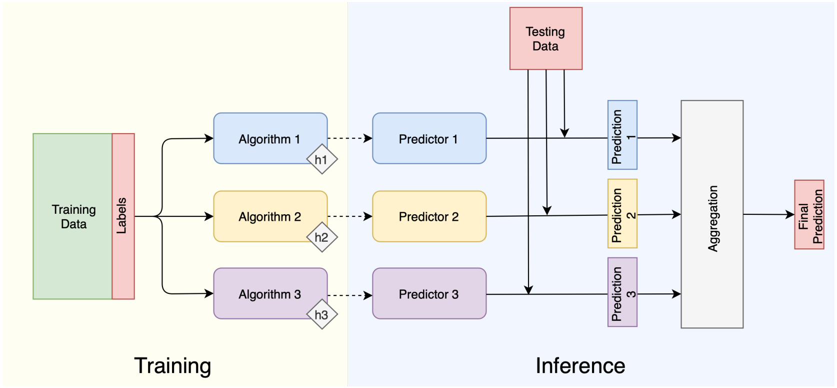
ML pipeline without ensemble



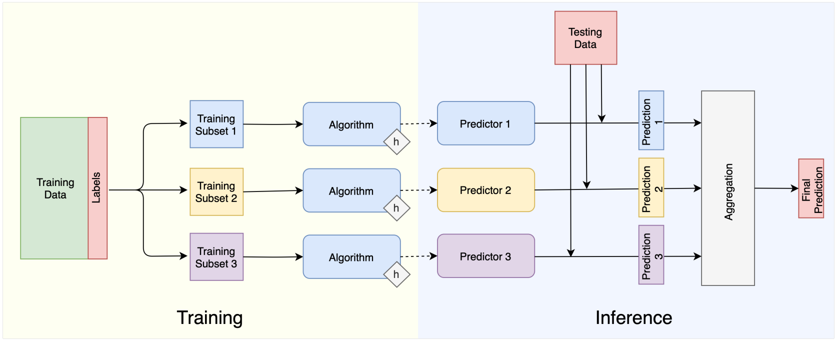
Ensemble: same algorithm with different hyperparameters



Ensemble: different algorithms



Ensemble: same algorithm with different training data subsets

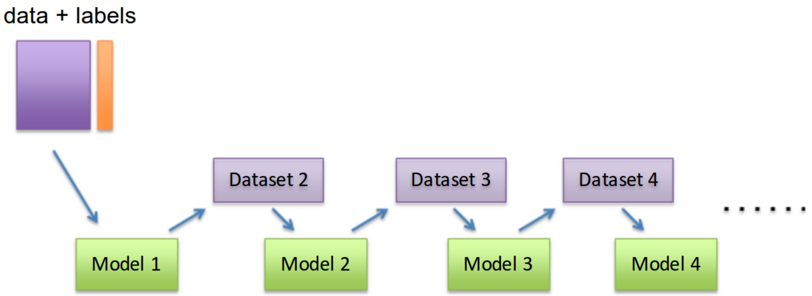


The order to generate base learners

- Sequentially: e.g. boosting
- Parallelly: e.g. bagging
- Hierarchically: e.g. stacking

Boosting

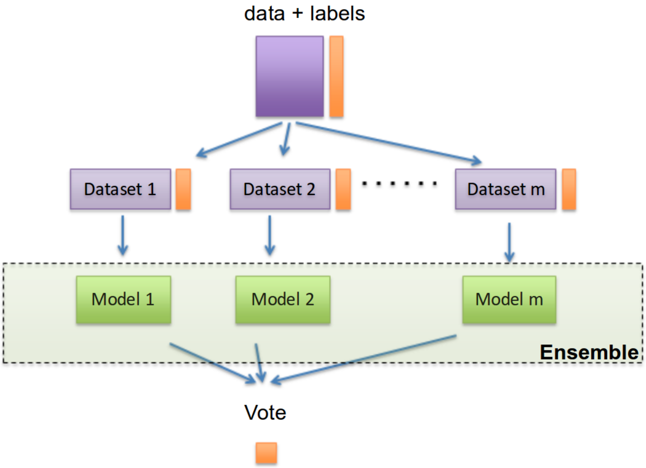
- Boosting involves incrementally building an ensemble by training each new model instance to emphasize the training instances that previous models mis-classified.



Each model corrects the mistakes of its predecessor.

