Ensemble Learning

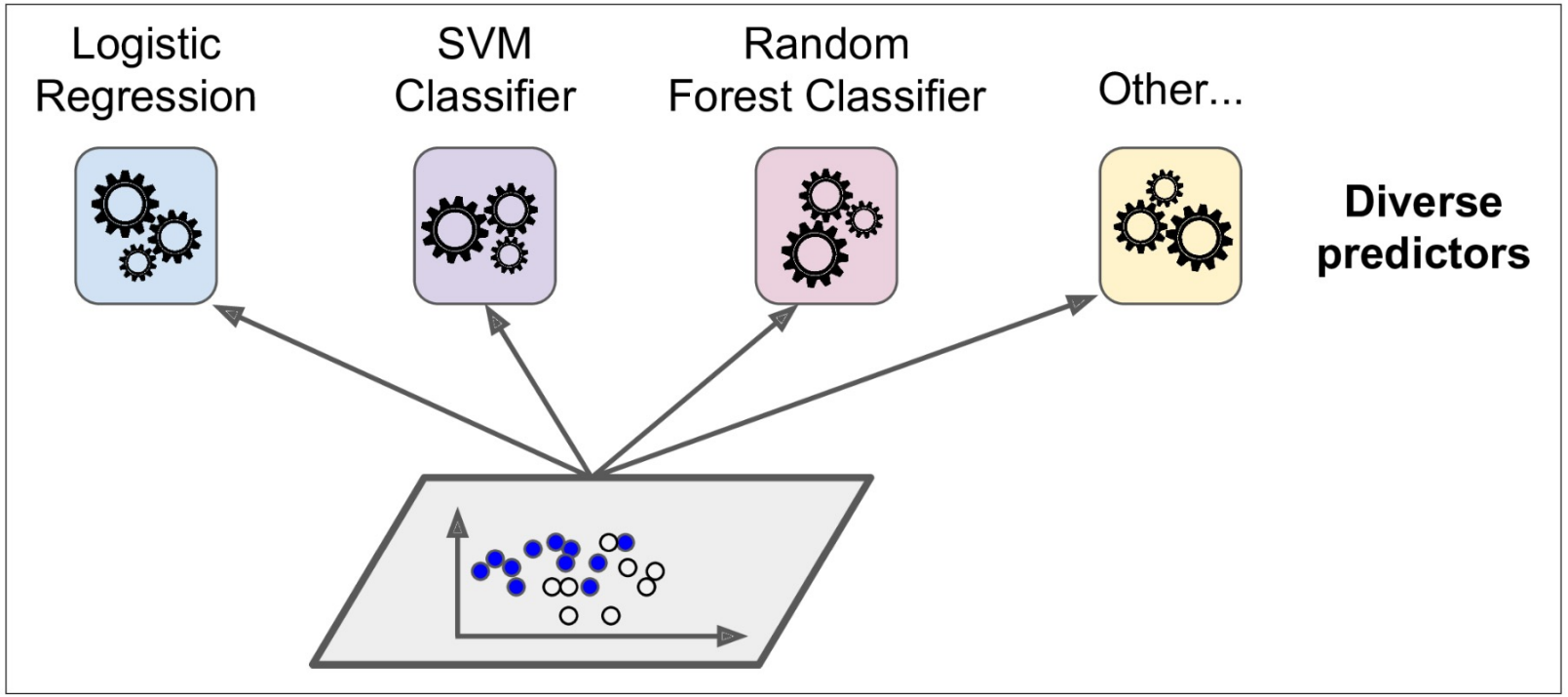
Introduction

- Suppose you ask a complex question to thousands of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert's answer. This is called the *wisdom of the crowd*.

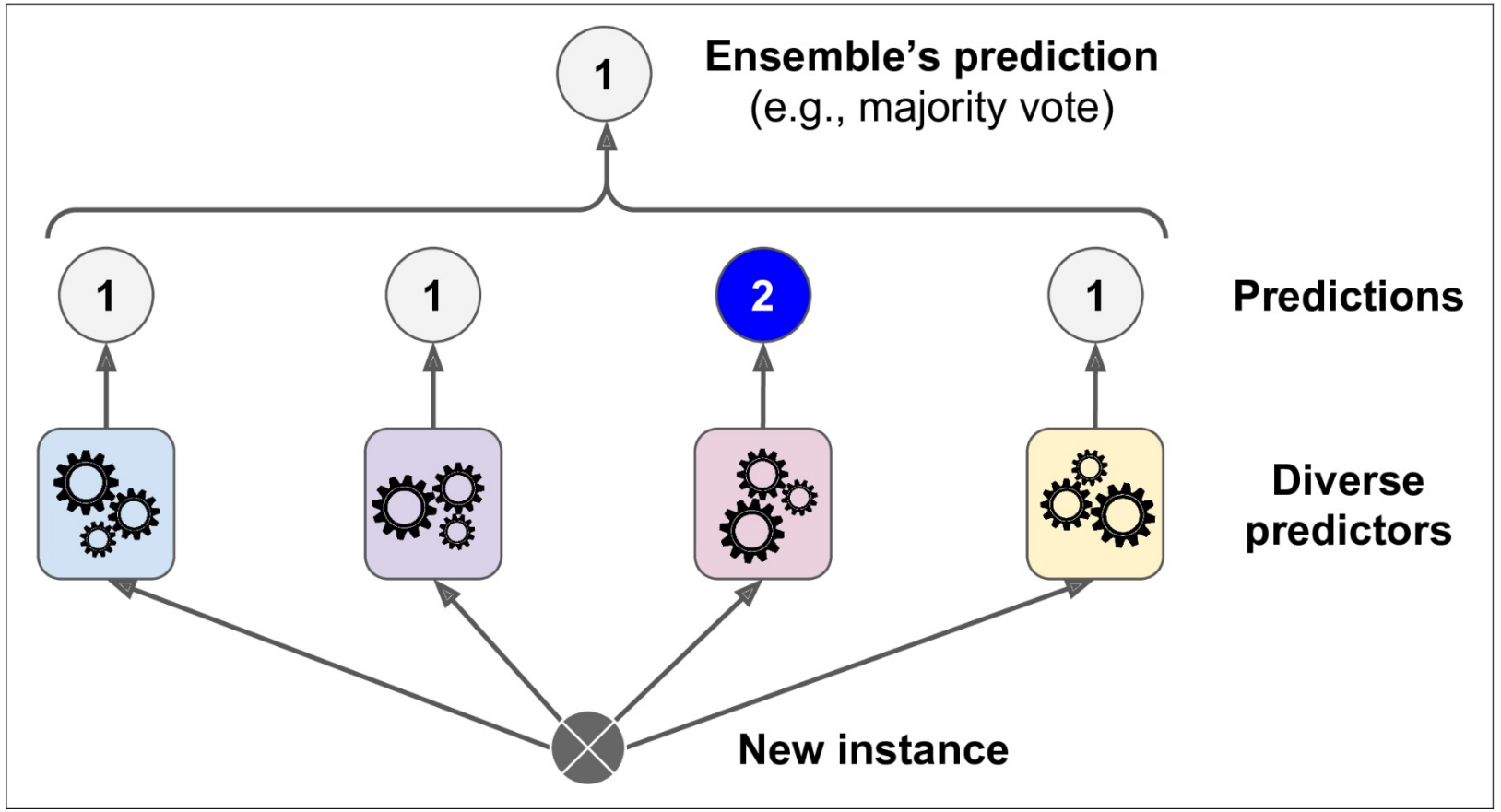
- the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an *ensemble*; thus, this technique is called *Ensemble Learning*, and an Ensemble Learning algorithm is called an *Ensemble method*.
- For example, you can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, you just obtain the predic- tions of all individual trees, then predict the class that gets the most votes

