I
Cross Validation

Data Science
Academy

Cross-validation

- Uses multiple train-test splits, not just a single one
- Each split used to train & evaluate a separate model
- Why is this better?
  - The accuracy score of a supervised learning method can vary, depending on which samples happen to end up in the training set.
  - Using multiple train-test splits gives more stable and reliable estimates for how the classifier is likely to perform on average.
  - Results are averaged over multiple different training sets instead of relying on a single model trained on particular training set.

Random_state	Test set accuracy
0	1.00
1	0.93
5	0.93
7	0.67
10	0.87

Accuracy of k-NN classifier (k=5) on fruit data set for different random_state values in train_test_split.

Original dataset		Model 1	Model 2	Model 3	Model 4	Model 5
	Fold 1	Test	Train	Train	Train	Train
	Fold 2	Train	Test	Train	Train	Train
	Fold 3	Train	Train	Test	Train	Train
	Fold 4	Train	Train	Train	Test	Train
	Fold 5	Train	Train	Train	Train	Test

Stratified Cross-validation

(Folds and dataset shortened for illustrations purposes.)

fruit_label	fruit_name
1	Apple
1	Apple
1	Apple
1	Apple
1	Apple
2	Mandarin
…	…
3	Orange
…	…
4	Lemon
4	Lemon
4	Lemon
4	Lemon
4	Lemon

Example has 20 data samples = 4 classes with 5 samples each.

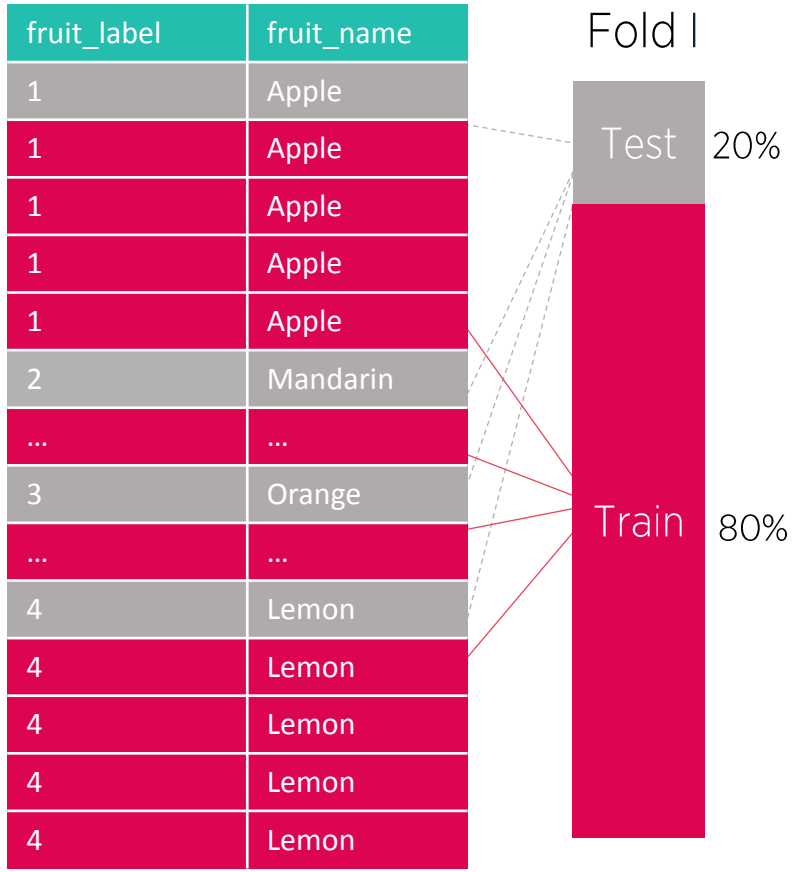
5-fold CV: 5 folds of 4 samples each.

Fold I uses the first 20% of the dataset as the test set, which only contains samples from class 1.

Classes 2, 3, 4 are missing entirely from test set and so will be missing from the evaluation.

Stratified folds each contain a proportion of classes that matches the overall dataset.

Now, all classes will be fairly represented in the test set.



II
Ridge and Lasso Regularization

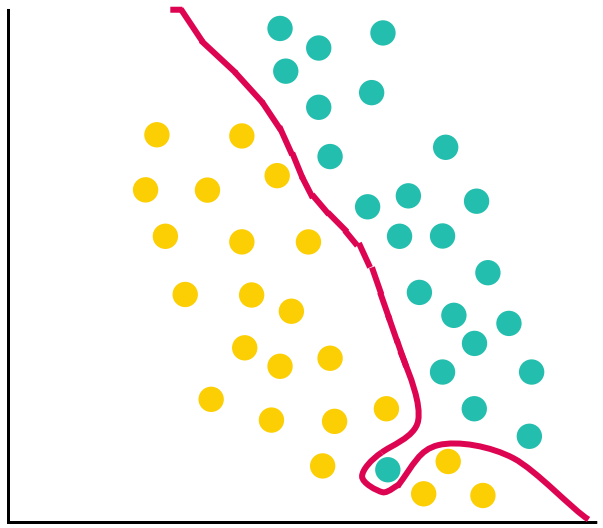
Data Science Academy

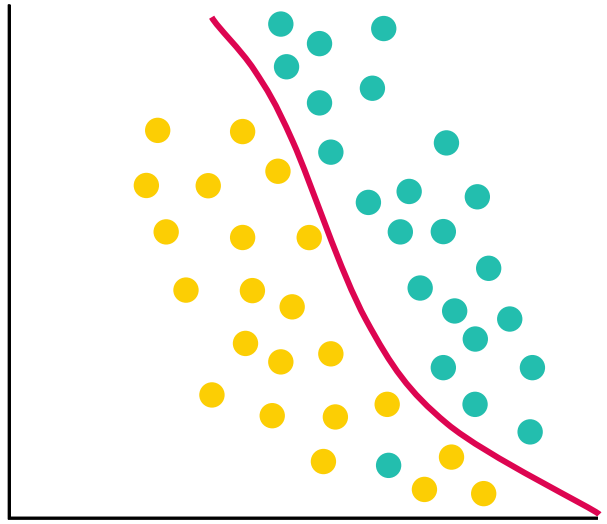
What makes a Good Model?

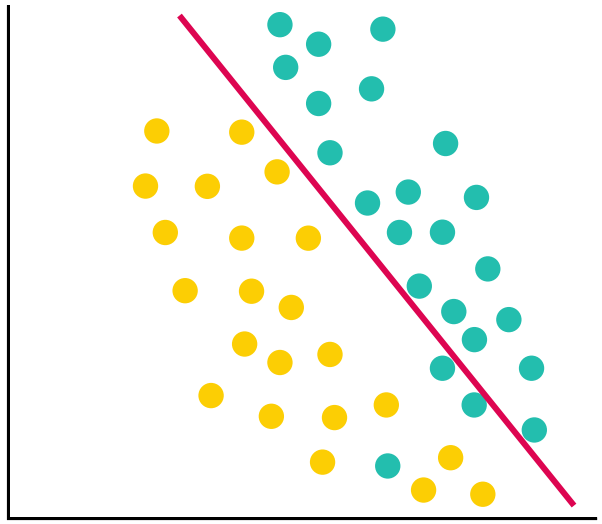
- A good model is accurate
- Generalizes well
- Does not overfit
- But what do these things mean?

Let’s look at Each

- Overfitting
- Ideal or Balanced
- Underfitting

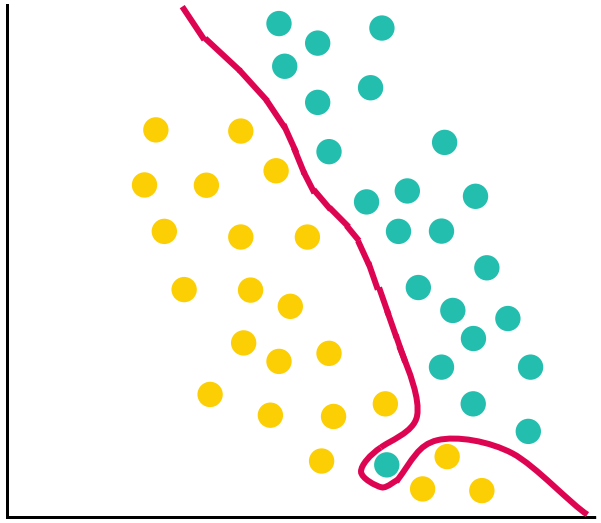






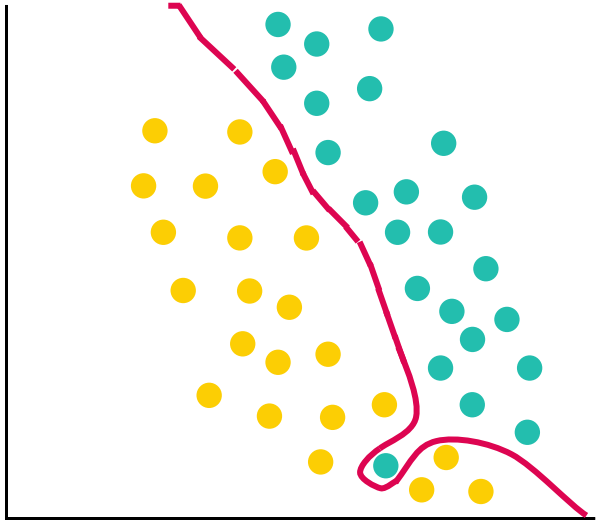
Overfitting and Underfitting

- Overfitting occurs when a statistical model or machine learning algorithm captures the noise of the data. Overfitting occurs if the model or algorithm show low bias but high variance.
- Underfitting occurs when a statistical model or machine learning algorithm cannot capture the underlying trend of the data.



Overfitting

- Overfitting leads to poor models and is one of the most common problems faced developing in AI/Machine Learning/Neural Nets.
- Overfitting occurs when our Model fits near perfectly to our training data, as we saw in the previous slide with Model A. However, fitting too closely to training data isn't always a good thing.



