Lecture 4: Model development and training - Part I [Draft]

CS 329S: Machine Learning Systems Design (cs329s.stanford.edu)
Prepared by [Chip Huyen](#), [Michael Cooper](#), & the CS 329S course staff
Reviewed by [Luke Metz](#)
Errata and feedback: please send to [chip@huyenchip.com](#)

Note:
The course, including lecture slides and notes, is a work in progress. This is the first time the course is offered and the subject of ML systems design is fairly new, so we (Chip + the course staff) are all learning too. We appreciate your:

1. **enthusiasm** for trying out new things
2. **patience** bearing with things that don't quite work
3. **feedback** to improve the course.

Table of contents

Sampling

Unless we have access to all possible data in the real-world, all datasets that we use to develop machine learning models are samples of real-world data. Sampling is an integral part of ML that is often overlooked in typical ML coursework. Understanding different sampling methods can help us use data more efficiently as well as avoid sampling biases.

There are generally two families of sampling: non-probability sampling and random sampling.

Non-probability sampling

Non-probability sampling is when selection of data isn't based on any probability criteria (e.g. select each sample with the probability of 10%). It is, therefore, not representative of the real-world data and is embedded with selection biases. Some of the criteria for non-probability sampling are:
- **Convenience sampling**: samples of data are selected based on their availability. This sampling method is popular because, well, it's convenient.
- **Snowball sampling**: future samples are selected based one existing samples.
  - E.g: to scrape legit Twitter accounts without having access to Twitter databases, you start with seed accounts then scrape their following, and so on.
- **Judgment sampling**: experts decide what samples to include.
- **Quota sampling**: you select samples based on quotas for certain slices of data without any randomization.

Because non-probability sampling is embedded with selection bias, you might think that it's a bad idea to train ML models using data selected by this family of sampling methods. You're right, but selection of data for ML models is still driven by convenience.

One example is language modeling. Language models are often trained not with data that is representative of all possible texts but with data that can be collected such as Wikipedia, CommonCrawl, articles linked in Reddit.

Another example is datasets for sentiment analysis or recommendation systems. They tend to come from sources with natural labels (e.g. ratings) such as IMDB reviews or Amazon reviews. These sources of data exclude people who don't have access to the Internet and aren't willing to put reviews online.

The third example is data for self-driving cars. Data collected for self-driving cars come largely from two areas: Phoenix in Arizona (because of lax regulations) and the Bay Area in California (because many companies that build self-driving cars are located here). Both areas have

generally sunny weather, which means that there's a lot more self-driving car data for sunny weather than for rainy or snowy weather.

Random sampling

Simple random sampling

The easiest method of random sampling is to give all samples in the population equal probabilities of being selected. For example: randomly select 10% of all samples.

Pros of this method is that it's easy to implement. Cons is that if you have a rare class (e.g. a class that appears in only 0.1% of the population), that rare class might not appear in your selection, and models trained on this selection might think that this rare class doesn't exist.

Stratified sampling

To avoid the drawback of simple random sampling, you can first divide your population into different classes and sample from each class separately. For example, to sample 10% of data that has two classes A and B, you can sample 10% of class A and 10% of class B. This method is called

You can also divide your data into subgroups based on different traits such as gender, category, age. For example, you can divide your users into male and female subgroups, and sample 10% of each subgroup.

Each group is called a strata, and this method is called stratified sampling.

One drawback of this sampling method is that it isn't possible when you can't divide all samples into subgroups or when one sample might belong to multiple groups. This is challenging when you have a multilabel task and a sample can be both class A and class B. For example, in an entity classification task, a name might be both a PERSON and a LOCATION.

Weighted sampling

In weighted sampling, each sample is given a weight, which determines the probability of it being selected. For example, if you want a sample to be selected 30% of the time, give it weight 0.3.

This method allows you to embed subject matter expertise. For example, if you know that more recent data is more valuable to your model, you can give recent data higher weight.

This also helps with the case when your available data comes from a different distribution compared to true data. For example, in your data, red samples account for only 25% and blue samples account for 75%, but you know that in the real world, red and blue have equal probability to happen, so you would give red samples weights three times the weights of blue samples.

