Ensemble Learning Part 1

17 March 2026

Alex Lyman

“Two heads are better than one, not because either is infallible, but because they are unlikely to go wrong in the same direction.”

— C.S. Lewis

Ensembles

- Learning algorithms have different biases
  - They probably do not make the same mistakes
  - If one makes a mistake, the others may not
- Solution: model combination
  - Exploit variation in algorithms
    - Voting Ensemble, Stacking, Cascade Generalization, Cascading, Delegating, Arbitrating
  - Exploit variation in data
    - Bagging, Boosting

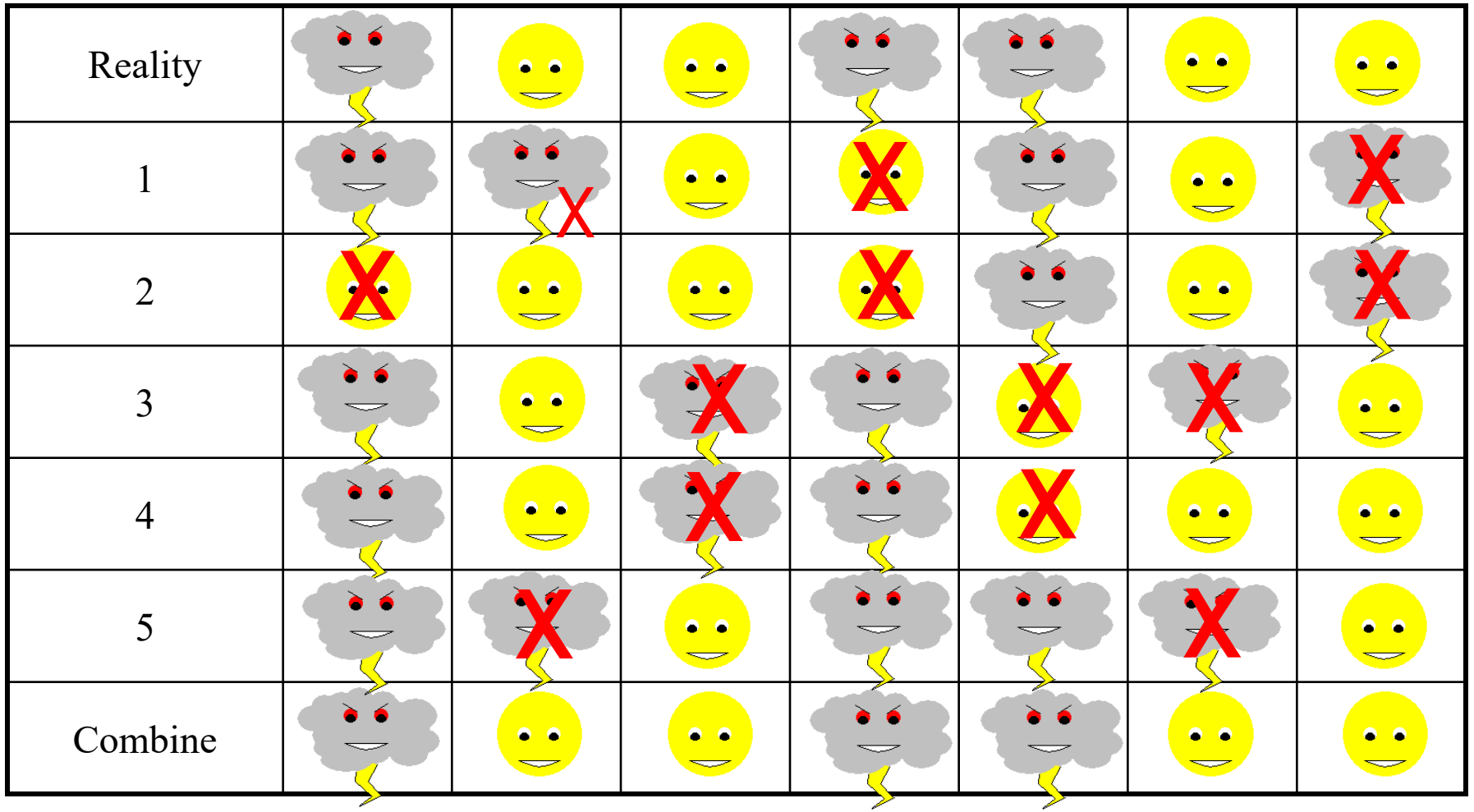
Value of Ensembles

- No single algorithm wins all the time!

- When combing multiple independent and diverse decisions each of which is at least more accurate than random guessing, random errors cancel each other out, correct decisions are reinforced.

- Examples: Human ensembles are demonstrably better
  - How many jelly beans in the jar?: Individual estimates vs. group average.
  - Who Wants to be a Millionaire: Audience vote.

Example: Weather Forecast



Historical Example

- 1906 county fair in Plymouth
- 800 people estimate the weight of an ox
- Francis Galton gathered all the estimates
- Median guess was accurate within 1%
- Published in Nature in 1907

Intuition

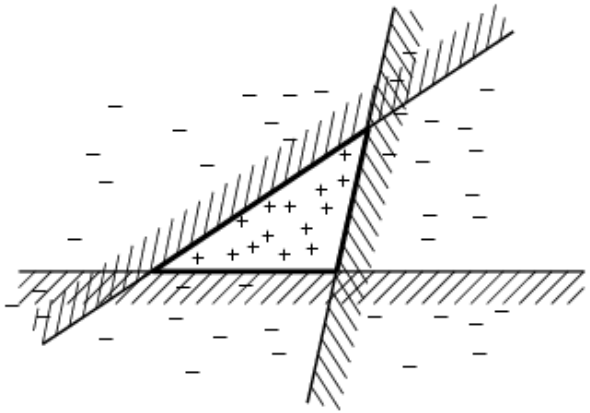
- Majority vote
- Suppose we have 5 completely independent classifiers…
  - If accuracy is 70% for each (30% chance of being wrong)
    - 0.16308 probability of being wrong
    - **83.692% majority vote accuracy**
  - 101 such classifiers
    - **99.9% majority vote accuracy**

$$\sum_{k=\frac{m+1}{2}}^{m} \binom{m}{k} r^k (1-r)^{m-k}$$

Ensemble Learning

- Another way of thinking about ensemble learning:

- → way of enlarging the hypothesis space, i.e., the ensemble itself is a hypothesis and the new hypothesis space is the set of all possible ensembles constructible from hypotheses of the original space.



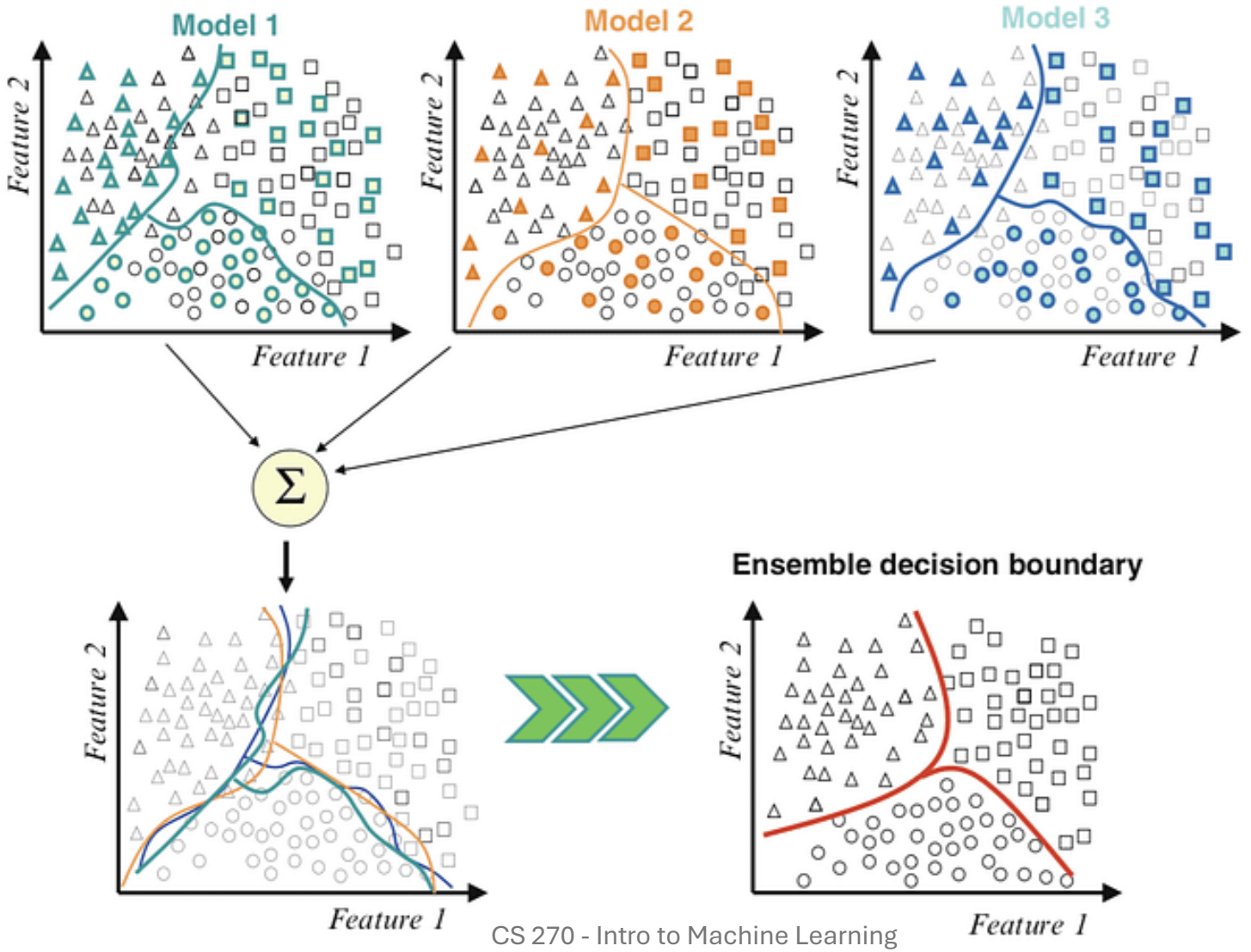
Increasing power of ensemble learning:

Three linear threshold hypothesis
(positive examples on the non-shaded side);
Ensemble classifies as positive any example classified positively be all three. The resulting triangular region hypothesis is not expressible in the original hypothesis space.

Four Important Criteria

- **Independence**: The various guesses have to be independent of one another. That is, each person must guess without the knowledge of what other people have guessed.
- **Diversity**: It is important to have a diverse set of guesses. In the guess the weight of the ox example, the people making the guesses ranged from farmers, butchers, livestock experts, housewives etc. That is, some people would be considered experts, while others would be considered as people with just a passing interest.
- **Decentralization**: The people making the guesses should be able to draw on their private, local knowledge.
- **Aggregation**: There must be some way of aggregating the guesses into a single collective guess. In the guess the weight of the ox example, this was done by taking the median guess. This is a common method, but others may also be used.

More Intuition



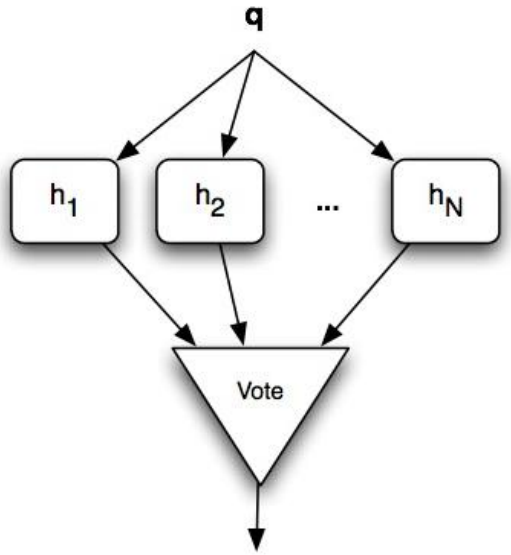
Variation in Algorithms

Exploiting Variation in Algorithms

- Ensembles must be made of different models (diversity criteria)
- What would happen if you ensembled together 3 KNN classifiers?
- Variation in Algorithms can provide difference.
- We'll be talking several ways to take advantage of this:
  - Voting Ensembles
  - Stacking (Stacked Generalization)
  - Cascading
  - Delegating (Mixture of Experts)

Voting Ensemble

- **Like a Democracy (Majority Rules)**
- **How it Works:** Train several completely different models (e.g., a Logistic Regression, an SVM, and a Decision Tree) on your training data. When a new data point arrives, ask all of them to make a prediction.
- **Hard Voting:** Every model gets one vote. The class with the most votes wins.
  - *Analogy:* A jury rendering a verdict.
- **Soft Voting:** Models output their *confidence probabilities* (e.g., "I am 90% sure this is Class A"). We average the probabilities together.
  - *Why Soft is Better:* It gives more weight to highly confident models and ignores models that are just guessing (51% vs 49%).



```
import numpy as np
from sklearn.linear_model import
LogisticRegression
from sklearn.naive_bayes import
GaussianNB
from sklearn.ensemble import
RandomForestClassifier, VotingClassifier

clf1 =
LogisticRegression(multi_class='multinom
ial', random_state=1)
clf2 =
RandomForestClassifier(n_estimators=50,
random_state=1)
clf3 = GaussianNB()
X = np.array([[-1, -1], [-2, -1], [-3, -
2], [1, 1], [2, 1], [3, 2]])
y = np.array([1, 1, 1, 2, 2, 2])

eclf1 = VotingClassifier(estimators=[
('lr', clf1), ('rf', clf2), ('gnb',
clf3)], voting='hard')
eclf1 = eclf1.fit(X, y)
```

sklearn.ensemble.VotingClassifier

class `sklearn.ensemble.`**`VotingClassifier`**(*estimators, *, voting='hard', weights=None, n_jobs=None, flatten_transform=True, verbose=False*) [source]

Soft Voting/Majority Rule classifier for unfitted estimators.