- What happens if we try to classify a brand new point that occurs at the position shown on the left? (who's true color is green)
- It will be misclassified because our model has overfit the test data
- Models don't need to be complex to be good

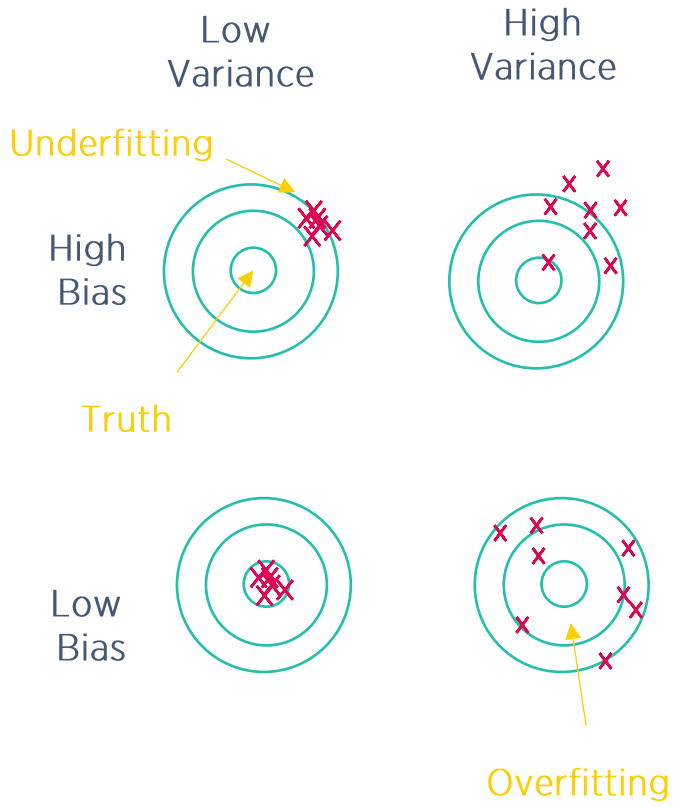
Overfitting – Bias and Variance

- Bias Error – Bias is the difference between the average prediction and the true value we're attempting the predict. Bias error results from simplifying assumptions made by a model to make the target function easier to learn. Common low bias models are Decision Trees, KNN and SVMs. Parametric models like Linear Regression and Logistic Regression have high-bias (i.e. more assumptions). They're often fast to train, but less flexible.

- Variance Error – Variance error originates from how much would your target model change if different training data, however, the underlying model should be generally the same. High variance models (Decision Trees, KNN and SVMs) are very sensitive to this whereas low variance models (Linear Regression and Logistic Regression) are not.

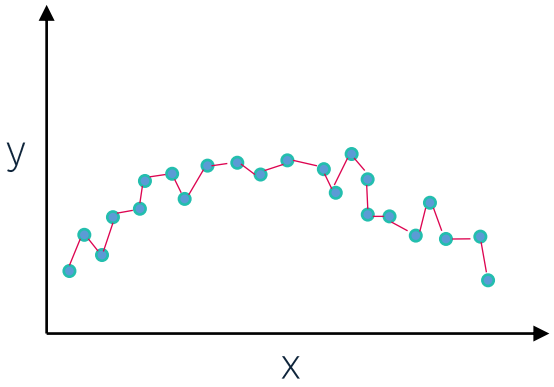
- Parametric or linear models have high bias and low variance.

- Non-parametric or non-linear have low bias and high variance.

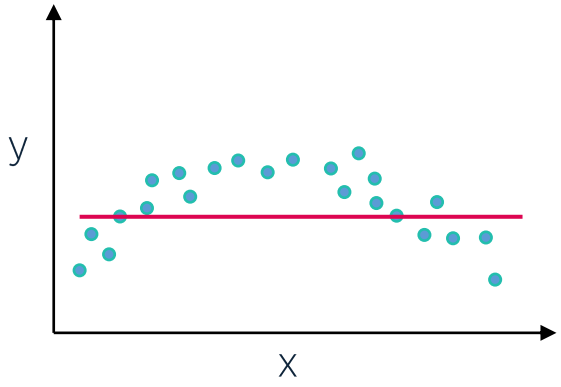


The Bias and Variance Tradeoff

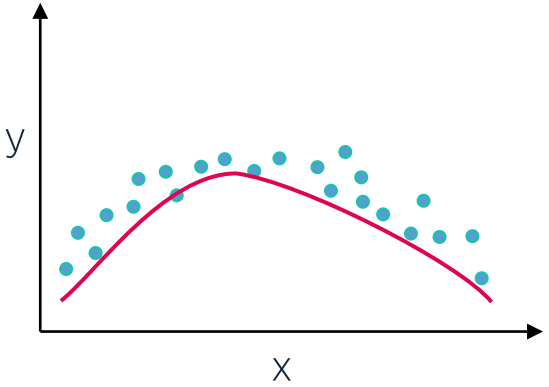
- Trade-Off – We want both low bias and low variance, however as we increase bias we decrease variance and vice versa. But how does this happen?
- In linear regression, if we increase the degrees in our polynomial, we are lowering the bias but increasing the variance, conversely if we reduce the complexity we increase bias but reduce variance.
- In K-nearest neighbors, increasing the number of clusters (k) increases bias but reduces variance.



Overfitting



Underfitting



Good balance

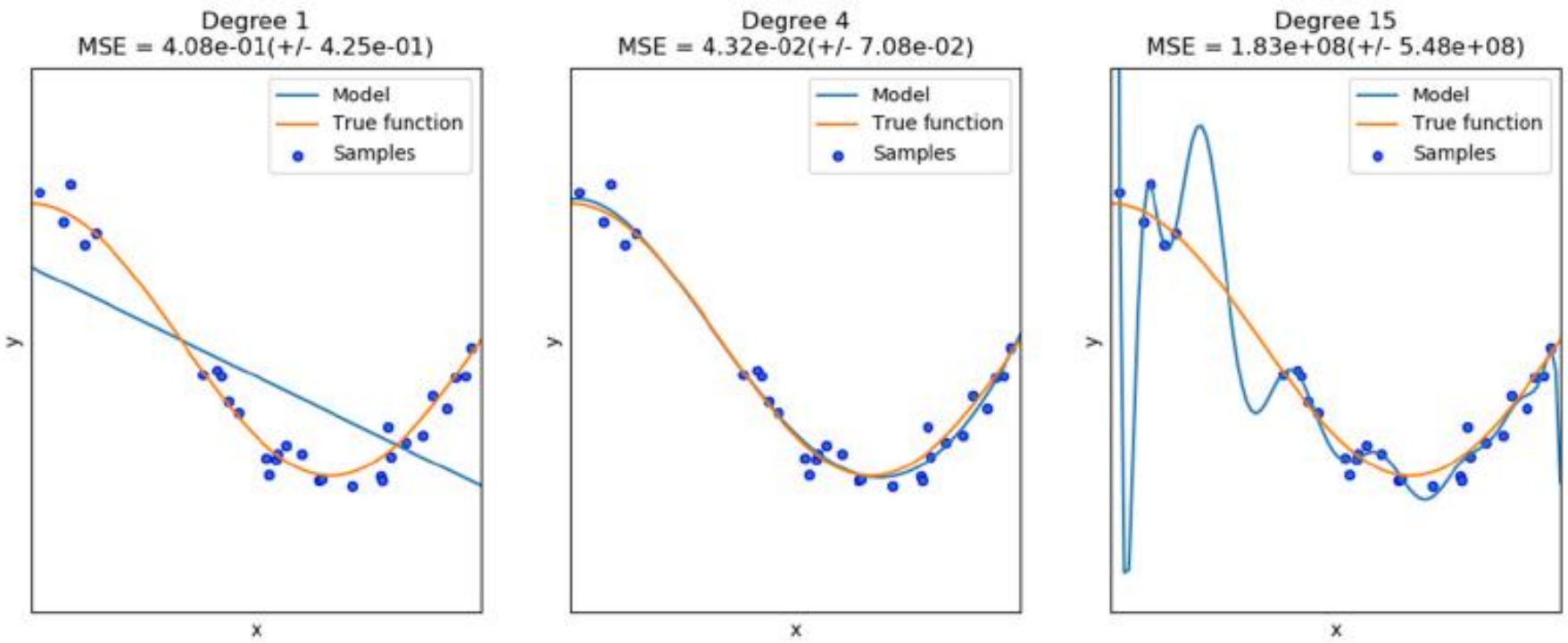
How do we know if we've Overfit?

- Test on your model on ….. Test Data!
- In all Machine Learning it is extremely important we hold back a portion of our data (10-30%) as pure untouched test data.

- Untouched meaning that this data NEVER seen by the training algorithm. It is used purely to test the performance of our model to access it's accuracy in classifying new never before seen data.
- Many times when Overfitting we can achieve high accuracy 95%+ on our training data, but then get abysmal (~70%) results on the test data. That is a perfect example of Overfitting.

How do we avoid overfitting?

- Rule of thumb is to reduce the complexity of your model



How do we avoid overfitting?

- Why do less complex models overfit less?
- Overly complex (i.e. less degrees in the polynomial or in Deep Learning, shallower networks) can sometimes find features or interpret noise to be important in data, due to their abilities to memorize more features (called memorization capacity)

Another method is to use Regularization!

- It is better practice to regularize than reduce our model complexity.

What is Regularization?

It is a method of making our model more general to our dataset.

Types of Regularization

- In most Machine Learning Algorithms we can use :
  - L1 & L2 Regularization.
  - Cross Validation
- In Deep Learning we can use:
  - Early Stopping
  - Drop Out
  - Dataset Augmentation

Ridge and Lasso Regularization

- L1 & L2 regularization are techniques we use to penalize large weights. Large weights or gradients manifest as abrupt changes in our model's decision boundary. By penalizing, we a really making them smaller.
- **λ** controls the degree of penalty we apply.
- The difference between them is that L1 brings the weights of the unimportant features to 0, thus acting as feature selection algorithm (known as sparse models or models with reduced parameter)

Ridge

$$\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda * Slope^2$$

- Shrinks some of the coefficient to near zero.
- Can not be used for feature selection.
- Makes correlated features coefficients smaller.
- Makes sense when all features are important.

Lasso

$$\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda * |Slope|$$

- Shrinks some of the coefficient to zero.
- Performs Embedded feature Selection.
- Makes some of the correlated features irrelevant.
- Can be used when some features can be eliminated.