In Python, you can do weighted sampling as following:

```
random.choices(population=[1, 2, 3, 4, 100, 1000],
               weights=[0.2, 0.2, 0.2, 0.2, 0.1, 0.1],
               k=2)
```

This is equivalent to:

```
random.choices(population=[1, 1, 2, 2, 3, 3, 4, 4, 100, 1000],
               k=2)
```

Weighted sampling vs. sample weights

Weighted sampling is related to, but different from sample weights. Weighted sampling is used to select samples to train your model with. Once you've had your training set, sample weights are used to assign weights to training samples to determine how much they can affect the loss function. Changing sample weights can change your model's decision boundaries.
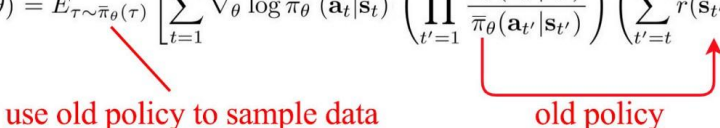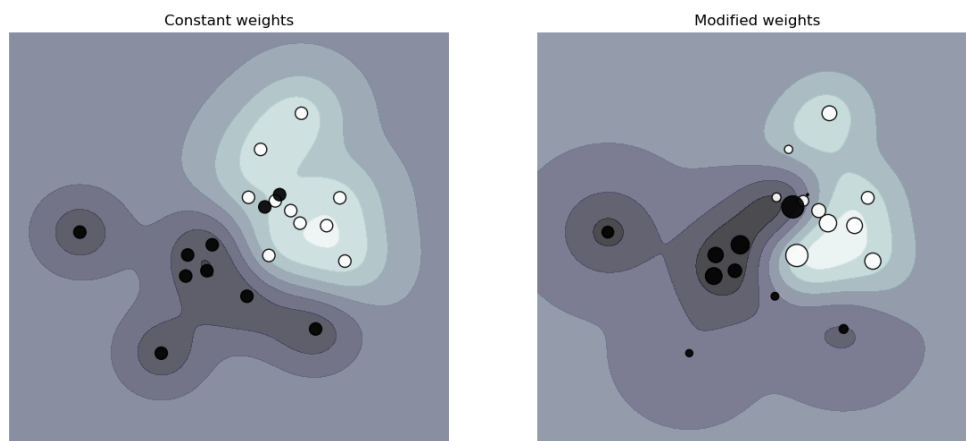

SVM: Weighted samples (sklearn)

Importance sampling

Imagine you have to sample x from a distribution P(x), but P(x) is really expensive, slow, or infeasible to sample from. However, you have a distribution Q(x) that is a lot easier to sample from. So you sample x from Q(x) instead -- Q(x) is called the proposal distribution -- and weight this sample by $\frac{P(x)}{Q(x)}$.

$$E_{P(x)}[x] = \sum_x P(x)\, x = \sum_x Q(x)\, x\, \frac{P(x)}{Q(x)} = E_{Q(x)}[x\frac{P(x)}{Q(x)}]$$

Importance sampling is common in reinforcement learning. For example, every time you update your policy, you want to estimate the value functions of the new policy , but calculating the total rewards of taking an action can be costly. However, if the new policy is relatively close to the old policy, you can calculate the total rewards based on the old policy instead and reweight them according to the new policy.

$$\nabla_\theta J(\theta) = E_{\tau \sim \bar{\pi}_\theta(\tau)} \left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta\,(\mathbf{a}_t|\mathbf{s}_t) \left(\prod_{t'=1}^{t} \frac{\pi_\theta\,(\mathbf{a}_{t'}|\mathbf{s}_{t'})}{\bar{\pi}_\theta(\mathbf{a}_{t'}|\mathbf{s}_{t'})} \right) \left(\sum_{t'=t}^{T} r(\mathbf{s}_{t'},\mathbf{a}_{t'}) \right) \right]$$

use old policy to sample data   old policy

[1]

Reservoir sampling

This is one really awesome sampling algorithm. Now imagine you have to sample k tweets from an incoming stream of tweets. You don't know how many tweets there are but you know you can't fit them all in memory, which means you don't know the probability at which a tweet should be selected. You want to:
- ensure that every tweet has an equal probability of being selected,
- you can stop the algorithm at any time and get the desired samples.