Read more in the User Guide.

New in version 0.17.

Parameters::

estimators : *list of (str, estimator) tuples*
Invoking the `fit` method on the `VotingClassifier` will fit clones of those original estimators that will be stored in the class attribute `self.estimators_`. An estimator can be set to `'drop'` using `set_params`.

Changed in version 0.21: `'drop'` is accepted. Using None was deprecated in 0.22 and support was removed in 0.24.

voting : *{'hard', 'soft'}, default='hard'*
If 'hard', uses predicted class labels for majority rule voting. Else if 'soft', predicts the class label based on the argmax of the sums of the predicted probabilities, which is recommended for an ensemble of well-calibrated classifiers.

weights : *array-like of shape (n_classifiers,), default=None*
Sequence of weights (`float` or `int`) to weight the occurrences of predicted class labels (`hard` voting) or class probabilities before averaging (`soft` voting). Uses uniform weights if `None`.

n_jobs : *int, default=None*
The number of jobs to run in parallel for `fit`. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

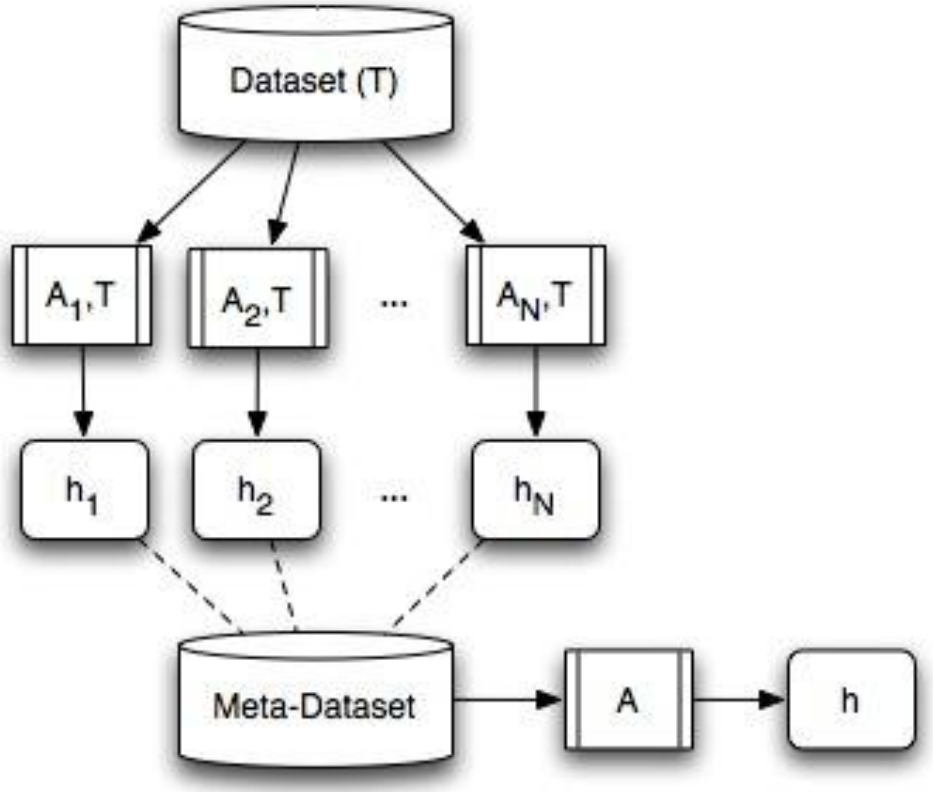
New in version 0.18.

flatten_transform : *bool, default=True*
Affects shape of transform output only when voting='soft' If voting='soft' and flatten_transform=True, transform method returns matrix with shape (n_samples, n_classifiers * n_classes). If flatten_transform=False, it returns (n_classifiers, n_samples, n_classes).

verbose : *bool, default=False*
If True, the time elapsed while fitting will be printed as it is completed.

Stacking

- **Like a Manager and the Department Heads**
- **The Problem with Voting:** Voting assumes all models are equally trustworthy all the time. But what if the SVM is terrible at certain types of data?
- **How Stacking Works:** * **Level 0 (Base Models):** Train your diverse models (KNN, SVM, Tree) just like a voting classifier.
  - **Level 1 (The Meta-Learner):** Instead of simply averaging their answers, we train a *new* machine learning model (usually a simple Logistic Regression) to look at the predictions of the base models and figure out the final answer.
- **The Analogy:** The Meta-Learner is a manager. The manager learns over time, "When the SVM and the KNN disagree, the KNN is usually right, so I will weigh its input heavier."



```
from sklearn.datasets import load_iris
from sklearn.ensemble import
RandomForestClassifier
from sklearn.svm import LinearSVC
from sklearn.linear_model import
LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.ensemble import StackingClassifier

X, y = load_iris(return_X_y=True)
estimators = [ ('rf',
RandomForestClassifier(n_estimators=10,
random_state=42)),
                          ('svr',
make_pipeline(StandardScaler(),
LinearSVC(random_state=42))) ]

clf = StackingClassifier( estimators=estimators,
final_estimator=LogisticRegression() )
```

sklearn.ensemble.StackingClassifier

class `sklearn.ensemble.StackingClassifier`(*estimators*, *final_estimator=None*, ***, *cv=None*, *stack_method='auto'*, *n_jobs=None*, *passthrough=False*, *verbose=0*) [source]

Stack of estimators with a final classifier.

Stacked generalization consists in stacking the output of individual estimator and use a classifier to compute the final prediction. Stacking allows to use the strength of each individual estimator by using their output as input of a final estimator.

Note that `estimators_` are fitted on the full `X` while `final_estimator_` is trained using cross-validated predictions of the base estimators using `cross_val_predict`.

Read more in the User Guide.

New in version 0.22.

Parameters::

estimators : *list of (str, estimator)*
Base estimators which will be stacked together. Each element of the list is defined as a tuple of string (i.e. name) and an estimator instance. An estimator can be set to 'drop' using `set_params`.

final_estimator : *estimator, default=None*
A classifier which will be used to combine the base estimators. The default classifier is a `LogisticRegression`.

cv : *int, cross-validation generator, iterable, or "prefit", default=None*
Determines the cross-validation splitting strategy used in `cross_val_predict` to train `final_estimator`. Possible inputs for cv are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a (Stratified) KFold,
- An object to be used as a cross-validation generator,
- An iterable yielding train, test splits,
- `"prefit"` to assume the `estimators` are prefit. In this case, the estimators will not be refitted.

For integer/None inputs, if the estimator is a classifier and y is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used. These splitters are instantiated with `shuffle=False` so the splits will be the same across calls.

Refer User Guide for the various cross-validation strategies that can be used here.

If "prefit" is passed, it is assumed that all `estimators` have been fitted already. The `final_estimator_` is trained on the `estimators` predictions on the full training set and are **not** cross validated predictions. Please note that if the models have been trained on the same data to train the stacking model, there is a very high risk of overfitting.

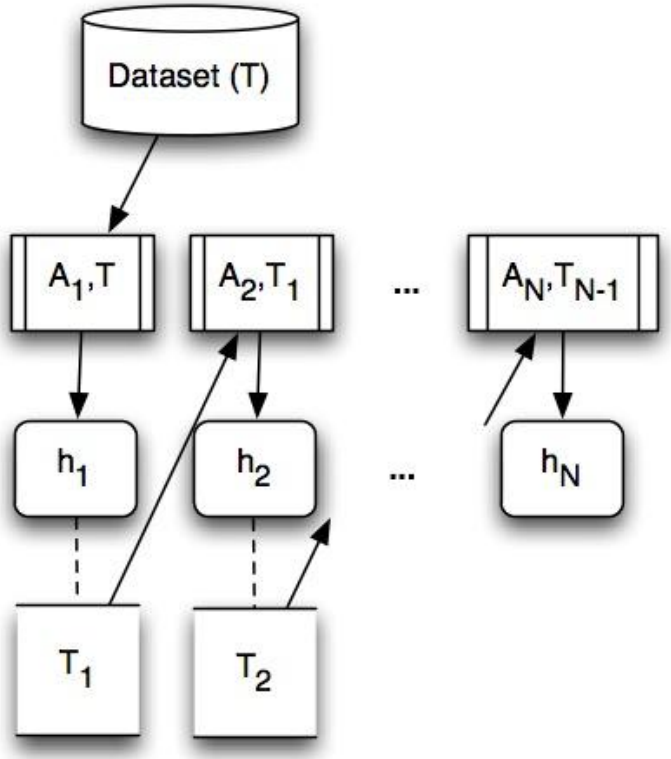
New in version 1.1: The 'prefit' option was added in 1.1

> **Note:** A larger number of split will provide no benefits if the number of training samples is large enough. Indeed, the training time will increase. `cv` is not used for model evaluation but for prediction.

stack_method : *{'auto', 'predict_proba', 'decision_function', 'predict'}, default='auto'*
Methods called for each base estimator. It can be:

Cascading

- **Assembly Line**
- **The Goal:** Efficiency and speed. We don't want to run a massive, slow model if the answer is obvious.
- **How it Works:** Models are arranged in a sequence, usually from simplest/fastest to most complex/slowest.
  - Pass data to Model 1 (e.g., a fast Logistic Regression).
  - If Model 1 is highly confident, output the answer immediately.
  - If Model 1 is uncertain, pass the data down the line to Model 2 (e.g., a deep Neural Net).
- **The Analogy:** Medical triage at a hospital. A triage nurse handles the obvious, simple cases immediately. The highly complex, borderline cases are passed up the chain to the specialist.



Delegating (Mixture of Experts)

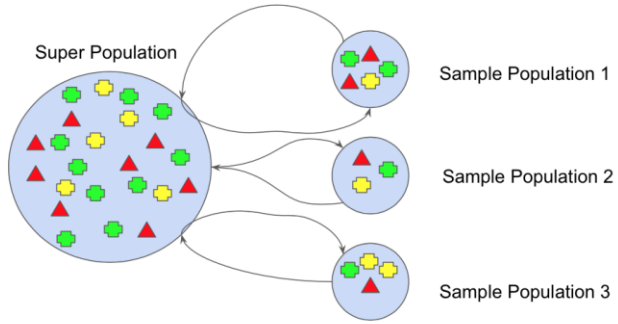
- **Specialists**
- **The Concept:** Instead of asking all models to solve the whole problem, we train models to become absolute experts in specific sub-regions of the data.
- **How it Works:** We train a "Gating Network" (a router).
  - When a new data point comes in, the Gating Network doesn't predict the answer. Instead, it predicts *which model* is best suited to handle this specific data point.
- **The Analogy:** A large consulting firm. You walk in the door and the receptionist (the Gating Network) looks at your problem. "Oh, you have a tax issue? I am sending you to Alice, our tax expert. You have a legal issue? I am sending you to Bob."

Variation in Data

Bootstrap Samples and Bagging

Bootstrap and Bagging

- Create ensembles by "*Bootstrap AGGregation*", i.e., repeatedly randomly resampling the training data (Brieman, 1996).



- Bootstrap:
  - draw N items from training data with replacement

- Bagging
  - Train *M* learners on *M* bootstrap samples
  - Combine outputs by voting (e.g., majority vote)

- Decreases error by decreasing the variance in the results due to ***unstable learners***, algorithms (like decision trees and neural networks) whose output can change dramatically when the training data is slightly changed.

Bootstrapping

- Not all samples are necessarily the same size.
- As you do more samples, you get lower chance of not training on certain data points.