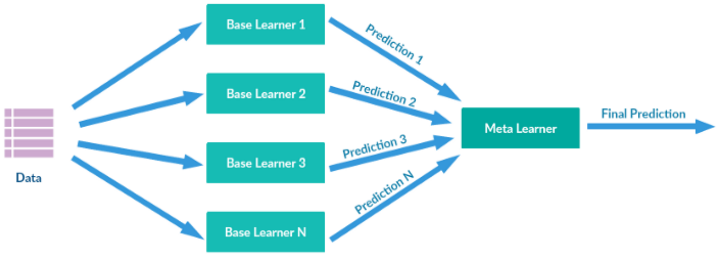
Bagging

- Simultaneously construct the base learners
- sometimes called Bootstrap AGGregatING



Stacking

- Use $[f_1(x), \ldots, f_B(x)]$ as features to fit a meta learner



AdaBoost

- The first practical boosting algorithm invented by Freund and Schapire (1995).
- Still useful nowadays
- It is a good illustration of ensemble learning

AdaBoost demo video

AdaBoost algorithm summary

- Sequentially construct a set of base learners
  - vertical/horizontal lines in the demo
- Each sample has a weight
  - emphasize more and more on the misclassified
- Weigh the base learners by their errors
- Final output is the weighted average

AdaBoost demo in `scikit-learn`

AdaBoost remarks

- **Generic**: accommodate any types of base learners
- **Convenient**: only one hyperparameter (num. of base learners)
- **Limitations**:
  - only admit a specific loss function
  - sensitive to noise and outliers
  - slower than XGBoost

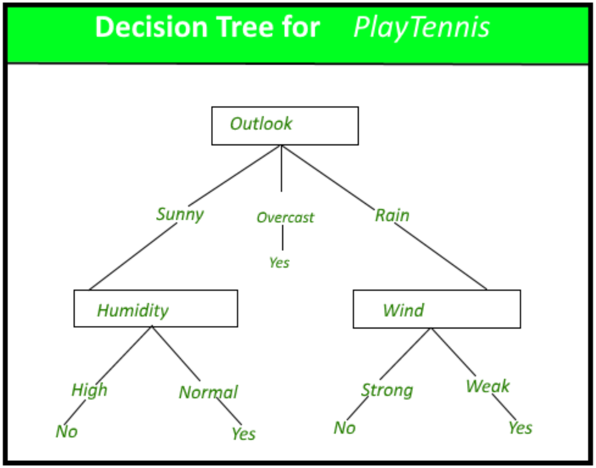
What is the most popular type of base learner?

- Logistic Regression
- Naive Bayes Classifier
- Support Vector Machines
- Decision Tree
- Neural Networks
- (k-)Nearest Neighbor
- etc.

What is the most popular type of base learner?

- Logistic Regression
- Naive Bayes Classifier
- Support Vector Machines
- **Decision Tree**
- Neural Networks
- (k-)Nearest Neighbor
- etc.

What is a decision tree?



Why decision tree?

- Highly accurate: almost half of data science challenges are won by tree based methods.
- Easy to use: invariant to input scale, get good performance with little tuning.
- Easy to interpret and control

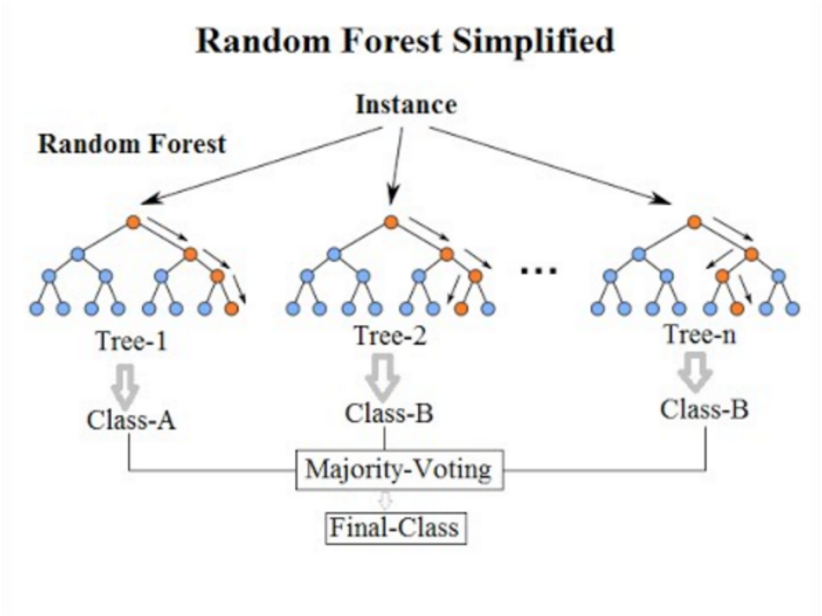
Random Forest

- A single decision tree is often not strong enough
- We need an ensemble of trees (i.e. forest)
- Can use AdaBoost to create the forest *sequentially*
- How can we create the ensemble *in parallel*?
- Can we just use different training subsets and then bagging?

