11 Ensemble models: combining multiple learners

- By suitably combining multiple learners the accuracy can be improved (but it need not to).
  - How to generate *base learners* that complement each other?
  - How to *combine* their outputs for maximum accuracy?
- Ex: train an ensemble of L decision trees on L different subsets of the training set and define the ensemble output for a test instance as the majority vote (for classification) or the average (for regression) of the L trees.
- Ensembles of decision trees (random forest, boosted decision trees) are practically among the most accurate models in machine learning.
- Disadvantages:
  - An ensemble of learners is computationally more costly in time and space than a single learner, both at training and test time.
  - A decision tree is interpretable (as rules), but an ensemble of trees is hard to interpret.

Generating diverse learners

- If the learners behave identically, i.e., the outputs of the learners are the same for any given input, their combination will be identical to any individual learner. The accuracy doesn't improve and the computation is slower.
  ☞ *Diversity*: we need learners whose decisions differ and complement each other.
- If each learner is extremely accurate, or extremely inaccurate, the ensemble will barely improve the individual learners (if at all).
  ☞ *Accuracy*: the learners have to be sufficiently accurate, but not very accurate.
- Good ensembles need a careful interplay of accuracy and diversity. Having somewhat inaccurate base learners can be compensated by making them diverse and independent from each other.
- Although one can use any kind of models to construct ensembles, practically it is best to use base learners that are simple and unstable.

Mechanisms to generate diversity

- *Different models*: each model (linear, neural net, decision tree…) makes different assumptions about the data and lead to different classifiers.
- *Different hyperparameters*: within the same model (polynomials, neural nets, RBF networks…), using different hyperparameters leads to different trained models. Ex: number of hidden units in multilayer perceptrons or of basis functions in RBF networks, k in k-nearest-neighbor classifiers, error threshold in decision trees, etc.
- *Different optimization algorithm or initialization*: for nonconvex problems (e.g. neural nets), each local optimum of the objective function corresponds to a different trained model. The local optimum found depends on the optimization algorithm used (gradient descent, alternating optimization, etc.) and on the initialization given to it.
- *Different features*: each learner can use a different (possibly random) subset of features from the whole feature vector. This also makes each learner faster, since it uses fewer features.

- *Different training sets*: each learner is trained on a different subset of the data. We can do this:
  - In parallel, by drawing independent random subsets from the training set, as in *bagging.*
  - Sequentially, by giving more emphasis to instances on which the preceding learners are not accurate, as in *boosting* or *cascading.*
  - By making the subsets local, e.g. obtained by clustering.
  - By defining the main task in terms of several subtasks to be implemented by the learners, as in *error-correcting output codes.*

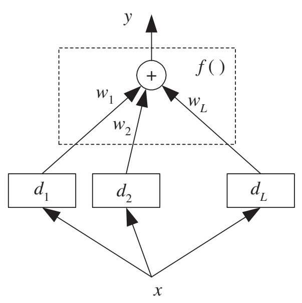
Model combination schemes

Given L trained models (learners), their outputs $\mathbf{y}_1, \dots, \mathbf{y}_L$ for an input $\mathbf{x}$ can be combined:

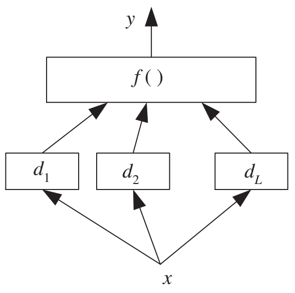
- *In parallel or multiexpert*:
  - *Global approach (learner fusion)*: all learners generate an output and all these outputs are combined. Ex: a fixed combination (*voting, averaging*) or a learned one (*stacking*).
  - *Local approach (learner selection)*: a "gating" model selects one learner as responsible to generate the final output. Ex: *mixture of experts.*
- *In sequence or multistage*: the learners are sorted (usually in increasing complexity) and we apply them in sequence to the input until one of them is confident. Ex: *boosting, cascading.*

In the case of K-class classification, each learner may have K outputs (for the discriminant of each class, or from a softmax). We can have each learner output a single class (the one with largest discriminant or softmax), or have the combination use all K outputs of all L learners.

