Unit 10
Random Forests and other Ensemble Methods

Prof. Phil Schniter

The Ohio State University

ECE 4300: Introduction to Machine Learning, Sp20

Learning objectives

- Understand intuition behind ensemble methods: "the wisdom of the crowd"
- Understand parallel ensemble methods
  - bagging, pasting
  - random feature selection
- Understand decision trees
  - feature thresholding and decision regions
  - training via top-down tree induction
  - homogeneity metrics: variance reduction, gini impurity
  - ensemble extension: random forests
- Understand boosting, or sequentially trained ensemble methods
  - Adaboost
  - gradient boosting
  - XGBoost

Outline

The wisdom of the crowd

- Suppose that you want to determine the heavy side of a biased coin
  - Although you don't know it, suppose "heads" is 51% and "tails" is 49%
- To predict the heavy side, you flip it N times and choose the majority vote
  - You'll be correct with probability

N	1	11	101	1001	10001
prob	51%	53%	58%	74%	97.7%

  - These values follow from the cdf of the binomial distribution
  - Interpretation: As you use more independent trials, the variance decreases
- Implications for classification:
  - Say you have 10001 binary classifiers, each correct only 51% of the time
    - Alone, each does barely better than random guessing ("weak learner")
  - A majority vote will classify with 97.7% accuracy! ("strong learner")
  - Caveat: The classifiers must generate independent outcomes

Voting classifiers

- Given an ensemble of base classifiers . . .
  - the hard voting classifier takes the majority vote of hard decisions
  - the soft voting classifier averages their soft outputs (i.e., pmfs) and then chooses the maximizing class . . . and usually works better
  - Both are implemented in `sklearn.VotingClassifier`
- The voting classifier will be better than the base classifiers if the base classifiers are sufficiently diverse (i.e., sufficiently independent)
- There are two ways to generate a diverse ensemble of classifiers:
  1. Train the classifiers with different data
  2. Make the classifiers structurally different:
     - Use different classification methods (e.g., LR, SVC, NN)
     - Use same method but add randomness (e.g., choose random features)
- Similar ideas apply to an ensemble of base regressors

Bagging and pasting

- Two main ways to train an ensemble using data diversity:
  1. pasting: draw training samples i without replacement
     - each sample is used exactly once by one predictor
  2. bagging (or bootstrap aggregating): draw training samples i with replacement
     - each sample may be used several times, or never
  - bagging usually works better than pasting
- Bagging:
  - Usually draw n samples (n is total # training samples)
  - For large n, this implies $e^{-1} \approx 37\%$ samples go unused (per predictor)
  - Can use the "out of bag" samples for cross-validation!
- Implemented in `sklearn.BaggingClassifier` & `sklearn.BaggingRegressor`:
  - Choose `bootstrap=True` for bagging (or `=False` for pasting)
  - Choose `oob_score=True` to report out-of-bag cross-validation

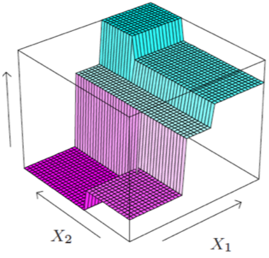
Random patches and random subspaces

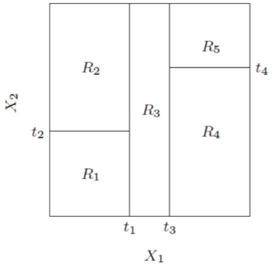
- Using random subsets of features is another way to add diversity
  - Useful when features are high-dimensional and redundant
  - Can sample features with replacement (i.e., "bootstrap") or without
- Terminology:
  - Using both random data and features is known as "random patches"
  - Using random features but full data is known as "random subspaces"
- Implemented in `sklearn.BaggingClassifier` & `sklearn.BaggingRegressor`:
  - Set `max_features` $< d$ for feature randomization
  - Set `bootstrap_features=True` for bootstrapping