Random Forest

- A single decision tree is often not strong enough
- We need an ensemble of trees (i.e. forest)
- Can use AdaBoost to create the forest *sequentially*
- How can we create the ensemble *in parallel*?
- Can we just use different training subsets and then bagging?
  A: No, the resulting trees tend to use similar splits. We need to also use different random feature subsets.

Random Forest diagram

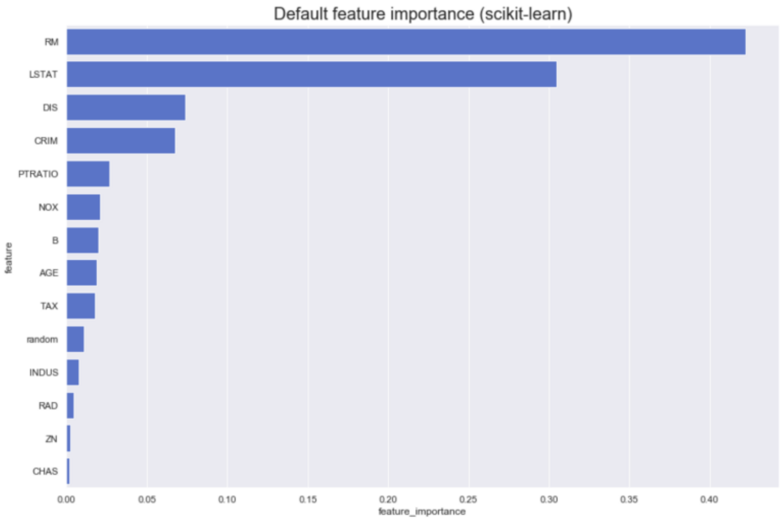


A walkthrough of IRIS classification

Obtain feature importance

- 'weight' or 'split': the number of times a feature is used to split the data across all trees
- 'gain': average gain of the feature when it is used in trees
- 'cover': the number of samples affected by the split

Feature selection

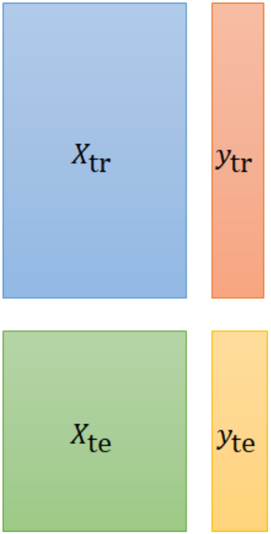


Tuning hyperparameters

- Every software (model) has tunable parameters
- Don't expect the software works optimally by default!
- How can we tune these parameters?

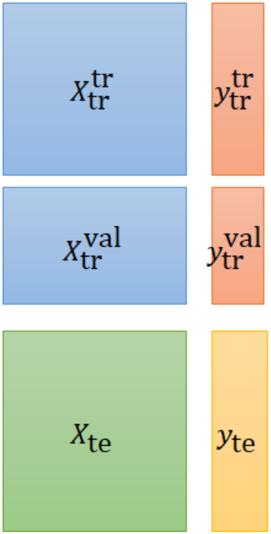
In Real-World Machine Learning Task

- You have X_{tr}, y_{tr}, and X_{te}
- You do not have y_{te}
- You use X_{tr} and y_{tr} to build a predictor
- Your predictor takes X_{te} as input, and outputs y_{pred}
- A good predictor gives y_{pred} close to y_{te} (Remember the P element in Machine Learning definition)