Image: Sebastian Raschka

Bootstrapping

- Not all samples are necessarily the same size.
- As you do more samples, you get lower chance of not training on certain data points.

$$P(\text{not chosen}) = \left(1 - \frac{1}{m}\right)^m,$$

$$\frac{1}{e} \approx 0.368, \quad m \to \infty .$$

$$P(\text{chosen}) = 1 - \left(1 - \frac{1}{m}\right)^m \approx 0.632$$

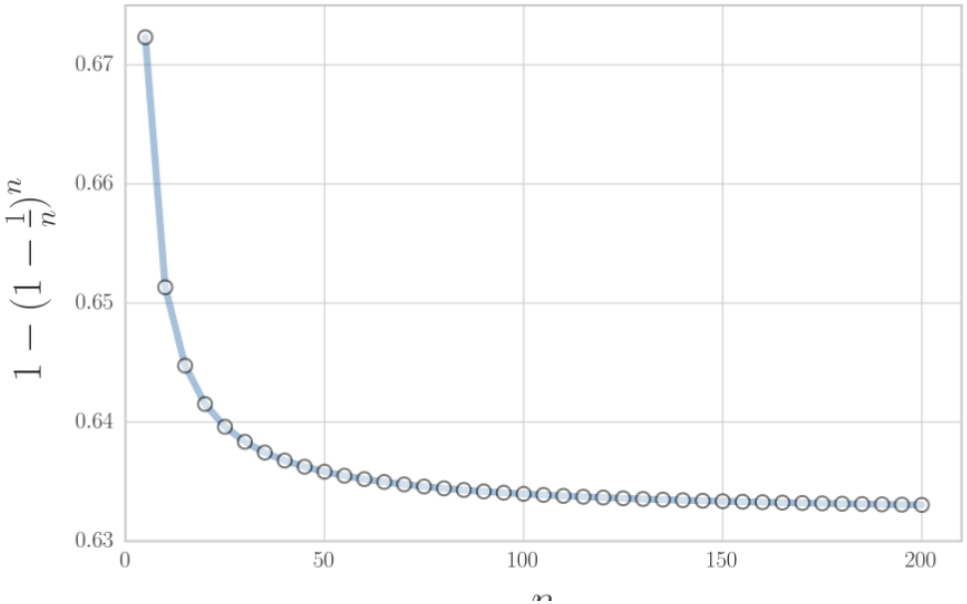
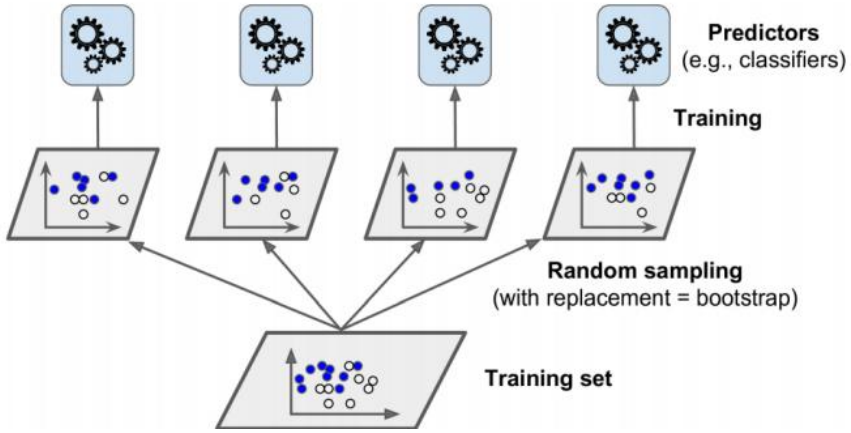


Image: Sebastian Raschka

Bagging - Aggregate Bootstrapping

Given a standard training set *D* of size *n*

- For i = 1 .. M
  - Draw a sample of size $n^* < n$ from *D* uniformly and with replacement
  - Learn classifier C_i
- Final classifier is a vote of C_1 .. C_M
- Increases classifier stability/reduces variance



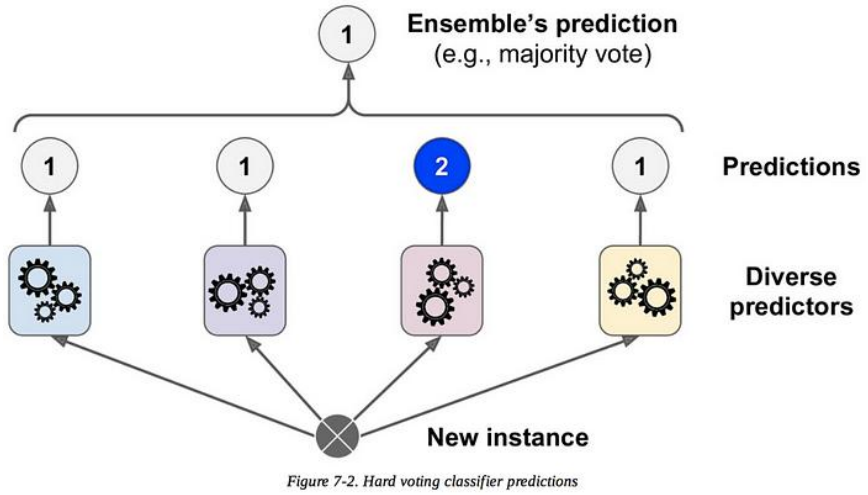


Figure 7-2. Hard voting classifier predictions

```
from sklearn.svm import SVC
from sklearn.ensemble import BaggingClassifier
from sklearn.datasets import
make_classification

X, y = make_classification(n_samples=100,
n_features=4, n_informative=2,

n_redundant=0, random_state=0, shuffle=False)

clf = BaggingClassifier(base_estimator=SVC(),
n_estimators=10, random_state=0).fit(X, y)

clf.predict([[0, 0, 0, 0]])
```

sklearn.ensemble.BaggingClassifier

class `sklearn.ensemble.`**`BaggingClassifier`**(*base_estimator=None, n_estimators=10, *, max_samples=1.0, max_features=1.0, bootstrap=True, bootstrap_features=False, oob_score=False, warm_start=False, n_jobs=None, random_state=None, verbose=0*) [source]

A Bagging classifier.

A Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

This algorithm encompasses several works from the literature. When random subsets of the dataset are drawn as random subsets of the samples, then this algorithm is known as Pasting [1]. If samples are drawn with replacement, then the method is known as Bagging [2]. When random subsets of the dataset are drawn as random subsets of the features, then the method is known as Random Subspaces [3]. Finally, when base estimators are built on subsets of both samples and features, then the method is known as Random Patches [4].

Read more in the User Guide.

New in version 0.15.

Parameters::

base_estimator : *object, default=None*
The base estimator to fit on random subsets of the dataset. If None, then the base estimator is a `DecisionTreeClassifier`.

n_estimators : *int, default=10*
The number of base estimators in the ensemble.

max_samples : *int or float, default=1.0*
The number of samples to draw from X to train each base estimator (with replacement by default, see `bootstrap` for more details).

- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples.

max_features : *int or float, default=1.0*
The number of features to draw from X to train each base estimator (without replacement by default, see `bootstrap_features` for more details).

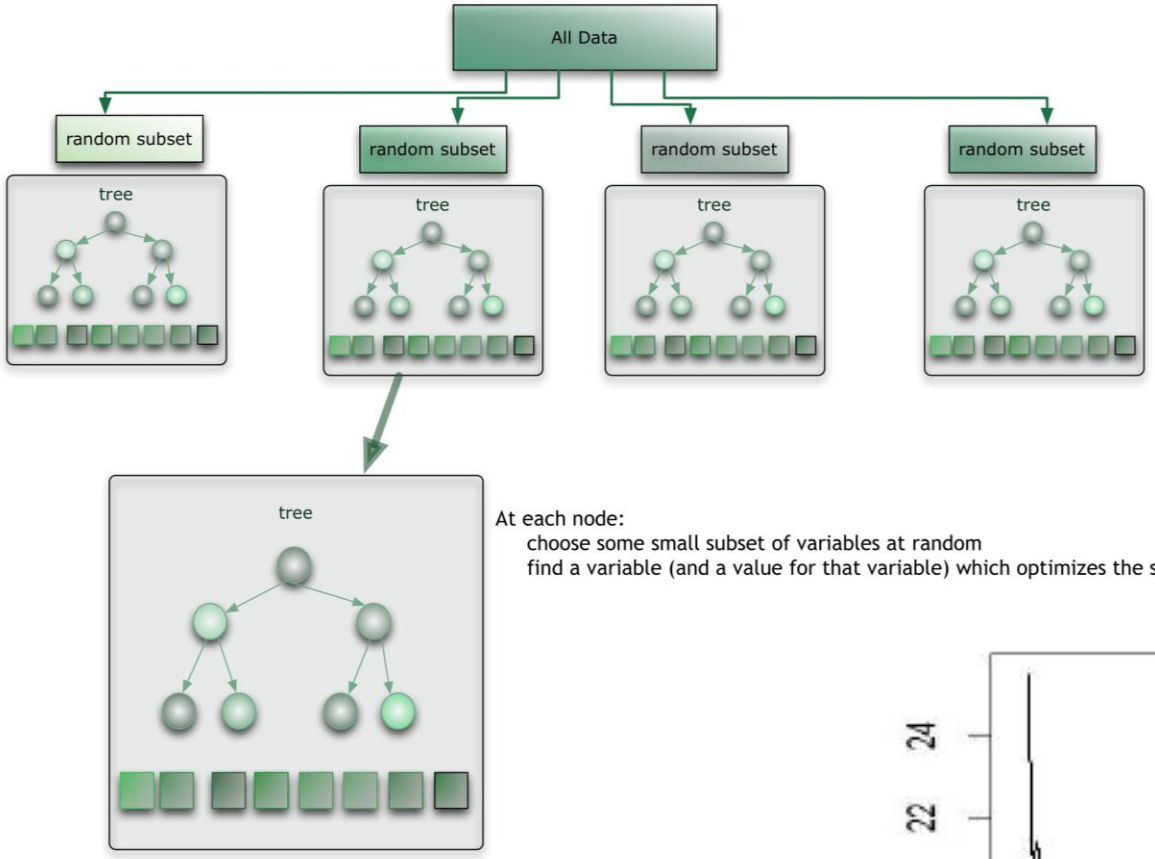
- If int, then draw `max_features` features.
- If float, then draw `max(1, int(max_features * n_features_in_))` features.

bootstrap : *bool, default=True*
Whether samples are drawn with replacement. If False, sampling without replacement is performed.

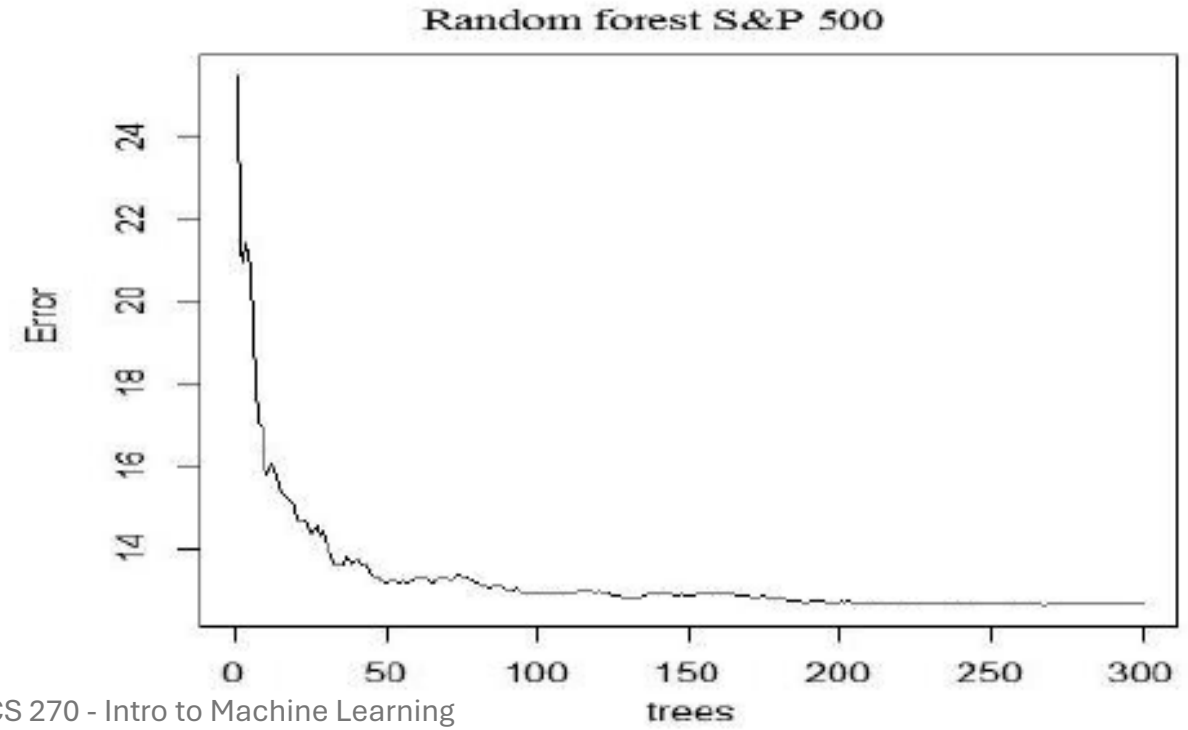
Random Forests

Random Forest

- An extension of bagging that also randomly selects subsets of features used in each data sample
- Build many decision trees
  - Each tree only uses a random subset of the features
    - For classification a good default is: m = sqrt(p)
    - For regression a good default is: m = p/3
- Two random variables
  - Bootstrap sample of the data
  - Features to consider for a given decision tree node



At each node:
choose some small subset of variables at random
find a variable (and a value for that variable) which optimizes the split



RandomForestClassifier

class **sklearn.ensemble.RandomForestClassifier(*n_estimators=100, *, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='sqrt', max_leaf_nodes=None, min_impurity_decrease=0.0, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None, monotonic_cst=None*)** [source]

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. Trees in the forest use the best split strategy, i.e. equivalent to passing `splitter="best"` to the underlying `DecisionTreeClassifier`. The sub-sample size is controlled with the `max_samples` parameter if `bootstrap=True` (default), otherwise the whole dataset is used to build each tree.