Outline

Decision trees

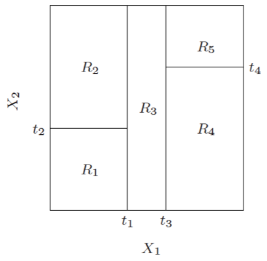
- Consider a supervised learning task (e.g., regression or classification)
  - Suppose data is $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ with $\boldsymbol{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$
- Approach:
  1. Partition domain $\mathbb{R}^d$ into L regions $\{R_\ell\}_{\ell=1}^L$
  2. Output $z_\ell \in \mathbb{R}$ whenever $\boldsymbol{x} \in R_\ell$
     - For regression, could set z_ℓ at sample mean of $\{y_i\}_{i \in S_\ell}$, where $S_\ell = \{i : \boldsymbol{x}_i \in R_\ell\}$
     - For classification, could set z_ℓ at sample mode, i.e., most common value in $\{y_i\}_{i \in S_\ell}$
- Regression example with dimension $d = 2$:

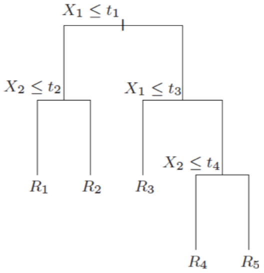




Why are they "trees"?

- Suppose the regions are constructed by thresholding one feature at a time
  - The decision boundaries are always parallel to the coordinate axes
- Then can view prediction as a decision tree:
  - The domain of $\boldsymbol{x}$ (i.e., $\mathbb{R}^d$) forms the root of the tree
  - The decision regions $\{R_\ell\}$ are the L leaves of the tree
- Example with dimension $d = 2$ and $L = 5$ leaves:





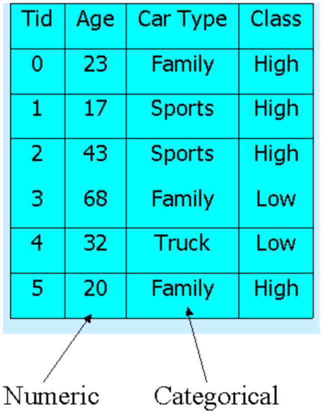
Top-down induction of decision trees

Decision trees are trained in a top-down manner:

- Start with entire data set: $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ and define set $S \triangleq \{1, \dots, n\}$
- First split S into subsets (S_1, S_2), so that labels $\{y_i\}$ are most homogeneous (i.e., similar) within each subset
  - Splitting is performed by thresholding some feature j:
    - Ordinal case: $S_1 = \{i : x_{ij} \leq t\}$ and $S_2 = \{i : x_{ij} > t\}$
    - Categorical case: $S_1 = \big\{i : x_{ij} \in \{A, C\}\big\}$ and $S_2 = \big\{i : x_{ij} \in \{B, D\}\big\}$
  - To find "best" split, must search jointly over feature j and threshold t
  - Many ways to measure homogeneity (will discuss later)
- Then split each subset S_1 and S_2 further, using the same procedure
- Repeat until . . .
  - labels are perfectly homogeneous within a subset (e.g., all y_i are same), or
  - stopping condition: e.g., subsets have min # samples, tree has max depth, etc

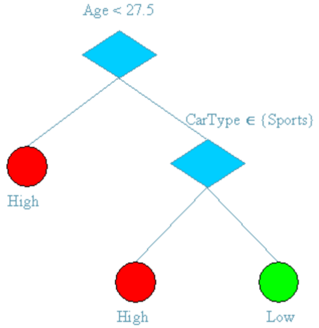
Decision tree example

Example: Risk prediction with $n = 6$ samples, $d = 2$ features, $L = 3$ leaves.
Feature $j = 1$ (age) is ordinal and feature $j = 2$ (car type) is categorical:



Tid	Age	Car Type	Class
0	23	Family	High
1	17	Sports	High
2	43	Sports	High
3	68	Family	Low
4	32	Truck	Low
5	20	Family	High