Ridge regression

- We can apply ridge regression using codes below:

In [ ]:

```
from sklearn.linear_model import Ridge
rr = Ridge (alpha=0.01)
rr.fit (X_train, y_train)
pred_train_rr= rr.predict (X_train)
print(np.sqrt(mean_squared_error(y_train,pred_train_rr
)))
print(r2_score(y_train, pred_train_rr)) pred_test_rr=
rr.predict(X_test)
print(np.sqrt(mean_squared_error(y_test,pred_test_rr))
)
print(r2_score(y_test, pred_test_rr))
```

Output

```
975.8314265299163
0.8672577596814723
1017.31106a62731054
0.8402957317988335
```

- A low alpha value can lead to over-fitting, whereas a high alpha value can lead to under-fitting.

Elastic Net

- Elastic Net penalize the size of the regression coefficients based on both their l^1 norm and their l^2 norm:

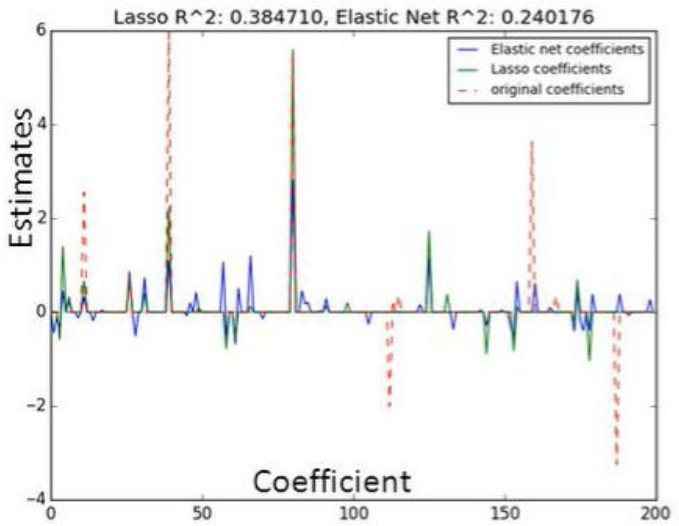
$$\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda_1 * Slope^2 + \lambda_2 * |Slope|$$

- The l^1 norm penalty generates a sparse model.
- The l^2 norm penalty:
  - Removes the limitation on the number of selected variables
  - Encourages grouping effect
  - Stabilizes the l^2 regularization path.

Elastic Net Application

```
# ElasticNet
from sklearn.linear_model import ElasticNet
alpha = 0.1
enet = ElasticNet(alpha=alpha, l1_ratio=0.7)
#Run model
#Make predictions
y_pred_enet = enet.fit(X_train, y_train).predict(X_test)
r2_score_enet = r2_score(y_test, y_pred_enet)
#Evaluate the predictions using R^2
```

Output



- Given the ill-posed problem (#features>># samples,) Elastic Net is ab is able to capture most of the non-zero coefficients.
- Elastic Net generates sparse estimates of the coefficients:
  most of the estimates are 0.
- In general LASSO gives larger coefficient estimates than Elastic Net.

III

Ensemble Learning

Data Science Academy

Ensemble Learning

- It is the process of running two or more related but different machine learning models and then synthesizing the results into single predictive or machine learning model
- It can have biases
- Presence of high variability
- Outright inaccuracies
- How ensemble models work
  - Producing a distribution called a simple ML model on the subset of original data.
  - Combining the distribution in one aggregated model.
  - Random Forest
  - It is the group of multiple decision trees which built on different sample data, evaluates different factors and/or weight common variables differently.

Advantages Ensemble Learning

Accuracy

less Variance

In Ensemble learning, the large variance of unstable learners is 'averaged out' across multiple learners.

less Overfitting

Diversity

Different classifiers to work on different random subsets of the null feature space or different subset of the training data.

- Imagine we have:
  - An ensemble of 5 independent classifiers.
  - Accuracy is 70% for each.
- What is the accuracy for the majority vote?

10 (.7^3)(.3^2)+5(.7^4)(.3)+(.7^5)
83.7% majority vote accuracy

How about if we have101 such classifiers
99.9% majority vote accuracy

Questions and Disadvantage

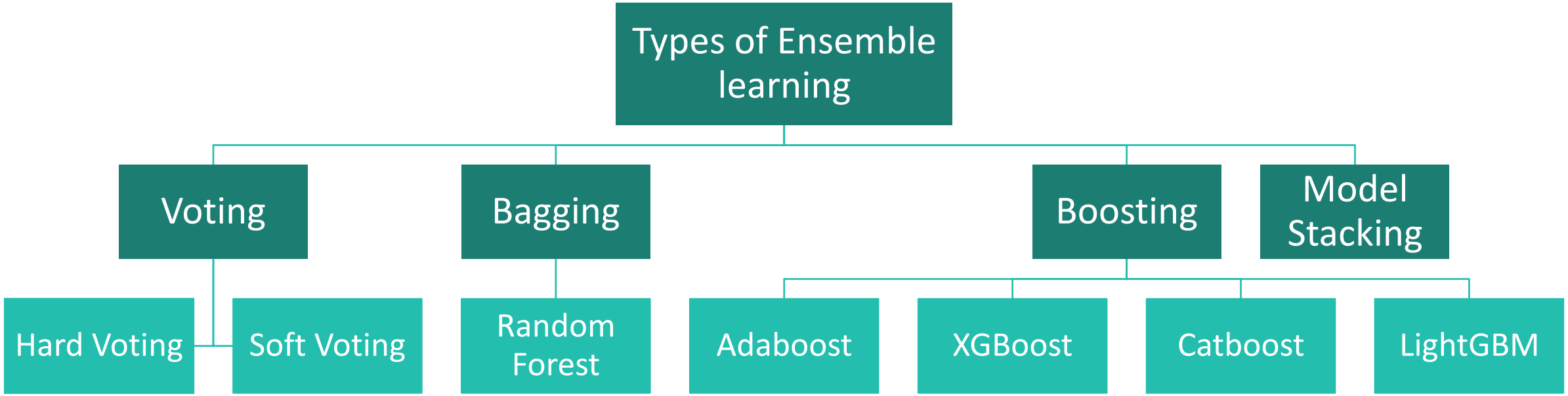
Questions

- Do we use all training data on every model
- What is M1, M2, M3 ...
- How to train all models
- How to combine output from all model

Disadvantage

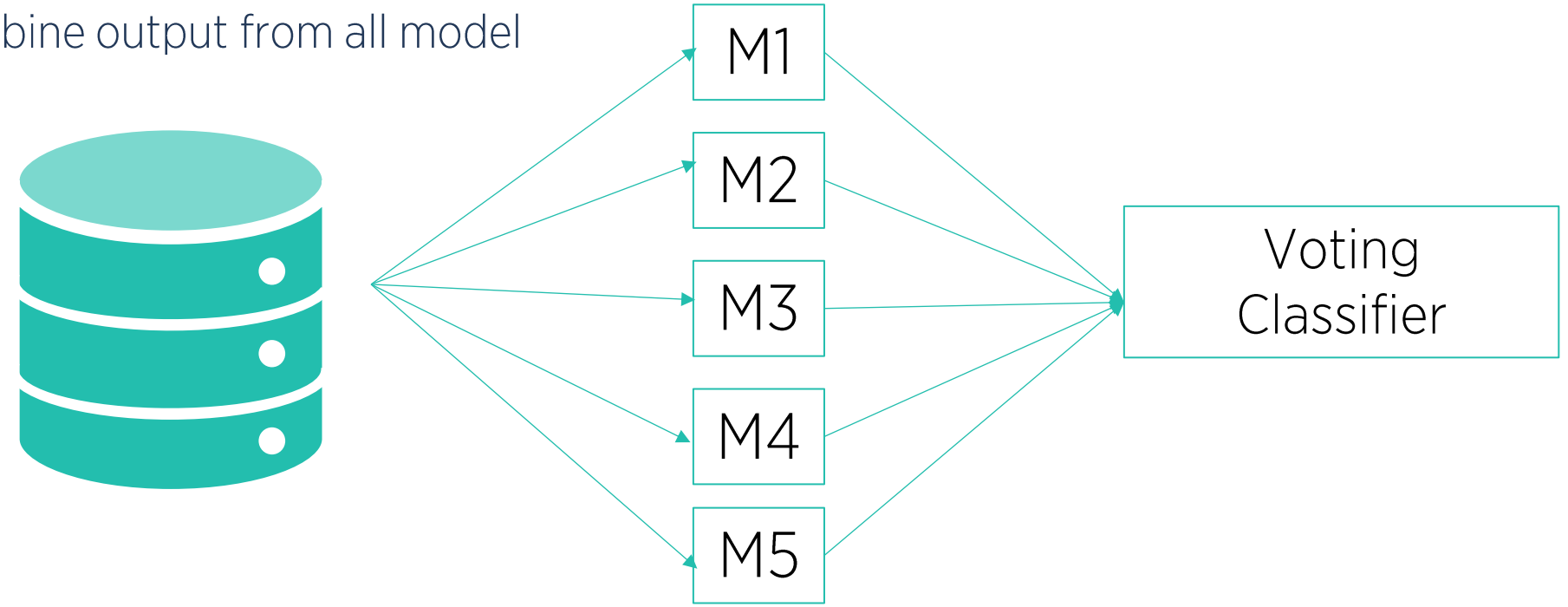
- High computational cost

Types of Ensemble learning



Voting Classifier

- Do we use all training data on every model
- What is M1, M2, M3 ...
- How to train all models
- How to combine output from all model