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=1000, n_features=4,
...                            n_informative=2, n_redundant=0,
...                            random_state=0, shuffle=False)
>>> clf = RandomForestClassifier(max_depth=2, random_state=0)
>>> clf.fit(X, y)
RandomForestClassifier(...)
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]
```

Break

Boosting

Strong and Weak Learners

- Strong Learner →Objective of machine learning
  - Take labeled data for training
  - Produce a classifier which can be *arbitrarily accurate*

- Weak Learner
  - Take labeled data for training
  - Produce a classifier which is more accurate than random guessing

Boosting

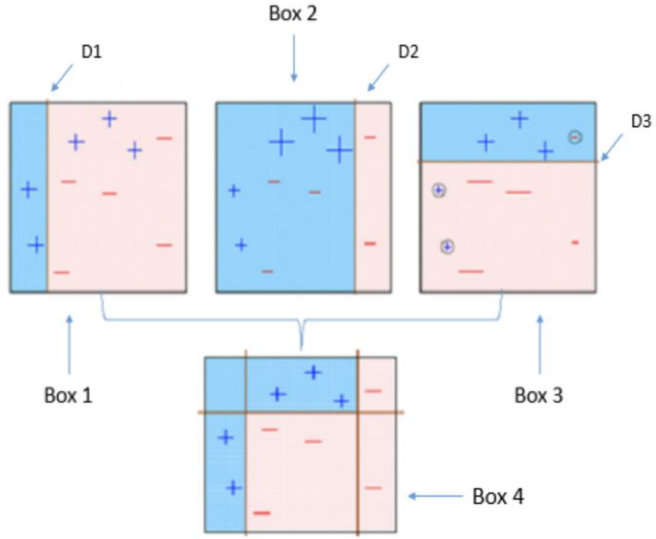
- Strong learners are very difficult to construct
  - Constructing weaker learners is relatively easy

- Weak learners only need to generate a hypothesis with a training accuracy greater than 0.5, i.e., < 50% error over any distribution

- Questions: Can a set of **weak learners** create a single **strong learner** ?

- YES ☺

  Boost weak classifiers to a strong learner



Boosting

- Originally developed by computational learning theorists to guarantee performance improvements on fitting training data for a ***weak learner*** that only needs to generate a hypothesis with a training accuracy greater than 0.5 (Schapire, 1990).

- Instead of sampling (as in bagging) re-weigh examples!
- Examples are given weights. At each iteration, a new hypothesis is learned (weak learner) and the examples are reweighted to focus the system on examples that the most recently learned classifier got wrong.

- Final classification based on weighted vote of weak classifiers

Construct Weak Classifiers

- Using Different Data Distribution
  - Start with uniform weighting
  - During each step of learning
    - Increase weights of the examples which are not correctly learned by the weak learner
    - Decrease weights of the examples which are correctly learned by the weak learner
- Idea
  - Focus on difficult examples which are not correctly classified in the previous steps