Voting Classifiers

- Suppose you have trained a few classifiers, each one achieving about 80% accuracy. You may have a Logistic Regression classifier, an SVM classifier, a Random Forest classifier, a K-Nearest Neighbors classifier, and perhaps a few more



- A very simple way to create an even better classifier is to aggregate the predictions of each classifier and predict the class that gets the most votes. This majority-vote classifier is called a *hard voting* classifier



See Lesson 5.pynb

- Code Cells

- `5.1 VoterClassifer`
- `5.2 VoterClassifer`

Bagging and Pasting

- One way to get a diverse set of classifiers is to use very different training algorithms, as just discussed.
- Another approach is to use the same training algorithm for every predictor, but to train them on different random subsets of the training set.
- When sampling is performed *with* replacement, this method is called *bagging* (short for *bootstrap aggregating*).
- When sampling is performed *without* replacement, it is called *pasting*.
- In other words, both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor.
- Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors. The aggregation function is typically the *statistical mode* (i.e., the most frequent prediction, just like a hard voting classifier) for classification, or the average for regression.

- Bootstrapping introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting, but this also means that predictors end up being less correlated so the ensemble's variance is reduced. Overall, bagging often results in better models, which explains why it is generally preferred. However, if you have spare time and CPU power you can use cross-validation to evaluate both bagging and pasting and select the one that works best.

- See Lesson 5.pynb
- Code Cell
- `5.3 Bagging`

Out-of-Bag Evaluation

- With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all. By default a BaggingClassifier samples *m* training instances with replacement (bootstrap=True), where *m* is the size of the training set.
- This means that only about 63% of the training instances are sampled on average for each predictor. The remaining 37% of the training instances that are not sampled are called *out-of-bag* (oob) instances. Note that they are not the same 37% for all predictors.
- Since a predictor never sees the oob instances during training, it can be evaluated on these instances, without the need for a separate validation set. You can evaluate the ensemble itself by averaging out the oob evaluations of each predictor.

Random Patches and Random Subspaces

- The BaggingClassifier class supports sampling the features as well. This is controlled by two hyperparameters: max_features and bootstrap_features.
- They work the same way as max_samples and bootstrap, but for feature sampling instead of instance sampling. Thus, each predictor will be trained on a random subset of the input features.
- This is particularly useful when you are dealing with high-dimensional inputs (such as images). Sampling both training instances and features is called the *Random Patches* method.

- Keeping all training instances (i.e., bootstrap=False and max_samples=1.0) but sampling features (i.e., bootstrap_features=True and/or max_features smaller than 1.0) is called the *Random Subspaces* method.

Random Patches and Random Subspaces Methods

- Random Patches Method: This method involves sampling both instances (rows) and features (columns) randomly. It's especially useful when the dataset is large or has many features. By setting both bootstrap and bootstrap_features to True (or False for pasting without replacement), and adjusting max_samples and max_features to values less than 1.0, you engage this method. This allows models to train on different random patches (subsets) of the data.
- Random Subspaces Method: This technique focuses solely on feature sampling while using all instances. By setting bootstrap=False and max_samples=1.0 (using all data points), and enabling bootstrap_features=True with max_features less than 1.0, you effectively train each model in the ensemble on all instances but only a subset of features. This can be beneficial for dealing with feature redundancy or when feature dimensionality is very high, as it helps in increasing the diversity of the ensemble without the risk of losing important information by excluding too many instances.

Random Forests

- As we have discussed, a Random Forest is an ensemble of Decision Trees, generally trained via the bagging method (or sometimes pasting), typically with max_samples set to the size of the training set. Instead of building a BaggingClassifier and passing it a DecisionTreeClassifier, you can instead use the RandomForestClassifier class, which is more convenient and optimized for Decision Trees (similarly, there is a RandomForestRegressor class for regression tasks).
- With a few exceptions, a RandomForestClassifier has all the hyperparameters of a DecisionTreeClassifier (to control how trees are grown), plus all the hyperparameters of a BaggingClassifier to control the ensemble itself.
- The Random Forest algorithm introduces extra randomness when growing trees; instead of searching for the very best feature when splitting a node , it searches for the best feature among a random subset of features. This results in a greater tree diversity, which (once again) trades a higher bias for a lower variance, generally yielding an overall better model.