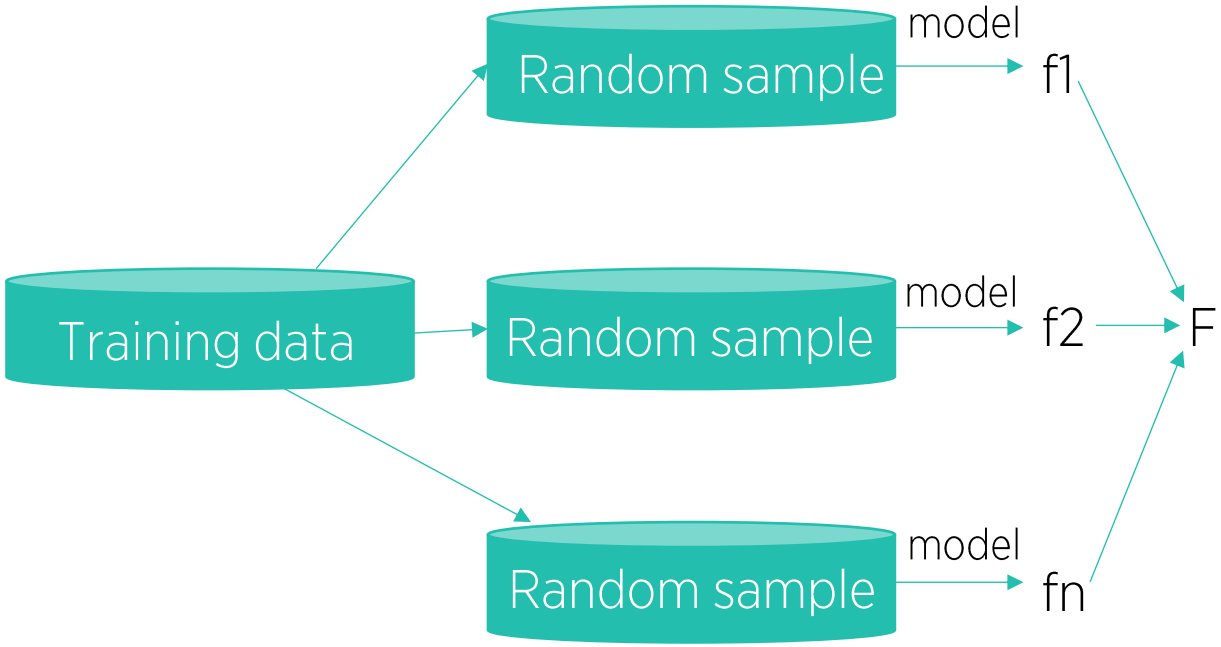


Bagging & Boosting

Bagging	Boosting
Both the ensemble methods get N learners from 1 leaner. But . .	
. . follows parallel ensemble techniques, i.e. base learners are formed independently.	. . follows Sequential ensemble technique, i.e. base leaners are dependent on the previous weak base learner.
Random sampling with replacement.	Random sampling with replacement over weighted data.
Both gives out the final prediction by taking average of N learners. But . .	
. . equal weights is given to all model. (equally weighted average)	more weight is given to the model with better performance. (weighted average)
Both are good at providing high model scalability. But . .	
. . it reduces variance and solves the problem of overfitting.	. . It reduces the bias but is more prone to overfitting. Overfitting can be avoided by tuning the parameters.

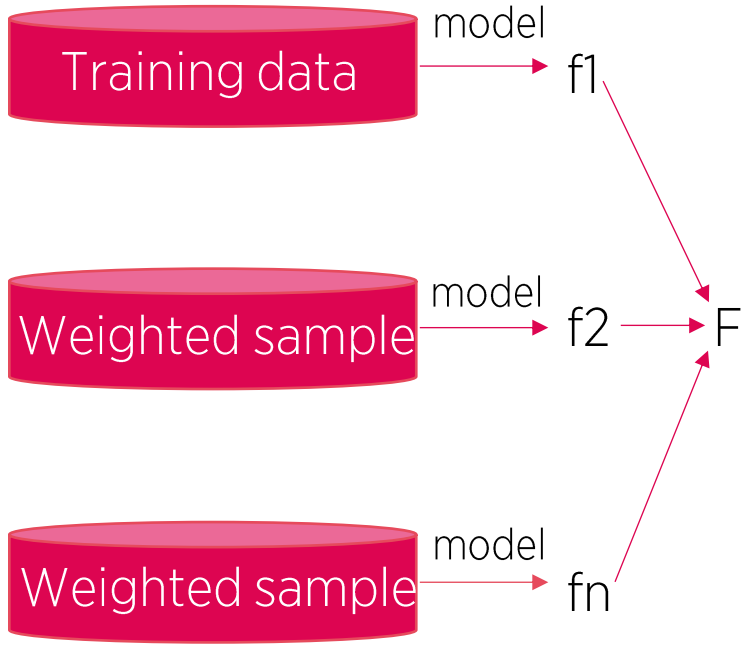
Bagging VS Boosting

- Bagging



Resampling
Uniform distribution
Parallel Style

- Boosting



Reweighting
Non-uniform distribution
Sequential Style

Voting Classifier

- Do we use all training data on every model
  - Yes
- What is M1, M2, M3 ...
  - As different as Possible:
- How to train all models
  - Train in Parallel
- How to combine output from all model with voting classifier

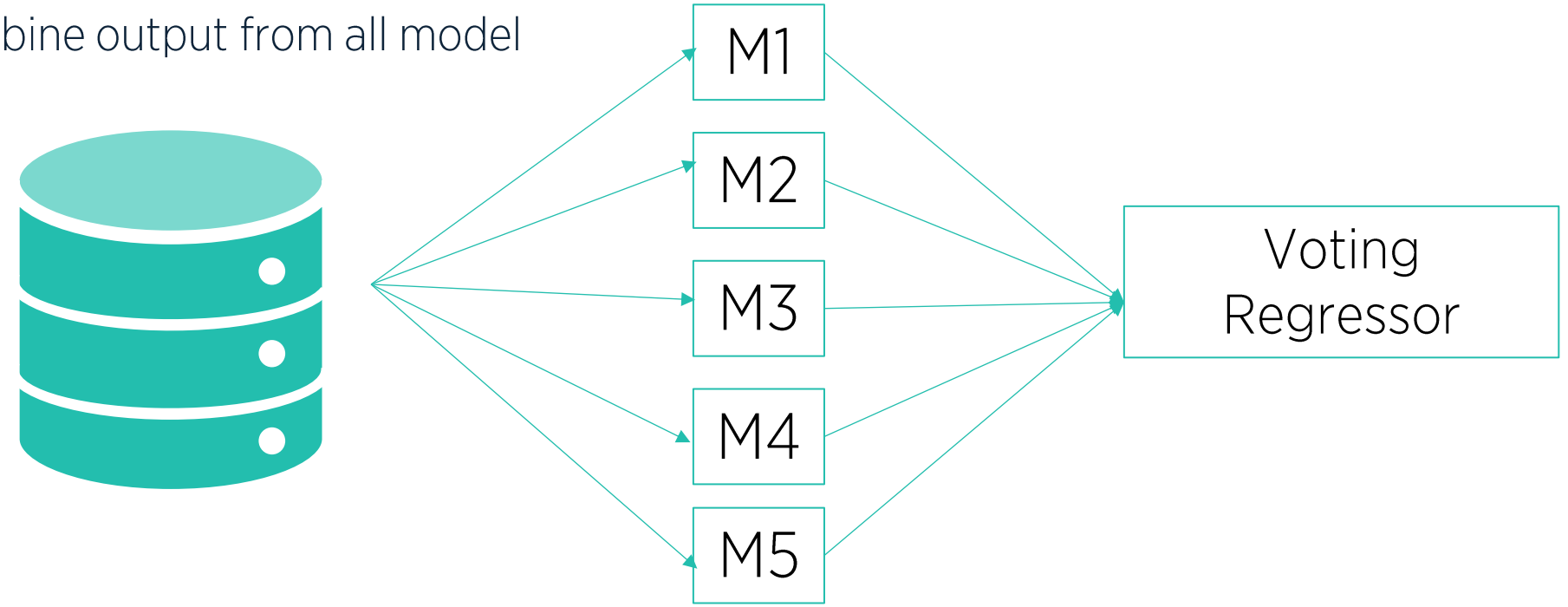
Voting Classifier

- Classification
  - Hard Voting
  - Soft Voting
- Regression
  - Averaging

	Hard Voting		Soft Voting	
M1	1	1	0.1, 0.9	0.2, 0.8
M2	0	0	0.95, 0.05	0.6, 0.4
M3	1	0	0.3, 0.7	0.75, 0.25
M4	1	1	0.2, 0.8	0.4, 0.6
M5	1	0	0.45, 0.55	0.62, 0.38
	1	0	0.4, 0.6 → 1	0.514, 0.486 → 0

Voting Regressor

- Do we use all training data on every model
- What is M1, M2, M3 ...
- How to train all models
- How to combine output from all model



Averaging

- Classification
  - Hard Voting
  - Soft Voting
- Regression
  - Averaging

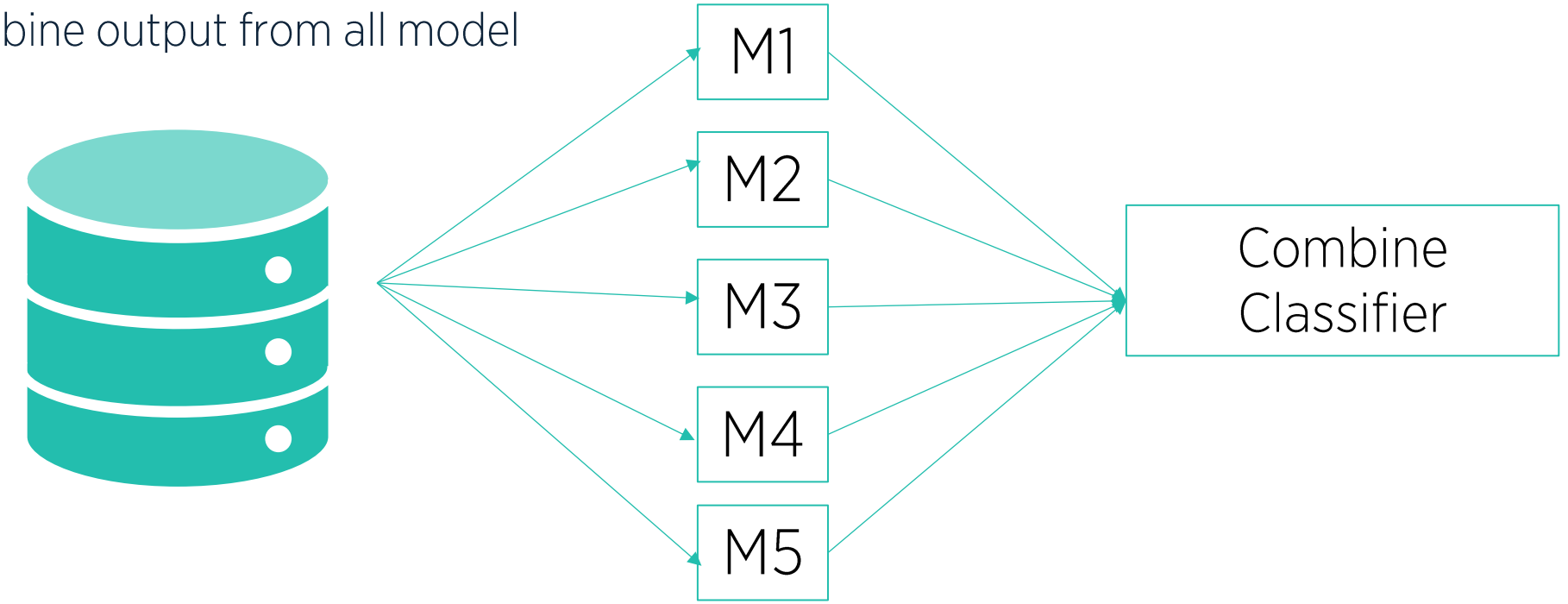
		Prediction
M1	0.5	$200
M2	0.2	$349
M3	0.9	$222
M4	0.7	$250
M5	0.65	$270

$$Average = \frac{200+349+222+250+270}{5} = 258.2$$

$$Weighted\ Average = \frac{0.5*200+0.2*349+0.9*222+0.7*250+0.65*270}{5} = 144.02$$