Numeric
Categorical



Age=40, CarType=Family $\Longrightarrow$ Class=Low

https://web.fhnw.ch/personenseiten/taoufik.nouri/Data%20Mining/Course/Course3/DM-Part%203.htm

Homogeneity metrics for regression

- Recall that the goal is to *maximize homogeneity* within S_1 and S_2
  - Equivalently, we want to *minimize inhomogeneity* within S_1 and S_2
- Let us first consider regression, where labels $y_i \in \mathbb{R}$
- Inhomogeneity could be measured by the variance after splitting $S \to (S_1, S_2)$:
  - The mean in subset S_ℓ is $\mu_\ell \triangleq \frac{1}{|S_\ell|}\sum_{i\in S_\ell} y_i$, for $\ell \in \{1,2\}$
  - The variance in subset S_ℓ is $v_\ell \triangleq \frac{1}{|S_\ell|}\sum_{i\in S_\ell}(y_i - \mu_\ell)^2$, for $\ell \in \{1,2\}$
  - Thus the (average) variance after splitting is $v \triangleq \frac{|S_1|}{|S|}v_1 + \frac{|S_2|}{|S|}v_2$
- Another option is to minimize the absolute error $v_\ell \triangleq \frac{1}{|S_\ell|}\sum_{i\in S_\ell}|y_i - \mu_\ell|$
- We'll see a more sophisticated metric when discussing XGBoost on page 31

Homogeneity metrics for classification

- Now consider K-ary classification, where $y_i \in \{a_1, \ldots, a_K\}\ \forall i$
  - For subset S_ℓ, the empirical label pmf is $\{p_{\ell k}\}_{k=1}^K$, where $p_{\ell k} = \frac{\sum_{i \in S_\ell} \mathbb{1}_{y_i = a_k}}{|S_\ell|}$
- Again, want to *maximize homogeneity*, or minimize inhomogeneity, in S_1 & S_2
- One popular measure of *in*homogeneity is Gini impurity:

$$I_{\mathsf{G}} = \tfrac{|S_1|}{|S|} I_{\mathsf{G},1} + \tfrac{|S_2|}{|S|} I_{\mathsf{G},2}, \quad \text{where} \quad I_{\mathsf{G},\ell} \triangleq 1 - \sum_{k=1}^K p_{\ell k}^2$$

  - Note that $I_{\mathsf{G},\ell} \in \big[0, 1 - \frac{1}{K}\big]$
  - $I_{\mathsf{G},\ell} = 0$ means perfectly pure/homogeneous, i.e., constant y_i for $i \in S_\ell$
- Other criteria include entropy, misclassification error, chi-square
  - See Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning, 2nd Ed.*, 2009

Advantages and disadvantages of decision trees

Advantages:

- Very general
  - Very few assumptions made on data; can work with any dataset
  - Don't need to standardize, but should "balance" the dataset
- Very interpretable: "white box model"
  - Easy to understand how a tree arrives at its prediction
- Prediction is fast: $O(\log_2 n)$ decisions to process a test sample $\boldsymbol{x}$

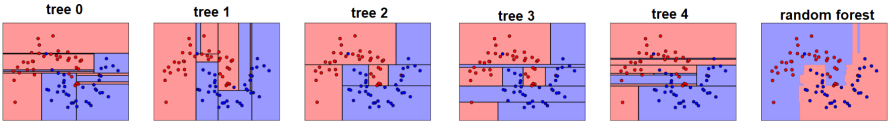
Disadvantages:

- Training can be expensive if all features j are considered at every split
  - Solution: Restrict to a few randomly chosen features j at each split
- Prone to overfitting
  - Highly dependent on training data: changing one sample can change entire tree!
  - Can regularize by enforcing min # samples per subset, max depth, etc

$\leadsto$ There is a good overview of decision trees in the sklearn documentation!