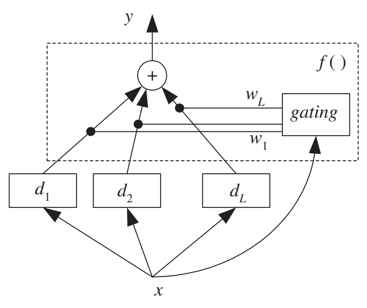
Voting or averaging



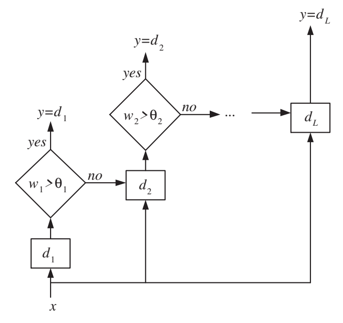
Stacking



Mixture of experts



Boosting or cascading



Voting and averaging

- For discrete outputs (classification):
  - *Majority vote*: the class with most votes wins.
  - *Weighted vote* with the posterior prob. $p_l(C_1|\mathbf{x}), \dots, p_l(C_K|\mathbf{x})$ from each learner $l = 1, \dots, L$.
- For continuous outputs (regression):
  - *Average* $\mathbf{y} = \frac{1}{L}\sum_{l=1}^{L} \mathbf{y}_l$. Possibly weighted: $\mathbf{y} = \frac{1}{L}\sum_{l=1}^{L} w_l \mathbf{y}_l$ with $\sum_{l=1}^{L} w_l = 1$ and $w_1, \dots, w_L \in (0, 1)$.
  - *Median*: more robust to outlying outputs.
- *Bayesian model combination*: in classification, the posterior prob. marginalized over all models is $p(C_k|\mathbf{x}) = \sum_{\text{all models } \mathcal{M}_i} p(C_k|\mathbf{x}, \mathcal{M}_i)\, p(\mathcal{M}_i)$, which can be seen as weighted averaging using as weights the model prior probabilities. Simple voting corresponds to a uniform prior (all models equally likely).

$$\boxed{\begin{array}{l} \mathrm{E}_{p(X,Y)}\{aX+bY\} = a\,\mathrm{E}_{p(X)}\{X\} + b\,\mathrm{E}_{p(Y)}\{Y\},\ a,b\in\mathbb{R}. \\ \mathrm{var}_{p(X)}\{aX\} = a^2\,\mathrm{var}_{p(X)}\{X\} \\ \mathrm{var}_{p(X,Y)}\{X+Y\} = \mathrm{var}_{p(X)}\{X\} + \mathrm{var}_{p(Y)}\{Y\} + 2\,\mathrm{cov}_{p(X,Y)}\{X,Y\} \end{array}}$$

Bias and variance Why (and when) does diversity help?

- Consider L *independent* binary classifiers with success probability $> \frac{1}{2}$ (i.e., better than random guessing) combined by taking a majority vote. One can prove the accuracy increases with L.
  Not necessarily true if they are not independent (e.g. if the L classifiers are equal the accuracy will not change).

- Consider L *iid* random variables $y_1,\dots,y_L$ with expected value $\mathrm{E}\{y_l\} = \mu$ and variance $\mathrm{var}\{y_l\} = \sigma^2$ (expectations wrt $p(y_l)$). Then, the average $y = \frac{1}{L}\sum_{l=1}^L y_l$ has the following moments (expectations wrt $p(y_1,\dots,y_L)$):

$$\mathrm{E}\{y\} = \mathrm{E}\left\{\frac{1}{L}\sum_{l=1}^L y_l\right\} = \frac{1}{L}\sum_{l=1}^L \mathrm{E}\{y_l\} = \mu$$

$$\mathrm{var}\{y\} = \mathrm{var}\left\{\frac{1}{L}\sum_{l=1}^L y_l\right\} = \frac{1}{L^2}\mathrm{var}\left\{\sum_{l=1}^L y_l\right\} = \frac{1}{L^2}\sum_{l=1}^L \mathrm{var}\{y_l\} = \frac{1}{L}\sigma^2.$$

  So the expected value (hence the bias) doesn't change, but the variance (hence the mean squared error) decreases as L increases. If $y_1,\dots,y_L$ are identically but *not independently* distributed:

$$\mathrm{var}\{y\} = \frac{1}{L^2}\mathrm{var}\left\{\sum_{l=1}^L y_l\right\} = \frac{1}{L^2}\left(\sum_{l=1}^L \mathrm{var}\{y_l\} + 2\sum_{i,j=1,\ i\neq j}^L \mathrm{cov}\{y_i,y_j\}\right) = \frac{1}{L}\sigma^2 + \frac{2}{L^2}\sum_{i,j=1,\ i\neq j}^L \sigma_{ij}.$$

  So, if the learners are positively correlated ($\sigma_{ij} > 0$), the variance (and error) is larger than if they are independent. Hence, a well-designed ensemble combination will aim at reducing, if not completely eliminating, positive correlation between learners.
  Further decrease in variance is possible with negatively correlated learners. However, it is impossible to have many learners that are both accurate and negatively correlated.
  Voting or averaging over models with low bias and high variance produces an ensemble with low bias but lower variance, if the models are (somewhat) uncorrelated.

Bagging

- *Bootstrap*: given a training set $\mathcal{X} = \{\mathbf{x}_1,\dots,\mathbf{x}_N\}$ containing N samples, the *bootstrap* generates L subsets $\mathcal{X}_1,\dots,\mathcal{X}_L$ of $\mathcal{X}$, each of size N, as follows: subset $\mathcal{X}_l$ is obtained by sampling at random *with replacement* N points from $\mathcal{X}$.
  - This means that some points $\mathbf{x}_n$ will appear repeated multiple times in $\mathcal{X}_l$ and some other points $\mathbf{x}_n$ will not appear in $\mathcal{X}_l$.
  - The L subsets are partly different from each other. On average, each subset contains $\approx 63\%$ of the training set. Proof: the probability we don't pick instance $\mathbf{x}_n$ after N draws is $(1-\frac{1}{N})^N \approx e^{-1} \approx 0.37$.
- *Stable vs unstable* learning algorithms:
  - A learning algorithm is *unstable* if small changes in the training set cause a large difference in the trained model, i.e., the training algorithm has large variance.
  - Running a stable algorithm on resampled versions of the training set lead to learners with high positive correlation, so little diversity. Using an unstable algorithm reduces this correlation and thus the ensemble has lower variance.
  - Ex. of unstable algorithms: decision trees (the bigger the more unstable), neural nets.
  - Ex. of stable algorithms: linear classifiers or regressors, nearest-neighbor classifier.
- *Bagging (bootstrap aggregating)*: applicable to classification and regression.
  - We generate L (partly different) subsets of the training set with the bootstrap.