See Lesson 5.pynb

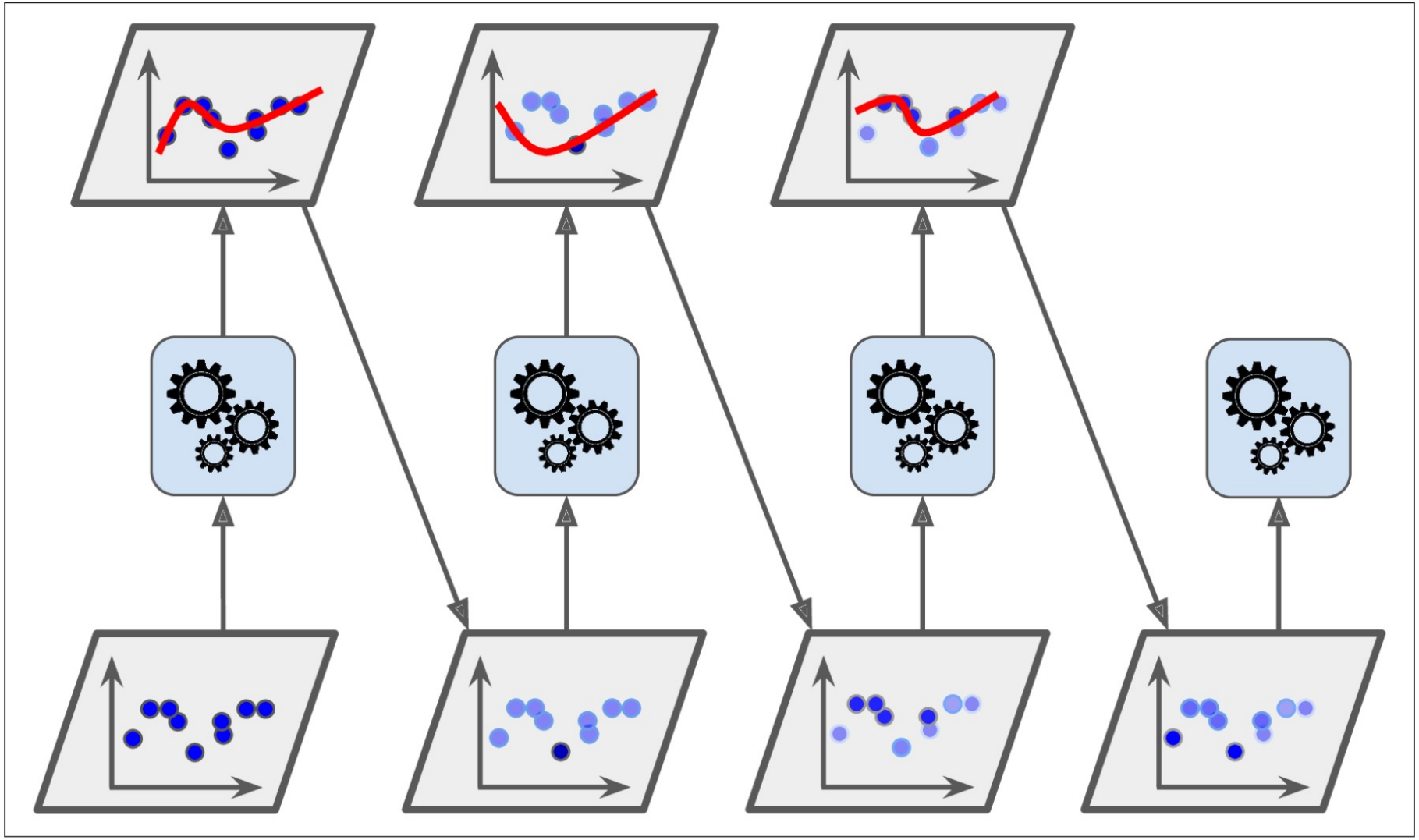
- Code Cell
- `5.4 Random Forests`

Boosting

- *Boosting* (originally called *hypothesis boosting*) refers to any Ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor.
- There are many boosting methods available, but by far the most popular are
- *AdaBoost* (short for *Adaptive Boosting*) and *Gradient Boosting*. Let's start with AdaBoost.

AdaBoost

- One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost.
- For example, to build an AdaBoost classifier, a first base classifier (such as a Decision Tree) is trained and used to make predictions on the training set. The relative weight of misclassified training instances is then increased. A second classifier is trained using the updated weights and again it makes predictions on the training set, weights are updated, and so on.
- Once all predictors are trained, the ensemble makes predictions very much like bagging or pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.



- The idea behind AdaBoost is to improve the classification accuracy by training a series of weak classifiers, each focusing on the mistakes of its predecessors, and combining their decisions to produce the final prediction.
- Here's how AdaBoost generally works:

1. **Initialize Weights**: Start by assigning equal weights to all instances in the training dataset.
2. **Train Weak Classifiers**: Iteratively train weak classifiers (usually simple models like decision stumps). In each iteration:
    1. Train a classifier using the weighted instances.
    2. Calculate the error of the classifier based on these weights.
    3. Assign more weight to instances that were misclassified, making it more likely for future classifiers to focus on these.
3. **Calculate Classifier Weights**: Assign a weight to each classifier based on its accuracy. More accurate classifiers have higher weights.
4. **Combine Classifiers**: Aggregate the decisions of all the classifiers, weighted by their respective accuracies, to make the final prediction.

- Weak classifiers, also known as weak learners, are simple models used in ensemble techniques like boosting. They are called "weak" because their performance is only slightly better than random guessing, but when used as part of a larger ensemble method like AdaBoost, they contribute to a much stronger overall classifier.
- Here are some characteristics and examples of weak classifiers:

1. **Simplicity**: Weak classifiers are generally simple due to their limited complexity and decision-making capabilities. They often use a simple rule or decision boundary to classify data.
2. **Examples**:
    1. **Decision Stumps**: These are decision trees with only one decision node (the root) and two leaves. They make a decision based on the value of just one input feature.
    2. **Small Decision Trees**: Trees with a few splits are slightly more complex than stumps but still considered weak.
    3. **Perceptrons**: As a simple form of neural networks, perceptrons can act as weak classifiers when they categorize data using a single linear boundary.
3. **Error Rate**: The key requirement for a weak classifier is that its error rate should be better than flipping a coin, meaning it should be correct more than 50% of the time on binary classification tasks.

- In AdaBoost and other ensemble boosting methods, weights are assigned to each instance in the dataset to indicate the importance of correctly classifying each instance during the training of subsequent classifiers. The process of updating these weights is crucial to the success of boosting methods. Here's how it typically works:
- **Initial Weight Assignment**

1. **Initial Weights**: Initially, all instances in the training dataset are given the same weight. For a dataset with N instances, each instance i is assigned a weight

2. $w_i=\frac{1}{N}$.

- **Updating Weights in Each Iteration**

2. **Training**: Train a weak classifier on the weighted instances. The goal for the classifier is to minimize the weighted error, where the error is calculated based on the weights of the misclassified instances.
3. **Calculate Error Rate**: After training the classifier, compute its error rate: $$\epsilon = \frac{\sum_{i=1}^{N} w_i \times \text{error}_i}{\sum_{i=1}^{N} w_i}$$

Here, error_i is an indicator that is 1 if the classification is wrong and 0 if it is correct.

- 4. **Compute Classifier Weight**: The weight α of the classifier is calculated using the formula:

$$\alpha = \frac{1}{2} \ln\left(\frac{1-\epsilon}{\epsilon}\right)$$

This value determines the influence of the classifier in the final decision. Classifiers with lower error rates have higher weights.

5. **Update Instance Weights**:

   Increase Weights of Misclassified Instances: For each instance that is misclassified by the classifier, update the weight: $w_i \leftarrow w_i \times e^{\alpha}$

   Decrease Weights of Correctly Classified Instances: For each instance that is correctly classified, update the weight: $w_i \leftarrow w_i \times e^{-\alpha}$

This adjustment makes misclassified instances more prominent in the dataset, encouraging subsequent classifiers to focus on them.

6. Normalize Weights: After updating the weights, they are normalized so that the sum of all weights equals 1. This normalization helps in maintaining a consistent scale of weights across all instances:

$$w_i \leftarrow \frac{w_i}{\sum_{j=1}^{N} w_j}$$

7. Repeat: This process of training weak classifiers and updating weights is repeated for a predefined number of iterations or until the error of the ensemble falls below a certain threshold. Each new classifier focuses more on the instances that previous classifiers misclassified.