Random forests

- Decision trees tend to overfit: low bias & high variance
- Idea: Use a random forest: A random collection of trees generated by ...
  - using a random training subset to construct each tree (i.e., bagging or pasting)
  - using a random subset of features j per *split*
- As with other ensemble methods, results are averaged to make a prediction
- Implemented in `sklearn.RandomForestClassifier` & `Regressor`

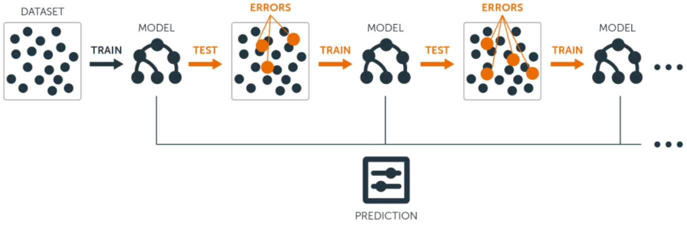


from machine-learning-algorithms-ensemble-methods-bagging-boosting-and-random-forests

Outline

Boosting

- Previously we described ensemble methods that train in parallel
- Now we discuss ensemble methods that train in series
  - This is called "boosting"
- Some of the most popular boosting methods are:
  - adaptive boosting, or Adaboost (1996)
  - gradient boosting (2001)
  - extreme gradient boosting or XGBoost (2016)



Adaboost

- Adaboost trains an ensemble of predictors $\{f_m(\cdot)\}_{m=1}^M$ sequentially as follows:

  For $m = 1 \dots M$,

  1. Train predictor $f_m(\cdot)$ to minimize some weighted loss $\sum_{i=1}^n w_{mi} L(y_i, f_m(\boldsymbol{x}_i))$
  2. Assign larger weights $w_{m+1,i}$ to samples i with higher loss

- The final prediction is done using a weighted average

$$F_M(\boldsymbol{x}) = \sum_{m=1}^M \alpha_m f_m(\boldsymbol{x}), \quad \text{for some learned weights } \{\alpha_m\}$$

- For simplicity, we will focus on the design of binary classifiers $f_m(\cdot) \in \{-1, 1\}$
  - Other versions of Adaboost exist for regression and non-binary classification

Adaboost: Derivation for binary classification

- Consider binary classification, with training data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ for $y_i = \pm 1$
- Adaboost's base classifiers $f_m(\cdot) = \pm 1$ are trained for $m = 1 \dots M$ as follows:
  - Define the step-m boosted classifier $F_m(\boldsymbol{x}) \triangleq F_{m-1}(\boldsymbol{x}) + \alpha_m f_m(\boldsymbol{x})$
  - Initialize $F_0(\boldsymbol{x}) = 0$. For $m = 1, \dots, M$, design α_m and f_m so that F_m minimizes the exponential loss $\mathcal{L}_m = \sum_{i=1}^n e^{-y_i F_m(\boldsymbol{x}_i)}$
- Plugging $F_m(\cdot)$ into $\mathcal{L}_m$, we find that

$$\begin{aligned}
\mathcal{L}_m &= \sum_{i=1}^n e^{-y_i(F_{m-1}(\boldsymbol{x}_i) + \alpha_m f_m(\boldsymbol{x}_i))} = \sum_{i=1}^n \overbrace{e^{-y_i F_{m-1}(\boldsymbol{x}_i)}}^{\triangleq\, w_{mi}} e^{-y_i \alpha_m f_m(\boldsymbol{x}_i)} \\
&= \sum_{i: y_i = f_m(\boldsymbol{x}_i)} w_{mi} e^{-\alpha_m} + \sum_{i: y_i \neq f_m(\boldsymbol{x}_i)} w_{mi} e^{\alpha_m} \quad \text{since } f_m \text{ and } y_i \text{ are } \pm 1 \\
&= \sum_{i=1}^n w_{mi} e^{-\alpha_m} + (e^{\alpha_m} - e^{-\alpha_m}) \sum_{i: y_i \neq f_m(\boldsymbol{x}_i)} w_{mi}
\end{aligned}$$

Adaboost: Derivation for binary classification (cont.)

