Topic 7: Ensemble Methods (Meta-Learning Algorithms)

Ensemble methods are "meta-algorithms" which combine two or more predictions from "basic" predictive models into one model which is more robust than the basic predictive models [Abbott, 2014, Chapter 10].

There are two families of ensemble methods:

- **Averaging Methods**: The driving principle is to build several estimators independently based on bootstrap sampling and then average their predictions. The combined estimator is usually better than any of the single base estimator because its variance is reduced. Examples:
  - Majority Voting: For a set of training data D, predictive model h_1 (e.g. kNN), h_2 (e.g. logreg), ..., h_n (e.g. naive bayes) are trained and all will be used in prediction, the output will be the majority. In Python, we have `VotingClassifier`, `VotingRegressor`.
  - Bagging (Section 7.1.1): For a set of training data D, we construct boostraped samples D_1, D_2, ..., D_T from D by sampling uniformly, then train on an algorithm h to get h_{D_1}, h_{D_2}, ..., h_{D_T}. Finally, take majority vote from them in prediction.
  - Random Forest (Section 7.1.2): We fit decision trees on different bootstrap samples and for each decision tree, we select a **random subset of features**, m, (e.g $\log_2 p + 1$) at each node to decide upon the optimal split.
  - Stacking, Blending (see `https://www.analyticsvidhya.com/blog/2018/06/comprehensive-guide-for-ensemble-models/`), etc. E.g. `StackingClassifier`, `StackingRegressor` are available in Python's scikit-learn.
- **Boosting Methods**: Base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble. Examples:
  - Adaptive Boosting (Section 7.2.1)
  - Gradient Boosting (Section 7.2.2)

Despite the interpretability, decision trees do not have the same level of predictive accuracy as other predictive models. Decision trees are **unstable** as they produce drastically different trees from slight different train sets. In technical terms, decision trees suffer from **high variance**. By using ensemble methods like bagging, random forests and boosting, the predictive performance of trees can be improved and averaging the trees leads to the reduction in variance [James et al., 2013, Section 8.1.4].

In this topic, $\mathbb{H}$ will denote the *hypothesis class*, a set of classifiers with large bias and usually high training error (e.g. CART trees with very limited depth.)

In his machine learning class project in 1988, `https://en.wikipedia.org/wiki/Michael_Kearns_(computer_scientist)` famously asked the question: Can weak learners h be combined to generate a strong learner with low bias?

In 1990, `https://en.wikipedia.org/wiki/Robert_Schapire` showed that we can! The solution is to create ensemble classifier

$$H_T(\vec{x}) = \sum_{t=1}^{T} \alpha_t h_t(\boldsymbol{x}).$$

built in an iterative fashion. In iteration t we add the classifier $\alpha_t h_t(\boldsymbol{x})$ to the ensemble. At test time we evaluate all classifier and return the weighted sum.

7.1 Average Methods

Popular averaging methods are `https://en.wikipedia.org/wiki/Bootstrap_aggregating`, `https://en.wikipedia.org/wiki/Random_forest` and extra trees.

7.1.1 Bagging

Bagging is an abbreviation for *bootstrap aggregation*:

- Bagging averages a given class of predictive models over many samples to reduce its variance and produces smoother decision boundaries.
- It reduces overfitting and works best with strong and complex models such as fully developed decision trees.

Example 7.1.1. An illustration of Bagging with 5 bootstraps for the predictive model h is shown below.

B	C	D	E	F	G
Original	Bootstrap1	Bootstrap2	Bootstrap3	Bootstrap4	Bootstrap5
Index	D1.Index	D2.Index	D3.Index	D4.Index	D5.Index
1	9	1	7	9	7
2	9	7	5	6	1
3	3	5	6	5	6
4	8	10	9	9	2
5	10	7	2	10	1
6	7	9	5	4	2
7	10	9	8	6	4
8	9	10	2	8	5
9	3	7	1	6	6
10	4	5	9	6	3
	h_D1	h_D2	h_D3	h_D4	h_D5
Predict(X)	A	A	B	A	B

Bagging is a strategy to reduce the variance term in the bias-variance decomposition (6.7), i.e. making $h_D \to \bar{h}$. The following theoretical analysis of bagging is based on Professor Kilian Q. Weinberger's CS4780.

The *weak law of large numbers* says for i.i.d. random variables X_i with mean $\bar{X}$, we have,

$$\frac{1}{B}\sum_{i=1}^{B} X_i \to \bar{X} \quad \text{as } B \to \infty.$$

Assume we have B training sets $D_1, D_2, ..., D_n$ drawn from P^n. If we apply this idea to classifiers, averaging the classifiers trained on each D_i should approximate a less bias classifier $\overline{h}$:

$$\hat{h} = \frac{1}{B}\sum_{i=1}^{B} h_{D_i} \to \overline{h} \quad as\ B \to \infty.$$

We refer to such an average of multiple classifiers as an ensemble of classifiers.

The good news is, if $\hat{h} \to \overline{h}$ the variance component of the error must also vanish, i.e. $\mathbb{E}_x[(\hat{h}(x) - \overline{h}(x))^2] \to 0$.

The problem is, we don't have B data sets $D_1, ..., D_B$, we only have D.

What we can do is to simulate drawing from P by drawing uniformly with replacement from the set D, i.e. let $Q(X, Y|D)$ be a probability distribution that picks a training sample $(\mathbf{x}_i, y_i)$ from D uniformly at random. More formally,

$$Q((\mathbf{x_i}, y_i)|D) = \frac{1}{n} \qquad \forall(\mathbf{x_i}, y_i) \in D, \quad n = |D|.$$

We sample the set $D_i \sim Q^n$, i.e. $|D_i| = n$, and D_i is picked with replacement from $Q|D$.

The bagged predictive model

$$\hat{h}_D = \frac{1}{B}\sum_{i=1}^{B} h_{D_i} \text{ (regressor)} \quad \hat{h}_D = \text{mode}\{h_{D_1}, \cdots, h_{D_B}\}, \text{ (classifier).} \tag{7.1}$$

won't approximate $\overline{h}$ in general:

$$\hat{h}_D \nrightarrow \bar{h}.$$

because the weak law of large number **only works** for i.i.d. samples.

However, in practice bagging still reduces variance very effectively.

Although we cannot prove that the new samples are i.i.d., we can show that they are drawn from the original distribution P. Assume P is discrete, with $P(X = x_i) = p_i$ over some set $\Omega = x_1, ...x_N$ (N very large) (let's ignore the label for now for simplicity)

$$Q(X = x_i) = \underbrace{\sum_{k=1}^{n}\binom{n}{k}p_i^k(1-p_i)^{n-k}}_{\substack{\text{Probability that are}\\ \text{k copies of } x_i \text{ in D}}} \underbrace{\frac{k}{n}}_{\substack{\text{Probability}\\ \text{pick one of}\\ \text{these copies}}} = \frac{1}{n}\underbrace{\sum_{k=1}^{n}\binom{n}{k}p_i^k(1-p_i)^{n-k}k}_{\substack{\text{Expected value of}\\ \text{Binomial Distribution}\\ \text{with parameter } p_i\\ \mathbb{E}[\mathbb{B}(p_i,n)]=np_i}} = \frac{1}{n}np_i = p_i$$

Each data set D'_l is drawn from P, but not independently.

There is a simple intuitive argument why $Q(X = x_i) = P(X = x_i)$. So far we assumed that we draw D from P^n and then Q picks a sample from D. However, we don't have to do it in that order. We can also view sampling from Q in reverse order: Consider that we first use Q to reserve a "spot" in D, i.e. a number from 1,...,n, where i means that we sampled the i^{th} data point in D. So far we only have the slot, i, and we still need to fill it with a data point (x_i, y_i). We do this by sampling (x_i, y_i) from P. It is now obvious that which slot we picked doesn't really matter, so we have $Q(X = x) = P(X = x)$.