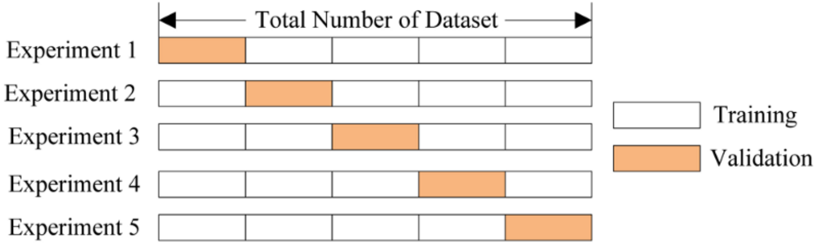


Validation: to mimic the test data

- You cannot get the exact performance without y_{te}
- But you can estimate it by validation
  - Split training data to training and validation subsets
  - Use $X_{\text{tr}}^{\text{tr}}$ and $y_{\text{tr}}^{\text{tr}}$ to build a predictor
  - The predictor takes $X_{\text{tr}}^{\text{val}}$ as input, and outputs $y_{\text{pred}}^{\text{val}}$
  - You score your predictor by comparing $y_{\text{pred}}^{\text{val}}$ and $y_{\text{tr}}^{\text{val}}$


$X_{\text{tr}}^{\text{tr}}$
$y_{\text{tr}}^{\text{tr}}$
$X_{\text{tr}}^{\text{val}}$
$y_{\text{tr}}^{\text{val}}$
X_{te}
y_{te}

Cross Validation: to make the estimate more reliable

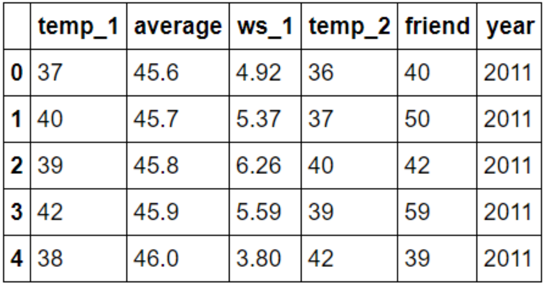


- Divide the data into several folds
- Use a fold for validation in an experiment and the others for training
- Loop over all folds
- Get average score
- You don't need to implement the above from scratch. You can use e.g., `cross_val_score()` in Scikit-Learn.

The whole procedure

1. Use X_{tr} and y_{tr} to get as high as possible (cross) validation score (i.e. run various predictors and tune their hyperparameters)
2. Fit the best predictor again, with its best hyperparamters and using the all X_{tr} and y_{tr}
3. Use the predictor on X_{te} and get y_{pred}

Example: predict the temperature tomorrow

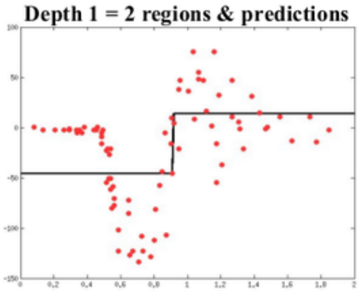


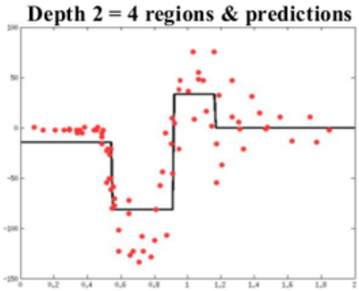
	temp_1	average	ws_1	temp_2	friend	year
0	37	45.6	4.92	36	40	2011
1	40	45.7	5.37	37	50	2011
2	39	45.8	6.26	40	42	2011
3	42	45.9	5.59	39	59	2011
4	38	46.0	3.80	42	39	2011

- temp_1 = max temperature (in F) one day prior
- average = historical average max temperature
- ws_1 = average wind speed one day prior
- temp_2 = max temperature two days prior
- friend = prediction from our “trusty” friend
- year = calendar year