- From the previous expression, we see that the $\mathcal{L}_m$-minimizing classifier $f_m(\cdot)$ is that which minimizes the weighted misclassification loss

$$\sum_{i:y_i \neq f_m(\boldsymbol{x}_i)} w_{mi} = \sum_{i=1}^{n} w_{mi} \mathbb{1}_{y_i \neq f_m(\boldsymbol{x}_i)}$$

  - At step $m=1$, use $w_{1i} = 1\ \forall i$, and thus train $f_1(\cdot)$ on $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ as usual
  - At step $m>1$, the weight w_{mi} is large if F_{m-1} made a wrong decision on y_i

- The $\mathcal{L}_m$-minimizing α_m can be found by solving $\partial \mathcal{L}_m / \partial \alpha_m = 0$, which gives

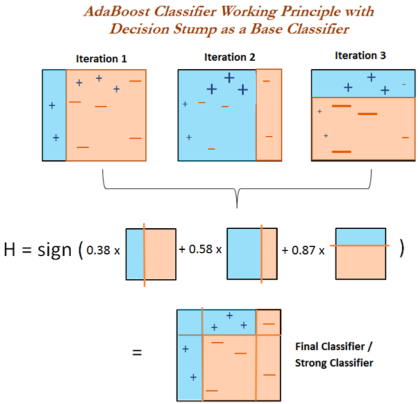
$$\alpha_m = \frac{1}{2}\left(\frac{1-\epsilon_m}{\epsilon_m}\right) \text{ for weighted error rate } \epsilon_m = \frac{\sum_{i:y_i \neq f_m(\boldsymbol{x}_i)} w_{mi}}{\sum_i w_{mi}}$$

- In summary, to train Adaboost, initialize $w_{1i} = 1\ \forall i$ and then, for $m = 1 \dots M$, compute $f_m(\cdot)$, ϵ_m, α_m, $F_m(\cdot)$, and $\{w_{m+1,i}\}_{i=1}^n$

- Once trained, predict the label of a test sample $\boldsymbol{x}$ using $F_M(\boldsymbol{x})$

Adaboost: Implementation

- Adaboost classification implemented via `sklearn.AdaBoostClassifier`
  - By default, it uses 50 decision tree classifiers of depth 1 (“stumps”)
- Adaboost regression implemented via `sklearn.AdaBoostRegressor`
  - By default, it uses 50 decision tree regressors of depth 3


AdaBoost Classifier Working Principle with Decision Stump as a Base Classifier
Iteration 1
Iteration 2
Iteration 3
H = sign (0.38 x
+ 0.58 x
+ 0.87 x
)
=
Final Classifier / Strong Classifier

Gradient boosting: Intuitions

- Gradient boosting is a generalization / re-interpretation of Adaboost
- For intuition, suppose we want to learn a weighted ensemble $F_M(\boldsymbol{x}) = \sum_{m=1}^{M} \alpha_m f_m(\boldsymbol{x})$ that minimizes the RSS $\mathcal{L} = \sum_{i=1}^{n} \big(y_i - F_M(\boldsymbol{x}_i)\big)^2$
- Like with Adaboost, the idea is to train $\{f_m(\cdot)\}$ sequentially
  - At step $m = 1$, we train $f_1(\cdot)$ to minimize RSS on $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$ as usual
  - At step $m = 2, \dots, M$, we construct $F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + \alpha_m f_m(\boldsymbol{x})$
  - But how do we train α_m and $f_m(\cdot) \in \mathcal{F}$?
- Intuition: A "perfect" predictor $F_m(\cdot)$ would yield zero error, i.e.,

$$\forall i : y_i = F_m(\boldsymbol{x}_i) \quad \Leftrightarrow \quad \forall i : \alpha_m f_m(\boldsymbol{x}_i) = y_i - F_{m-1}(\boldsymbol{x}_i) \triangleq r_{mi}$$

  - So $\alpha_m f_m(\cdot)$ should predict the residual error r_{mi} of the previous step!
- Note that the residual $y - F$ is the negative gradient of $\frac{1}{2}(y - F)^2$ w.r.t. F
  - Suggests a way to generalize from RSS cost to a generic cost ...