- In AdaBoost and similar boosting algorithms, the mechanism by which weights influence the learning process is crucial for understanding how the algorithm directs focus towards harder-to-classify instances and handles those that are easier.
- **Influence of High Weights**
- When an instance in the training dataset is misclassified by a weak learner, its weight is increased. This means that in the overall cost function that the next weak learner aims to minimize, these instances now play a more significant role. The cost function in many machine learning models, particularly those used in boosting like decision trees, is sensitive to the weights of the instances:
- **Higher Weights Increase Influence**: When a weak learner is trained, it typically tries to minimize some form of weighted error. This error is calculated as a sum where each term is the product of an instance's weight and the error for that instance. Thus, instances with higher weights contribute more to the total error. If these instances are misclassified, they result in a larger penalty, prompting the learner to focus on getting these instances right to minimize the overall weighted error.

- **Effect on the Learning Process**
- **Weighted Sampling or Cost Adjustment**: Depending on the implementation, the algorithm might sample instances with a probability proportional to their weights for training subsequent classifiers, or it might adjust the cost of misclassification based on weights directly. In either case, instances with higher weights are either more likely to be included in the training set or they make the error costlier if misclassified again.
- **Handling of Low Weight Instances**
- **Decreased Influence**: Conversely, instances that are correctly classified see their weights decreased. This reduction in weight means that these instances become less influential in the training process of subsequent learners. They contribute less to the error term, and as a result, the model may not focus as much on getting these already correctly classified instances right.
- **Risk of Underfitting**: While the emphasis on hard-to-classify instances helps to improve the model's performance on difficult parts of the data, there is a potential risk. If the weights of some instances become too small, they might be effectively ignored by the model, possibly leading to underfitting on these parts of the data. However, because the weights are normalized so that their sum remains constant, no instance's weight becomes absolute zero, ensuring they still have some influence.
- **Dynamic Adjustment Ensures Balance**
- The dynamic adjustment of weights after each iteration ensures that the training focus can shift back and forth between different parts of the data as needed. If a previously hard-to-classify instance starts getting classified correctly, its weight decreases, and other still-misclassified instances' weights increase, shifting the focus continually. This adaptability helps AdaBoost and similar methods remain balanced, focusing on different challenges in the dataset over the course of many iterations.

Example

- we'll start with a small dataset of 5 instances, each initially with equal weights, and go through one iteration of training a weak classifier and updating weights.
- **Step 1: Initial Setup**
- Suppose we have 5 instances in our dataset, and each instance initially receives the same weight.
- Number of instances, *N*=5
- Initial weights for each insta $w_i = \frac{1}{N} = \frac{1}{5} = 0.2$

- **Step 2: Assume Classifier Outcome**
- Let's say we have trained our first weak classifier, and the outcomes are as follows:
- Instance 1: Correctly classified
- Instance 2: Misclassified
- Instance 3: Correctly classified
- Instance 4: Misclassified
- Instance 5: Correctly classified

- **Step 3: Calculate the Weighted Error Rate (ϵ)**
- The error for each instance ($error_i$) is 1 if misclassified, and 0 if correctly classified.

$$\epsilon = \sum_{i=1}^{N} w_i \times \text{error}_i = 0.2 \times 0 + 0.2 \times 1 + 0.2 \times 0 + 0.2 \times 1 + 0.2 \times 0 = 0.4$$

- **Step 4: Calculate the Classifier's Weight (α)**
- The weight of the classifier is calculated using:

$$\alpha = \frac{1}{2} \ln\left(\frac{1-\epsilon}{\epsilon}\right) = \frac{1}{2} \ln\left(\frac{1-0.4}{0.4}\right) = \frac{1}{2} \ln\left(\frac{0.6}{0.4}\right) = \frac{1}{2} \ln(1.5) \approx 0.2027$$

- **Step 5: Update the Weights**
- Weights are updated based on whether instances were misclassified or not:
- If correctly classified , update weight: $w_i \leftarrow w_i \times e^{\alpha}$
- If misclassified, update weight: $w_i \leftarrow w_i \times e^{-\alpha}$
- Calculating new weights:
- Instance 1: $0.2 \times e^{-0.2027} \approx 0.2 \times 0.8165 \approx 0.1633$
- Instance 2: $0.2 \times e^{0.2027} \approx 0.2 \times 1.2247 \approx 0.2449$
- Instance 3: $0.2 \times e^{-0.2027} \approx 0.1633$
- Instance 4: $0.2 \times e^{0.2027} \approx 0.2449$
- Instance 5: $0.2 \times e^{-0.2027} \approx 0.1633$

- **Step 6: Normalize the Weights**
- Sum of new weights: 0.1633+0.2449+0.1633+0.2449+0.1633=0.97970.1633+0.2449+0.1633+0.2449+0.1633=0.9797
- Normalized weights:
- Instance 1: 0.1633/0.9797≈0.1667
- Instance 2: 0.2449/0.9797≈0.2500
- Instance 3: 0.1633/0.9797≈0.1667
- Instance 4: 0.24490/9797≈0.2500
- Instance 5: 0.16330/9797≈0.1667