One solution for that is reservoir sampling. The algorithm goes like this:
1. First k elements are put in the reservoir.
2. For each incoming i^{th} element, generate a random number j between 1 and i
3. If $1 \leq j \leq k$: replace j^{th} in reservoir with i^{th}

Each incoming i^{th} element has $\frac{k}{i}$ probability of being in the reservoir. You can also prove that each element in the reservoir has $\frac{k}{i}$ probability of being there.
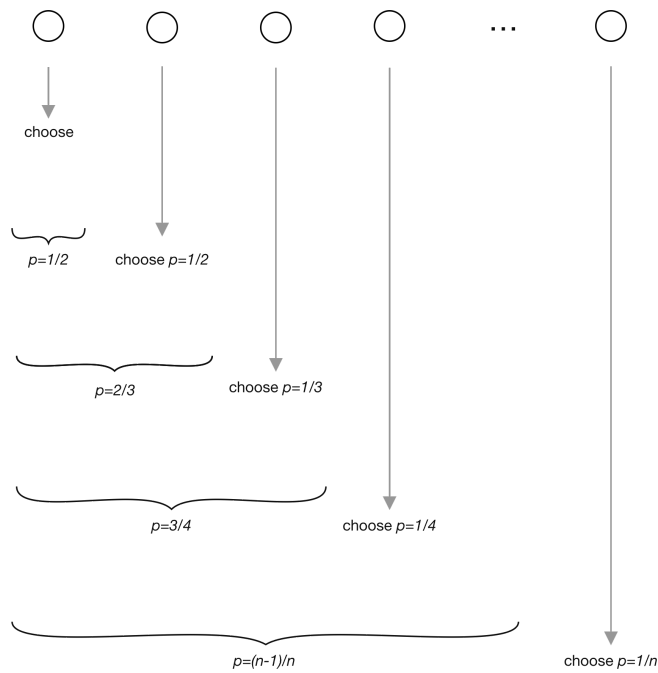
---

[1] RL — Importance Sampling (Jonathan Hui)

Image from Analytics Vidhya

Class imbalance

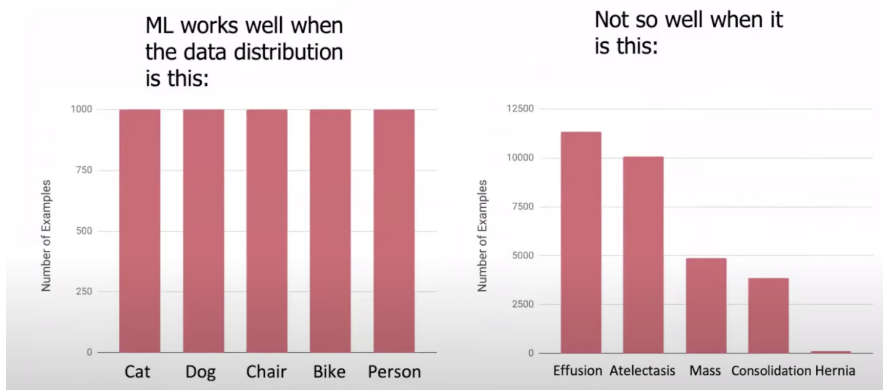Image from Andrew Ng: Bridging AI's Proof-of-Concept to Production Gap

Class imbalance - a substantial difference in the number of samples in each class of the training data - can make learning difficult for a number of reasons:

- **Insufficient signal** - if the absolute number of training examples corresponding to a minority class is low, the problem then becomes one of few-shot learning on examples from the minority class. (Which, though far from intractable, makes the learning problem more difficult).
- **"Satisfactory" defaults** - by learning a very simple prediction heuristic (e.g. "always predict the majority class"), the model can obtain low loss and high accuracy, while learning relatively little about the underlying structure of the data. Such a point may be a local loss minimum from which it may be difficult to extricate a given descent algorithm.

ML models can default to the majority option.

- **Asymmetric cost of error** - the human cost of misclassifying a minority class may be asymmetrically great (e.g. if a model fails to correctly identify a particularly rare but aggressive form of cancer). Though this isn't, per se, a reason why it's *difficult* to train models on imbalanced data, it's a reason why poor performance on imbalanced data is often *insufficient* for machine learning systems.

In the type of real-world data on which machine learning systems are trained, class imbalance is the norm (typically biased toward the negative class):
- Fraud detection - only a tiny minority of credit card transactions (fortunately!) are fraudulent.
- Spam detection - only a tiny minority of incoming emails are spam.
- Disease screening - most people are healthy.
- Churn prediction - most customers are not planning on cancelling their subscription (or, if they are, your business has more to worry about than its churn prediction algorithm).
- Resume screening - very few resumes pass automated screening.
- Object detection - most possible bounding boxes over an image do not contain a relevant object.
- Tree classification - in Vermont, there are far more 🌲 (pines) than 🌴 (palm trees).