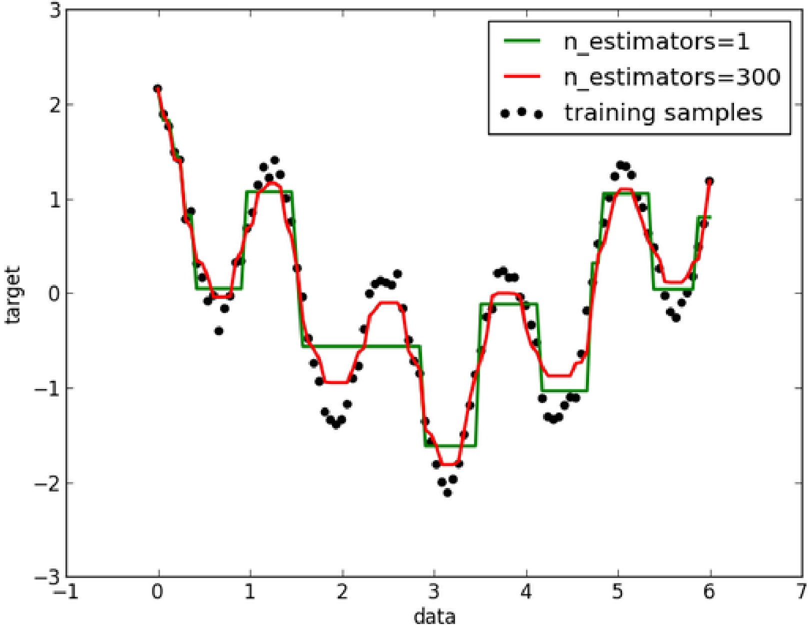
Regression Tree

- Not only binary outputs
- Includes real numbers in the leave nodes





Regression Forest



Random Forest hyperparameters

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(random_state = 42)
from pprint import pprint
pprint(rf.get_params())

{'bootstrap': True,
 'criterion': 'mse',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 10,
 'n_jobs': 1,
 'oob_score': False,
 'random_state': 42,
 'verbose': 0,
 'warm_start': False}
```

Common tunable hyperparameters

1. bootstrap = with or without replacement
2. max_depth = max #levels in each decision tree
3. max_features = max #features considered for splitting a node
4. min_samples_leaf = min #data points allowed in a leaf node
5. min_samples_split = min #data points placed in a node before the node is split
6. n_estimators = #trees in the forest

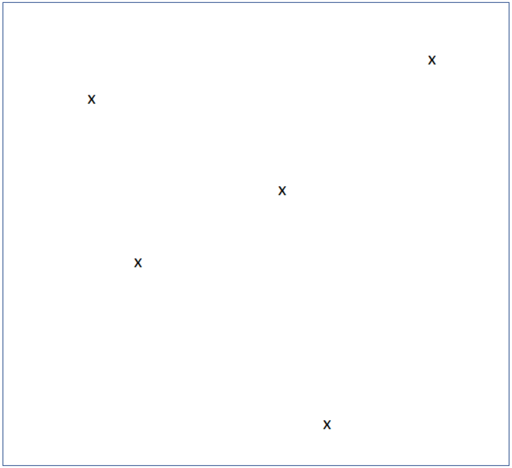
```
{'bootstrap': [True, False],
'max_depth': [10,20,30,40,50,60,70,80,90,100,110,None],
'max_features': ['auto', 'sqrt'],
'min_samples_leaf': [1, 2, 4],
'min_samples_split': [2, 5, 10],
'n_estimators':[200,400,600,800,1000,1200,1400,1600,1800,2000]}
```

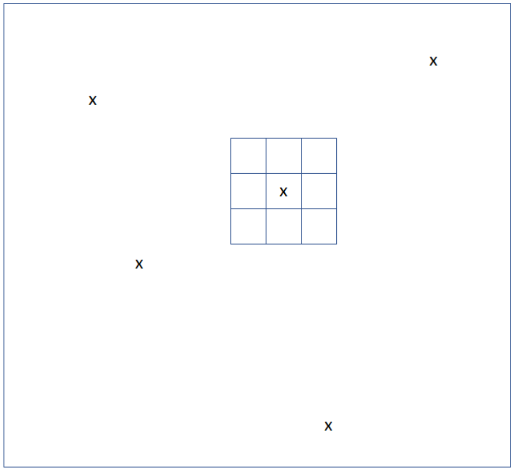
Grid search: $2 \times 12 \times 2 \times 3 \times 3 \times 10 = 4320$ trials!

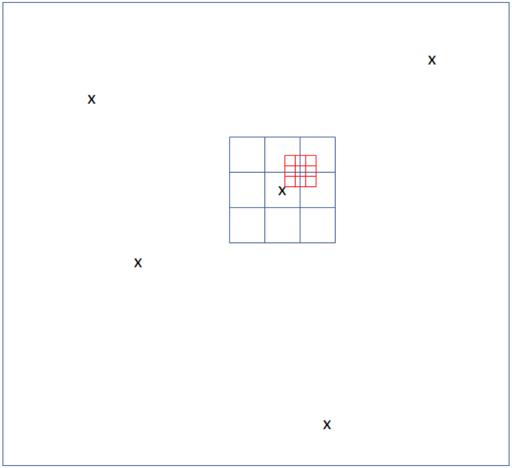
Budgeted hyperparameter selection

1. Random grid search (e.g. 100 trials)