Bagging

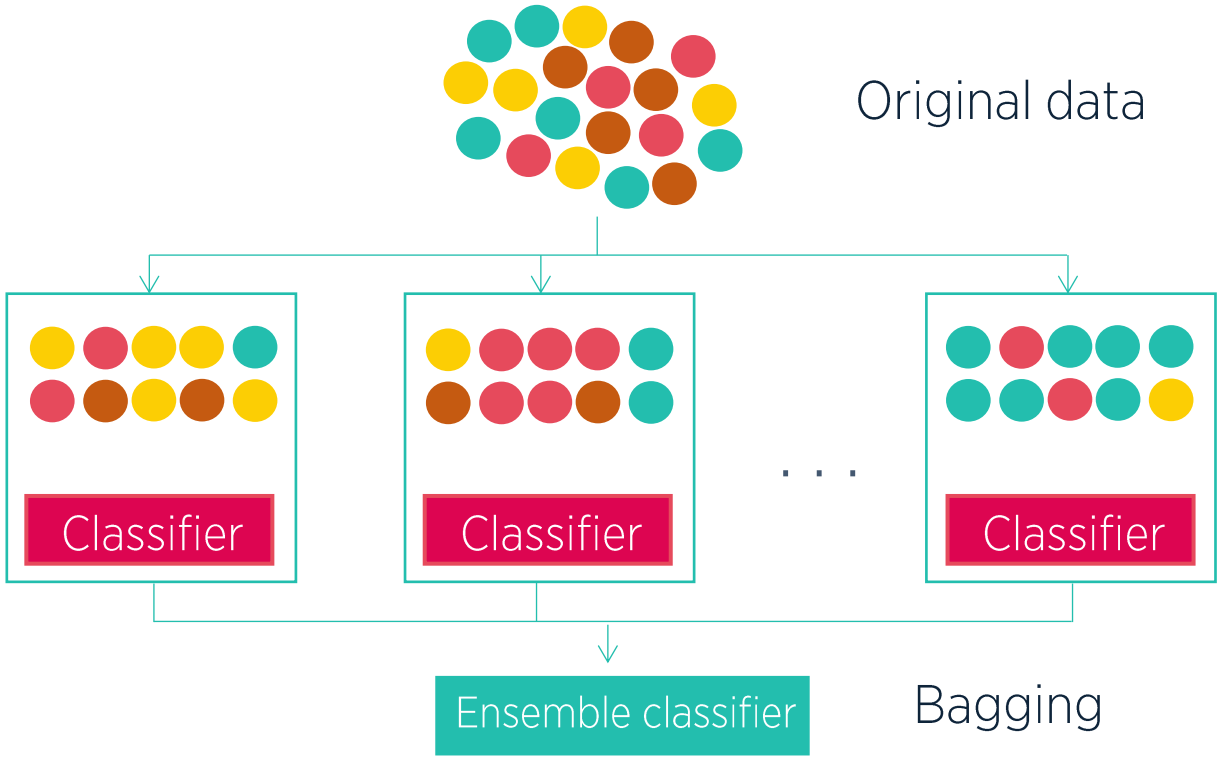
- Do we use all training data on every model
- What is M1, M2, M3 ...
- How to train all models
- How to combine output from all model



Bagging

- Bagging stands for Bootstrapped Aggregation.
- Bagging is the way to decrease variance of your prediction by generating additional training data from the original data with different combination and replications.
- Bagging Algorithm
    1. Samples (with replacement) are repeatedly taken from the data set, so that each record has an equal probability of being selected, and each sample is of the same size as the original training data set. These are bootstrapped samples.
    2. Train the model and record the predictions for each sample.
    3. Bagging ensembles will be defined as the class with most votes or the average of prediction made.

Bagging learning procedure



Bootstrapping

Aggregating

Bagging

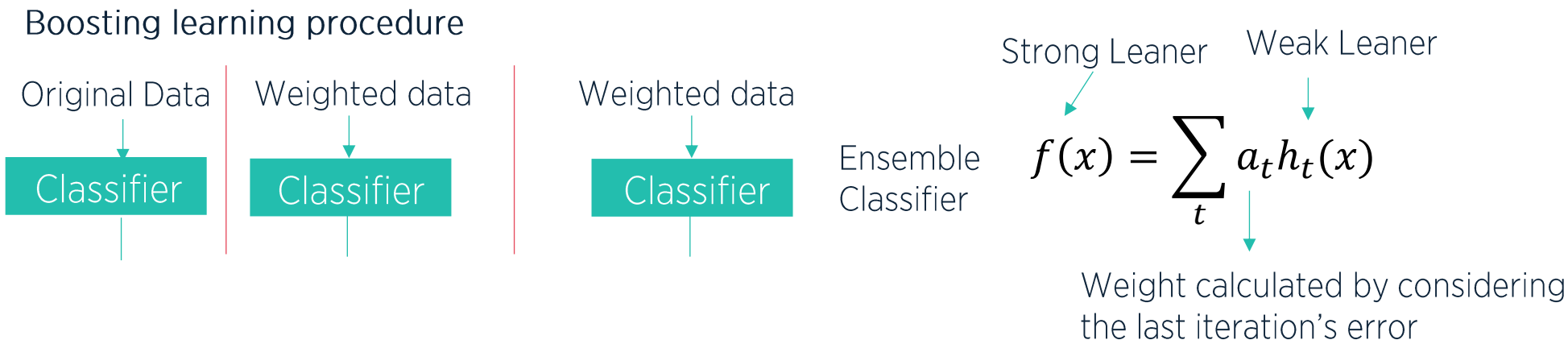
$$f(x) = \frac{1}{B}\sum_{b=1}^{B} f_b(x)$$

Generates Bootstrapping Sets

Weak Leaners

Boosting

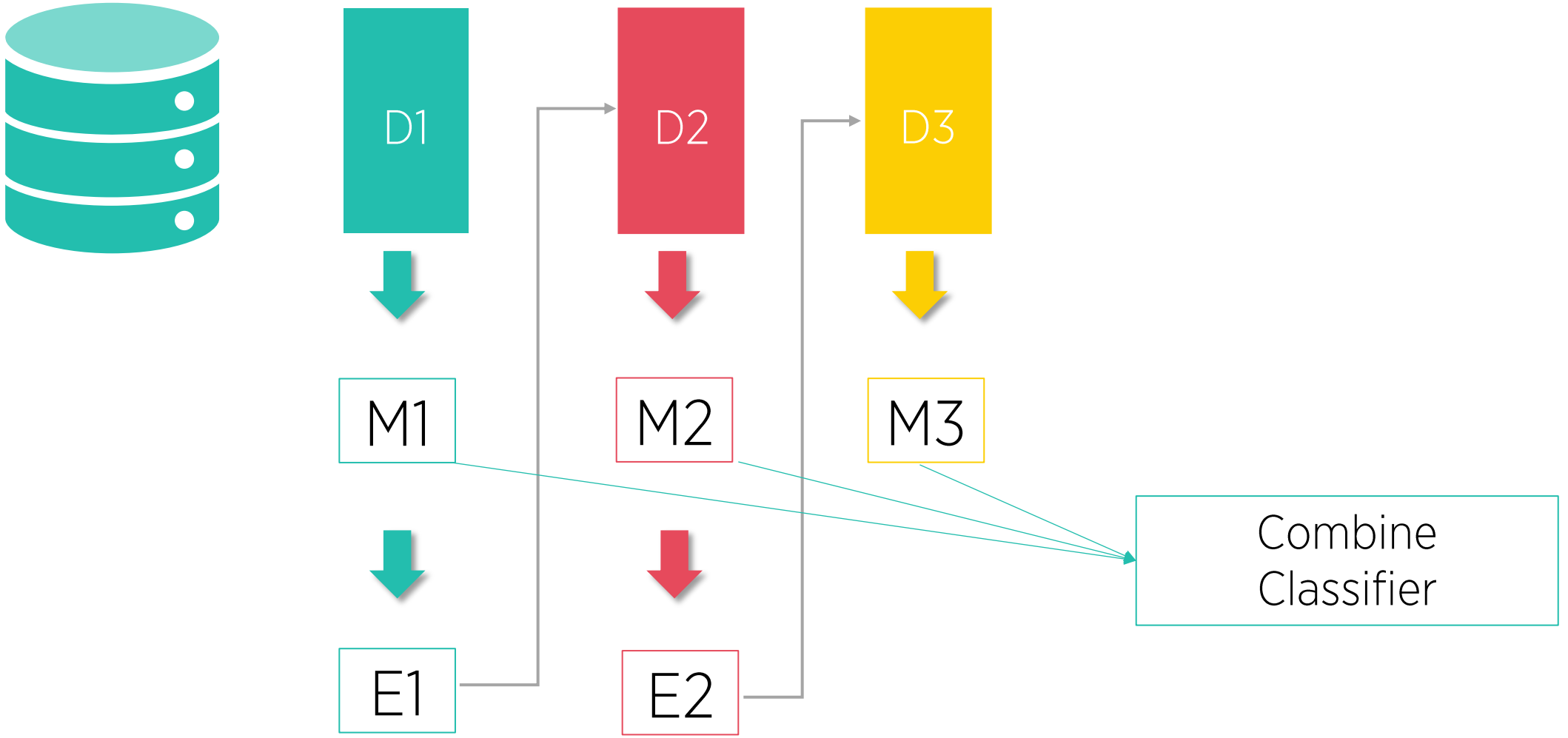
- Boosting is a form of *sequential learning* technique. The algorithm works by training a model with the entire training set, and subsequent models are constructed by fitting the residual error values of the initial model.
- In this way, Boosting attempts to give higher weight to those observations that were poorly estimated by the previous model.
- Once the sequence of the models are created the predictions made by models are weighted by their accuracy scores and the results are combined to create a final estimation.
- Models that are typically used in Boosting technique are XGBoost (Extreme Gradient Boosting), GBM (Gradient Boosting Machine), ADABoost (Adaptive Boosting), etc.