---

[2] https://twitter.com/computerfact/status/974364686685794304

Causes of class imbalance

Causes of class imbalance can include:
- **Sampling bias** - perhaps the way the dataset was constructed excludes certain cases. E.g. Google's self driving cars are trained mostly on roads in California and Arizona, and on urban (rather than rural) roads. This selection bias can create class imbalance in the training data.
- **Domain specific** - perhaps some labels are just very rare (see the above list) given the domain in which the model is operating.
- **Labeling Errors** - a less common source of error, but incorrect labeling could result in an imbalance of classes as well.

Solutions

There are many possible techniques to mitigate the class imbalance problem, one or a combination of them might be necessary but not sufficient. In this lecture, we'll examine three of them.
- Resampling: add more minority samples or remove majority samples.
- Weight balancing: adjust the loss function to incentivize the model to focus more on rare classes.
- Ensembles: choose a learning algorithm more robust to class imbalance.

⚠ **Note** ⚠
1. Not all class imbalance problems are equal. Class imbalance for binary problems is a much easier problem than class imbalance for problems with more than 2 classes.
2. Some might argue that you shouldn't try to *fix* class imbalance if that's how the data is in the real world. A good model should learn to work with that!
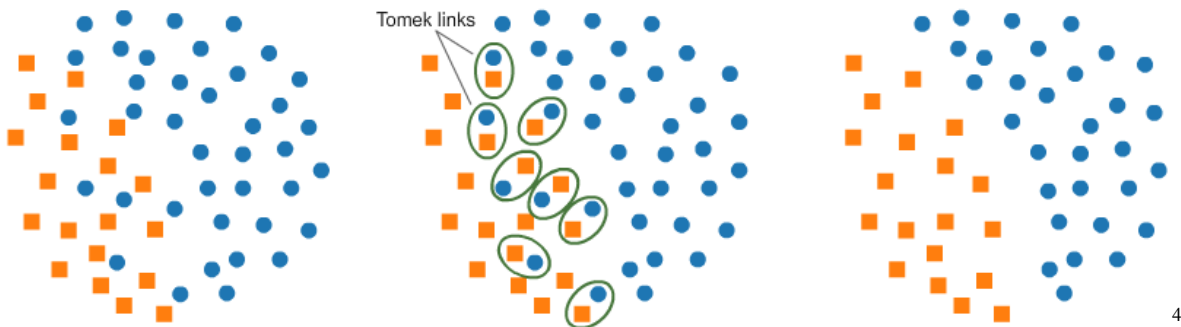
Resampling

Resampling means either adding more minority samples (oversampling) or removing majority samples (undersampling). Undersampling runs the risk of losing important data from removing data, while undersampling runs the risk of overfitting on training data, especially if the added copies of the minority class are replicas of existing data.

Undersampling / Oversampling diagrams
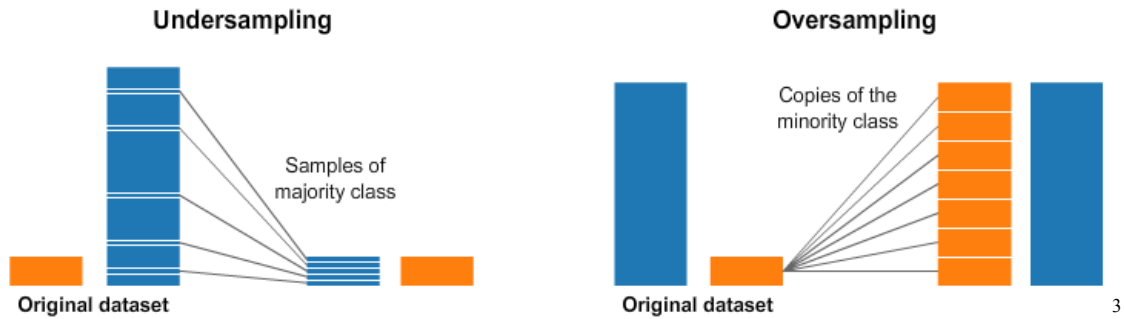
Undersampling: Tomek Links

A popular method of undersampling low-dimensional data is Tomek Links. Find pairs of samples from opposite classes that are close in proximity, and remove the sample of the majority class in each pair.