Combine Weak Classifiers

- Weighted Voting
  - Construct strong classifier by
  - weighted voting of the weak classifiers
- Idea
  - Better weak classifier gets a larger weight
  - Iteratively add weak classifiers
    - Increase accuracy of the combined classifier through minimization of a cost function

Adaptive Boosting (AdaBoost)

AdaBoost (Adaptive Boosting)

- **The "Hard Test" Strategy**
- Instead of training models randomly, we train a sequence of models where each new model is explicitly forced to focus on the data points the previous model got wrong.
- Think of this like studying for a final exam
  - *Model 1:* You take a practice test. You get all the easy questions right, but fail the calculus questions.
  - *The Update:* You grab a red marker and draw a giant circle around the calculus questions. You assign them a **higher weight** of importance.
  - *Model 2:* You take a new practice test, but this time you *obsess* over the red-circled calculus questions. You get them right, but now you miss the probability questions.
  - *The Update:* You draw circles around the probability questions…
- You build a team of "specialists." One model is the calculus expert, one is the probability expert. Together, they pass the test.

Adaptive Boosting in Pseudocode

C =0; /* counter*/
M = m; /* number of hypotheses to generate*/

1 Set same weight for all the examples (typically each example has weight = 1);

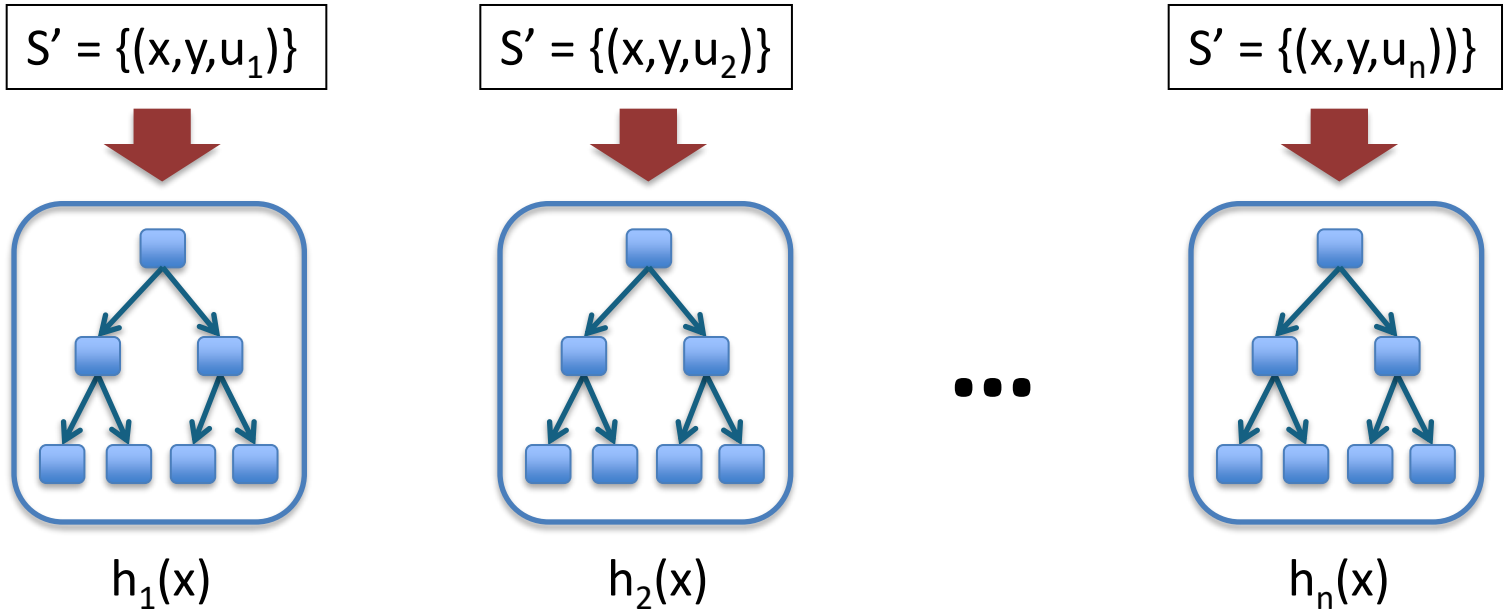
2 While (C < M)
    2.1 Increase counter C by 1.
    2.2 Generate hypothesis h_C . (Train a model using the weighted data)
    2.3 Increase the weight of the misclassified examples in hypothesis h_C

3 Weighted majority combination of all M hypotheses (weights according to how well it performed on the training set).

Many variants depending on how to set the weights and how to combine the hypotheses.
    ADABOOST → quite popular!!!!

Boosting (AdaBoost)

$$h(x) = a_1h_1(x) + a_2h_2(x) + \ldots + a_nh_n(x)$$

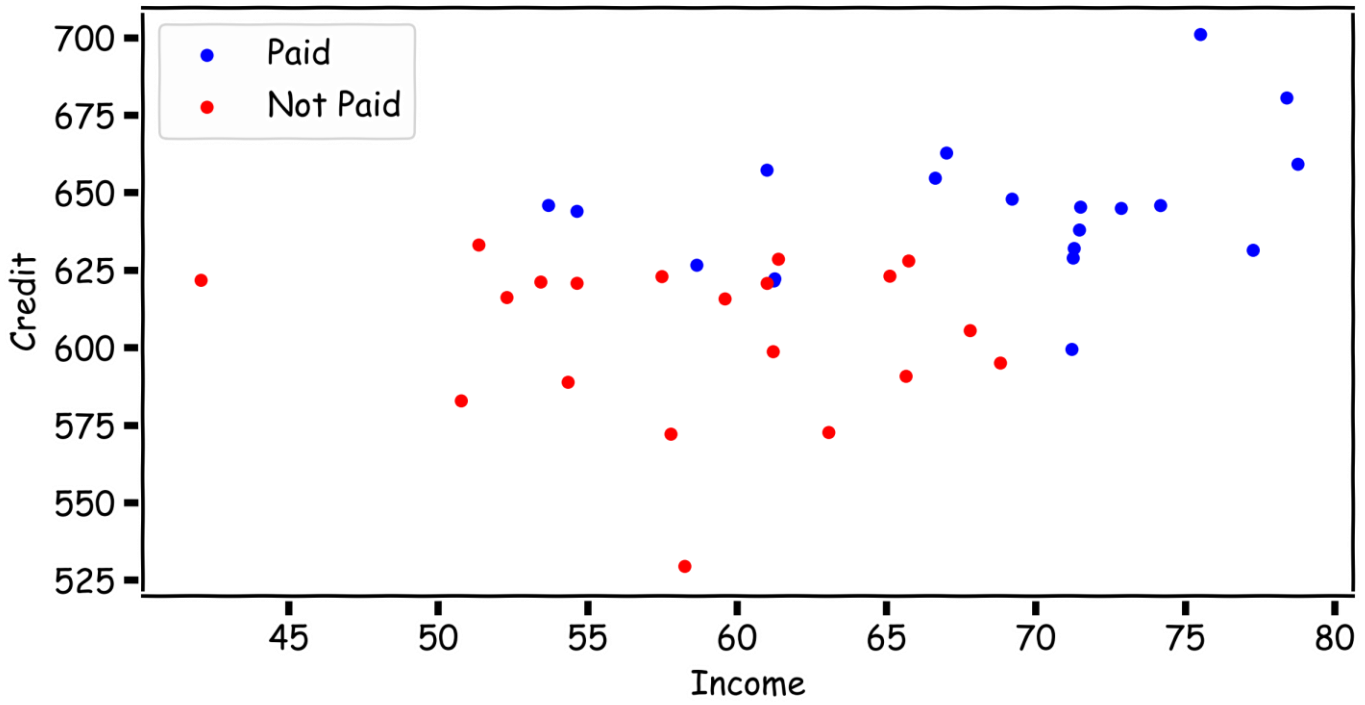