See Lesson 5.pynb

- Code Cell

- `5.5 adaboost`

Gradient Boosting

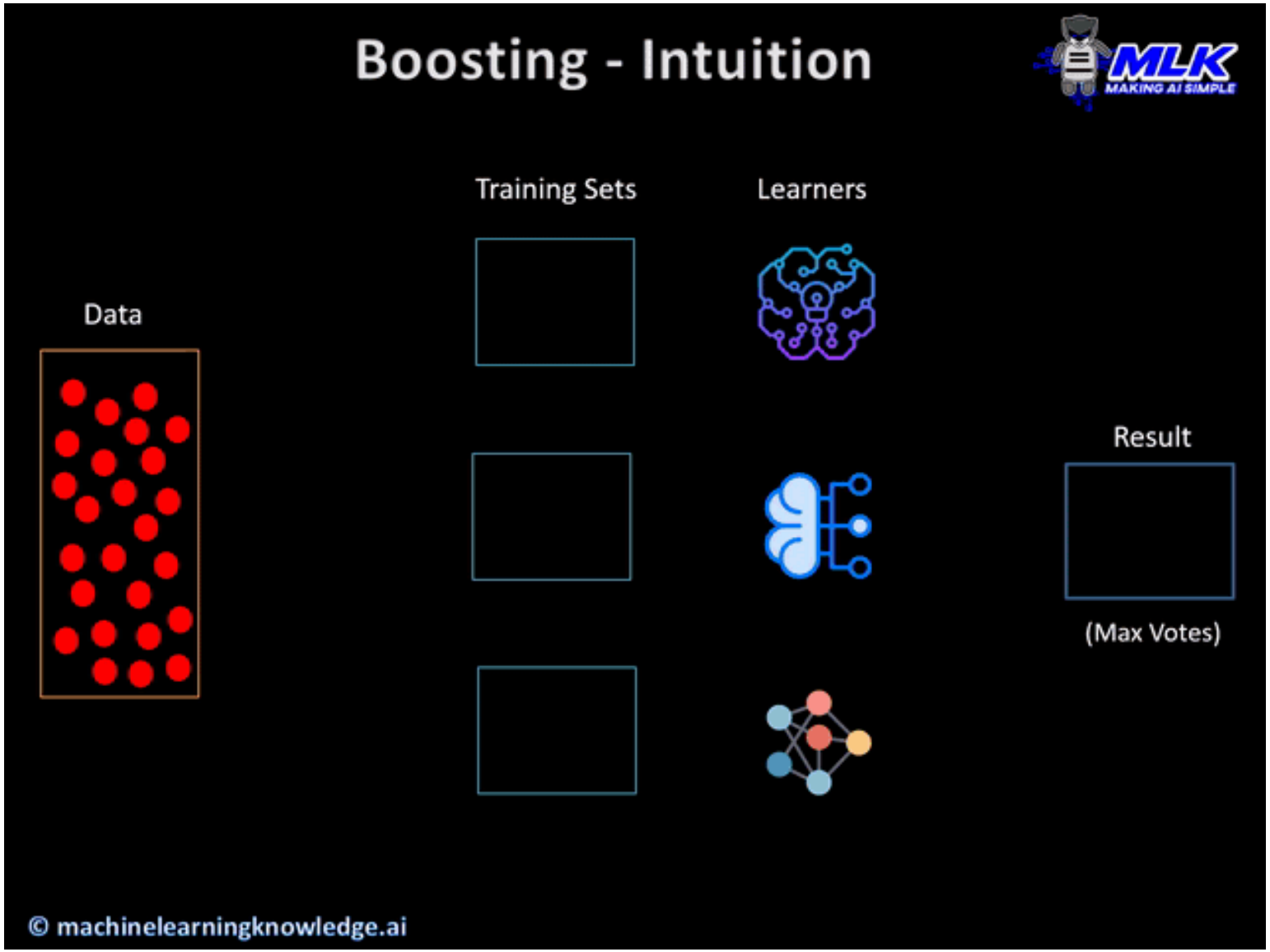
- Another very popular Boosting algorithm is *Gradient Boosting*. Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.

- See Lesson 5.pynb
- Code Cell
- `5.5 Gradient Boosting`

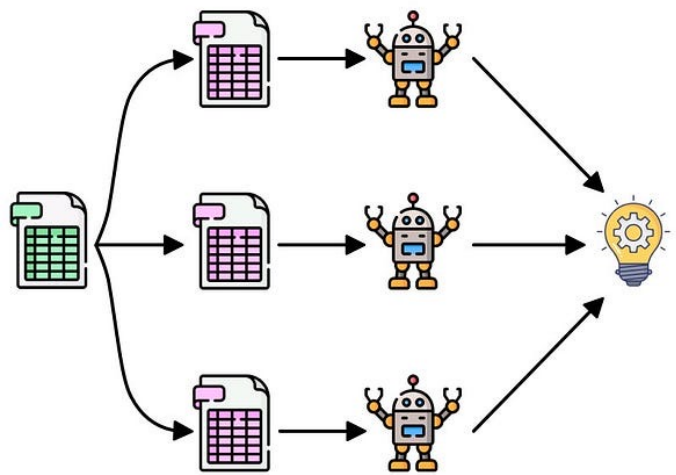
Algorithm

1. **Initial Prediction**: Start with a basic prediction for each instance. This could be something simple like the mean of the target values. This is your initial prediction for all instances.
2. **Calculate Residuals**: Compute the residuals for each instance. The residual is the difference between the actual target value and your current prediction. Essentially, it represents how much your model is off for each instance.
3. **Train Model on Residuals**: Train a new model, not on the original target, but on these residuals. The goal of this model is to predict these residuals based on the input features. This means that this model is trying to learn the errors of the previous model.
4. **Update Predictions**: Use the predictions from this new model (which are predictions of the residuals) to update your initial predictions. You adjust your existing predictions by adding the predicted residuals (usually scaled by a learning rate) to them. This gives you a new set of improved predictions.
5. **Recalculate Residuals**: With your updated predictions, calculate new residuals. These new residuals are again the differences between the actual target values and your updated predictions.
6. **Repeat**: Train another model on these new residuals and use its output to update your predictions again. This cycle continues for a fixed number of iterations, or until the improvements become negligibly small.
7. **Final Model**: The ensemble of all these sequentially trained models, each correcting the predecessor's errors, forms your final predictive model.

Boosting - Intuition

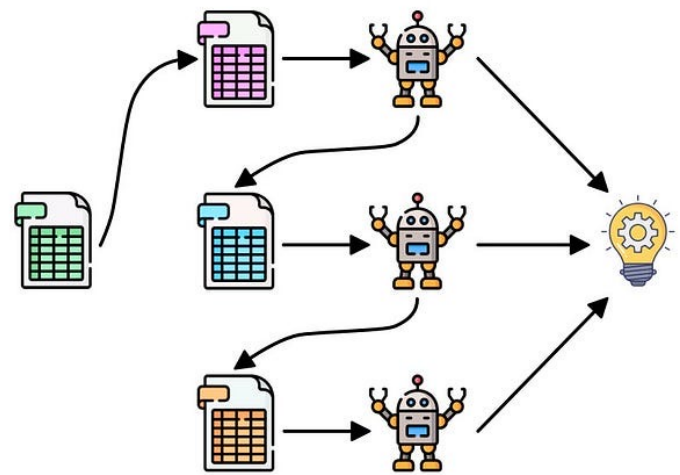


Bagging



Parallel

Boosting



Sequential

Example Scenario: Predicting Exam Outcomes

- Dataset
- We have a dataset that records study hours and whether a student passed (1) or failed (0) an exam:

Student ID	Study Hours	Passed Exam
1	1	0
2	3	0
3	4	1
4	6	1
5	8	1

- Step 1: Initial Model
- The initial model predicts the probability of passing, typically the mean of the target variable (Pass or Fail).

$$\text{Probability of Passing} = \frac{0+0+1+1+1}{5} = 0.6$$

- So, every prediction initially is 0.6 (60% chance of passing).
- Step 2: Calculate Residuals
- Residuals are the differences between actual outcomes and predicted probabilities.
-

Student ID	Actual	Predicted	Residual
1	0	0.6	-0.6
2	0	0.6	-0.6
3	1	0.6	0.4
4	1	0.6	0.4
5	1	0.6	0.4