In summary, a bagging of the algorithm h involves three steps:

1. Sample B data sets $D_1, \ldots, D_B$ from D with replacement.

2. For each D_i train a classifier h_{D_i}.

3. The final classifier is (7.1).

Note that larger B results in a better ensemble, however at some point we will obtain diminishing returns. Note that setting B unnecessarily high will only slow down the classifier but will not increase the error of the classifier.

A Naive Implementation of Bagging of CART regressor in R

```
B=100
bagged_models=list()
set.seed(12)
for(i in 1:B) {
  bootstrap = sample(idx.train, size=length(idx.train), replace=T)
  bagged_models = c(bagged_models, list(rpart(Y~.,
    d.f.train[bootstrap],control=rpart.control(minsplit=6))))
}
bagged_result=NULL
i=0
for (from_bag_model in bagged_models) {
  if (is.null(bagged_result))
    bagged_result=predict(from_bag_model,new_X)
  else
    bagged_result=
      (i*bagged_result+predict(from_bag_model,new_X))/(i+1)
  i=i+1
}
```

In R, there are a few libraries providing the bagging meta-algorithm. However, they are not popularly. The `ipred` library, which stands for "improved predictive models by indirect classification and bagging for classification, regression and survival problems" depends on the `lava` (latent variable models) library, which in turn too many libraries. It provides a primitive bagging method `bagging`. The package `adabag` implements Freund and Schapire's Adaboost.M1 algorithm and Breiman's Bagging algorithm using classification trees as individual classifiers [Kuhn and Johnson, 2013, Chapter 8]. The package `autoBagging` implements rank bagging. Both `adabag` and `autoBagging` depend on `caret` library, which in turn too many libraries.

In Python, bagging is available from the scikit-learn library.

```
class sklearn.ensemble.BaggingClassifier(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)

class sklearn.ensemble.BaggingRegressor(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)

#from sklearn.neighbors import BaggingClassifier, KNeighborsClassifier
#bg=BaggingClassifier(KNeighborsClassifier(),max_samples=0.6,max_features=0.4)
```

Here, `max_samples` and `max_features` control the size of the subsets (in terms of samples and features), while `bootstrap` and `bootstrap_features` control whether samples and features are drawn with or without replacement. When using a subset of the available samples the generalisation accuracy can be estimated with the out-of-bag samples by setting `oob_score=True`.

Advantages of Bagging

- Reduces variance, so has a strong beneficial effect on high variance classifiers.
- As the prediction is an average of many classifiers, we obtain a mean score and variance. The latter can be interpreted as the uncertainty of the prediction. Especially in regression

tasks, such uncertainties are otherwise hard to obtain. For example, imagine the prediction of a house price is \$300,000. If a buyer wants to decide how much to offer, it would be very valuable to know if this prediction has standard deviation $\pm\$10,000$ or $\pm\$50,000$.

- Bagging provides an unbiased estimate of the test error, which we refer to as the out-of-bag error. The idea is that each training point was not picked and all the data sets D_k. If we average the classifiers h_k of all such data sets, we obtain a classifier (with a slightly smaller B) that was not trained on $(\mathbf{x}_i, y_i)$ ever and it is therefore equivalent to a test sample. If we compute the error of all these classifiers, we obtain an estimate of the true test error. The beauty is that we can do this without reducing the training set. We just run bagging as it is intended and obtain this so called out-of-bag error for free.

  More formally, for each training point $(\mathbf{x}_i, y_i) \in D$ let $S_i = \{k | (\mathbf{x}_i, y_i) \notin D_k\}$ - in other words S_i is a set of all the training sets D_k, which do not contain $(\mathbf{x}_k, y_k)$. Let the averaged classifier over all these data sets be

$$\tilde{h}_i(\mathbf{x}) = \frac{1}{|S_i|} \sum_{k \in S_i} h_k(\mathbf{x}).$$

  The out-of-bag error becomes simply the average error/loss that all these classifiers yield

$$\epsilon_{\text{OOB}} = \frac{1}{n} \sum_{(\mathbf{x}_i, y_i) \in D} \ell(\tilde{h}_i(\mathbf{x_i}), y_i).$$

  This is an estimate of the test error, because for each training point we used the subset of classifiers that never saw that training point during training. If B is sufficiently large, the fact that we take out some classifiers has no significant effect and the estimate is pretty reliable.

Despite the fact that bagging is not a complex method but calculation by hand is impossible, therefore, relevant exam questions are usually theoretical.

Example 7.1.2 (SRM-02-18, Q12)**.** Determine which of the following statements is true

(A) Linear regression is a flexible approach

(B) Lasso is more flexible than a linear regression approach

(C) Bagging is a low flexibility approach

(D) There are methods that have high flexibility and are also easy to interpret

(E) None of (A), (B), (C), or (D) are true

Solution:
(A) is false, linear regression is considered inflexible because the number of possible models is restricted to a certain form.
(B) is false, the lasso determines the subset of variables to use while linear regression allows the analyst discretion regarding adding or moving variables.
(C) is false, bagging provides additional flexibility.
(D) is false, there is a tradeoff between being flexible and easy to interpret.
Answer: (E)

7.1.2 Random Forest & Extremely Randomised Trees

- When random subsets of the dataset are drawn as random subsets of the samples **without replacement**, then this algorithm is known as **Pasting**. E.g. R's sample(nrow(d.f), 0.8*nrow(d.f))

- If samples are **drawn with replacement**, then the method is known as **Bagging**. E.g. R's sample(nrow(d.f), replace=TRUE)
- When random subsets of the dataset are drawn as random subsets of the features, then the method is known as **Random Subspaces**.
- When base estimators are built on subsets of both samples and features, then the method is known as **Random Patches**.

A *random forest (RF)*, developed by Breiman and Cutler, is essentially bagged decision trees, with a slightly modified splitting criteria.

- Sample $m < p$ columns/features (`mtry`) randomly from the p features from D with replacement as data D_t. (Random Subspaces. If $m = p$, we have Bagging.)
- Grow a simple decision tree of level 1 or a CART tree for the bootstrap data D_t;
- Stop when T number of decision trees (`ntree`) are constructed. The random forest is the collection of decision trees. The final classifier and regression are respectively [Breiman, 2001]

$$h(\boldsymbol{x}) = \underset{h_i \in H}{\text{mode}}\{h_i(\boldsymbol{x})\}, \quad h(\boldsymbol{x}) = \frac{1}{n_T} \sum_{h_i \in H} h_i(\boldsymbol{x}).$$

Example 7.1.3. An illustation of random tree with $n = 10$, $p = 4$, $m = 2$ and $T = 3$ is given below.

mtry = 2

	Original				Bootstrap1			Bootstrap2			Bootstrap3		
Index	C1	C2	C3	C4	D1.Index	C1	C2	D2.Index	C4	C2	D3.Index	C3	C1
1					9			1			7		
2					9			7			5		
3					3			5			6		
4					8			10			9		
5					10			7			2		
6					7			9			5		
7					10			9			8		
8					9			10			2		
9					3			7			1		
10					4			5			9		
					h_D1			h_D2			h_D3		
Predict(X)					A			A			B		

Remark 7.1.4. According to the paper "Bias in random forest variable importance measures: Illustrations, sources and a solution", bioinformatics face the so-called "small n large p"' problem: Usual data sets in genomics often contain 10^3 of genes or markets that serve as predictor variables but only for a comparatively small number n of subjects or tissue types. Logistic regression and neural networks won't work but the random forest works fine.