u – weighting on data points
a – weight of linear combination

Stop when validation performance plateaus

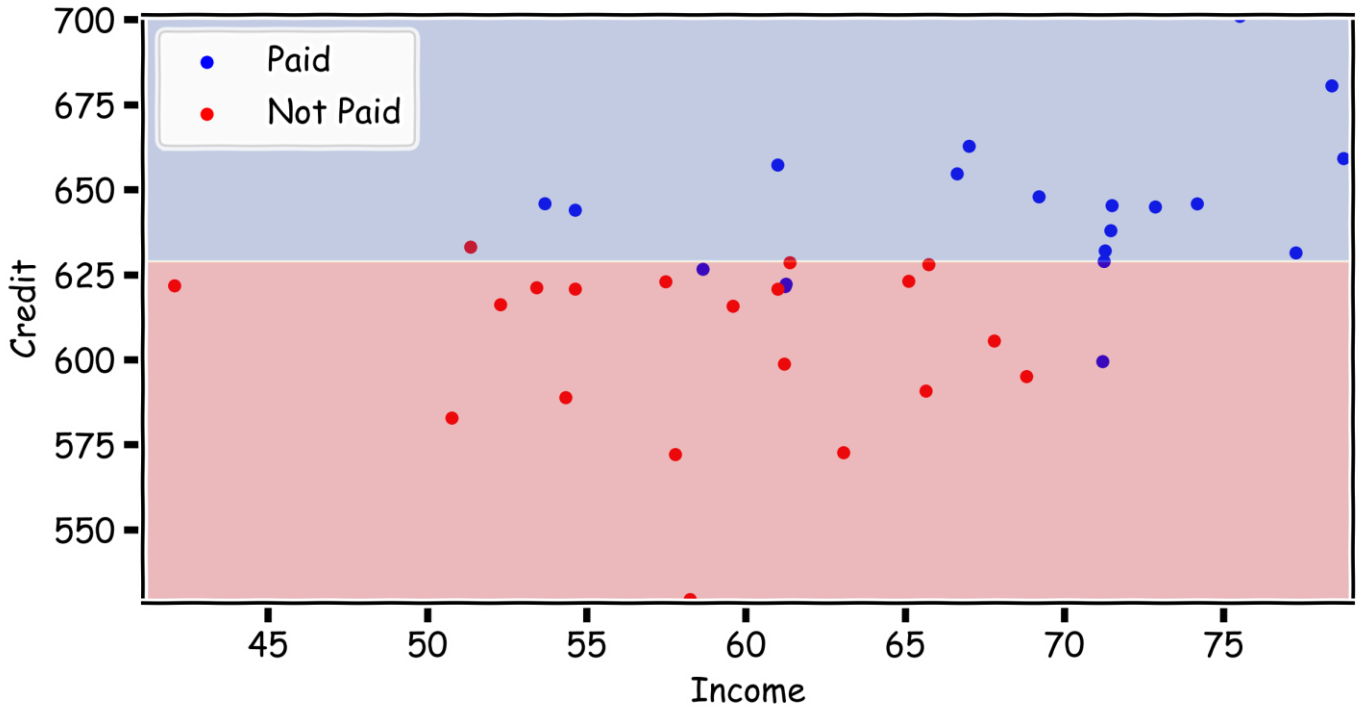
https://www.cs.princeton.edu/~schapire/papers/explaining-adaboost.pdf

AdaBoost: start with equal weights



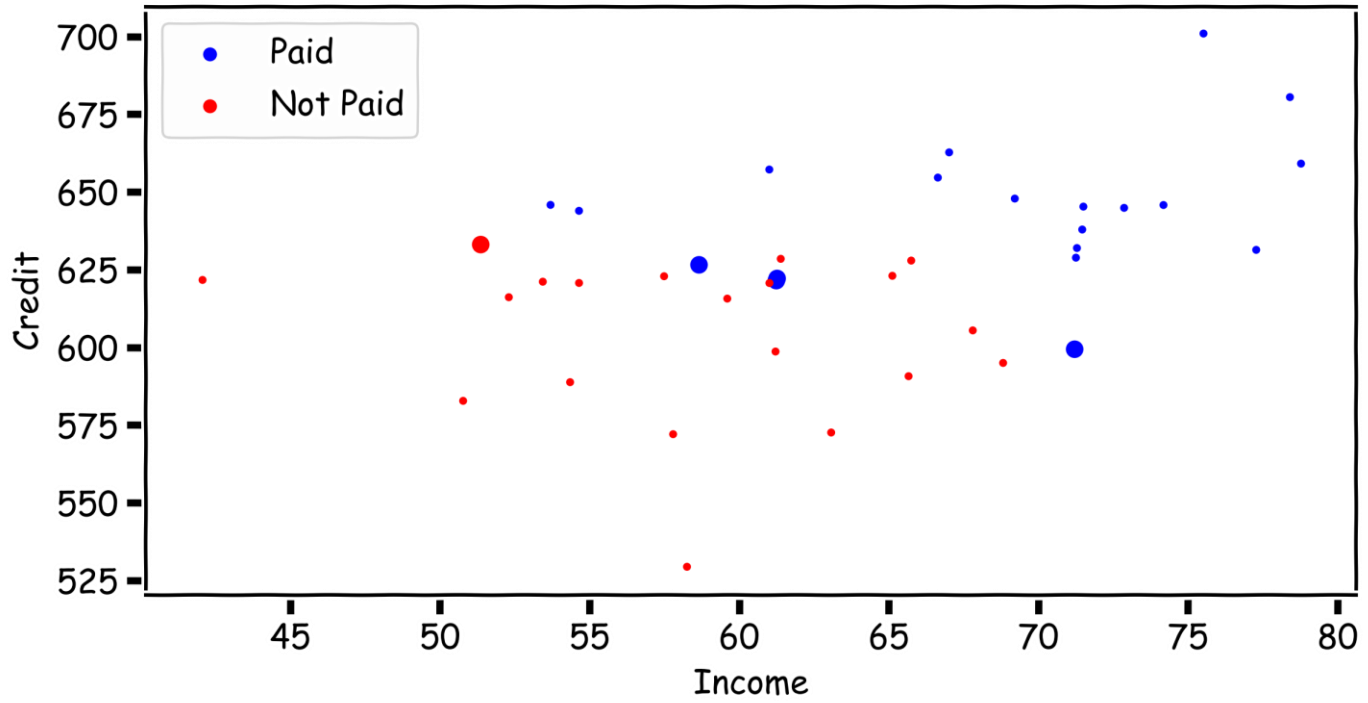
AdaBoost: fit a simple decision tree

fit a simple classifier $T^{(i)}$



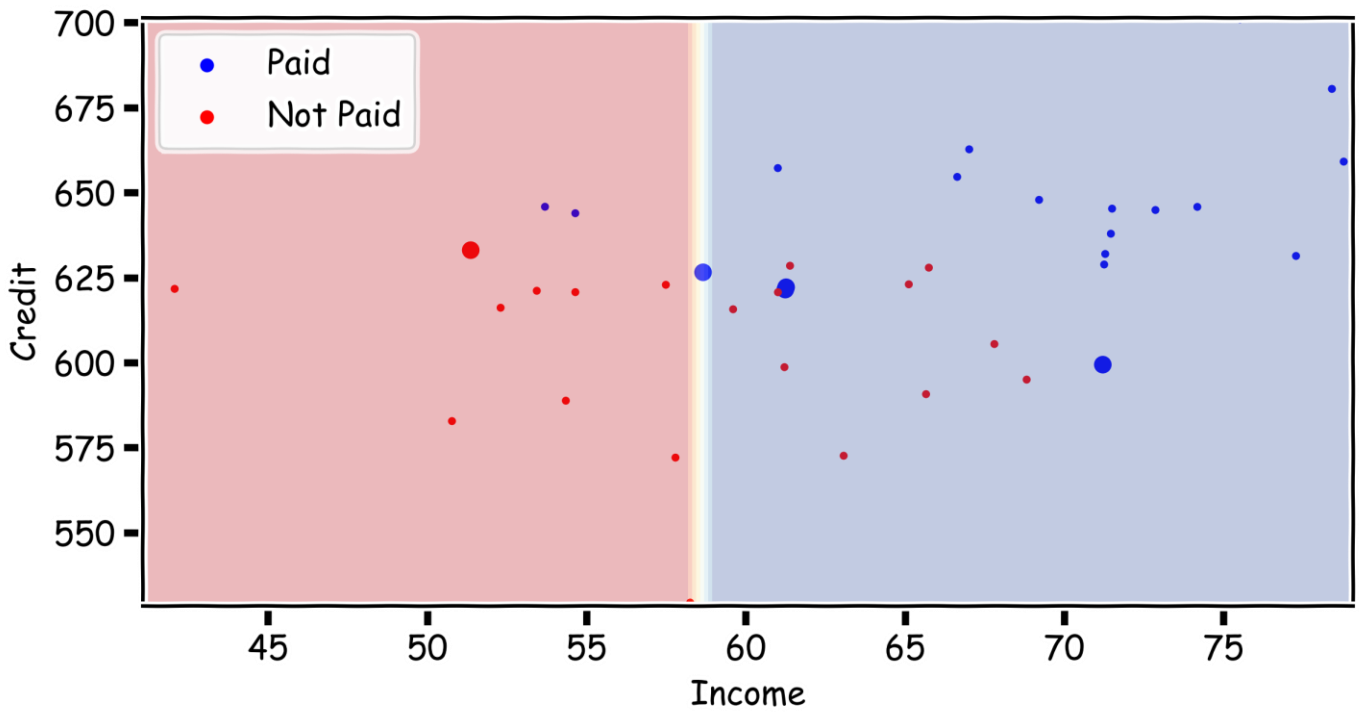
AdaBoost: update the weights

Update the weights: $$w_n \leftarrow \frac{w_n \exp(-\lambda^{(i)} y_n T^{(i)}(x_n))}{Z}$$



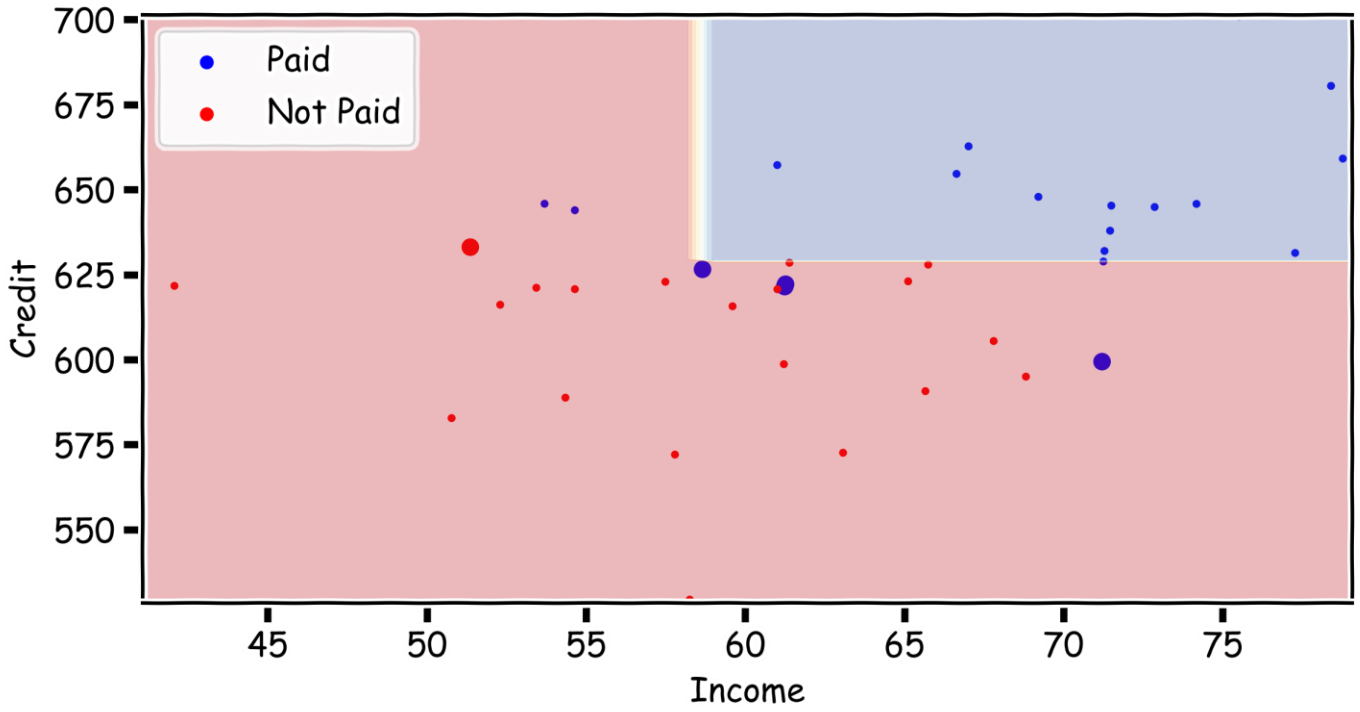
AdaBoost:

fit another simple decision tree on re-weighted data



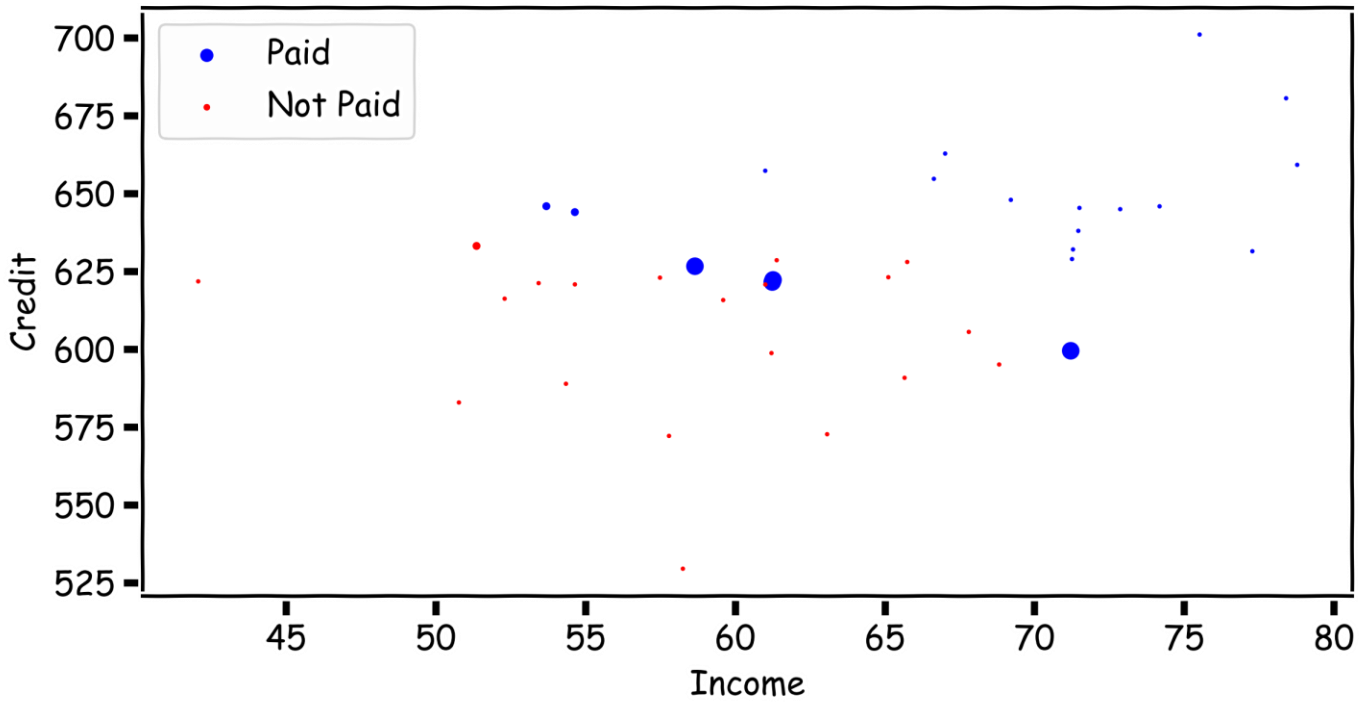
AdaBoost:

add the new model to the ensemble: $T \leftarrow T + \lambda^{(i)} T^{(i)}$



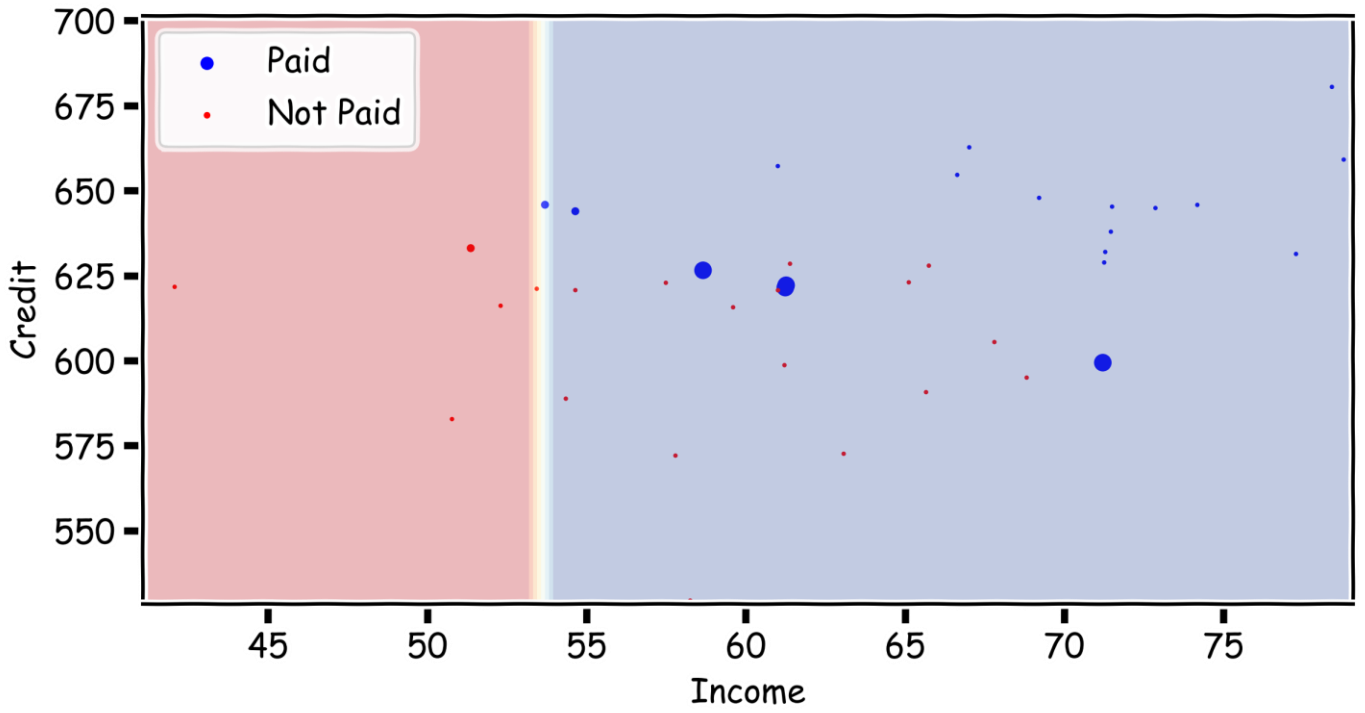
AdaBoost: update the weights

Update the weights: $$w_n \leftarrow \frac{w_n \exp(-\lambda^{(i)} y_n T^{(i)}(x_n))}{Z}$$



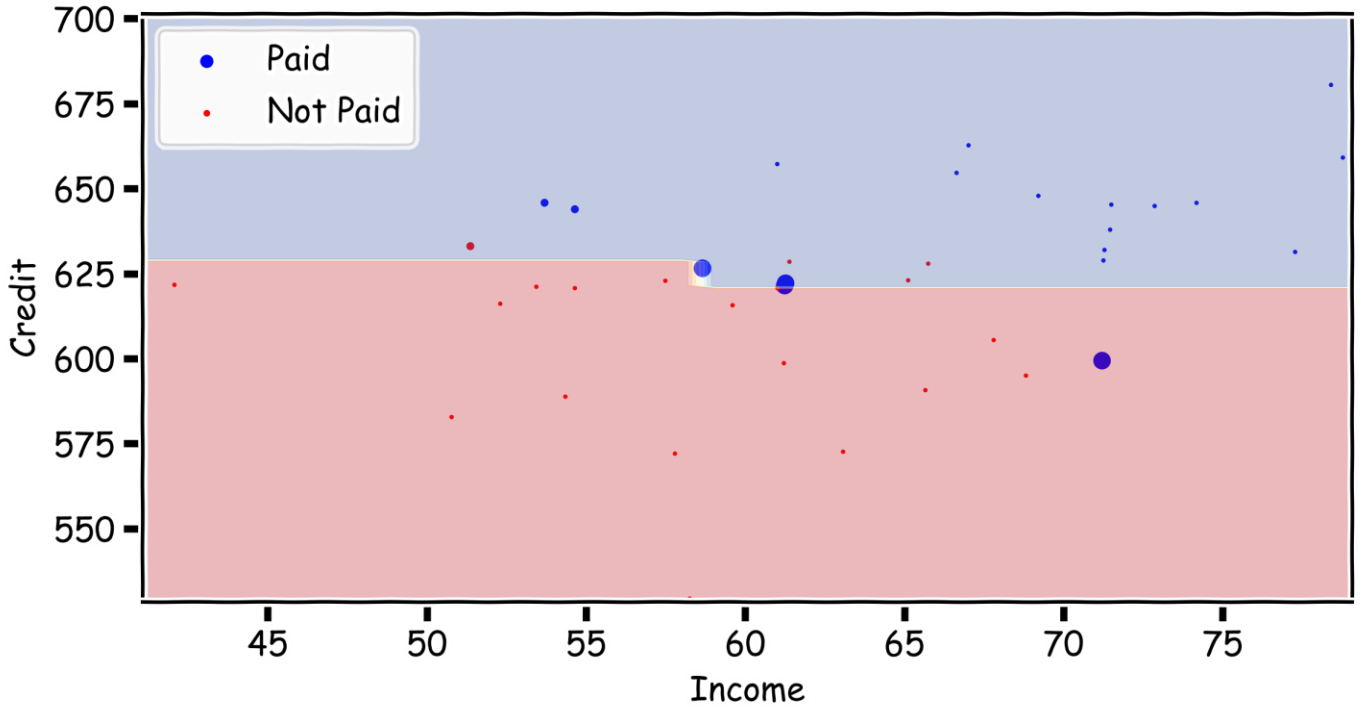
AdaBoost

fit another simple decision tree on re-weighted data



AdaBoost: add the new model to the ensemble, repeat…

add the new model to the ensemble: $T \leftarrow T + \lambda^{(i)} T^{(i)}$



Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in \mathcal{X}$, $y_i \in \{-1, +1\}$.

Initialize $D_1(i) = 1/m$ for $i = 1, \ldots, m$. ← Initial Distribution of Data

For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t.
- Get weak hypothesis $h_t : \mathcal{X} \to \{-1, +1\}$. ← Train model
- Aim: select h_t with low weighted error:

$$\varepsilon_t = \Pr_{i \sim D_t}\left[h_t(x_i) \neq y_i\right].$$ ← Error of model

- Choose $\alpha_t = \frac{1}{2} \ln\left(\frac{1 - \varepsilon_t}{\varepsilon_t}\right)$. ← Coefficient of model
- Update, for $i = 1, \ldots, m$:

$$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$ ← Update Distribution

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right).$$ ← Final average

Theorem: training error drops exponentially fast

https://www.cs.princeton.edu/~schapire/papers/explaining-adaboost.pdf