- Step 3: Train a Weak Learner on Residuals
- Let’s assume a decision tree splits on study hours to predict residuals:
-
- - If Study Hours ≤ 2: Residual = -0.6 (captures students who studied very little)
- - If Study Hours > 2: Residual = 0.4 (captures students who studied more)
-
- Step 4: Update Predictions
- Using a learning rate of α= 0.1, we update the predictions as follows:

$$\text{New Prediction} = \text{Old Prediction} + (\alpha \times \text{Tree Prediction})$$

- Updating the predictions:

Student ID	Old Prediction	Tree Prediction	Learning Rate (0.1)	New Prediction
1	0.6	-0.6	0.1	0.6 - 0.06 = 0.54
2	0.6	-0.6	0.1	0.6 - 0.06 = 0.54
3	0.6	0.4	0.1	0.6 + 0.04 = 0.64
4	0.6	0.4	0.1	0.6 + 0.04 = 0.64
5	0.6	0.4	0.1	0.6 + 0.04 = 0.64

- Step 5: Repeat Steps 2-4
- Continue the process, each time:
- 1. Calculating new residuals between actual and updated predictions.
- 2. Training a new decision tree on the latest residuals.
- 3. Updating the predictions using the new tree's predictions and the learning rate.

- After the first iteration, the model predictions were updated as follows:

Student ID	Old Prediction	Tree Prediction	Updated Prediction
1	0.6	-0.6	0.54
2	0.6	-0.6	0.54
3	0.6	0.4	0.64
4	0.6	0.4	0.64
5	0.6	0.4	0.64

- Step 6: Calculate New Residuals
- We calculate new residuals based on the updated predictions:
-

Student ID	Actual	Updated Prediction	New Residual
1	0	0.54	-0.54
2	0	0.54	-0.54
3	1	0.64	0.36
4	1	0.64	0.36
5	1	0.64	0.36

- Step 7: Train Another Weak Learner on the New Residuals
- Let’s assume the next decision tree finds a better split at 5 study hours:
-
- - If Study Hours < 5: Residual = -0.54 (averaging -0.54 and -0.54 for those who studied very little or moderately)
- - If Study Hours >= 5: Residual = 0.36 (captures those who studied more)
-
- Step 8: Update Predictions Again
- With the learning rate still at 0.1, update the predictions using the results from this new tree:

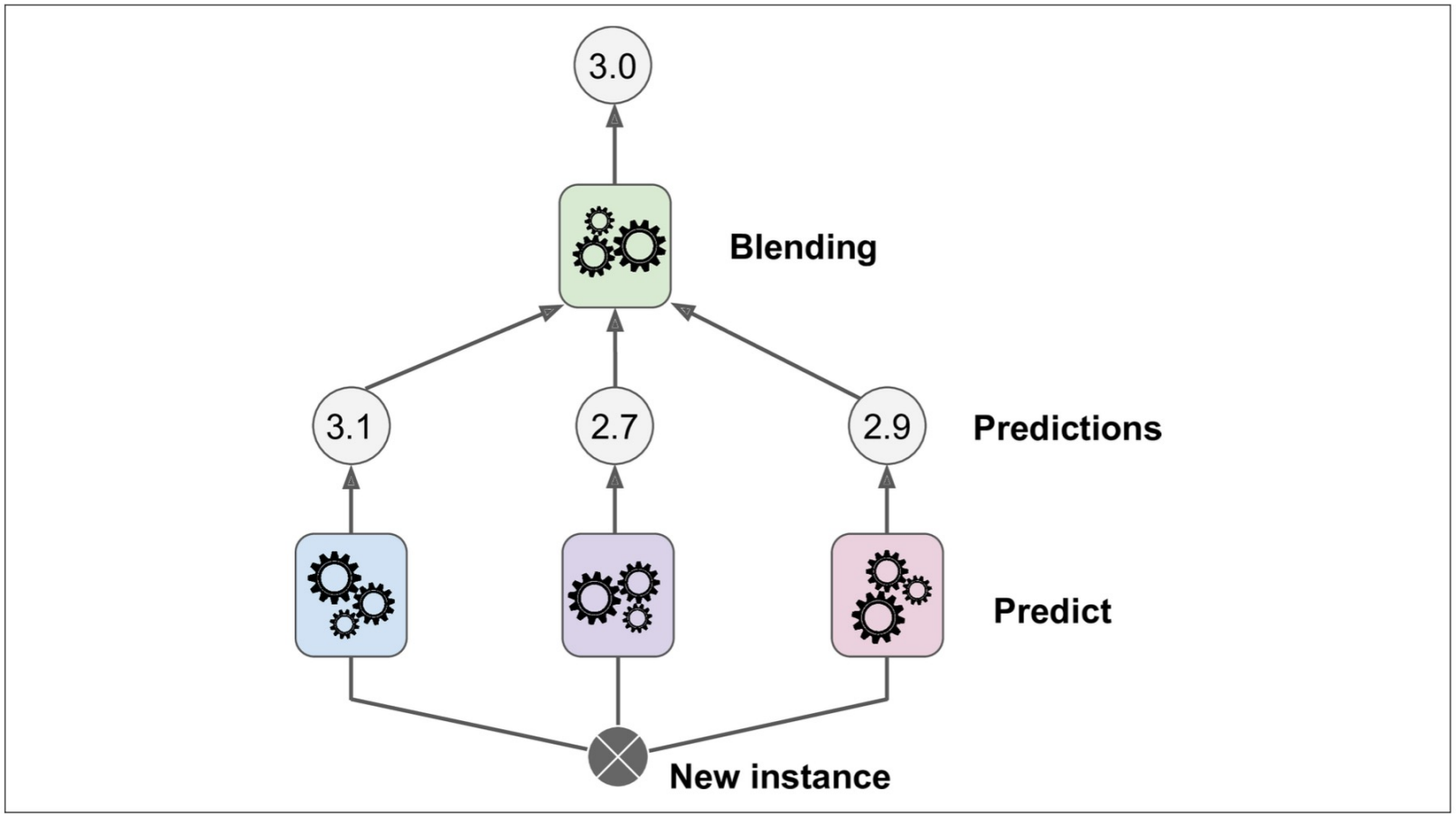
$$\text{New Prediction} = \text{Previous Prediction} + (\alpha \times \text{Tree Prediction})$$

Student ID	Previous Prediction	Tree Prediction	Learning Rate (0.1)	Updated Prediction
1	0.54	-0.54	0.1	0.54 - 0.054 = 0.486
2	0.54	-0.54	0.1	0.54 - 0.054 = 0.486
3	0.64	0.36	0.1	0.64 + 0.036 = 0.676
4	0.64	0.36	0.1	0.64 + 0.036 = 0.676
5	0.64	0.36	0.1	0.64 + 0.036 = 0.676