In R, `randomForest`, which implements Breiman's random forest algorithm based on Breiman and Cutler's original Fortran code is the most popular library for random forest (and bagging as a special case).

```
randomForest(x, y=NULL,  xtest=NULL, ytest=NULL, ntree=500,
  mtry=if (!is.null(y) && !is.factor(y))
         max(floor(ncol(x)/3), 1) else floor(sqrt(ncol(x))),
  replace=TRUE, classwt=NULL, cutoff, strata,
  sampsize = if (replace) nrow(x) else ceiling(.632*nrow(x)),
  nodesize = if (!is.null(y) && !is.factor(y)) 5 else 1,
  maxnodes = NULL, importance=FALSE, localImp=FALSE, nPerm=1,
```

```
  proximity, oob.prox=proximity, norm.votes=TRUE, do.trace=FALSE,
  keep.forest=!is.null(y) && is.null(xtest), corr.bias=FALSE,
  keep.inbag=FALSE, ...)
```

The `cforest` in `party` or `partykit` implements the random forest with conditional inference trees (Section 5.4.3).

```
cforest(formula, data = list(), subset = NULL, weights = NULL,
  controls = cforest_unbiased(),
  xtrafo = ptrafo, ytrafo = ptrafo, scores = NULL)
```

Hyper parameters in `cforest_control` are:

1. The number of randomly preselected variables `mtry`, which is fixed to the value 5 by default here for technical reasons, while in `randomForest` the default values for classification and regression vary with the number of input variables.
2. The number of trees `ntree`. Use more trees if there are more input variables.
3. The depth of the trees, regulated by `mincriterion`. Usually unstopped and unpruned trees are used in random forests. To grow large trees, set `mincriterion` to a small value.

The aggregation scheme works by averaging observation weights extracted from each of the `ntree` trees and NOT by averaging predictions directly as in `randomForest`.

In Python, RF is available in scikit-learn ensemble module.

```
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)
# criterion: gini, entropy, log_loss

class sklearn.ensemble.RandomForestRegressor(n_estimators=100, *,
  criterion='squared_error', max_depth=None, min_samples_split=2,
  min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=1.0,
  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, ccp_alpha=0.0, max_samples=None)
# criterion: squared_error, absolute_error, friedman_mse, poisson
```

Remarks on Python's RandomForestClassifier:

- The default values for the parameters controlling the size of the trees (e.g. "max_depth", "min_samples_leaf", etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

- The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data, "max_features=n_features" and "bootstrap=False", if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, "random_state" has to be fixed.

The RF is a popular and easy to use out-of-the-box classifier due to reasons below:

- Simple RF only has two hyper-parameters, T and m. It is extremely insensitive to both of these. A good choice for m is $m = \sqrt{p}$. We can set T as large as we can afford.

- Decision trees do not require a lot of preprocessing. For example, the features can be of different scale, magnitude, or slope. This can be highly advantageous in scenarios with heterogeneous data, for example the medical settings where features could be things like blood pressure, age, gender, ..., each of which is recorded in completely different units.
- Random forests can be used to rank the importance of variables in a regression or classification problem in a natural way.

  The first step in measuring the variable importance in a data set

  $$\mathcal{D}_n = \{(X_i, Y_i)\}_{i=1}^n$$

  is to fit a random forest to the data. During the fitting process the out-of-bag error for each data point is recorded and averaged over the forest (errors on an independent test set can be substituted if bagging is not used during training).

  To measure the importance of the j-th feature after training, the values of the j-th feature are permuted among the training data and the out-of-bag error is again computed on this perturbed data set. The importance score for the j-th feature is computed by averaging the difference in out-of-bag error before and after the permutation over all trees. The score is normalised by the standard deviation of these differences.

  Features which produce large values for this score are ranked as more important than features which produce small values.

  This method of determining variable importance has some drawbacks. For data including categorical variables with different number of levels, random forests are biased in favour of those attributes with more levels. Methods such as partial permutations and growing unbiased trees can be used to solve the problem. If the data contain groups of correlated features of similar relevance for the output, then smaller groups are favoured over larger groups.

Manual calculation is impossible. Therefore, exam questions on RF are normally philosophical.

Example 7.1.5 (SRM-02-18, Q10)**.** Determine which of the following statements about random forests is/are true?

I. If the number of predictors used at each split is equal to the total number of available predictors, the result is the same as using bagging.

II. When building a specific tree, the same subset of predictor variables is used at each split.

III. Random forests are an improvement over bagging because the trees are decorrelated.

(A) None

(B) I and II only

(C) I and III only

(D) II and III only

(E) The correct answer is not given by (A), (B), (C), or (D).

Solution: II is false because with random forest a new subset of predictors is selected for each split.

Answer: (C)

Example 7.1.6 (SRM-02-18, Q41)**.** For a random forest, let p be the total number of features and m be the number of features selected at each split.

Determine which of the following statements is/are true.

I: When $m = p$, random forest and bagging are the same procedure.

II: $\frac{p-m}{p}$ is the probability a split will not consider the strongest predictor.

III: The typical choice of m is $\frac{p}{2}$.

(A) None

(B) I and II only

(C) I and III only

(D) II and III only

(E) The correct answer is not given by (A), (B), (C), or (D).

Solution:
I is true. Random forests differ from bagging by setting $m < p$.
II is true. $p - m$ represents the splits not chosen.
III is false. Typical choices are the square root of p or $p/3$.
Answer: (B)

Extremely Randomised Trees or **Extra Trees** (ET), construct multiple trees like RF algorithms during training time over the entire dataset. During training, the ET will construct trees over every observation in the dataset but with different subsets of features usually without bootstrapping. Furthermore, when constructing each decision tree, the ET algorithm splits nodes randomly. The differences between ET and RF are stated in the following table.

RF	ET
Sample subsets through bootstrapping	samples the entire training dataset
Nodes are split by selecting the best split	randomised node split
medium variance	low variance
it takes time to find the best split	faster since node splits are random

The main advantage of ET is the reduction in bias. This is in terms of sampling from the entire dataset during the construction of the trees. Different subsets of the data may introduce different biases in the results obtained, hence ET prevents this by sampling the entire dataset.

Another advantage of ET is that they reduce variance. This is a result of the randomised splitting of nodes within the decision trees, hence the algorithm is not heavily influenced by certain features or patterns in the dataset.

```
class sklearn.ensemble.ExtraTreesClassifier(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=False,
  oob_score=False, n_jobs=None, random_state=None, verbose=0,
  warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None)
# criterion: gini, entropy, log_loss

class sklearn.ensemble.ExtraTreesRegressor(n_estimators=100, *,
  criterion='squared_error', max_depth=None, min_samples_split=2,
  min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=1.0,
  max_leaf_nodes=None, min_impurity_decrease=0.0, bootstrap=False,
  oob_score=False, n_jobs=None, random_state=None, verbose=0,
  warm_start=False, ccp_alpha=0.0, max_samples=None)
# criterion: squared_error, absolute_error, friedman_mse, poisson
```

7.2 Boosting

Boosting tries to boost "weak predictive models" (e.g. decision tree which takes only one feature!) to become "strong learners".

There are two broad categories of boosting: (a) adaptive boosting; (b) gradient boosting; and a few less popular methods such as `https://en.wikipedia.org/wiki/LogitBoost` (implemented in R's `ada` library).

7.2.1 AdaBoost (Adaptive Boosting)

Assume that we have binary classification problem with $y_i \in \{+1, -1\}$ and weak learners $h \in \mathbb{H}$ are binary, $h(\boldsymbol{x}_i) \in \{-1, +1\}$, $\forall x$. We perform line-search to obtain best **step-size** α. Let the loss function be the **exponential loss** [Schapire and Freund, 2012]:

$$\ell(H) = \sum_{i=1}^{n} e^{-y_i H(\mathbf{x}_i)}.$$

To find the best weak learner, we first compute the gradient

$$r_i = \frac{\partial \ell}{\partial H(\mathbf{x}_i)} = -y_i e^{-y_i H(\mathbf{x}_i)}.$$

For notational convenience, let us define $w_i = \frac{1}{Z} e^{-y_i H(\boldsymbol{x}_i)}$, where $Z = \sum_{i=1}^{n} e^{-y_i H(\mathbf{x}_i)}$ is a normalising constant so that $\sum_{i=1}^{n} w_i = 1$. Note that the normalising constant Z is identical to the loss function. Each weight w_i therefore has a very nice interpretation. It is the relative contribution of the training point $(\boldsymbol{x}_i, y_i)$ towards the overall loss.

