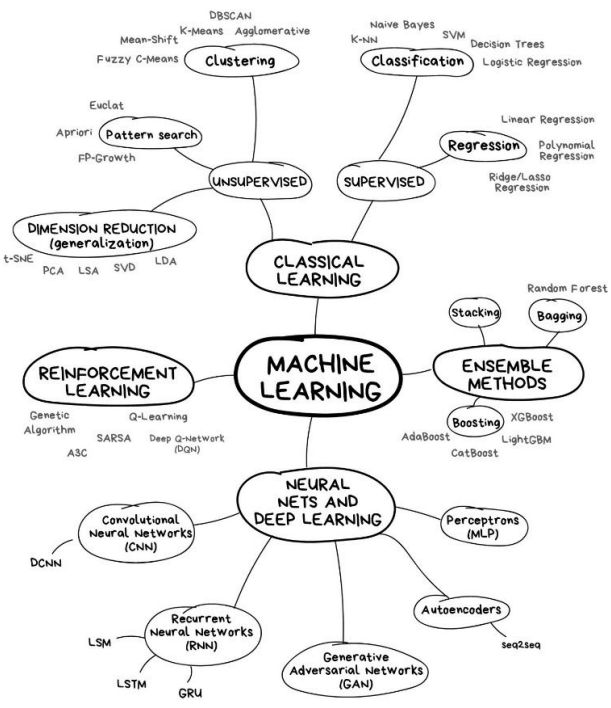
Big Data for Public Policy

Supervised ML

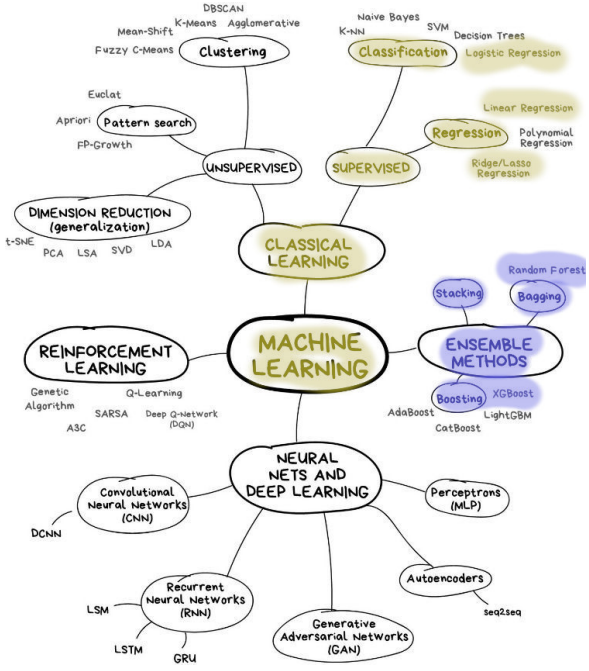
Sergio Galletta

ETHZ Zurich

20/04/2023


MACHINE LEARNING
CLASSICAL LEARNING
Clustering
DBSCAN
K-Means
Agglomerative
Mean-Shift
Fuzzy C-Means
Classification
Naive Bayes
K-NN
SVM
Decision Trees
Logistic Regression
UNSUPERVISED
Pattern search
Euclat
Apriori
FP-Growth
SUPERVISED
Regression
Linear Regression
Polynomial Regression
Ridge/Lasso Regression
DIMENSION REDUCTION (generalization)
t-SNE
PCA
LSA
SVD
LDA
ENSEMBLE METHODS
Stacking
Bagging
Random Forest
Boosting
AdaBoost
XGBoost
LightGBM
CatBoost
REINFORCEMENT LEARNING
Genetic Algorithm
A3C
SARSA
Q-Learning
Deep Q-Network (DQN)
NEURAL NETS AND DEEP LEARNING
Convolutional Neural Networks (CNN)
DCNN
Perceptrons (MLP)
Recurrent Neural Networks (RNN)
LSM
LSTM
GRU
Autoencoders
seq2seq
Generative Adversarial Networks (GAN)

What we will do today



Supervised Learning

- Supervised learning is a type of machine learning where a model is trained on labeled data to make predictions.
- It involves input-output pairs, where the model learns to map inputs to corresponding outputs.
- Common applications include image classification, speech recognition, and spam detection.
- Supervised learning is typically divided into two categories: **classification** and **regression**.

Classification

- **Classification** is a type of supervised learning where the output variable is a **discrete** or **categorical value**. Examples:
  - Identifying potential fraud or illegal activities
  - Predicting disease outbreaks
  - Identifying vulnerable populations
- Popular algorithms for classification include **logistic regression**, **decision trees**
- Evaluation metrics for classification include **accuracy**, **precision**, **recall**, and **F1 score**.

Regression

- **Regression** is a type of supervised learning where the output variable is a **continuous value**. Examples:
  - Predicting life expectation
  - Predicting income
  - Predict population dynamics
- Popular algorithms for regression include **linear regression**, **polynomial regression**.
- Evaluation metrics for regression include **mean squared error**, **root mean squared error**, and **R-squared**.