2. Coarse grid search (e.g. $1 \times 4 \times 2 \times 3 \times 3 \times 4 = 288$ trials)
3. Fine grid search (e.g. $1 \times 3 \times 2 \times 3 \times 1 \times 3 = 54$ trials)

In total $100 + 288 + 54 = 442$ trials.


x
x
x
x
x


x
x
x
x
x


x
x
x
x
x

A walk through of hyperparameter selection in RF

Random Forest: pros and cons

- Advantages
  - no need for feature normalization
  - parallel training
  - widely used
  - perform reasonably well with default hyperparameter
- Disadvantages
  - not easily interpretable
  - time-consuming in inference
  - hard to control model complexity

Other algorithms to learn Tree Ensembles

- Random Forest is often the first choice for prototyping
- Other options for higher accuracy:
  - Random Forest (Breiman 1997)
  - Gradient Tree Boosting (Friedman 1999)
  - Gradient Tree Boosting with Regularization (variant of original GBM)
    - Regularized Greedy Forest (RGF)
    - XGBoost
    - (LightGBM)
    - (CatBoost)

What is XGBoost?

- A Scalable System for Learning Tree Ensembles
  - Model improvement
    - Regularization to control model complexity
  - Systems optimizations
    - Out-of-core computing
    - Cache optimization
    - Distributed computing
  - Algorithm improvements
    - Sparse aware algorithm
    - Weighted approximate quantile sketch

Why XGBoost?

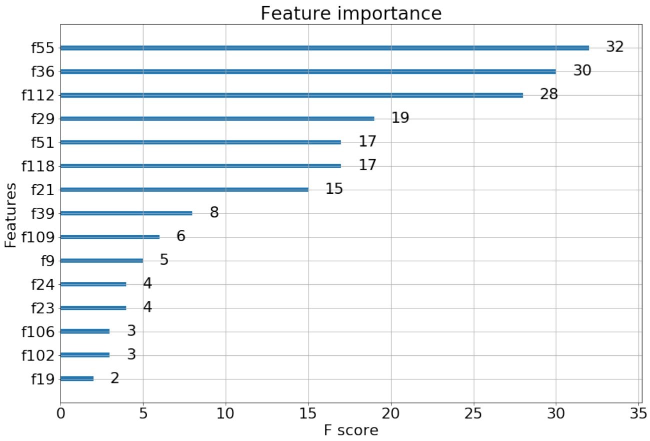
- More than half teams in Kaggle uses XGBoost
- Many industrial applications use XGBoost or its variants

XGBoost demo