In order to find the best next weak learner, we need to solve the optimization problem: (in the following, we will make use of the fact that $h(\mathbf{x}_i) \in \{+1, -1\}$.)

$$\begin{aligned}
h(\boldsymbol{x}_i) &= \operatorname*{argmin}_{h \in \mathbb{H}} \sum_{i=1}^{n} r_i h(\boldsymbol{x}_i) && \Big(\text{substitute in: } r_i = e^{-H(\boldsymbol{x}_i) y_i}\Big) \\
&= \operatorname*{argmin}_{h \in \mathbb{H}} -\sum_{i=1}^{n} y_i e^{-H(\boldsymbol{x}_i) y_i} h(\boldsymbol{x}_i) && \Big(\text{substitute in: } w_i = \tfrac{1}{Z} e^{-H(\boldsymbol{x}_i) y_i}\Big) \\
&= \operatorname*{argmin}_{h \in \mathbb{H}} -\sum_{i=1}^{n} w_i y_i h(\boldsymbol{x}_i) && \Big(y_i h(\boldsymbol{x}_i) \in \{+1, -1\} \text{ with } h(\boldsymbol{x}_i) y_i = 1 \iff h(\boldsymbol{x}_i) = y_i\Big) \\
&= \operatorname*{argmin}_{h \in \mathbb{H}} \sum_{i : h(\boldsymbol{x}_i) \neq y_i} w_i - \sum_{i : h(\boldsymbol{x}_i) = y_i} w_i && \Big(\sum_{i : h(\boldsymbol{x}_i) = y_i} w_i = 1 - \sum_{i : h(\boldsymbol{x}_i) \neq y_i} w_i\Big) \\
&= \operatorname*{argmin}_{h \in \mathbb{H}} \sum_{i : h(\boldsymbol{x}_i) \neq y_i} w_i && \Big(\text{This is the weighted classification error.}\Big)
\end{aligned}$$

Let us denote this weighted classification error as $\epsilon = \sum_{i : h(\mathbf{x}_i) y_i = -1} w_i$. So for AdaBoost, we only need a classifier that can take training data and a distribution over the training set (i.e. normalzied weights w_i for all training samples) and which returns a classifier $h \in H$ that reduces the weighted classification error of these training samples. It doesn't have to do all that well, in order for the inner-product $\sum_i r_i h(\mathbf{x}_i)$ to be negative, it just needs less than $\epsilon < 0.5$ weighted training error.

In GBRT, we set the stepsize α to be a small constant. As it turns out, in the AdaBoost setting we can find the optimal stepsize (i.e. the one that minimises ℓ the most) in closed form every time we take a "gradient" step.

When we are given ℓ, H, h, we would like to solve the following optimization problem:

$$\alpha = \operatorname*{argmin}_{\alpha} \ell(H + \alpha h) = \operatorname*{argmin}_{\alpha} \sum_{i=1}^{n} e^{-y_i [H(\mathbf{x}_i) + \alpha h(\mathbf{x}_i)]}.$$

We differentiate w.r.t. α and equate with zero:

$$\sum_{i=1}^{n} y_i h(\mathbf{x}_i) e^{-(y_i H(\mathbf{x}_i) + \alpha y_i h(\mathbf{x}_i))} = 0 \quad \Big(y_i h(\mathbf{x}_i) \in \{+1, -1\}\Big)$$

$$-\sum_{i:h(\mathbf{x}_i)y_i=1} e^{-(y_i H(\mathbf{x}_i) + \alpha \underbrace{y_i h(\mathbf{x}_i)}_{1})} + \sum_{i:h(\mathbf{x}_i)y_i=-1} e^{-(y_i H(\mathbf{x}_i) + \alpha \underbrace{y_i h(\mathbf{x}_i)}_{-1})} = 0 \quad \Big(w_i = \frac{1}{Z} e^{-y_i H(\mathbf{x}_i)}\Big)$$

$$-\sum_{i:h(\mathbf{x}_i)y_i=1} w_i e^{-\alpha} + \sum_{i:h(\mathbf{x}_i)y_i=-1} w_i e^{\alpha} = 0 \quad \Big(\epsilon = \sum_{i:h(\mathbf{x}_i)y_i=-1} w_i\Big)$$

$$-(1-\epsilon)e^{-\alpha} + \epsilon e^{\alpha} = 0$$

$$\therefore \quad e^{2\alpha} = \frac{1-\epsilon}{\epsilon} \Rightarrow \alpha = \frac{1}{2} \ln \frac{1-\epsilon}{\epsilon}.$$

It is unusual that we can find the optimal step-size in such a simple closed form. One consequence is that AdaBoost converges extremely fast.

After you take a step, i.e. $H_{t+1} = H_t + \alpha h$, you need to re-compute all the weights and then re-normalize. It is however straight-forward to show that the unnormalized weight $\hat{w}_i$ is updated as

$$\hat{w}_i \leftarrow \hat{w}_i * e^{-\alpha h(\mathbf{x}_i) y_i}$$

and that the normalizer Z becomes

$$Z \leftarrow Z * 2\sqrt{\epsilon(1-\epsilon)}.$$

Putting these two together we obtain the following multiplicative update rule:

$$w_i \leftarrow w_i \frac{e^{-\alpha h(\mathbf{x}_i) y_i}}{2\sqrt{\epsilon(1-\epsilon)}}.$$

The (adaptive) boosting algorithm is as follows.

1. Initialise all weights to $w = \frac{1}{n}$ where n is the number of instances in the dataset
2. For $t := 0$ to T, the number of models to be grown, do:
   - Create a model and get the hypothesis $h_t(x_n)$ for all datapoints x_n in the dataset;
   - Calculate the error ϵ of the training set summing over all datapoints x_n in the training set with:
     $$\epsilon_t \;=\; \frac{\sum_{n=1}^{N} w_n^{(t)} \cdot I(y_n \neq h_t(x_n))}{\sum_{n=1}^{N} w_n^{(t)}}$$
     where $I(cond)$ returns 1 if $I(cond)$ == True and 0 otherwise
   - Compute α with:
     $$\alpha_t \;=\; \log(\frac{1-\epsilon_t}{\epsilon_t})$$
   - Update the weights for the N training instances in the next $(t+1)$ model with:
     $$w_n^{(t+1)} \;=\; w_n^{(t)} \cdot \exp(\alpha_t \cdot I(y_n \neq h_t(x_n)))$$
3. After the T iterations, calculate the final output with: $h(x) \;= \text{sign}(\sum_{t}^{T} \alpha_t \cdot h_t(x))$.

Remark 7.2.1.

- As long as H is negation closed (this means for every $h \in H$ we must also have $-h \in H$), it cannot be that the error $\epsilon > \frac{1}{2}$. The reason is simply that if h has error ϵ, it must be that $-h$ has error $1-\epsilon$. So you could just flip h to $-h$ and obtain a classifier with smaller error. As h was found by minimizing the error, this is a contradiction.

- The inner loop can terminate as the error $\epsilon = \frac{1}{2}$, and in most cases it will converge to $\frac{1}{2}$ over time. In that case the latest weak learner h is only as good as a coin toss and cannot benefit the ensemble (therefore boosting terminates). Also note that if $\epsilon = \frac{1}{2}$ the step-size α would be zero.