Gradient boosting: Approach

- Goal: Design predictor $F_M(\cdot)$ to minimize a training loss of the form

$$\mathcal{L} = \sum_{i=1}^{n} L\big(y_i, F_M(\boldsymbol{x}_i)\big) \quad \text{for some } L(\cdot,\cdot)$$

  where $F_M(\cdot)$ is sequentially learned from base predictors $f_m(\cdot) \in \mathcal{F}$
  - Adaboost did this with exponential loss $L(y,F) = e^{-yF}$ and binary $f_m = \pm 1$
  - What about general L and $\mathcal{F}$?
- Ideally, we would like to do the following:
  - Train $f_1 \in \mathcal{F}$ to minimize $\sum_{i=1}^{n} L\big(y_i, f_1(\boldsymbol{x}_i)\big)$, set $F_1 = f_1$
  - For $m = 2 \dots M$: $\quad F_m = F_{m-1} + \displaystyle\arg\min_{\alpha_m, f_m \in \mathcal{F}} \sum_{i=1}^{n} L\big(y_i, F_{m-1}(\boldsymbol{x}_i) + \alpha_m f_m(\boldsymbol{x}_i)\big)$
  - But this optimization is too difficult!
- Idea: Instead of exact optimization, settle for a gradient step, i.e.,

$$\text{For } m = 2 \dots M: \quad F_m = F_{m-1} - \alpha_m \nabla \mathcal{L}(F_{m-1})$$

Gradient boosting: Details

- Problem:
  - We are limited to updates of the form

    $$F_m = F_{m-1} + \alpha_m f_m, \quad \text{for } f_m \in \mathcal{F} \text{ where } \mathcal{F} \text{ is set of base predictors}$$

  - But gradient descent does not constrain the update to $\mathcal{F}$:

    $$F_m = F_{m-1} + \alpha_m \big[-\nabla\mathcal{L}(F_{m-1}) \big], \text{ with } \nabla\mathcal{L}(F_{m-1}) = \sum_{i=1}^{n} \underbrace{\frac{\partial L(y_i, F_{m-1}(\boldsymbol{x}_i))}{\partial F}}_{\triangleq\, -r_{mi}}$$

- Solution: Train $f_m \in \mathcal{F}$ to be *close to* $-\nabla\mathcal{L}(F_{m-1})$
- In particular, for $m = 2, \dots, M$, we do the following:
  - First train $f_m \in \mathcal{F}$ to minimize $\sum_{i=1}^{n} \big(r_{mi} - f_m(\boldsymbol{x}_i)\big)^2$
  - Then choose α_m via line-search:
    $\alpha_m = \arg\min_\alpha \sum_{i=1}^{n} L(y_i, F_{m-1}(\boldsymbol{x}_i) + \alpha f_m(\boldsymbol{x}_i))$
  - Finally, update boosted predictor: $F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + \alpha_m f_m(\boldsymbol{x})$

Gradient boosting: Implementation

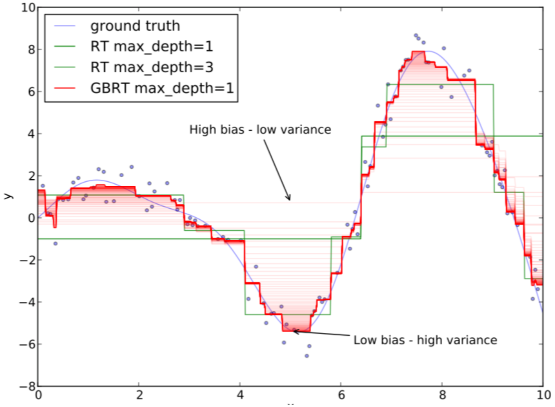
- In practice, it helps to slow down the updates using "learning rate" $\mu \in (0,1]$:

$$F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + \mu \alpha_m f_m(\boldsymbol{x})$$