```
import xgboost as xgb

# read in data
dtrain = xgb.DMatrix('data/agaricus.txt.train')
dtest = xgb.DMatrix('data/agaricus.txt.test')

# specify parameters via map
param = {'max_depth':2, 'eta':1, 'silent':1, 'objective':'binary:logistic' }
num_round = 10
model = xgb.train(param, dtrain, num_round)

# make prediction
preds = model.predict(dtest)
```

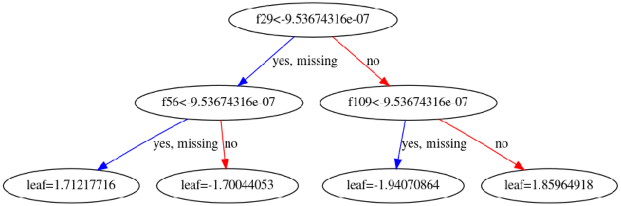
XGBoost feature importance

```
from matplotlib import pyplot
xgb.plot_importance(model, max_num_features=15)
pyplot.show()
```



XGBoost tree plot

```
xgb.plot_tree(model)
pyplot.show()
```

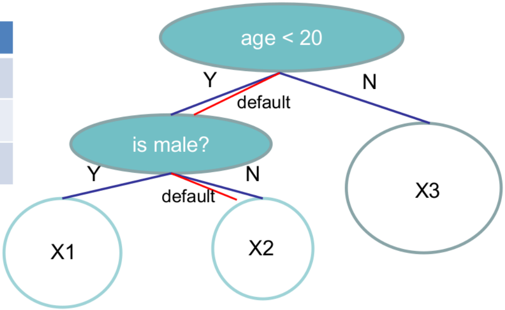


Automatic Missing Value Handling

XGBoost learns the best direction for missing values

Data

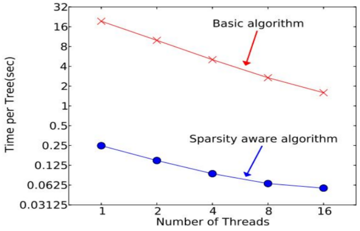
Example	Age	Gender
X1	?	male
X2	15	?
X3	25	female



Speedup for sparse data

Useful for categorical encoding and other cases (e.g. Bag of words)

$$\begin{pmatrix}
1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\
0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\
0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\
0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\
2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\
0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\
0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0
\end{pmatrix}$$



Hyperparameter tuning

Parameter tuning is a dark art in machine learning, the optimal parameters of a model can depend on many scenarios.

—XGBoost Documentation

XGBoost hyperparameters

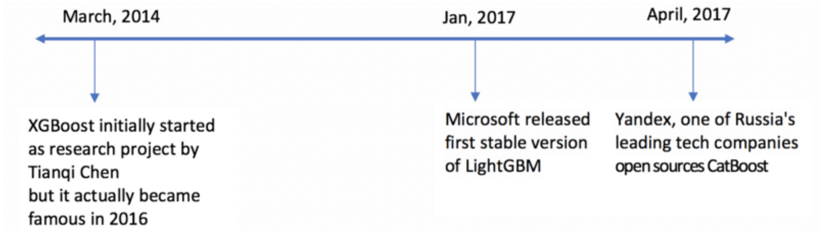
- General Parameters: Guide the overall functioning
  - booster [default=gbtree]
  - silent [default=0]:
  - nthread [default to maximum number of threads available if not set]
- Booster Parameters: Guide the individual booster (tree/regression) at each step
  - eta
  - min_child_weight
  - max_depth
  - max_leaf_nodes
  - gamma
  - max_delta_step
  - subsample
  - colsample_bytree
  - colsample_bylevel
  - lambda
  - alpha
  - scale_pos_weight
- Learning Task Parameters: Guide the optimization performed
  - objective
  - eval_metric
  - seed

A complete list of parameters are in

`https://xgboost.readthedocs.io/en/latest/parameter.html`

A plausible "guide" can be found in

`https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/`


March, 2014
Jan, 2017
April, 2017
XGBoost initially started as research project by Tianqi Chen but it actually became famous in 2016
Microsoft released first stable version of LightGBM
Yandex, one of Russia's leading tech companies open sources CatBoost

LightGBM

Motivation

- GBDT requires all data in each boosting step
- This is too expensive for large data sets

LightGBM

Motivation

- GBDT requires all data in each boosting step
- This is too expensive for large data sets
- Previous workarounds:
  - uniform downsampling
  - histograms (e.g. in XGBoost)

LightGBM

Motivation

- GBDT requires all data in each boosting step
- This is too expensive for large data sets
- Previous workarounds:
  - uniform downsampling
  - histograms (e.g. in XGBoost)