Example 7.2.2. In Adaboost, the "shortcomings" are identified by high-weight datapoints. For a data with 10 samples, we have

Data

Index	X1	X2	X3	Y	D1[i]	Err
1					0.1	
2					0.1	
3					0.1	Yes
4					0.1	
5					0.1	
6					0.1	Yes
7					0.1	
8					0.1	
9					0.1	
10					0.1	

Stumps:

Choose X1, X2, X3 for h_1

Let's say X2 wins out

X2	Y	D2[i] formula	D2[i]'	D2[i]
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(J17)	0.2	0.25
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(J17)	0.2	0.25
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(-J17)	0.05	0.063

h_1

0.8

Z_1

err: 0.2

alpha_1=

0.693147

Choose X1, X2, X3 for h_2

Let's say X3 wins out

Index	X3	Y
3		
6		
3		
6		
6		
1		
10		
9		
7		
4		

h_2

(Tree) boosting averages many trees, each grown to re-weighted versions of the training data, hence classifies the data by weighted majority vote. Boosting grows trees sequentially: each tree is grown using information from previously grown trees. It does not involve bootstrap sampling; instead each tree is fit on a modified version of the original data set. Unlike bagging, boosting fit a decision tree using **residuals** (for regression problem, it is usually the RSS, in general, it is **loss function**) of a model, rather than the outcome Y. The mathematical theory of boosting is given in Mohri et al. [2018, Chapter 7].

Let us examine the two kinds of updates.

The weight update:

$$\hat{w}_i \leftarrow \hat{w}_i * e^{-\alpha h(\mathbf{x}_i) y_i},$$

as, $h(\mathbf{x}_i)y_i$ is either $+1$ (if classified correctly by this weak learner) or -1 (otherwise), it follows that this weight update multiplies the weight w_i either by a factor $e^{\alpha} > 1$ if it was classified incorrectly (i.e. increases the weights), or by a factor $e^{-\alpha} < 1$ if it was classified correctly (i.e. decreases the weight).

The normalisation update:

$$Z \leftarrow Z * 2\sqrt{\epsilon(1-\epsilon)}.$$

Previously we established that the normalizer Z is identical to the loss. We can therefore use it to bound the loss function after T iterations:

$$\ell(H) = Z = n \prod_{t=1}^{T} 2\sqrt{\epsilon_t(1-\epsilon_t)},$$

(the factor n comes from the fact that the initial $Z_0 = n$, when all weights are $\frac{1}{n}$.) If we define $c = \max t\epsilon_t$, we can establish

$$\ell(H) \leq n \left[2\sqrt{c(1-c)}\right]^T.$$

The function $c(1-c)$ is maximized at $c = \frac{1}{2}$. But we know that each $\epsilon_t < \frac{1}{2}$ (or else the algorithm would have terminated). Therefore $c(1-c) < \frac{1}{4}$ and we can re-write it as $c(1-c) = \frac{1}{4} - \gamma^2$, for some γ. This leaves us with

$$\ell(H) \leq n\left(1 - 4\gamma^2\right)^{\frac{T}{2}}.$$

In other words, the training loss is decreasing exponentially!

In fact, we can go even further and compute after how many iterations we must have zero training error. Note that the training loss is an upper bound on the training error (defined as $\sum_{i=1}^{n} \delta_{H(\mathbf{x}_i) \neq y_i}$) - simply because $\delta_{H(\mathbf{x}_i) \neq y_i} < e^{-y_i H(\mathbf{x}_i)}$ in all cases. We can then compute the number of steps required until the loss is less than 1, which would imply that not a single training input is misclassified.

$$n\left(1 - 4\gamma^2\right)^{\frac{T}{2}} < 1 \Rightarrow T > \frac{2\log(n)}{\log(\frac{1}{1-4\gamma^2})}.$$

This is an amazing result. It shows that after $O(\log(n))$ iterations your training error must be zero. In practice it often makes sense to keep boosting even after we make no more mistakes on the training set.

In R, adaboost is implemented in `https://github.com/benob/icsiboost`, `ada`, `JOUSBoost`, `adabag` (depends on caret), and the no longer supported `fastAdaboost`.

```
library(JOUSBoost)
adaboost(X, y, tree_depth = 3, n_rounds = 100, verbose = FALSE,
       control = NULL)  # uses rpart.control
```

In Python, random forest is available in the scikit-learn ensemble module.

```
sklearn.ensemble.AdaBoostClassifier(estimator=None, *, n_estimators=50,
  learning_rate=1.0, algorithm='SAMME.R', random_state=None)
sklearn.ensemble.AdaBoostRegressor(estimator=None, *, n_estimators=50,
  learning_rate=1.0, loss='linear', random_state=None)
```

7.2.2 (Generalised) Gradient Boosting

(Generalised) Gradient boosting (in particular, Gradient Boosted Regression Tree) is one of the most popular class of predictive models for learning to rank, the branch of machine learning focused on learning ranking functions, for example for web search engines.

It is an extension to Freund and Schapire's AdaBoost algorithm (Section 7.2.1 and Friedman's gradient boosting machine, which adopts AdaBoost's exponential loss function and Friedman's gradient descent algorithm [Ridgeway, 2007]. It incorporates different loss functions and therefore includes adaptive boosting as a special case. Let ℓ denote a (convex and differentiable) loss function. With a little abuse of notation we write

$$\ell(H) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, H(\boldsymbol{x}_i)). \tag{7.2}$$

Assume we have already finished t iterations and already have an ensemble classifier $H_t(\boldsymbol{x})$. Now in iteration $t+1$ we want to add one more weak learner h^{t+1} to the ensemble. To this end we search for the weak learner that minimises the loss the most,

$$h^{t+1} = \operatorname*{argmin}_{h \in \mathbb{H}} \ell(H_t + \alpha h^t). \tag{7.3}$$

Once h^{t+1} has been found, we add it to our ensemble, i.e. $H_{t+1} := H_t + \alpha h^{t+1}$.

Apart from the boostrapping method, we can find such $h \in \mathbb{H}$ using gradient descent in function space. In function space, the inner product can be defined as $\langle h, g\rangle = \int_x h(x)g(x)dx$. Since we only have training set, we define

$$\langle h, g\rangle = \sum_{i=1}^{n} h(\boldsymbol{x}_i)g(\boldsymbol{x}_i).$$

Given H, we want to find the step-size α and (weak learner) h to minimise the loss $\ell(H+\alpha h)$. By applying the Taylor Approximation on $\ell(H + \alpha h)$,

$$\ell(H + \alpha h) \approx \ell(H) + \alpha\langle\nabla\ell(H), h\rangle. \tag{7.4}$$

This approximation (of ℓ as a linear function) only holds within a small region around $\ell(H)$, i.e. as long as α is small. We therefore fix it to a small constant (e.g. $\alpha \approx 0.1$). With the step-size α fixed, we can use the approximation above to find an almost optimal h:

$$\operatorname*{argmin}_{h\in H} \ell(H + \alpha h) \approx \operatorname*{argmin}_{h\in H}\langle\nabla\ell(H), h\rangle = \operatorname*{argmin}_{h\in\mathbb{H}} \sum_{i=1}^{n} \frac{\partial\ell}{\partial[H(\mathbf{x}_i)]}h(\mathbf{x}_i). \tag{7.5}$$

We can write $\ell(H) = \sum_{i=1}^{n}\ell(H(\mathbf{x}_i)) = \ell(H(x_1), ..., H(x_n))$ (each prediction is an input to the loss function) $\frac{\partial\ell}{\partial H}(\mathbf{x}_i) = \frac{\partial\ell}{\partial[H(\mathbf{x}_i)]}$. So we can do boosting if we have an algorithm $\mathbb{A}$ to solve

$$h^{t+1} = \operatorname*{argmin}_{h\in\mathbb{H}} \sum_{i=1}^{n} \underbrace{\frac{\partial\ell}{\partial[H(\mathbf{x}_i)]}}_{r_i} h(x).$$

We need a function

$$\mathbb{A}(\{(\boldsymbol{x}_1, r_1), \ldots, (\boldsymbol{x}_n, r_n)\}) = \operatorname*{argmin}_{h\in\mathbb{H}} \sum_{i=1}^{n} r_i h(\boldsymbol{x}_i).$$

To make progress, h does not have to be great. We still make progress as long as $\sum_{i=1}^{n} r_i h(\boldsymbol{x}_i) < 0$. A generic boosting (a.k.a Anyboost) algorithm is summarised below.