While this makes the decision boundary more clear and arguably helps models learn the boundary better, it may make the model less robust (by removing some of the subtleties of the true decision boundary).


Oversampling: SMOTE (Synthetic Minority Over-sampling TEchnique)

A popular method of oversampling low-dimensional data is SMOTE. It synthesizes novel samples of the minority class through sampling convex (which approximately means linear) combinations of existing data points within the minority class.

[3] Image from Resampling strategies for imbalanced datasets (Rafael Alencar, Kaggle 2018)
[4] Image from Resampling strategies for imbalanced datasets (Rafael Alencar, Kaggle 2018)
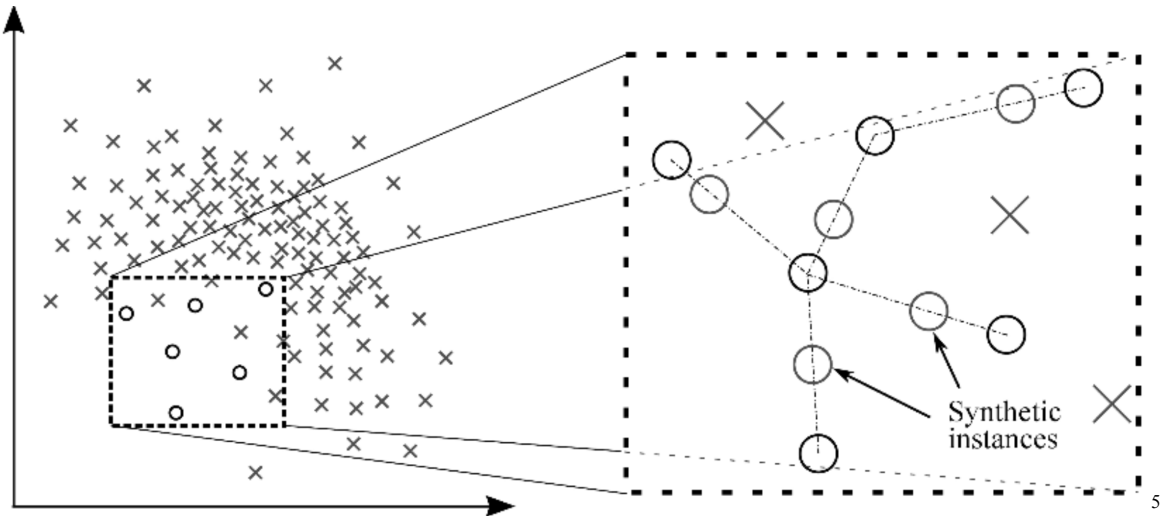
Both SMOTE and Tomek Links have only been proven effective in low-dimensional data. Effectiveness in high dimensional regimes is an open question.

Loss adjustment: weight-balancing

The key idea behind weight-balancing loss is that if we give the loss resulting from the wrong prediction on a data sample higher weight, we'll incentivize the ML model to focus more on learning that sample correctly.

There are two major methods for weight-balancing loss:
- Biasing toward rare classes
- Biasing toward difficult samples

Biasing toward rare classes

What might happen with a model trained on an imbalance dataset is that it'll bias toward majority classes and make wrong predictions on minority classes. What if we punish the model for making wrong predictions on minority classes to correct this bias?

The non-weighted loss might look like the below. The loss is the average of losses on individual samples.

$$L(X;\ \theta)\ =\ \sum_i L(x_i; \theta)$$

The naive weighted might look like the below. The weight for a class is inversely proportional to the number of samples in that class, so that the rarer classes have higher weights.

$$L(X; \theta) = \sum_i W_{y_i} L(x_i; \theta)$$

$$W_c = \frac{N}{number\ of\ samples\ of\ class\ C}$$

A more sophisticated version of this loss can take in account the overlapping among existing samples, such as Class-Balanced Loss Based on Effective Number of Samples (Cui et al., CVPR 2019).