```
from sklearn.ensemble import
AdaBoostClassifier
from sklearn.datasets import
make_classification
X, y =
make_classification(n_samples=1000,
n_features=4,

n_informative=2, n_redundant=0,

random_state=0, shuffle=False)
clf =
AdaBoostClassifier(n_estimators=100,
random_state=0)
clf.fit(X, y)
clf.predict([[0, 0, 0, 0]])
clf.score(X, y)
```

sklearn.ensemble.AdaBoostClassifier

class `sklearn.ensemble.`**`AdaBoostClassifier`**(*base_estimator=None, *, n_estimators=50, learning_rate=1.0, algorithm='SAMME.R', random_state=None*) [source]

An AdaBoost classifier.

An AdaBoost [1] classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

This class implements the algorithm known as AdaBoost-SAMME [2].

Read more in the User Guide.

New in version 0.14.

Parameters::

base_estimator : *object, default=None*
The base estimator from which the boosted ensemble is built. Support for sample weighting is required, as well as proper `classes_` and `n_classes_` attributes. If `None`, then the base estimator is `DecisionTreeClassifier` initialized with `max_depth=1`.

n_estimators : *int, default=50*
The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early. Values must be in the range `[1, inf)`.

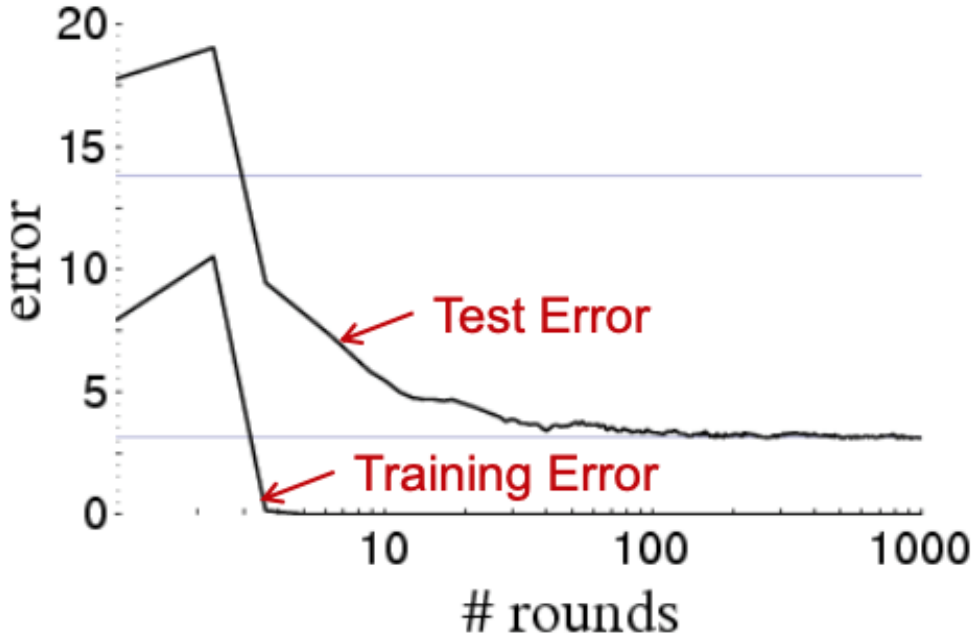
learning_rate : *float, default=1.0*
Weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier. There is a trade-off between the `learning_rate` and `n_estimators` parameters. Values must be in the range `(0.0, inf)`.

algorithm : *{'SAMME', 'SAMME.R'}, default='SAMME.R'*
If 'SAMME.R' then use the SAMME.R real boosting algorithm. `base_estimator` must support calculation of class probabilities. If 'SAMME' then use the SAMME discrete boosting algorithm. The SAMME.R algorithm typically converges faster than SAMME, achieving a lower test error with fewer boosting iterations.

random_state : *int, RandomState instance or None, default=None*
Controls the random seed given at each `base_estimator` at each boosting iteration. Thus, it is only used when `base_estimator` exposes a `random_state`. Pass an int for reproducible output across multiple function calls. See Glossary.

Performance of Adaboost

- Learner = Hypothesis = Classifier
- Weak Learner: < 50% error over any distribution
- M number of hypothesis in the ensemble.



- **Theory:** If the input learning is a Weak Learner, then ADABOOST will return a hypothesis that classifies the training data perfectly for a large enough M, boosting the accuracy of the original learning algorithm on the training data.
- Strong Classifier: thresholded linear combination of weak learner outputs.

Gradient Boosting

Gradient Boosting – A Different Way to Learn from Mistakes

- **AdaBoost's Strategy:** "I missed that data point. I will make that data point physically larger/more important so the next tree is forced to look at it."
  - *Modifies the Data Weights*.
- **Gradient Boosting's Strategy:**
  - "I predicted the house costs $300k. The real price is $350k. My error is $50k. I will tell the next tree to completely ignore the house price, and just try to predict the number '$50k'."
  - *Modifies the Target Variable (Labels)*.

Gradient Boosting

- The key to boosting is to learn from mistakes
- AdaBoost learns from the mistakes by increasing the weight of misclassified data points.

- Gradient Boosting learns from the mistake — residual error, directly rather than update the weights of data points.
  - Learn the error – use it to correct

Gradient Boosting

1. Train a model
2. Apply the model to predict
3. Calculate the residual based on the error function
4. Train a model to predict the residual
5. Use the original model results and all residual models to correct those results – determine error
6. Go to step 2 until you reach the error rate you are looking for

Gradient Boosting: Golf Analogy