- Require:
  - An inner product space $(X, \langle,\rangle)$ containing functions mapping from X to some set Y.
  - A class of base classifiers $H \subset X$.
  - A differentiable cost functional $C : lin(H) \to \mathbb{R}$
  - A weak learner $C(H)$ that accepts $H \in lin(H)$ and returns $h \in H$ with a large value of $-\langle\nabla C(H), h\rangle$.
- Let $H_0(x) := O$.
- for $t := 0$ to T do
  - Let $h_{t+1} := L(H_t)$.
  - if $-\langle\nabla C(H_t), h_{t+1}) \leq 0$ then return H_t.
  - Choose w_{t+1}.
  - Let $H_{t+1} := H_t + w_{t+1}h_{t+1}$
- return H_{T+1}.

A few additional things to know:

- The step size α is often referred to as shrinkage.
- Some people do not consider gradient boosting algorithms to be part of the boosting family, because they have no guarantee that the training error decreases exponentially. Often these algorithms are referred to as stage-wise regression instead.
- Inspired by Breiman's Bagging, stochastic gradient boosting subsamples the training data for each weak learner. This combines the benefits of bagging and boosting. One variant is to subsample only $n/2$ data points without replacement, which speeds up the training process.
- One advantage of boosted classifiers is that during test time the computation $H(\boldsymbol{x}) = \sum_{t=1}^{T} \alpha_t h^t(\boldsymbol{x})$ can be stopped prematurely if it becomes clear which way the prediction goes. This is particularly interesting in search engines, where the exact ranking of results is typically only interesting for the top 10 search results. Stopping the evaluation of lower ranked search results can lead to tremendous speed ups. A similar approach is also used by the Viola-Jones algorithm to speed up face detection in images. Here, the algorithm scans regions of an image to detect possible faces. As almost all regions and natural images do not contain faces, there are huge savings if the evaluation can be stopped after just a few weak learners are evaluated. These classifiers are referred to as cascades, that spend very little time on the common case (no face), but more time on the rare interesting case (face). With this approach Viola and Jones were the first to solve face recognition in real-time on low performance hardware (e.g. cameras).

A **gradient boosting procedure** builds iteratively a sequence of approximations

$$F^t : \mathbb{R}^p \to \mathbb{R}, \quad t = 0, 1, 2, ...$$

in a greedy fashion, i.e. F^t is obtained from the previous approximation F^{t-1} in an additive manner: $F^t = F^{t-1} + \alpha h^t$ where α is a **step size** (a hyperparameter) and function $h^t : \mathbb{R}^p \to \mathbb{R}$ is a **base predictor** chosen from a family of functions H (e.g. decision trees of a fixed-depth) in order to minimise the expected loss:

$$h^t = \operatorname*{argmin}_{h \in H} \mathbb{E}\left[\ell(y, F^{t-1}(\boldsymbol{x}) + h(\boldsymbol{x}))\right] = \operatorname*{argmin}_{h \in H} \mathcal{L}(F^{t-1} + h). \tag{7.6}$$

The minimisation problem is usually solved by the *Newton method* or the *steepest descent method* using a second-order approximation of $\mathcal{L}(F^{t-1} + h^t)$ at F^{t-1} or by taking a (negative) gradient step. In particular, the gradient step h^t is chosen to approximate

$$-g'(\boldsymbol{x}, y) := -\frac{\partial}{\partial s} \ell(y, s)\Big|_{s=F^{t-1}(\boldsymbol{x})}.$$

When ℓ is the square loss/gaussian and $\alpha = 1$, the gradient boosting algorithm is summarised below.

Input: training set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ a differentiable loss function $\ell(y, F(x))$.

1. Initialise model with a constant value

$$F^0(\boldsymbol{x}) := \operatorname*{argmin}_{\gamma} \sum_{i=1}^{n} \ell(y_i, \gamma).$$

2. For $t := 1$ to T do

   (a) Compute the negative gradient as the working response: For all $i = 1$ to n,

$$r_{i,t} = -\frac{\partial}{\partial s} \ell(y_i, s)\Big|_{s=F^{t-1}(\boldsymbol{x}_i)} = y_i - F^{t-1}(\boldsymbol{x}_i).$$

(b) Fit a base learner (or weak learner, e.g. tree) $h^t(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{i,t})\}_{i=1}^n$.

(c) Compute multiplier γ_t by solving the following 1D optimisation problem:

$$\gamma_t := \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} \ell\left(y_i, F^{t-1}(\boldsymbol{x}_i) + \gamma h^t(\boldsymbol{x}_i)\right) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} (F^{t-1}(\boldsymbol{x}_i) + \gamma h^t(\boldsymbol{x}_i) - y_i)^2.$$

(d) Let $F^t(\boldsymbol{x}) := F^{t-1}(\boldsymbol{x}) + \gamma_t h^t(\boldsymbol{x})$

3. Return $F^T(\boldsymbol{x})$.

The gradient boosting is implemented in the package `gbm` and has the following syntax.

```
gbm(formula = formula(data), distribution = "bernoulli",
  data = list(), weights, var.monotone = NULL, n.trees = 100,
  interaction.depth = 1, n.minobsinnode = 10, shrinkage = 0.1,
  bag.fraction = 0.5, train.fraction = 1, cv.folds = 0, keep.data = TRUE,
  verbose = FALSE, class.stratify.cv = NULL, n.cores = NULL)

### Practical 12 (regression problem)
#model  = gbm(Sales ~ ., "gaussian", data=data.train)
#yhat = predict(reg.model, data.train)
```

The `n.trees` is the number of iterations T; the `interaction.depth` is the depth of each tree; the `shrinkage` is the learning rate α, the `bag.fraction` is the subsampling rate; the `distribution` is the loss function ℓ in (7.6) and has the options: `gaussian` (squared error), `laplace` (absolute loss), `tdist` (t-distribution loss), `bernoulli` (logistic regression for 0-1 outcomes), `huberized` (huberized hinge loss for 0-1 outcomes), `adaboost` (the AdaBoost exponential loss for 0-1 outcomes), `poisson` (count outcomes), `coxph` (right censored observations), `quantile` or `pairwise` (ranking measure using the LambdaMart algorithm).

In Python, the implementations in scikit-learn library are shown below.

```
sklearn.ensemble.GradientBoostingClassifier(*, loss='log_loss',
  learning_rate=0.1, n_estimators=100, subsample=1.0,
  criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1,
  min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0,
  init=None, random_state=None, max_features=None, verbose=0,
  max_leaf_nodes=None, warm_start=False, validation_fraction=0.1,
  n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0)

sklearn.ensemble.GradientBoostingRegressor(*, loss='squared_error',
  learning_rate=0.1, n_estimators=100, subsample=1.0,
  criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1,
  min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0,
  init=None, random_state=None, max_features=None, alpha=0.9, verbose=0,
  max_leaf_nodes=None, warm_start=False, validation_fraction=0.1,
  n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0)
```

7.2.3 Various Open-Source Extension: XGBoost

Various generalisations/variations of gradient boosting algorithm are developed by commercial coorporations as open source software. The most popular are the Distributed (Deep) Machine Learning Community (DMLC) group's XGBoost and the (Microsoft) LightGBM and a newer model is the (Yandex) CatBoost.

`https://en.wikipedia.org/wiki/XGBoost` (eXtreme Gradient Boosting) is an open-source software library which provides a regularizing gradient boosting framework. It is an optimised

distributed gradient boosting library designed to be highly efficient, flexible and portable. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.

It works as Newton-Raphson in function space unlike gradient boosting that works as gradient descent in function space, a second order Taylor approximation is used in the loss function to make the connection to Newton Raphson method.

A generic unregularized XGBoost algorithm is given in `https://en.wikipedia.org/wiki/XGBoost`:

Input: training set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ a differentiable loss function $\ell(y, F(x))$, a number of weak learners M and a learning rate α.