Biasing toward difficult samples

In our dataset, some samples are easier to classify than others, and our model might learn to classify them quickly. We want to incentivize our model to focus on learning the samples they still have difficulty classifying. What if we adjust the loss so that if a sample has a lower probability of being right, it'll have a higher weight? This is exactly what Focal Loss does[6].
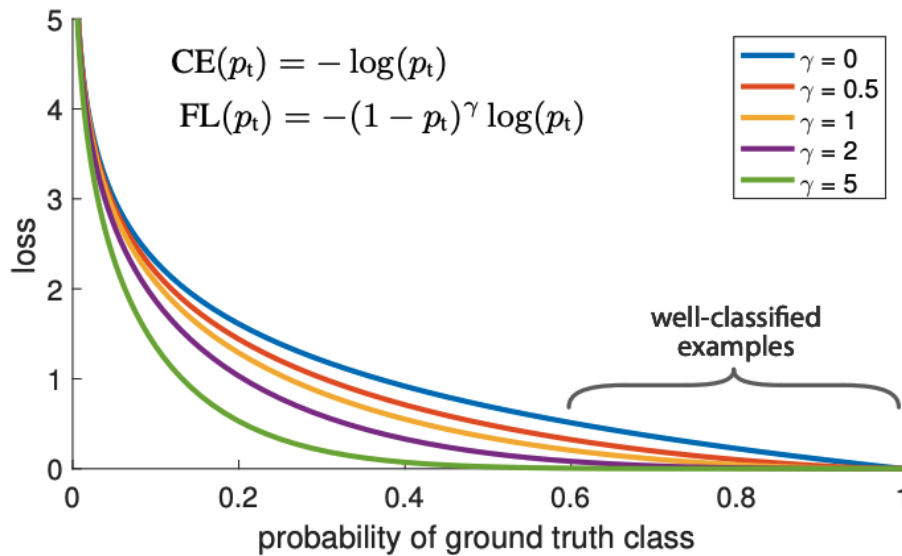

Figure 1. We propose a novel loss we term the *Focal Loss* that adds a factor $(1 - p_t)^\gamma$ to the standard cross entropy criterion. Setting $\gamma > 0$ reduces the relative loss for well-classified examples ($p_t > .5$), putting more focus on hard, misclassified examples. As our experiments will demonstrate, the proposed focal loss enables training highly accurate dense object detectors in the presence of vast numbers of easy background examples.

---

[6] Focal Loss for Dense Object Detection (Lin et al., 2017)

Algorithms

According to several survey papers[78], ensembling methods such as boosting and bagging, together with resampling, have shown to perform well on imbalance datasets.

> ⚠ **Missing information** ⚠
> Experiments have shown that boosting and bagging have helped, but I haven't been able to find much literature on algorithms robust to the class imbalance problem. If you have more info, ideas, or feedback on this, please let me know. Thank you!
>
> I included bagging and boosting algorithms here anyway because I believe they are important algorithms that students should know.

Bagging

Bagging, shortened for bootstrap aggregating, is designed to improve the stability and accuracy of ML algorithms. It reduces variance and helps to avoid overfitting.

Given a dataset, instead of training one classifier on the entire dataset, you sample with replacement to create different datasets, called bootstraps, and train a classification or regression model on each of these bootstraps. Sampling with replacement ensures each bootstrap is independent from its peers.

If the problem is classification, the final prediction is decided by the majority vote of all models. For example, if 10 classifiers vote SPAM and 6 models vote NOT SPAM, the final prediction is SPAM.

If the problem is regression, the final prediction is the average of all models' predictions.

Bagging generally improves unstable methods, such as neural networks, classification and regression trees, and subset selection in linear regression. However, it can mildly degrade the performance of stable methods such as k-nearest neighbors[9].

---

[7] A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches (Galar et al., 2011)
[8] Solving class imbalance problem using bagging, boosting techniques, with and without using noise filtering method (Rekha et al., 2019)
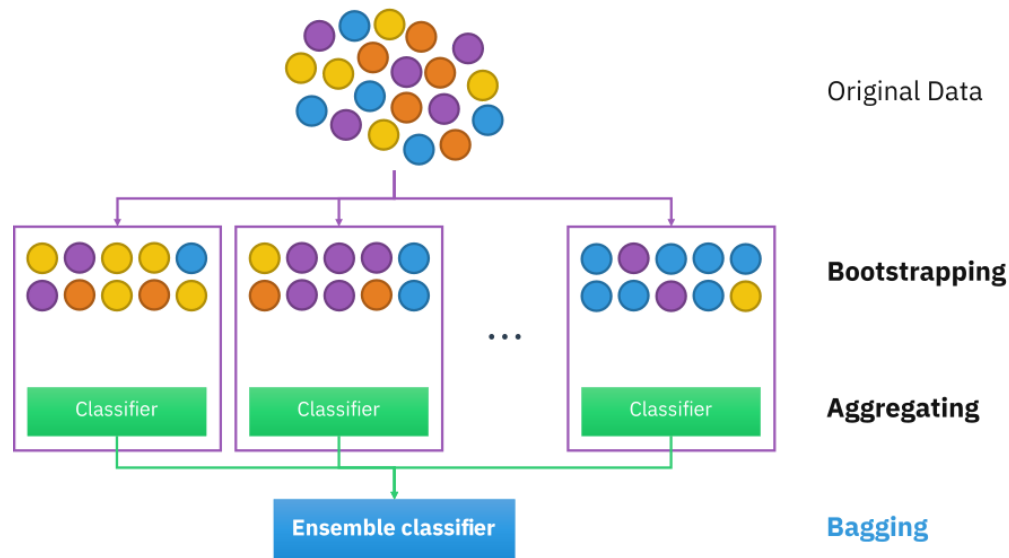[9] Bagging Predictors (Leo Breiman, 1996)