- How to create a better subset (with less info loss)?
- A better downsampling requires sample weights
  - Example: learn more on the misclassified

LightGBM

Motivation

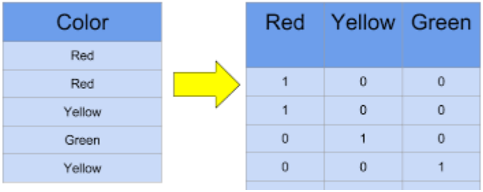
- GBDT requires all data in each boosting step
- This is too expensive for large data sets
- Previous workarounds:
  - uniform downsampling
  - histograms (e.g. in XGBoost)
- How to create a better subset (with less info loss)?
- A better downsampling requires sample weights
  - Example: learn more on the misclassified
- AdaBoost has sample weights, but GBDT has no.
- In LightGBM, leaf with higher gradient/error is used for growing further

CatBoost

- CatBoost mainly overcomes target leakage in
  - category features
  - boosting
- target leakage here means wrongly using y_i's
- target leakage theoretically leads to wrong fitting (conditional distribution shift)

Categorical features

- Conventionally use one-hot encoding



Color
Red
Red
Yellow
Green
Yellow

Red	Yellow	Green
1	0	0
1	0	0
0	1	0
0	0	1

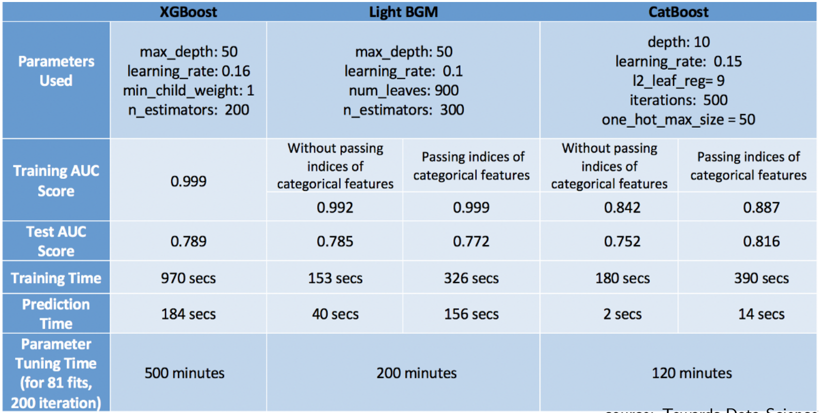
- apply one-hot encoding before using XGBoost
- but problematic if #category is large

Target statistics (TS)

- A target statistic is a number $\hat{x}_k^i \approx \mathbb{E}(y|x^i = x_k^i)$, where k indexes sample and i indexes a category feature.
- substitutes a category value with target statistic
- TS is much more efficient if #category is large
- CatBoost uses Ordered TS:
    - With a random permutation, TS is calculated using other y_i's before the data point
    - Proven to be no target leakage

An example notebook of comparing XGBoost, LightGBM and CatBoost

Results

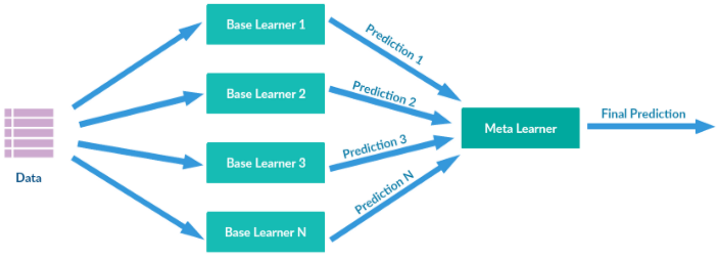


	XGBoost	Light BGM		CatBoost	
Parameters Used	max_depth: 50 learning_rate: 0.16 min_child_weight: 1 n_estimators: 200	max_depth: 50 learning_rate: 0.1 num_leaves: 900 n_estimators: 300		depth: 10 learning_rate: 0.15 l2_leaf_reg= 9 iterations: 500 one_hot_max_size = 50	
Training AUC Score	0.999	Without passing indices of categorical features	Passing indices of categorical features	Without passing indices of categorical features	Passing indices of categorical features
		0.992	0.999	0.842	0.887
Test AUC Score	0.789	0.785	0.772	0.752	0.816
Training Time	970 secs	153 secs	326 secs	180 secs	390 secs
Prediction Time	184 secs	40 secs	156 secs	2 secs	14 secs
Parameter Tuning Time (for 81 fits, 200 iteration)	500 minutes	200 minutes		120 minutes	

source: Towards Data Science

Stacking: chase the accuracy extreme

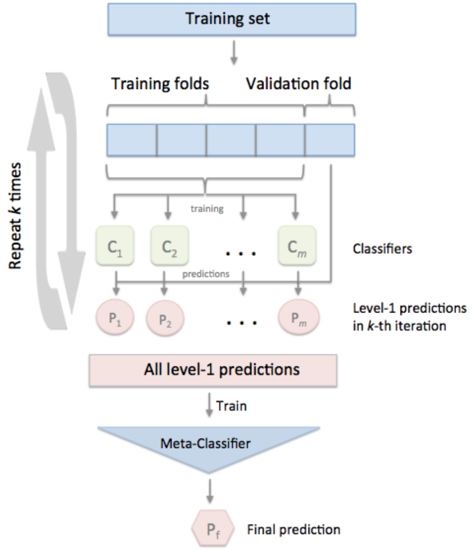
- Use $[f_1(x), \ldots, f_B(x)]$ as features to fit a meta learner



Choice of stacking bases

- Tier 1: **state-of-the-art**
  - XGBoost, LightGBM, CatBoost
- Tier 2: **good cost-performance ratio**
  - Random Forest, AdaBoost, K-Nearest-Neighbors
- Tier 3: **sometimes helps**
  - Deep Neural Networks, Logistic Regression, Support Vector Machines, Gaussian Process (for regression)
- Tier 4: **if you have extra time**
  - Fisher's linear discriminant, Naive Bayes, Learning Vector Quantization, etc.

Stacking using cross-validation



A stacking demo with grid search