- Classification implemented via `sklearn.GradientBoostingClassifier`
  - By default, it uses 100 decision tree classifiers of depth 3
  - By default, uses $\mu = 0.1$
- Regression implemented via `sklearn.GradientBoostingRegressor`
  - By default, it uses 100 decision tree regressors of depth 3
  - By default, uses $\mu = 0.1$

Gradient boosting: Example

```
from sklearn.ensemble import GradientBoostingRegressor
est = GradientBoostingRegressor(n_estimators=2000, max_depth=1).fit(X, y)
for pred in est.staged_predict(X):
    plt.plot(X[:, 0], pred, color='r', alpha=0.1)
```



from Gradient Boosted Regression Trees by Prettenhofer & Louppe

Extreme gradient boosting (XGBoost)

- XGBoost is an evolution of gradient boosting that has won many recent machine-learning contests
  - For low-dimensional problems, XGBoost often performs as good as neural nets, but it is much easier to design/train!
- Compared to gradient boosting, XGBoost uses. . .
  - second-order Taylor series approximation
  - regularization
  - sophisticated tree-search
  - many parallelization and hardware acceleration tricks
- The XGBoost package is not part of sklearn, but compatible with it
  - Once you download the XGBoost package, you can use it just like sklearn

XGBoost: Derivation

- XGBoost sequentially designs a boosted predictor $F_M(\boldsymbol{x}) = \sum_{m=1}^{M} f_m(\boldsymbol{x})$
- Similar to gradient boosting, we'd ideally like to do the following:
  - Train $f_1 \in \mathcal{F}$ to minimize $\sum_{i=1}^{n} L(y_i, f_1(\boldsymbol{x}_i))$, set $F_1 = f_1$
  - For $m = 2...M$: $\quad F_m = F_{m-1} + \operatorname*{arg\,min}_{f_m \in \mathcal{F}} \sum_{i=1}^{n} L\big(y_i, F_{m-1}(\boldsymbol{x}_i) + f_m(\boldsymbol{x}_i)\big) + \phi(f_m)$
  - Note that we added regularization to the cost function
  - Unfortunately, this optimization problem is too difficult to solve exactly
- Idea: Simplify using a 2nd-order Taylor series approximation
  - Use $L\big(y_i, F_{m-1}(\boldsymbol{x}_i) + f_m(\boldsymbol{x}_i)\big) \approx L\big(y_i, F_{m-1}(\boldsymbol{x}_i)\big) + g_{mi} f_m(\boldsymbol{x}_i) + \frac{1}{2} h_{mi} f_m^2(\boldsymbol{x}_i)$ where $g_{mi} \triangleq \frac{\partial L(y_i, F_{m-1}(\boldsymbol{x}_i))}{\partial F}$ and $h_{mi} \triangleq \frac{\partial^2 L(y_i, F_{m-1}(\boldsymbol{x}_i))}{\partial^2 F}$
  - Then the optimization problem becomes

$$\operatorname*{arg\,min}_{f_m \in \mathcal{F}} \sum_{i=1}^{n} \left[g_{mi} f_m(\boldsymbol{x}_i)) + \tfrac{1}{2} h_{mi} f_m^2(\boldsymbol{x}_i) \right] + \phi(f_m) \tag{1}$$