Boosting

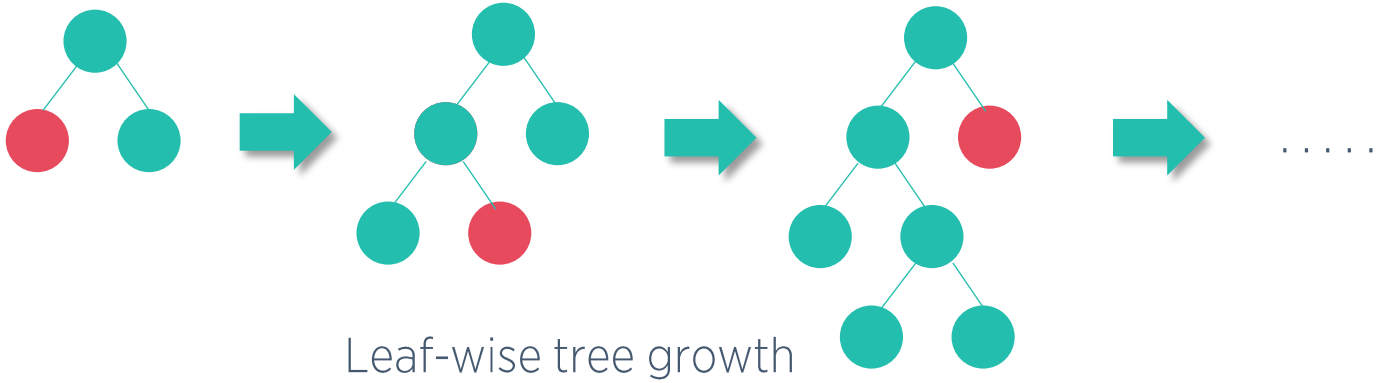
- Instead of assigning equal weighting to models, boosting assigns weights to classifiers, and derives its ultimate result based on weighted voting
- Operates via weighted voting
- Boosting algorithm proceeds iteratively, new models are influenced by previous ones
- New models become experts for instances classified incorrectly by earlier models
- Can be used without weights by using resampling, with probability determined by weights
- Works well if classifiers are not too complex
- Also works well with weak learners like decision trees

Boosting



LightGBM

- Light GBM is a gradient boosting framework that uses tree based learning algorithm.
- Light GBM grows tree vertically while other algorithm grows trees horizontally meaning that Light GBM grows tree leaf-wise while other algorithm grows level-wise. It will choose the leaf with max delta loss to grow. When growing the same leaf, Leaf-wise algorithm can reduce more loss than a level-wise algorithm.
- Light GBM was developed by Microsoft machine learning group.
- Light GBM has the capability for distributed computing and has GPU support.
- Light GBM is built for model training speed and churning high performing model.
- Light GBM is capable on of big data with its in-built parallel computing capacity.
- Light GBM is optimized for low memory usage.



Leaf-wise tree growth

LightGBM

We can install and use LightGBM using code below.

In [ ]:
```
pip install lightgbm
or
conda install lightgbm
```

In [ ]:
```
from lightgbm import LGBMClassifier
model = LGBMClassifier()
model.fit(X_train, y_train)
pred = model.predict(X_test)
```

In [ ]:
```
from lightgbm import LGBMRegressor
model = LGBMRegressor()
model.fit(X_train, y_train)
pred = model.predict(X_test)
```

XGBoost

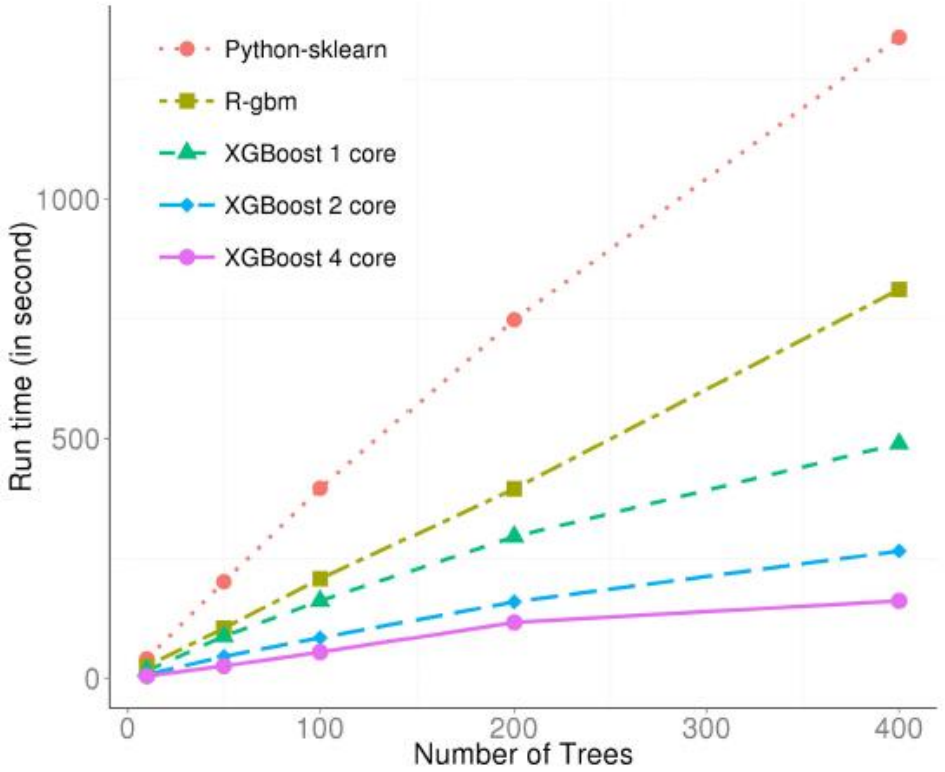
- Extreme gradient Boosting Tree.
- It is not different algorithm but works on same principal used in GBM.
- Sequential model creation with keep on reducing Error.

Feature of XGBoost

- Faster Execution Speed
- Better Performance
  - Many Machine Learning competition won by XGBoost
- Parallel Processing.
- XGBoost can handle missing value.
- It has Built-in cross validation, Regularization.
- It can do incremental training. (For Large Dataset which Can not fit in memory)

What is XGBoost

- A scalable System for Learning Tree Ensembles
  - Model improvement
    - Regularized objective for better model
  - Systems optimizations
    - Out of core computing
    - Parallelization
    - Cache optimization
    - Distributed computing
  - Algorithm improvements
    - Sparse aware algorithm
    - Weighted approximate quantile sketch
  - In short, faster tool for learning better models



Industry Use cases

- XGBoost was first released on March 27, 2014
- Used by Google, MS Azure, Tencet, Alibaba, ...
- Quotes from some users:
  - Hanjiing Su from Tencent data platform team: “We use distributed XGBoost for click through prediction in wechat shopping and lookalikes. The problems involve hundreds millions of users and thousands of features. XGBoost is cleanly designed and can be easily integrated into our production environment, reducing our cost in developments.”
  - Cnevd from autohome.com ad platform team: “Distributed XGBoost is used for click through rate prediction in our display advertising, XGBoost is highly efficient and flexible and can be easily used on our distributed platform, our ctr made a great improvement with hundred millions samples and millions of features due to this awesome XGBoost”.

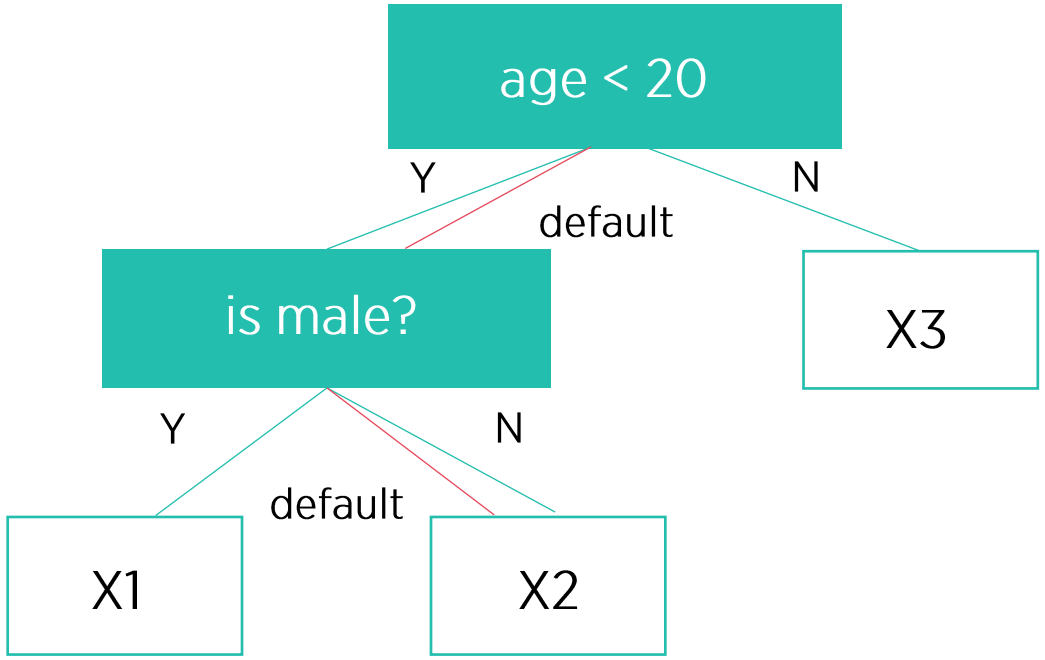
What can XGBoost do for you

- Push the limit of computation resources to solve one problem.
  - Gradient tree boosting.
- Automatic handle missing value.
- Interactive Feature analysis.
- Extendible system for more functionalities.
- Deployment on the Cloud.