- **Goal:** Sink a golf ball into a hole 100 yards away.
- **Swing 1 (Tree 1):** You hit the ball 70 yards.
  - *The Residual (Error):* 100 - 70 = **30 yards remaining**.
- **Swing 2 (Tree 2):** You walk up to the ball. You completely forget about the original 100-yard goal. Your *new* goal is exclusively to hit the ball exactly **30 yards**. You hit it 35 yards.
  - *The Residual (Error):* 30 - 35 = **-5 yards remaining** (You overshot).
- **Swing 3 (Tree 3):** Your new goal is to hit the ball exactly **-5 yards** (tap it backwards).
- **The Final Ensemble:** To find the final prediction, you simply add up the distance of all your swings: 70 + 35 - 5 = 100.

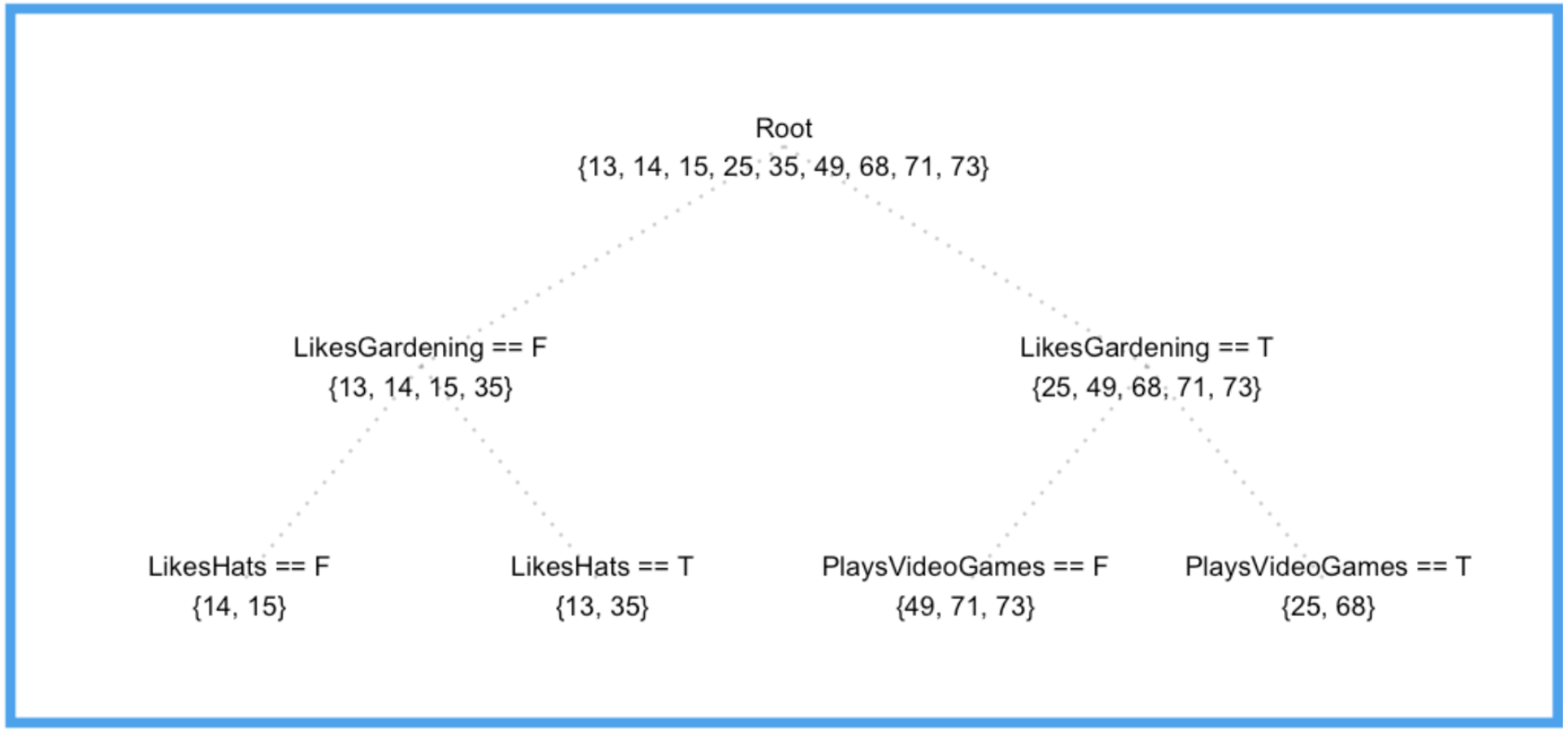
Gradient Boosting Example

PersonID	Age	LikesGardening	PlaysVideoGames	LikesHats
1	13	FALSE	TRUE	TRUE
2	14	FALSE	TRUE	FALSE
3	15	FALSE	TRUE	FALSE
4	25	TRUE	TRUE	TRUE
5	35	FALSE	TRUE	TRUE
6	49	TRUE	FALSE	FALSE
7	68	TRUE	TRUE	TRUE
8	71	TRUE	FALSE	FALSE
9	73	TRUE	FALSE	TRUE

Goal: Predict age based on other attributes

Gradient Boosting Example

Overfit Tree



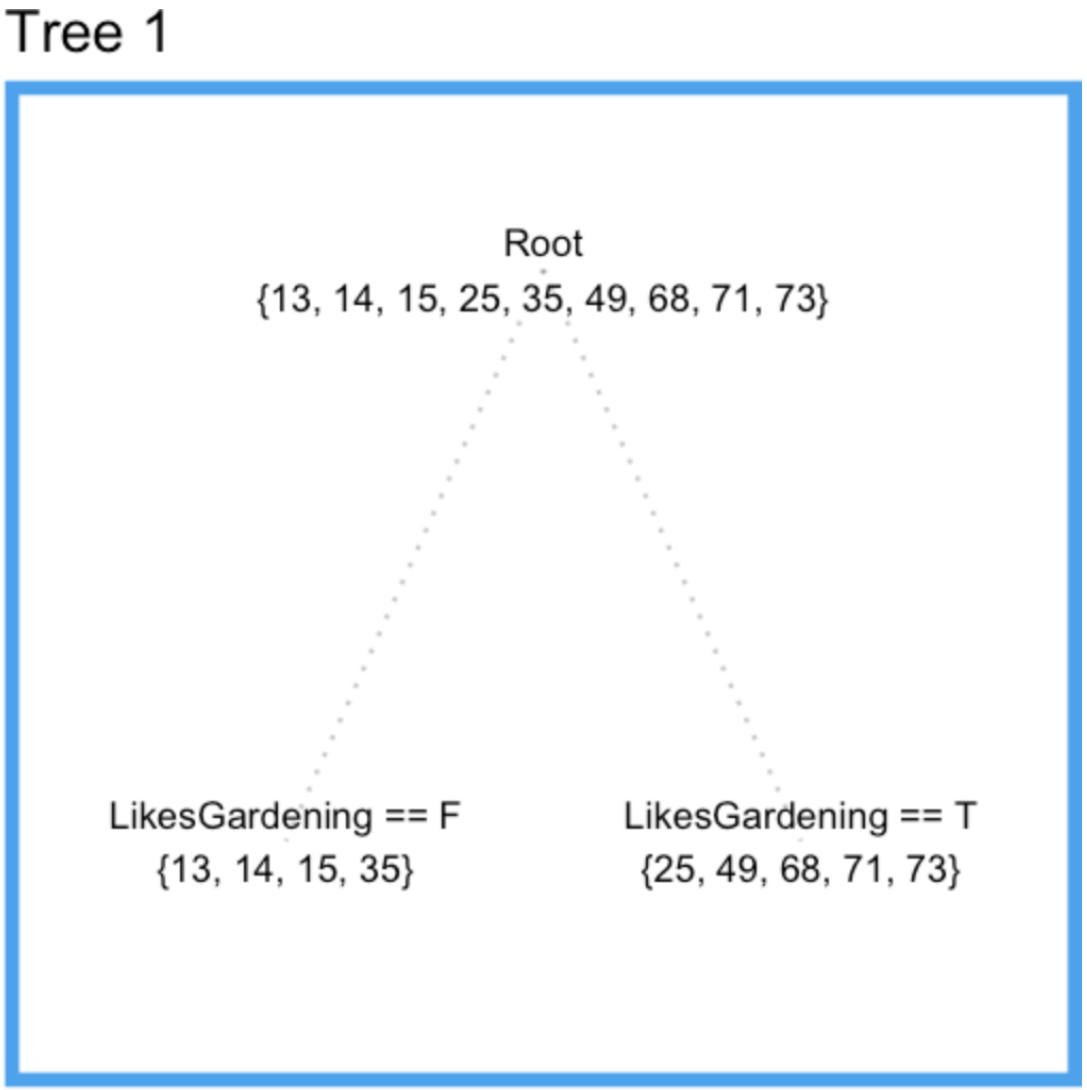
Gradient Boosting Example

Intuitively, we might expect
– The people who like gardening are probably older
– The people who like video games are probably younger
– *LikesHats* is probably just random noise

We can do a quick and dirty inspection of the data to check these assumptions:

Feature	FALSE	TRUE
LikesGardening	{13, 14, 15, 35}	{25, 49, 68, 71, 73}
PlaysVideoGames	{49, 71, 73}	{13, 14, 15, 25, 35, 68}
LikesHats	{14, 15, 49, 71}	{13, 25, 35, 68, 73}

Gradient Boosting Example



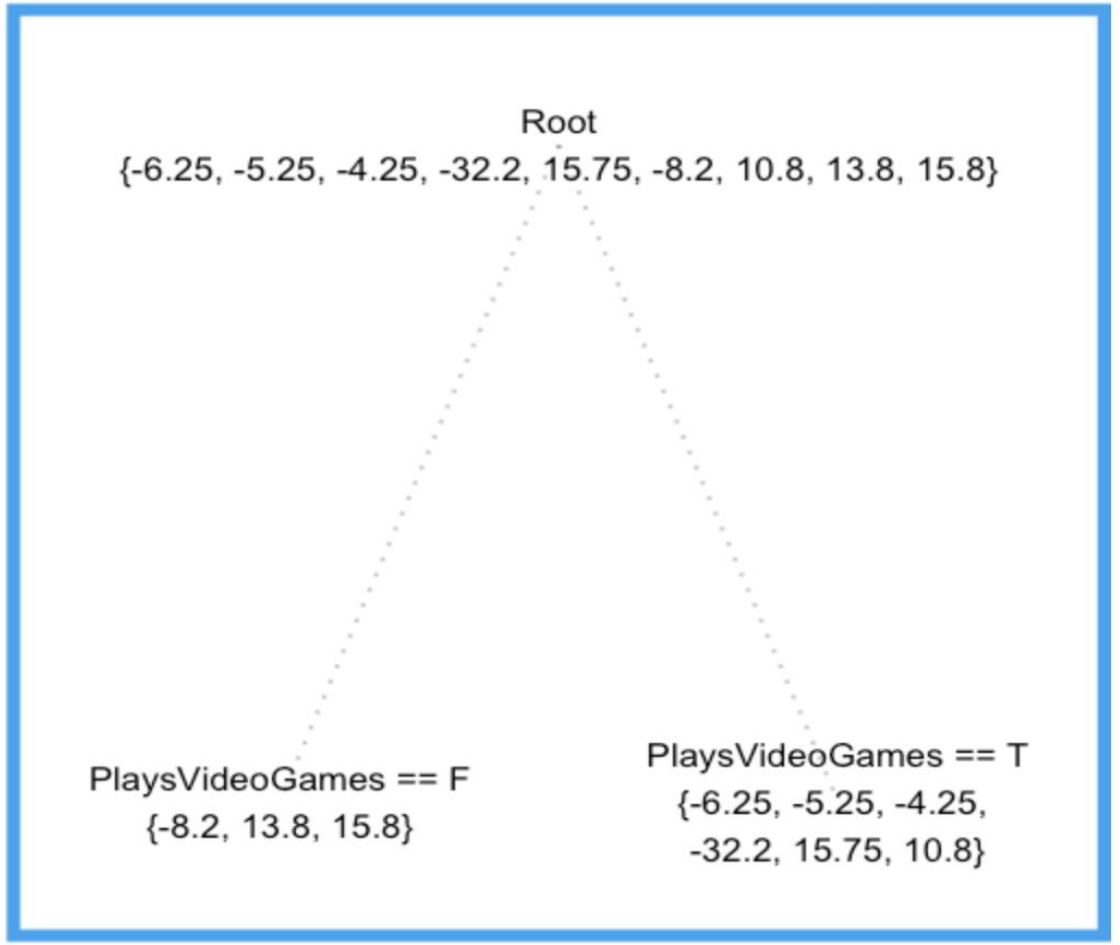
Gradient Boosting Example

PersonID	Age	Tree1 Prediction	Tree1 Residual
1	13	19.25	-6.25
2	14	19.25	-5.25
3	15	19.25	-4.25
4	25	57.2	-32.2
5	35	19.25	15.75
6	49	57.2	-8.2
7	68	57.2	10.8
8	71	57.2	13.8
9	73	57.2	15.8

Calculate Residuals – How far off to correct answer

Gradient Boosting Example

Tree2 to predict the residuals



Use the predicted residual to correct the result from the first tree

Gradient Boosting Example

Residual Tree

PersonID	Age	Tree1 Prediction	Tree1 Residual	Tree2 Prediction	Combined Prediction	Final Residual
1	13	19.25	-6.25	-3.567	15.68	2.683
2	14	19.25	-5.25	-3.567	15.68	1.683
3	15	19.25	-4.25	-3.567	15.68	0.6833
4	25	57.2	-32.2	-3.567	53.63	28.63
5	35	19.25	15.75	-3.567	15.68	-19.32
6	49	57.2	-8.2	7.133	64.33	15.33
7	68	57.2	10.8	-3.567	53.63	-14.37
8	71	57.2	13.8	7.133	64.33	-6.667
9	73	57.2	15.8	7.133	64.33	-8.667

Tree1 SSE	Combined SSE
1994	1765

Gradient Boosting

- Generalize this idea to gradient descent
- Works with any differentiable loss function

Initialize the model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$$

For m = 1 to M:

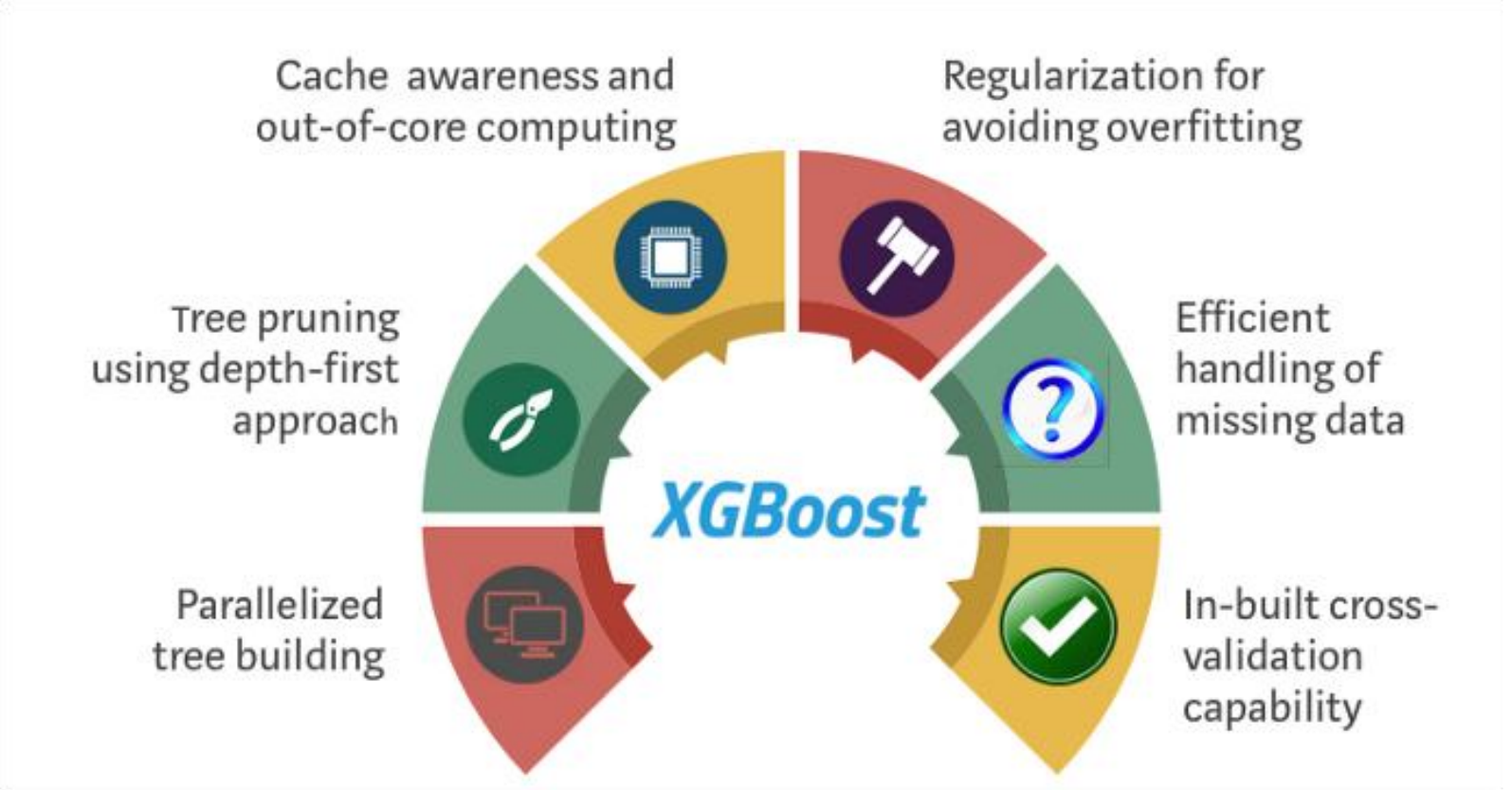
Compute *pseudo* residuals, $r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$ for $i = 1, \ldots, n$.

Fit base learner, $h_m(x)$ to pseudo residuals

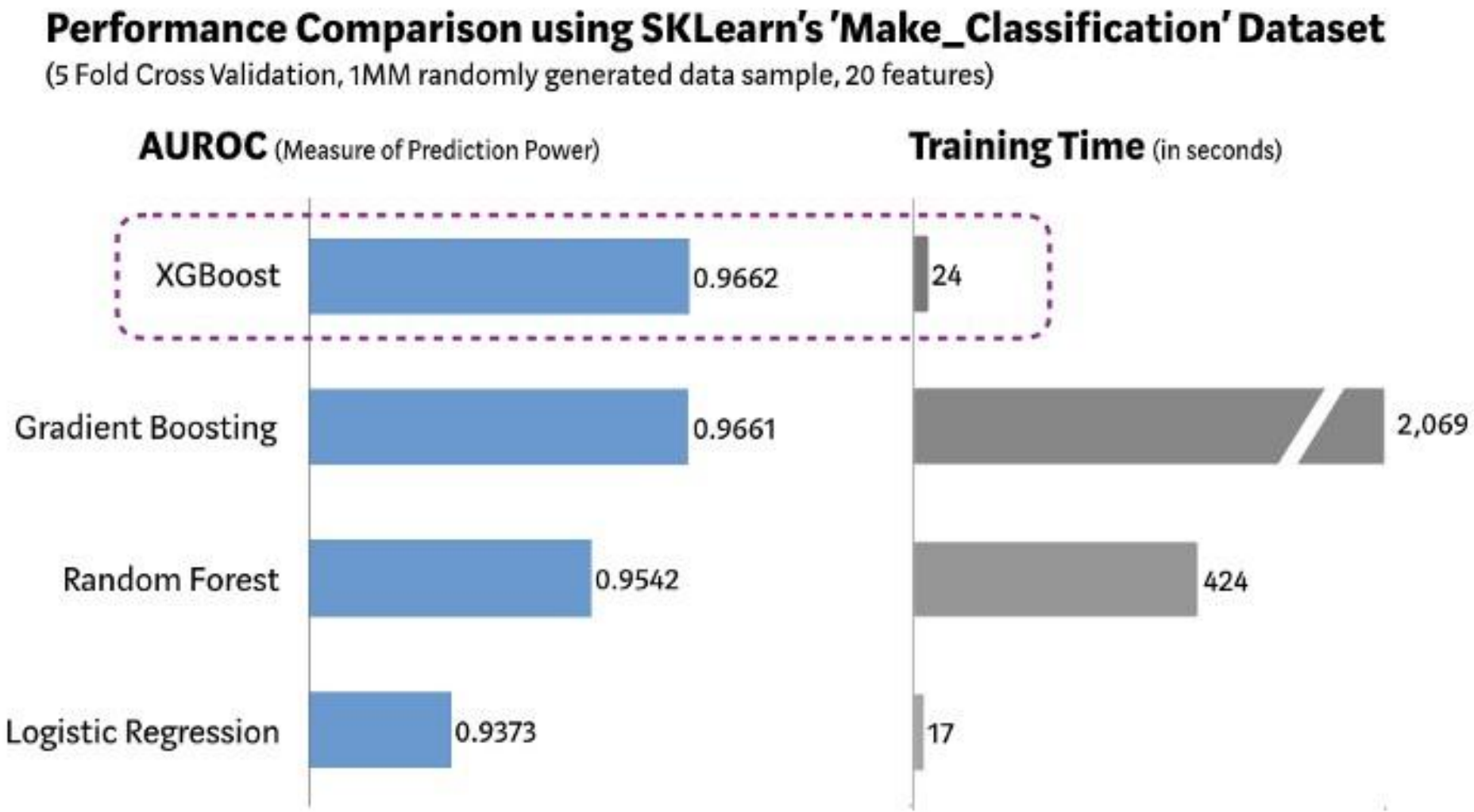
Compute step magnitude multiplier γ_m. (In the case of tree models, compute a different γ_m for every leaf.)

Update $F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$

- Add in a learning rate to scale and adjust as you get closer
- XGBoost and LightGBM



- Gradient Boosting Decision Trees
- Adds in L1 and L2 regularization
  - Like LASSO and Ridge regression
- With the right hyperparameters can outperform deep neural nets (much faster to train)



"When in doubt, use XGBoost"

— Owen Zhang, Winner of Avito Context Ad Click Prediction competition on Kaggle

XGBoost

- XGBoost has **a lot of hyperparameters** to tune
- Categories
  - General parameters – overall functioning
    - booster (gbtree), silent, nthread (set to your number of cores)
  - Booster parameters
  - Learning Task parameters
- Start with the defaults

Quiz