Illustration by Sirakorn

A random forest is an example of bagging. A random forest is a collection of decision trees constructed by both bagging and feature randomness, each tree can pick only from a random subset of features to use.

Due to its ensembling nature, random forests correct for decision trees' overfitting to their training set.

Boosting

Boosting is a family of iterative ensemble algorithms that convert weak learners to strong ones. Each learner in this ensemble is trained on the same set of samples but the samples are weighted differently among iterations. Thus, future weak learners focus more on the examples that previous weak learners misclassified.

1. You start by training the first weak classifier on the original dataset.
2. Samples are reweighted based on how well the first classifier classifies them, e.g. misclassified samples are given higher weight.
3. Train the second classifier on this reweighted dataset. Your ensemble now consists of the first and the second classifiers.
4. Samples are weighted based on how well the ensemble classifies them.
5. Train the third classifier on this reweighted dataset. Add the third classifier to the ensemble.
6. Repeat for as many iterations as needed.
7. Form the final strong classifier as a weighted combination of the existing classifiers -- classifiers with smaller training errors have higher weights.
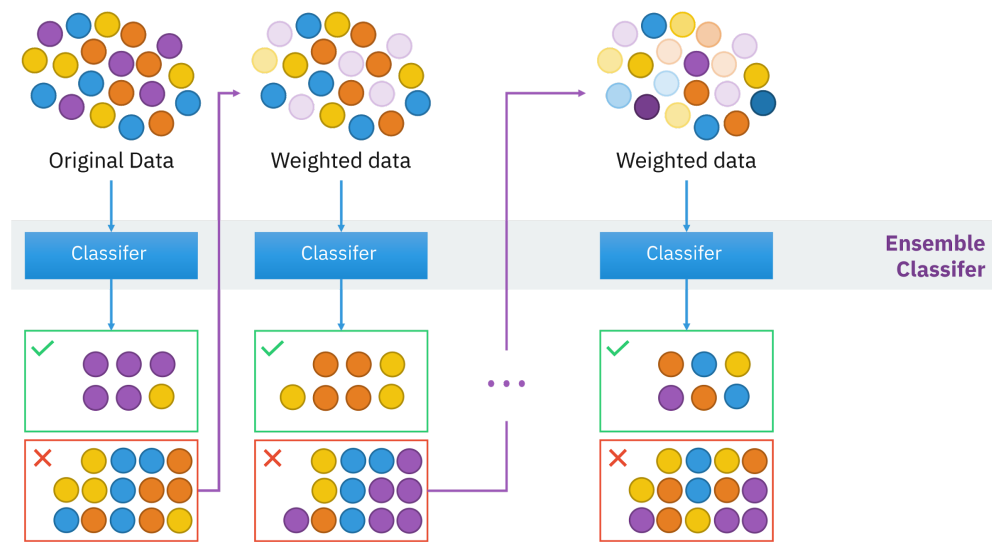
Illustration by Sirakorn

An example of a boosting algorithm is Gradient Boosting Machine which produces a prediction model typically from weak decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

XGBoost, a variant of GBM, used to be the algorithm of choice for many winning teams of machine learning competitions. It's been used in a wide range of tasks from classification, ranking, to the discovery of the Higgs Boson[10]. However, many teams have been opting for LightGBM, a distributed gradient boosting framework that allows parallel learning which generally allows faster training on large datasets.

---

[10] Higgs Boson Discovery with Boosted Trees (Tianqi Chen and Tong He, 2015)