Automatic Missing Value Handling

Data

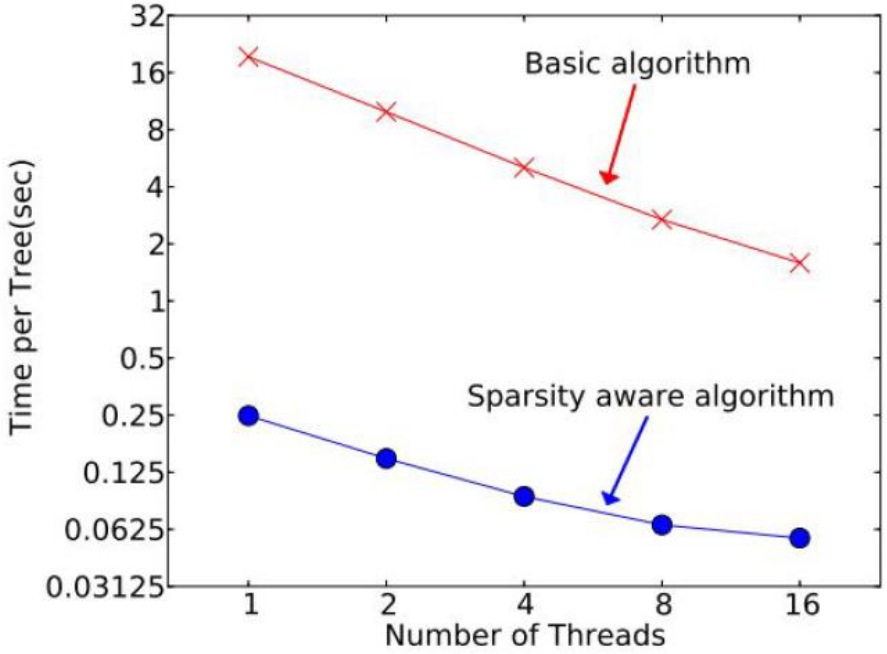
Example	Age	Gender
X1	?	male
X2	15	?
X3	25	female



- XGBoost learns the best direction for missing values.

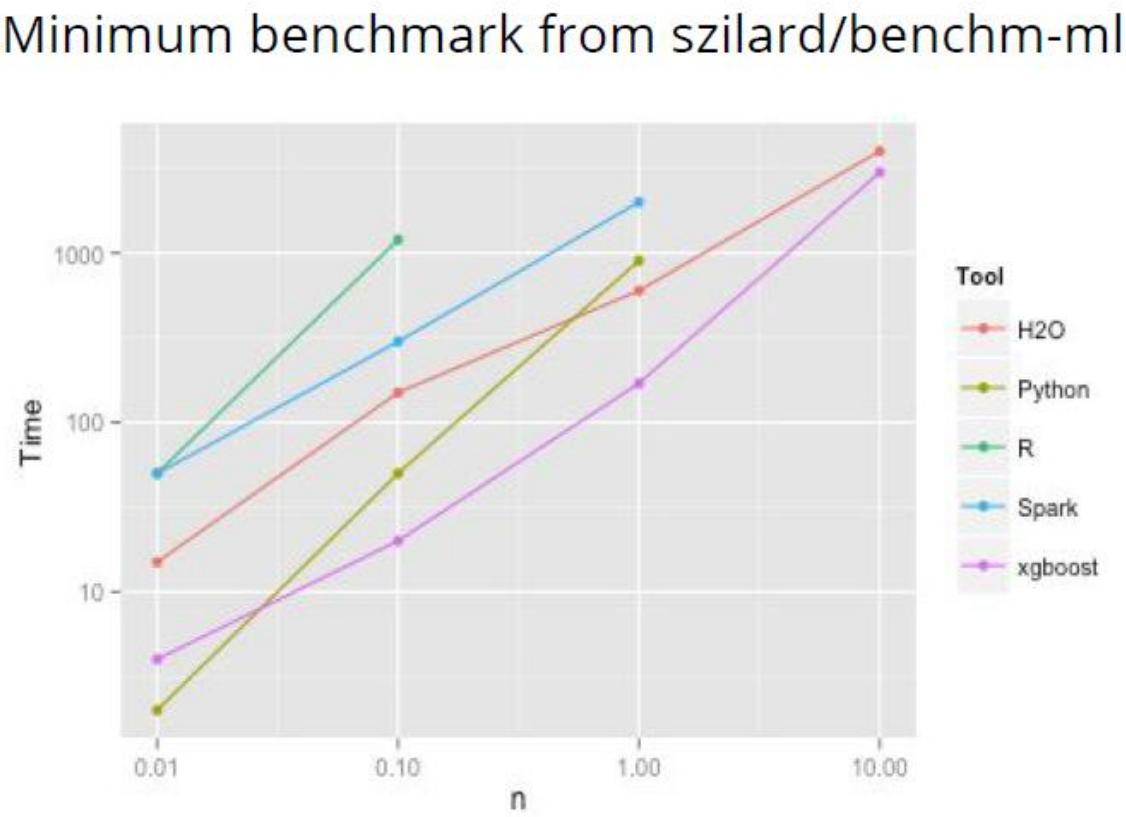
Automatic Sparse Data Optimization

- Useful for categorical encoding and other cases (e.g. Bag of words)
- User do not need to worry about large sparse matrices
- Impact of sparse aware vs basic algorithm on allstate dataset.



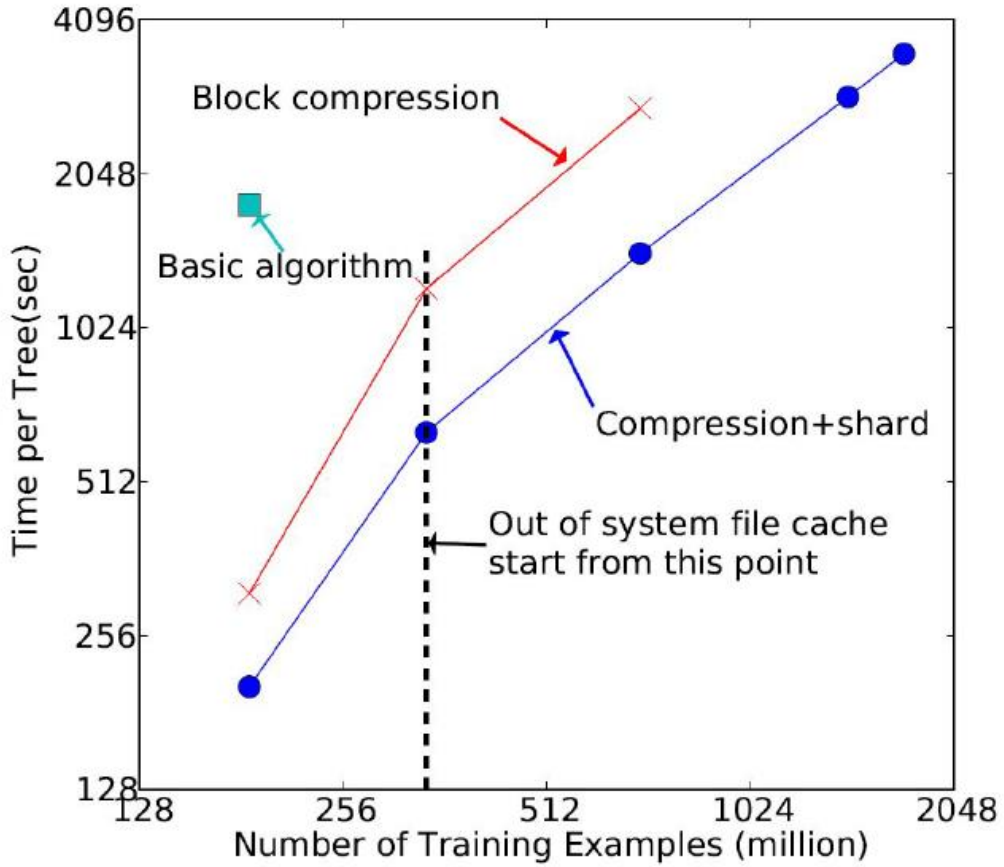
Faster Training Speed via Parallel Training

- Push limit of machine in all cases
- Law memory footprint
- Hackable native codes
  - Instead of everything in backend
  - Early-stopping
  - Checkpointing
  - Customizable objective



XGBoost with Out of Core Computation

- Impact of out of core optimizations
- On a single EC2 machine with two SSD

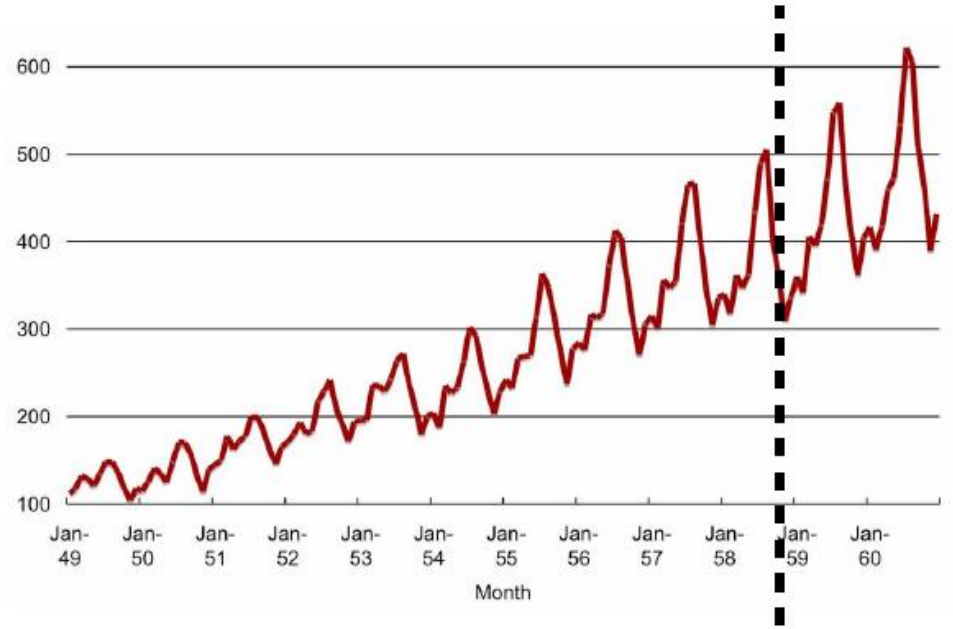


What XGBoost cannot do for you

- Feature engineering
- Hyper parameter tuning
- A lot more case . . .