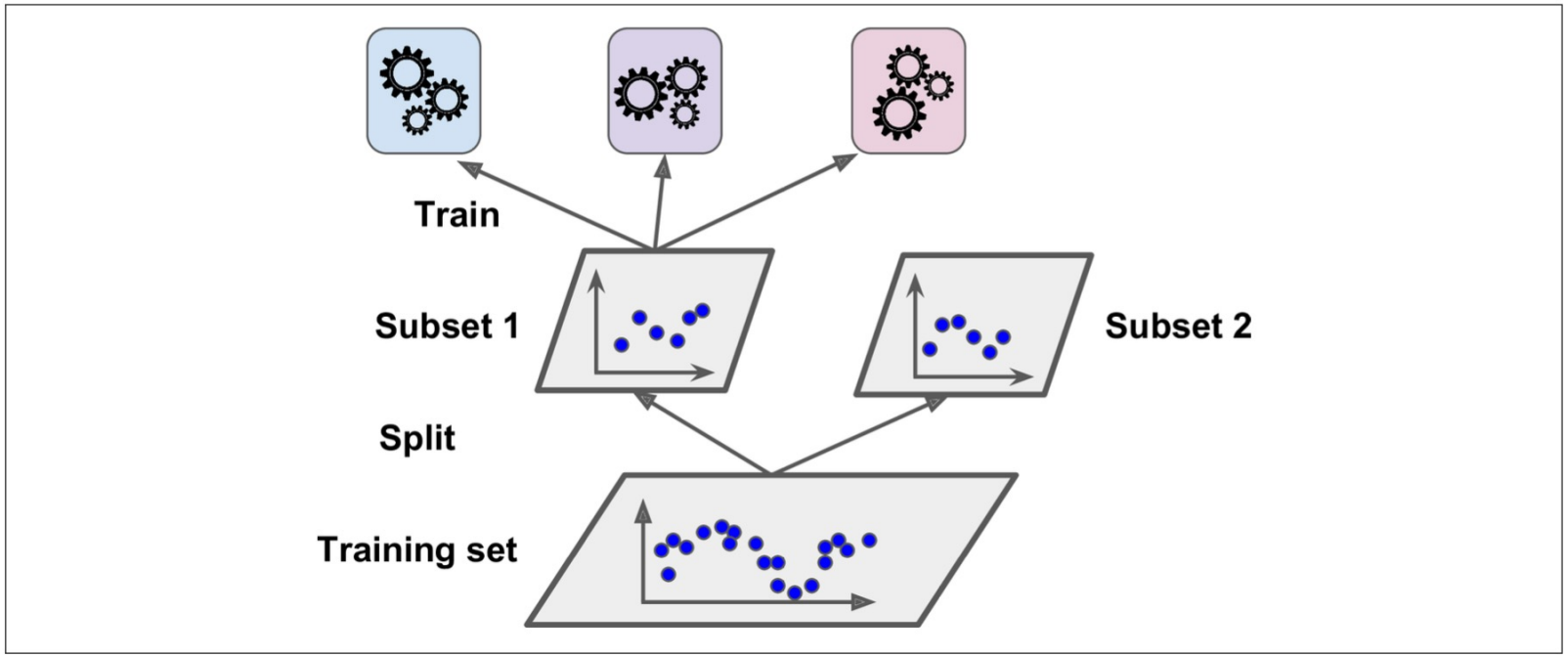
- Step 9: Repeat Steps 6-8
- This cycle is repeated, recalculating residuals and fitting new trees, until no significant improvement is observed or a predetermined number of iterations are completed.

Stacking

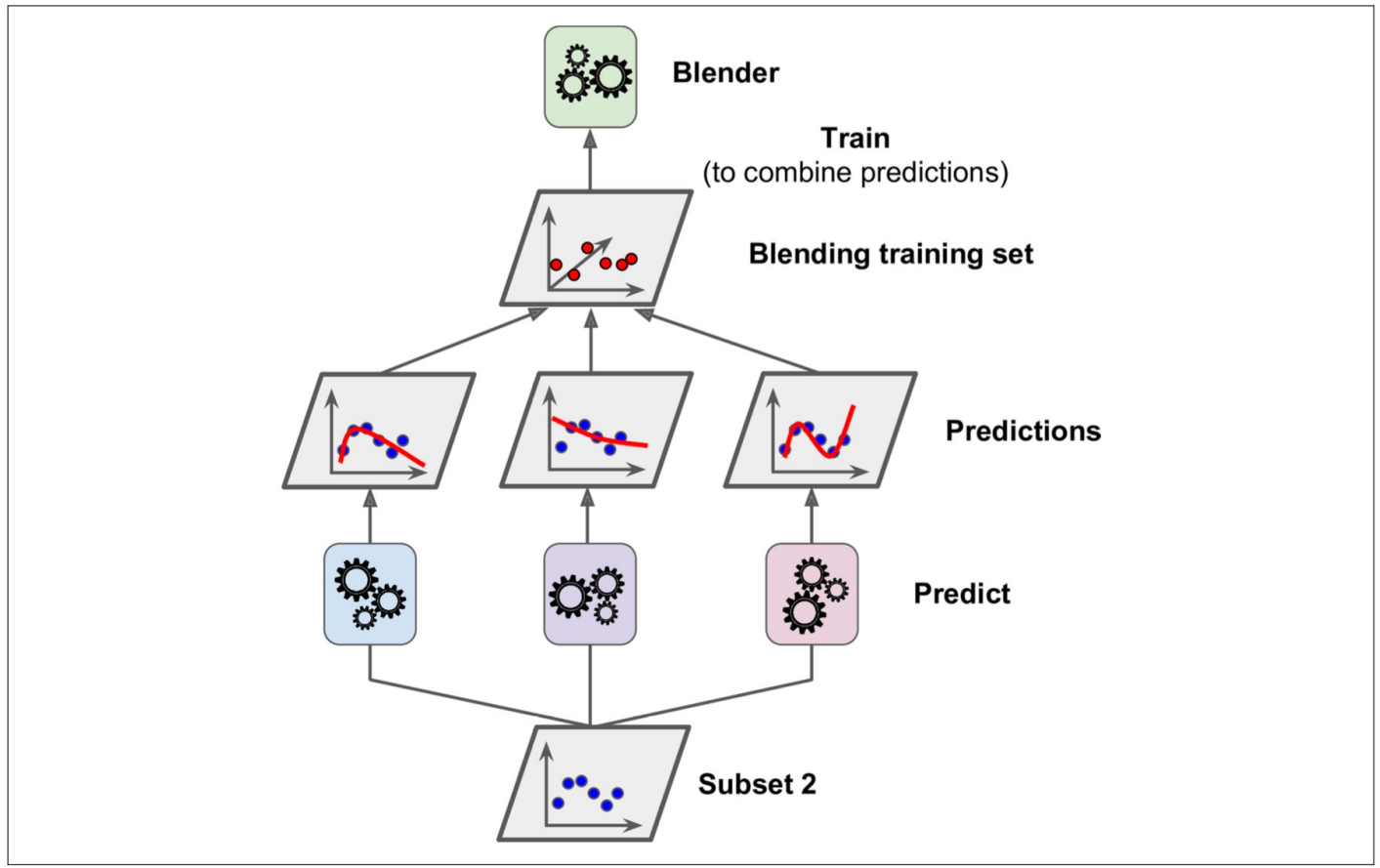
- It is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation?