1. Initialise model with a constant value
$$F^0(\boldsymbol{x}) := \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} \ell(y_i, \gamma).$$
2. For $t := 1$ to T do
   (a) Compute the 'gradients' and 'hessians':
   $$\hat{u}^t(\boldsymbol{x}_i) = \frac{\partial}{\partial s}\ell(y_i, s)\Big|_{s=F^{t-1}(\boldsymbol{x}_i)}, \quad \hat{v}^t(\boldsymbol{x}_i) = \frac{\partial^2}{\partial s^2}\ell(y_i, s)\Big|_{s=F^{t-1}(\boldsymbol{x}_i)}.$$
   (b) Fit a base learner (or weak learner, e.g. tree) using the training set
   $$\left\{x_i, -\frac{\hat{u}^t(x_i)}{\hat{v}^t(x_i)}\right\}_{i=1}^n$$
   by solving the optimisation problem below:
   $$h^t(\boldsymbol{x}) = \alpha\hat{\phi}^t(\boldsymbol{x}), \quad \hat{h}^t = \underset{h\in H}{\operatorname{argmin}} \sum_{i=1}^{n} \frac{1}{2}\hat{v}^t(\boldsymbol{x}_i)\left[-\frac{\hat{u}^t(\boldsymbol{x}_i)}{\hat{v}^t(\boldsymbol{x}_i)} - h(\boldsymbol{x}_i)\right]^2$$
   (c) Let $F^t(\boldsymbol{x}) := F^{t-1}(\boldsymbol{x}) + h^t(\boldsymbol{x})$.
3. Return $F^T(\boldsymbol{x}) = \sum_{t=0}^{T} h^t(\boldsymbol{x})$.

In R, the `xgboost` package has more advantages over the more popular `gbm` package mentioned earlier:

1. Regularisation: `xgboost` implements a 'regularised boosting' technique which helps to reduce overfitting.
2. Parallel Processing: `xgboost` implements parallel processing and is blazingly faster as compared to `gbm`. `xgboost` also supports implementation on Hadoop.
3. High Flexibility: `xgboost` allow users to define **custom optimisation objectives and evaluation criteria**. This adds a whole new dimension to the model and there is no limit to what we can do.
4. Handling Missing Values:
   - `xgboost` has an in-built routine to handle missing values.
   - User is required to supply a different value than other observations and pass that as

a parameter. `xgboost` tries different things as it encounters a missing value on each node and learns which path to take for missing values in future.

5. Tree Pruning:
   - A `gbm` would stop splitting a node when it encounters a negative loss in the split. Thus it is more of a greedy algorithm.
   - `xgboost` on the other hand make splits upto the max_depth specified and then start pruning the tree backwards and remove splits beyond which there is no positive gain.
   - Another advantage is that sometimes a split of negative loss say -2 may be followed by a split of positive loss +10. `gbm` would stop as it encounters -2. But `xgboost` will go deeper and it will see a combined effect of +8 of the split and keep both.
6. Built-in Cross-Validation: `xgboost` allows user to run a cross-validation at each iteration of the boosting process and thus it is easy to get the exact optimum number of boosting iterations in a single run. This is unlike `gbm` where we have to run a grid-search and only a limited values can be tested.
7. Continue on Existing Model: User can start training an `xgboost` model from its last iteration of previous run. This can be of significant advantage in certain specific applications. `gbm` implementation of sklearn also has this feature so they are even on this point.

7.2.4 Various Open-Source Extension: LightGBM

`https://en.wikipedia.org/wiki/LightGBM`, short for light gradient-boosting machine, supports different algorithms including GBDT, GBRT, GBM, multiple-additive regression tree (MART) and random forest. LightGBM has many of XGBoost's advantages, including sparse optimisation, parallel training, multiple loss functions, regularisation, bagging, and early stopping. A major difference between the two lies in the construction of trees. LightGBM does not grow a tree level-wise (row by row) as most other implementations do but grows trees **leaf-wise**. It chooses the leaf it believes will yield the largest decrease in loss. Besides, LightGBM does not use the widely-used sorted-based decision tree learning algorithm, which searches the best split point on sorted feature values, as XGBoost do, but it implements a highly optimised histogram-based decision tree learning algorithm, which yields great advantages on both efficiency and memory consumption. The LightGBM algorithm utilizes two novel techniques called Gradient-Based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB) which allow the algorithm to run faster while maintaining a high level of accuracy.

7.2.5 Various Open-Source Extension: CatBoost

`https://en.wikipedia.org/wiki/CatBoost` provides a gradient boosting framework which among other features attempts to solve for Categorical features using a permutation driven alternative compared to the classical algorithm.

According to `https://deep-and-shallow.com/2020/02/29/the-gradient-boosters-v-catboost/`, CatBoost deals with Categorical variables in a native way. Many studies have shown that One-Hot encoding high cardinality categorical features is not the best way to go for many predictive models, especially in tree based algorithms. Other popular alternatives all come under the umbrella of *Target Statistics* (TS) — Target Mean Encoding (Greedy TS), Leave-One-Out Encoding, etc.

The basic idea of Greedy TS is simple: we replace a categorical value by the mean of all the targets for the training samples with the same categorical value. For example, we have a Categorical value called weather, which has four values — sunny, rainy, cloudy, and snow. We replace "sunny" with the average of the target value for all the training samples where weather was "sunny".

If X_j is the categorical feature we are encoding and k is the specific value in X_j, and n_k is the number of training samples with $X_j = k$, i.e.

$$GreedyTS(X_j = k) = \frac{\sum_{i=1}^{n} y_i|_{X_{ij}=k}}{n_k}.$$

But this is unstable when the number of samples with k is too low or zero. Therefore we use the Laplace Smoothing used in Naive Bayes Classifier to make the statistics much more robust:

$$GreedyTS(X_j = k) = \frac{\sum_{i=1}^{n} y_i|_{X_{ij}=k} + \alpha p}{n_k + \alpha}.$$

where $\alpha > 0$ is a hyperparameter. A common setting for the prior p is the average target value in the dataset.

But these methods usually suffer from something called Target Leakage because we are using our targets to calculate a representation for the categorical variables and then using those features to predict the target. Leave-One-Out Encoding tries to reduce this by excluding the sample for which it is calculating the representation, but is not fool proof.

CatBoost authors propose another idea here, which they call Ordered TS. This is inspired from Online Learning algorithms which get the training examples sequentially in time. And in such cases, the TS will only rely on the training examples in the past. To adapt this idea to a standard offline training paradigm, they imagine a concept of artificial time, but randomly permuting the dataset and considering them sequential in nature.

Then they calculate the TS using only the samples which occured before that particular sample in the artificial time. It is important to note that if we use just one permutation as the artificial time, it would not be very stable and to this end they do this encoding with multiple permutations.

The main motivation for the CatBoost algorithm is, as argued by the authors, the target leakage, which they call Prediction Shift, inherent in the traditional Gradient Boosting models. The high-level idea is quite simple. As we know, any Gradient Boosting model works iteratively by building base learners over base learners in an additive fashion. But since each base learner is build based on the same dataset, the authors argue that there is a bit of target leakage which affects the generalisation capabilities of the model. Empirically, we know that Gradient Boosted Trees has an overwhelming tendency to overfit the data. The only countermeasures against this leakage are features like subsampling, which they argue is a heuristic way of handling the problem and only alleviates it and not completely removes it.

The target shift or the bias is found to be inversely proportional to the size of the dataset, i.e. if the dataset is small, the target leak is much more pronounced. This observation also agrees with our empirical observation that Gradient Boosted Trees tend to overfit to a small dataset.

To combat this issue, they propose a new variant of Gradient Boosting, called Ordered Boosting. The main problem with previous Gradient Boosting was the reuse of the same dataset for each iteration. So, if we have a different dataset for each of the iteration, we would be solving the problem of leakage. But since none of the datasets are infinite, this idea, purely applied, will not be feasible. So, the authors have proposed a practical implementation of the above concept.