Limitations of XGBoost

- Can you guess of the limitations of the XGBoost?
  - Appropriate algorithm for supervised learning.
  - The more complex the data, the more likely it will not work properly. Because, it is based on Decision Tree.
  - Inappropriate for time-series data. Why?

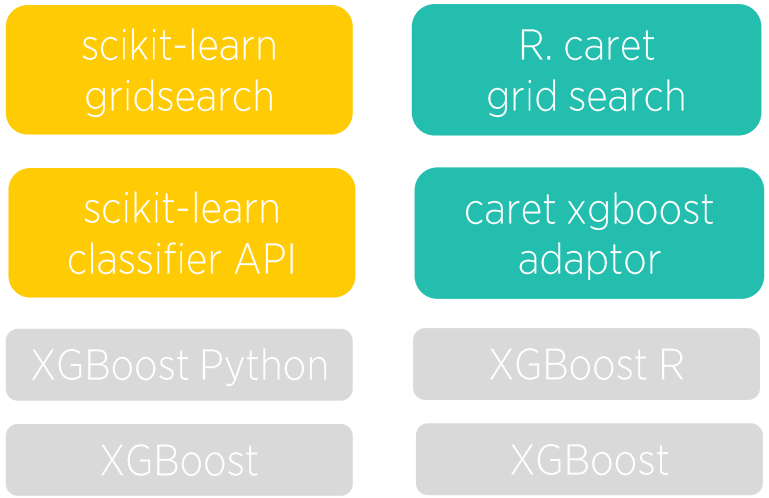


  - Not randomly splitting training and test data. But using historical data as training and future data as test data.
  - Since XGBoost is based on a decision tree, it will have difficulty in predicting.
  - Do you have any idea to handle this issue in XGBoost?

Limitations of XGBoost

- Deeply integrate with existing ecosystem
- Directly interact with native data structures

- Expose native standard APIs



XGBoost

- We can install and use XGBoost using codes below:
- Install commands

In [ ]:
```
pip install xgboost
```

- XGBoost can be used for both classification and regression problem.

In [ ]:
```
from xgboost import XGBRegressor
model = XGBRegressor()
model.fit(train_X, train_y)
test_y_hat = model.predict(test_X)
```

CatBoost

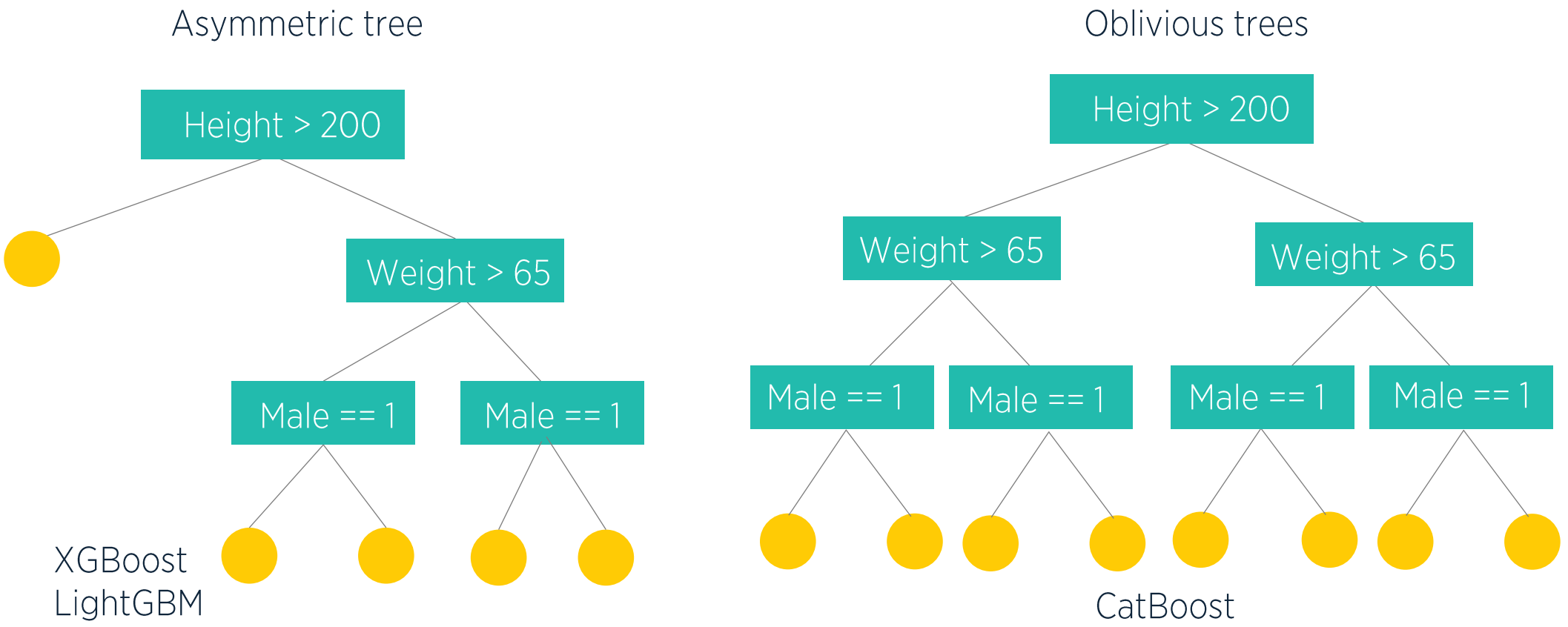
- Catboost can be used for solving problems, such as regression, classification, multi-class classification and ranking. Modes differ by the objective function, that we are trying to minimize during gradient descend. Moreover, Catboost have pre-built metrics to measure the accuracy of the model.
- Categorical + Boosting (CatBoost) is a boosting algorithm developed by Yandex researchers and engineers.
- CatBoost produces good results even without extensive hyper-parameter tuning.
- CatBoost has the inbuilt capacity to handle categorical features even non-numerical ones.
- CatBoost offers improved performance as it reduces overfitting when building model.
- CatBoost is fast and scalable and provides GPU support.

CatBoost advantage

- Catboost introduces the following algorithmic advances:
  1. An innovative algorithm for processing categorical features. No need to preprocess features on your own — it's performed out of the box. For data with categorical features the accuracy would be better compare to other algorithm.
  2. The implementation of ordered boosting, a permutation-driven alternative to the classic bosting algorithm. On small datasets, the GB is quickly overfitted. In Catboost there is a special modification for such cases. That is, on those datasets where other algorithms had a problem with overfitted you won't observe the same problem on Catboost.
  3. Fast and easy to-use GPU-training. You can simply install it via pip-install.
  4. Other useful features: missing value support, great visualization.
     - The main advantage of catboost is a smart preprocessing of categorical data. You don't have to preprocess data on your own. Some of the most popular practices to encode categorical data are:
       - One-hot encoding
       - Label encoding
       - Hashing encoding
       - Target encoding and etc..

CatBoost

- CatBoost uses oblivious decision trees, where the same splitting criterion is used across an entire level of the tree. Such trees are balanced, less prone to overfitting, and allow speeding up prediction significantly at testing time.



CatBoost

Among described advantages also need to mention the following one:

- Overfitting detector. Usually in gradient boosting we adjust learning rate to the stable accuracy. But the smaller learning rate — more iterations needed.
- Missing variables. Just left NaN
- In Catboost you are able to write your own loss function
- Feature importance
- CatBoost provides tools for the Python package that allow plotting charts with different training statistics. This information can be accessed both during and after the training procedure

CatBoost

- We can use CatBoost using code.

In [ ]:

```
from catboost import CatboostRegressor
# Initialize data
cat_feature = [0, 1,  2]
train_date = [[“a”, “b”, 1, 4, 5, 6], [“a”, “b”, 4, 5, 6, 7], [“c”, “d”, , 30, 40, 50, 60]]
test_data = [[“a”, “b”, 2, 4, 6, 8], [“a”, “b”, 1, 4, 50, 60]]
train_labels = [10, 20, 30]
# Initialize CatBoostRegressor
model = CatBoostRegressor(iterations=2, learning_rate=1, depth=2)
# Fit model
model.fit(train_data, train_labels, cat_features)
```

Code Implementation

- The main drivers of building the classifier model are Sci-Kit Learn's **sklearn.tree** and **sklearn.ensemble.**
- Before starting the classifier implementation, it is important to call the Decision Tree from **sklearn.tree.** Only the will the **sklearn.ensemble** the function work.

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier,
GradientBoostingClassifier,AdaBoostClassifier
from sklearn import datasets # import inbuild datasets
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.model_selection import cross_val_score, cross_val_predict
from sklearn.metrics import confusion_matrix
```

- Load the Iris data from ski-kit learn datasets.

In [ ]:
```
Iris = datasets.load_iris()
X = iris.data
y = iris.target
```

- The next step is to split the data set into 70% training and 30% testing using the hold-out method.

In [ ]:
```
X_train, X_test, y_train, y_train = train_test_split(x, y, test_size=0.3)
```

Bagging: Random Forest

In [ ]:
```
rf = RandomForestClassifier(n_estimators=100)
```

In [ ]:
```
bag_clf = BaggingClassifier(base_estimator=rf, n_estimators=100,
                            bootstrap=True, n_jobs=-1,
                            random_state=42)
bag_clf.fit(X_train, y_train)
```

In [ ]:
```
bag_clf.score(X_train,y_train),bag_clf.score(X_test,y_test)
```

In [ ]:
```
(0.9904761904761905, 0.9777777777777777)
```

- The accuracy is around 98%, and the model solves the problem of overfitting. Amazing!

Bagging: Random Forest

In [ ]:
```
rf = RandomForestClassifier(n_estimators=100)
```

In [ ]:
```
gb = GradientBoostingClassifier(n_estimators=100).fit
(X_train, y_train) gb.fit(X_train, y_train)
)
```

In [ ]:
```
gb.score (X_test,y_test),gb.score (X_train,y_train)
```

In [ ]:
```
(0.9333333333333333, 1.0)
```

Thank you!

Data Science
Academy