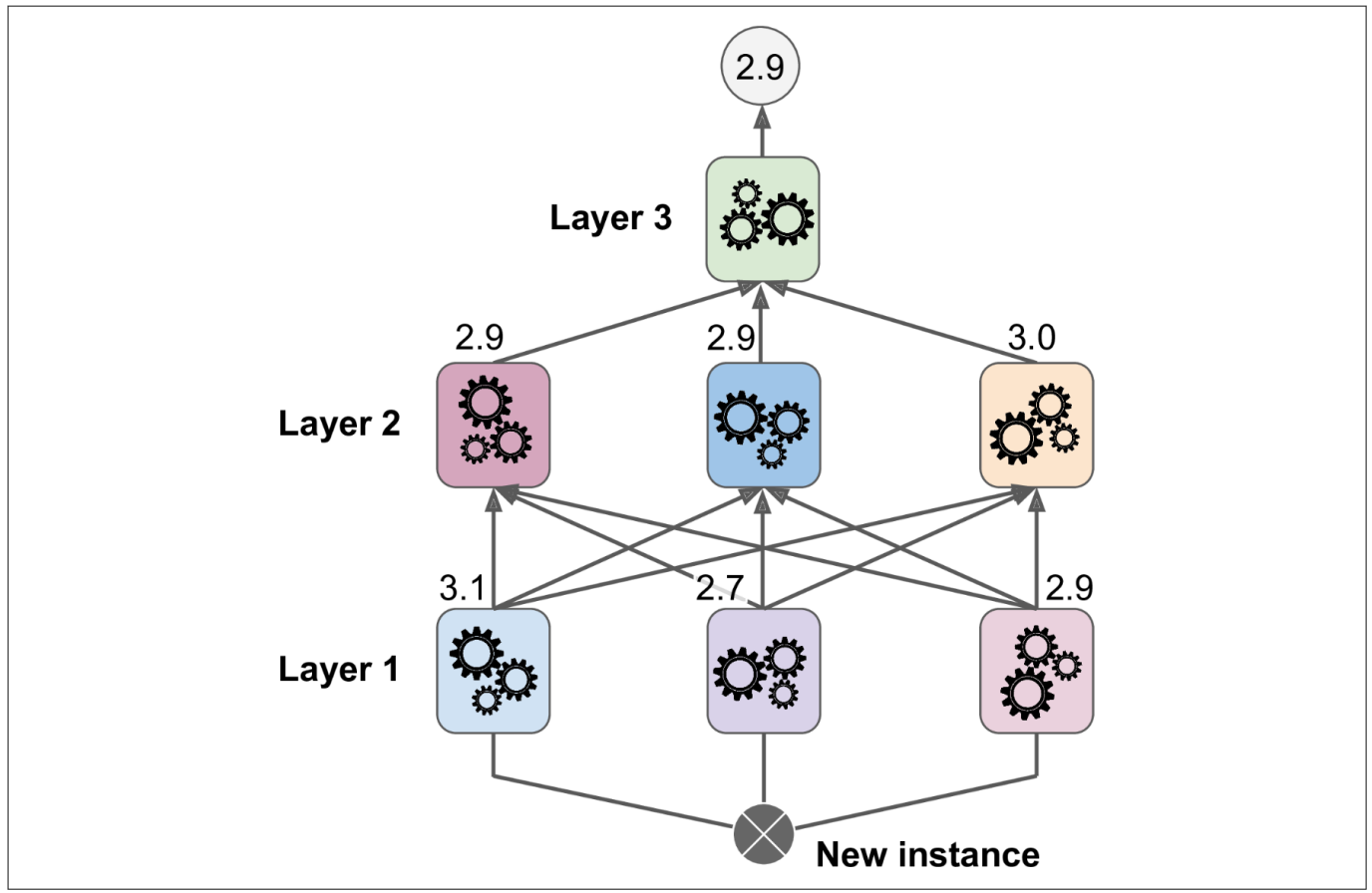
- To train the blender, a common approach is to use a hold-out set. Let's see how it works. First, the training set is split in two subsets. The first subset is used to train the predictors in the first layer.



- Next, the first layer predictors are used to make predictions on the second (held-out) set. This ensures that the predictions are “clean,” since the predictors never saw these instances during training. Now for each instance in the hold-out set there are three predicted values.
- We can create a new training set using these predicted values as input features (which makes this new training set three-dimensional), and keeping the target values. The blender is trained on this new training set, so it learns to predict the target value given the first layer’s predictions.


Blender
Train
(to combine predictions)
Blending training set
Predictions
Predict
Subset 2

- It is actually possible to train several different blenders this way (e.g., one using Linear Regression, another using Random Forest Regression, and so on): we get a whole layer of blenders.
- The trick is to split the training set into three subsets: the first one is used to train the first layer, the second one is used to create the training set used to train the second layer (using predictions made by the predictors of the first layer), and the third one is used to create the training set to train the third layer (using pre- dictions made by the predictors of the second layer).
- Once this is done, we can make a prediction for a new instance by going through each layer sequentially, as shown in


2.9
Layer 3
2.9
2.9
3.0
Layer 2
3.1
2.7
2.9
Layer 1
New instance

See Lesson 5.pynb

- Code Cell
- `5.6 StackingClassifier`

Interview Questions

1. **High-Dimensional Data Scenario**:
    1. **Question**: You are tasked with creating a predictive model for genetic data classification, where each sample may have thousands of features. Explain how you would apply the Random Patches and Random Subspaces methods of ensemble learning to handle the high dimensionality. Discuss the potential benefits and limitations of these methods in terms of feature selection, model accuracy, and computational efficiency.
2. **Stacking Ensemble Scenario**:
    1. **Question**: Consider you have three different machine learning models: a Logistic Regression classifier, a Decision Tree classifier, and a Support Vector Machine, all performing with moderate accuracy on a text classification problem. You want to improve the performance using a stacking ensemble approach. Detail the steps you would take to implement this ensemble, including how you would generate training sets for each layer, how you would choose the blender or meta-learner, and how you would validate the performance of your ensemble model compared to the individual classifiers.

Main Point

- The predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor.
- Science of Consciousness: Creation arises from the collapse of the unbounded value of wholeness to a point; the re-emergence of wholeness results in the laws (algorithms of nature) that provide the balance, order, and efficiency in creation. Contact with this field improves the quality of life (order, balance, simplicity, efficiency) of the individual and society.