XGBoost: Derivation 2

- Now suppose that $f(\cdot)$ is a decision tree with L leaves (recall page 10)
  - Then we can write $f(\boldsymbol{x}) = z_{q(\boldsymbol{x})}$
  - The function $q(\boldsymbol{x}) : \mathbb{R}^d \to \{1, \dots, L\}$ assigns $\boldsymbol{x}$ to a leaf
  - $z_\ell \in \mathbb{R}$ is the predictor output for leaf ℓ
- XGBoost uses the regularization

$$\phi(f) = \frac{\lambda}{2} \sum_{\ell=1}^{L} z_\ell^2 + \gamma L, \quad \text{for some } \lambda > 0 \text{ and } \gamma > 0$$

- With $f_m(\cdot)$ and $\phi(f_m)$ as above, the optimization problem (1) becomes

$$\begin{aligned} &\arg\min_{q,\boldsymbol{z}} \sum_{\ell=1}^{L} \sum_{i:q(\boldsymbol{x}_i)=\ell} \Big(g_i z_\ell + \frac{1}{2} h_i z_\ell^2 \Big) + \frac{\lambda}{2} \sum_{\ell=1}^{L} z_\ell^2 + \gamma L \\ &= \arg\min_{q,\boldsymbol{z}} \sum_{\ell=1}^{L} \Bigg[\Big(\sum_{i:q(\boldsymbol{x}_i)=\ell} g_i \Big) z_\ell + \frac{1}{2} \Big(\lambda + \sum_{i:q(\boldsymbol{x}_i)=\ell} h_i \Big) z_\ell^2 + \gamma \Bigg] \qquad (2) \end{aligned}$$

XGBoost: Derivation 3

- We can optimize over $\boldsymbol{z} \in \mathbb{R}^L$ to yield
$$z_\ell = -\frac{\sum_{i:q(\boldsymbol{x}_i)=\ell} g_i}{\lambda + \sum_{i:q(\boldsymbol{x}_i)=\ell} h_i}, \quad \ell = 1, \dots, L$$
and plug this back into (2) to obtain an optimization problem over $q(\cdot)$:
$$\arg\min_q \sum_{\ell=1}^{L} \left[2\gamma - \frac{\left(\sum_{i:q(\boldsymbol{x}_i)=\ell} g_i\right)^2}{\lambda + \sum_{i:q(\boldsymbol{x}_i)=\ell} h_i} \right]$$

- Using the above loss, the tree $q(\cdot)$ can be designed as on page 11: Choose a feature j and search for the threshold t that maximally reduces loss after a split

- For example, consider the split $S \to (S_1, S_2)$. The loss reduction would be
$$-\frac{\left(\sum_{i\in S} g_i\right)^2}{\lambda + \sum_{i\in S} h_i} - \left[2\gamma - \frac{\left(\sum_{i\in S_1} g_i\right)^2}{\lambda + \sum_{i\in S_1} h_i} - \frac{\left(\sum_{i\in S_2} g_i\right)^2}{\lambda + \sum_{i\in S_2} h_i} \right]$$
If the maximum loss reduction is negative, then it's better *not* to split!

- Further details can be found in the original paper

XGBoost: Example

```python
# First XGBoost model for Pima Indians dataset
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
# split data into train and test sets
seed = 7
test_size = 0.33
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=test_size, ran
# fit model no training data
model = XGBClassifier()
model.fit(X_train, y_train)
# make predictions for test data
y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]
# evaluate predictions
accuracy = accuracy_score(y_test, predictions)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

Running this example produces the following output.

```
Accuracy: 77.95%
```

This is a good accuracy score on this problem, which we would expect, given the capabilities of the model and the modest complexity of the problem.

from develop-first-xgboost-model-python-scikit-learn

Learning objectives

- Understand intuition behind ensemble methods: "the wisdom of the crowd"
- Understand parallel ensemble methods
  - bagging, pasting
  - random feature selection
- Understand decision trees
  - feature thresholding and decision regions
  - training via top-down tree induction
  - homogeneity metrics: variance reduction, gini impurity
  - ensemble extension: random forests
- Understand boosting, or sequentially trained ensemble methods
  - Adaboost
  - gradient boosting
  - XGBoost