It starts out with creating $s + 1$ permutations of the dataset. This permutation is the artificial time that the algorithm takes into account. Let's call it $\sigma_0 \to \sigma_s$. The permutations $\sigma_1 \to \sigma_s$ is used for constructing the tree splits and σ_0 is used to choose the leaf values b_j. In the absence of multiple permutations, the training samples with short "history" will have high variance and hence having multiple permutations ease out that defect.

The gradient statistics required for the tree splits and the target statistics required for the categorical encoding are calculated using the sampled permutation. Once all the trees are built,

the leaf values of the final model F are calculated by the standard gradient boosting procedure using permutation σ_0. When the final model F is applied to new examples from test set, the target statistics are calculated on the entire training data.

CatBoost also differs from the rest of the flock in another key aspect — the kind of trees that is built in its ensemble, by default, is Symmetric Trees or Oblivious Trees. These are trees the same features are responsible in splitting learning instances into the left and the right partitions for each level of the tree.

This has a two-fold effect in the algorithm:

- Regularization: Since we are restricting the tree building process to have only one feature split per level, we are essentially reducing the complexity of the algorithm and thereby regularisation.

- Computational Performance: One of the most time consuming part of any tree-based algorithm is the search for the optimal split at each nodes. But because we are restricting the features split per level to one, we only have to search for a single feature split instead of L splits, where L is the number of nodes in the level. Even during inference these trees make it lightning fast. It was shown to be 8X faster than XGBoost in inference.

Another important detail of CatBoost is that it considers combinations of categorical variables implicitly in the tree building process. This helps it consider joint information of multiple categorical features. But since the total number of combinations possible can explode quickly, a greedy approach is undertaken in the tree building process. For each split in the current tree, CatBoost concatenates all previously used Categorical Features in the leaf with all the rest of the categorical features as combinations and target statistics are calculated on the fly.

Another interesting feature in CatBoost is the inbuilt Overfitting Detector. CatBoost can stop training earlier than the number of iterations we set, if it detects overfitting. there are two overfitting detectors implemented in CatBoost:

- Iter is the equivalent of early stopping where the algorithm waits for n iterations since an improvement in validation loss value before stopping the iterations;

- IncToDec is more slightly involved. It takes a slightly complicated route by keeping track of the improvement of the metric iteration after iteration and also smooths the progression using an approach similar to exponential smoothing and sets a threshold to stop training whenever that smoothed value falls below it.

Following XGBoost's footsteps, CatBoost also deals with missing values separately. There are two ways of handling missing values in CatBoost — Min and Max.

If we select "Min", the missing values are processed as the minimum value for the feature. And if we select "Max", the missing values are processed as the maximum value for the feature. In both cases, it is guaranteed that the split between missing values and others are considered in every tree split.

If LightGBM had a lot of hyperparameters, CatBoost has even more. With so many hyperparameters to tune, GridSearch stops being feasible. It becomes more of an art to get the right combination of parameters for any given problem:

- `one_hot_max_size`: This sets the maximum number of unique values in a categorical feature below which it will be one-hot encoded and not using Target statistics. It is recommended that we do not do our one-hot encoding before we feed in the feature set, because it will hurt both accuracy and performance of the algorithm.

- `iterations`: The number of trees to be built in the ensemble. This has to be tuned with a cv or one of the overfitting detection methods should be employed to make the iteration stop at the ideal iteration.

- `od_type`, `od_pval`, `od_wait`: These three parameters configure the overfitting detector.

  `od_type` is the type of overfitting detector;

  `od_pval` is the threshold for IncToDec (Recommended Range: [10e-10, 10e-2]). Larger the value, earlier Overfitting is detected;

  `od_wait` has different meaning depending on the `od_type`. If it is IncToDec, the `od_wait` is the number of iterations it has to run before the overfitting detector kicks in. If it is Iter, the `od_wait` is the number of iterations it will wait without an improvement of the metric before it stops training.

- `learning_rate`: CatBoost automatically set the learning rate based on the dataset properties and the number of iterations set.

- `depth`: This is the depth of the tree. Optimal values range from 4 to 10. Default Value: 6 and 16 if `growing_policy` is Lossguide

- `l2_leaf_reg`: This is the regularisation along the leaves. Any positive value is allowed as the value. Increase this value to increase the regularisation effect.

- `has_time`: We have already seen that there is an artificial time which is taken to accomplish ordered boosting. But what if our data actually have a temporal order? In such cases set `has_time = True` to avoid using permutations in ordered boosting, but instead use the order in which the data was provided as the one and only permutation.

- `grow_policy`: As discussed earlier, CatBoost builds "SymmetricTree" by default. But sometimes "Depthwise" and "Lossguide" might give better results.

- `min_data_in_leaf` is the usual parameter to control the minimum number of training samples in each leaf. This can only be used in "Lossguide" and "Depthwise".

- `max_leaves` is the maximum number of leaves in any given tree. This can only be used in Lossguide. It is not recommended to have values greater than 64 here as it significantly slow down the training process.

- `rsm` or `colsample_bylevel`: The percentage of features to be used in each split selection. This helps us control overfitting and the values range from (0,1].

- `nan_mode`: Can take values "Forbidden", "Min", "Max" as the three options. "Forbidden" does not allow missing values and will throw an error. Min and Max are mentioned earlier.

The following are some real-world applications:

- JetBrains uses CatBoost for Code Completion;

- Cloudflare uses CatBoost for bot detection;

- Careem uses CatBoost to predict future destinations of the rides.

7.3 Summary

Discriminative models have the **advantage** that they directly address finding an accurate classifier $P(Y|X)$ based on modelling the decision boundary.

The **disadvantage** of the discriminative approach is that they are usually trained as 'black-box' classifiers, with little prior knowledge built used to describe how data for a given class is distributed. E.g. we don't know whether the output is imbalance or how it is distributed.

Generative models have the **advantage** that the information about the structure of the data D (e.g. domain knowledge) is often specified through a generative model $P(X, Y)$. E.g. it is easier to describe certain illness (output) in relation to the age, weight, etc. rather that based on age, weight, etc. to determine the probability of a person to have certain illness.

The **disadvantage** of the generative approach is that it does not target the classification model $P(Y|X)$ directly. When the data D is complex, finding a suitable data model $P(X|Y)$ is difficult!

	Nonparametric	**Parametric**
discriminative	• kNN, wkNN ($k \in \mathbb{Z}$) • Decision tree models, Random Forest, Gradient Boosting Models • Kernel SVM	• LR • Multilogit & ANN • Linear SVM
generative	• NB with curve fitting(?) • Gaussian mixture model	• Naive Bayes (NB) • LDA, QDA

- kNN: Based on distance/similarity measure, inputs need to be numeric, data scaling is recommended. Need to be careful with high dimension data (curse of dimensionality).
- LR: Based on linear algebra and nonlinear iteration, data scaling is usually unnecessary. Only suitable for closely linear data. It has a nice statistical theory.
- Multilogit with regularisation (ElasticNet) & ANN: Based on optimisation, data scaling is recommended. Need large samples of data. Superpowerful but very difficult to train.
- LDA, QDA: The input needs to be numeric and not too nonlinear. Data scaling is recommended otherwise large covariance matrix. The covariance matrix performs terribly with low samples.
- Naive Bayes: Data scaling is not necessary. Tends to be 'bias'. It is used in older email spam filtering system.
- Decision tree models & Random Forest: Scaling is not necessary. Tend to be 'bias'.

Dealing with Imbalanced Data (`https://developers.google.com/machine-learning/data-prep/construct/sampling-splitting/imbalanced-data`):

- Use the right performance metrics:
  - Use the right recall (sensitivity, specificity) and the right precision (PPV, NPV)
  - F1 Score, Cohen's kappa, ROC Curves, etc.
- Try `https://en.wikipedia.org/wiki/Oversampling_and_undersampling_in_data_analysis`or Generate Synthetic Samples, in particular, SMOTE (Synthetic Minority Over-sampling TEchnique, based on NN).
- Try collecting more data: This is possible for spam filtering but impossible for customer survey.