  - We train L learners, each on a different subset, using an unstable learning algorithm.
    Since the training sets are partly different, the resulting learners are diverse.
  - The ensemble output is defined as the vote or average (or median) of the learners' outputs.

- *Random forest*: a variation of bagging using *decorrelated* decision trees as base learners.
  - As in bagging, each tree is trained on a bootstrap sample of the training set, and the ensemble output is defined as the vote or average (or median) of the learners' outputs.
  - In addition, the lth tree is constructed with a randomized CART algorithm, independently of the other trees: at each node of the tree we use only a random subset of $m \leq D$ of the original D features. The tree is fully grown (no pruning) so that it has low bias.
    Typically one sets $m = \lfloor\sqrt{D}\rfloor$ for classification.

  Random forests are among the best classifiers in practice, and simpler to train than boosting.

Boosting

- Bagging generates complementary learners through random sampling and unstable learning algorithms. *Boosting actively generates complementary learners by training the next learner on the mistakes of the previous learners.*
- *Weak learner*: a learner that has probability of error $< \frac{1}{2}$ (i.e., better than random guessing on binary classification). Ex: decision trees, *decision stumps* (tree grown to only 1 or 2 levels).
  Strong learner: a learner that can have arbitrarily small probability of error. Ex: neural net.
- There are many versions of boosting, we focus on *AdaBoost.M1*, for classification (it can be applied to regression with some modifications):
  - It combines L weak learners. They have high bias, but the decrease in variance in the ensemble compensates for that.
  - Each learner $l = 1, \dots, L$ is trained on the entire training set $\mathcal{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, but each point $\mathbf{x}_n$ has an associated probability $p_n^{(l)}$ indicating how "important" it is for that learner.
    The training algorithm must be able to use these probabilities $\mathbf{p}^{(l)}$ together with the training set $\mathcal{X}$. Otherwise, this can be simulated by sampling a training set of size N from $\mathcal{X}$ according to $\mathbf{p}^{(l)}$.
  - The first learner uses $p_n^1 = \frac{1}{N}$ (all points equally important).
  - After training learner l on $(\mathcal{X}, \mathbf{p}^{(l)})$, let its error rate be $\epsilon_l = \sum_{n \text{ misclassified by learner } l} p_n^{(l)} \in [0, 1]$. We update the probabilities as follows: for each point $\mathbf{x}_n$, $n = 1, \dots, N$:
    $$p_n^{(l+1)} = \begin{cases} \beta_l p_n^{(l)} & \text{if learner } l \text{ correctly classifies } \mathbf{x}_n \\ p_n^{(l)} & \text{otherwise} \end{cases} \qquad \text{where } \beta_l = \frac{\epsilon_l}{1-\epsilon_l} \in [0, \tfrac{1}{2})$$
    and then renormalize the probabilities so they sum 1: $p_n^{(l+1)} = p_n^{(l+1)} / \sum_{n=1}^N p_n^{(l+1)}$.
    This decreases the probability of a point if it is correctly classified, so the next learner focuses on the misclassified points. That is why learners are chosen to be weak. If they are strong (= very accurate), the next learner's training set will emphasize a few points, many of which could be very noisy or outliers.
- After training, the ensemble output for a given instance is given by a weighted vote, where the weight of learner l is $w_l = \log(1/\beta_l)$ (instead of 1), so the weaker learners have a lower weight.
  Other variations of boosting make the overall decision by applying learners in sequence, as in cascading.
- Boosting usually achieves very good classification performance, although it does depend on the dataset and the type of learner used (it should be weak but not too weak). It is sensitive to noise and outliers.
  A very successful application in computer vision: the Viola-Jones face detector. This is a cascade of AdaBoost ensembles of decision stumps, each trained on a large set of Haar features. It is robust and (with clever image-based operations) very fast for test images.

Cascading is similar to boosting, but the learners are trained sequentially. For example, in boosting the next learner could be trained on the residual error of the previous learner. Another way:

- When applied to an instance $\mathbf{x}$, each learner gives an output (e.g. class label) and a confidence (which can be defined as the largest posterior probability $p(C_k|\mathbf{x})$).
- Learner l is trained on instances for which the previous learner is not confident enough.
- When applying the ensemble to a text instance, we apply the learners in sequence until one is confident enough. We use learner l only if the previous learners $1, \dots, l-1$ are not confident enough on their outputs.
- The goal is to order the learners in increasing complexity, so the early learners are simple and classify the easy instances quickly.