What do ML Algorithms do? Minimize a cost function

What do ML Algorithms do? Minimize a cost function

- A typical cost function (or loss function) for regression problems is Mean Squared Error (MSE):

$$\text{MSE}(\theta) = \frac{1}{n_D} \sum_{i=1}^{n_D} (h(x_i; \theta) - y_i)^2$$

  - n_D, the number of rows/observations
  - x, the matrix of predictors, with row x_i
  - y, the vector of outcomes, with item y_i
  - $h(x_i; \theta) = \hat{y}$ the model prediction (hypothesis)

Loss functions, more generally

- The loss function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ assigns a score based on prediction and truth:
  - Should be bounded from below, with the minimum attained only for cases where the prediction is correct.
- The average loss for the test set is

$$\mathcal{L}(\theta) = \frac{1}{n_D} \sum_{i=1}^{n_D} L(h(\boldsymbol{x}_i; \theta), \boldsymbol{y}_i)$$

- The estimated parameter matrix θ solves

$$\hat{\theta} = \arg\min_{\theta} \mathcal{L}(\theta)$$

$\hookrightarrow$ optimizes over parameter space; treats the data as constants.

OLS Regression is Machine Learning

- Ordinary Least Squares Regression (OLS), also called simple linear regression, assumes the functional form $h(x;\theta) = x_i'\theta$ and minimizes the mean squared error (MSE)

$$\min_{\hat{\theta}} \frac{1}{n_D} \sum_{i=1}^{n_D} (x_i'\hat{\theta} - y_i)^2$$

OLS Regression is Machine Learning

- Ordinary Least Squares Regression (OLS), also called simple linear regression, assumes the functional form $h(x;\theta) = x_i'\theta$ and minimizes the mean squared error (MSE)

$$\min_{\hat{\theta}} \frac{1}{n_D} \sum_{i=1}^{n_D} (x_i'\hat{\theta} - y_i)^2$$

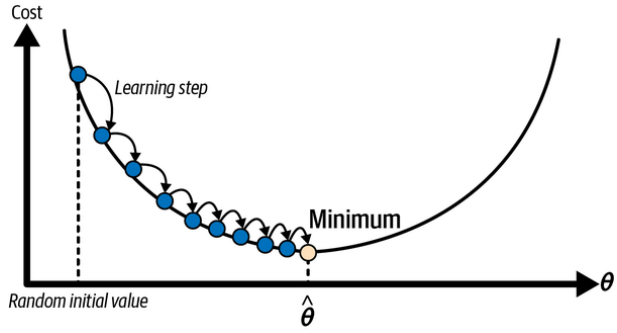
- This minimand has a closed-form solution

$$\hat{\theta} = (\boldsymbol{x}'\boldsymbol{x})^{-1}\boldsymbol{x}'\boldsymbol{y}$$

  - most machine learning models do **not** have a closed form solution $\rightarrow$ use numerical optimization (gradient descent).

Gradient Descent for Loss Function Optimization

- **Gradient descent** is a popular optimization algorithm used in machine learning to minimize the loss function.
- Gradient descent uses the negative gradient to update the model parameters iteratively.
- It moves toward the steepest decrease of the loss function to reach the optimal parameter values.



Gradient Descent for Loss Function Optimization (cont.)

- The partial derivative for feature j is

$$\frac{\partial \text{MSE}}{\partial \theta_j} = \frac{2}{n_D} \sum_{i=1}^{n_D} \underbrace{(h(\theta; \boldsymbol{x}_i) - y_i)}_{\text{error for this obs}} \underbrace{\frac{\partial h(\theta; \boldsymbol{x}_i)}{\partial \theta_j}}_{\text{how } \theta_j \text{ shifts } h(\cdot)}$$

  - $\rightarrow$ estimates how changing θ_j would reduce the error across the whole dataset.

Gradient Descent for Loss Function Optimization (cont.)

- The partial derivative for feature j is

$$\frac{\partial \text{MSE}}{\partial \theta_j} = \frac{2}{n_D} \sum_{i=1}^{n_D} \underbrace{(h(\theta; \mathbf{x}_i) - y_i)}_{\text{error for this obs}} \underbrace{\frac{\partial h(\theta; \mathbf{x}_i)}{\partial \theta_j}}_{\text{how } \theta_j \text{ shifts } h(\cdot)}$$

  - $\rightarrow$ estimates how changing θ_j would reduce the error across the whole dataset.

- The ***gradient*** ∇ gives the vector of these partial derivatives for all features:

$$\nabla_\theta \text{MSE} = \begin{bmatrix} \frac{\partial \text{MSE}}{\partial \theta_1} \\ \frac{\partial \text{MSE}}{\partial \theta_2} \\ \vdots \\ \frac{\partial \text{MSE}}{\partial \theta_j} \end{bmatrix}$$

Gradient Descent for Loss Function Optimization (cont.)

- The partial derivative for feature j is

$$\frac{\partial \text{MSE}}{\partial \theta_j} = \frac{2}{n_D} \sum_{i=1}^{n_D} \underbrace{(h(\theta; \mathbf{x}_i) - y_i)}_{\text{error for this obs}} \underbrace{\frac{\partial h(\theta; \mathbf{x}_i)}{\partial \theta_j}}_{\text{how } \theta_j \text{ shifts } h(\cdot)}$$

  - $\rightarrow$ estimates how changing θ_j would reduce the error across the whole dataset.

- The ***gradient*** ∇ gives the vector of these partial derivatives for all features:

$$\nabla_\theta \text{MSE} = \begin{bmatrix} \frac{\partial \text{MSE}}{\partial \theta_1} \\ \frac{\partial \text{MSE}}{\partial \theta_2} \\ \vdots \\ \frac{\partial \text{MSE}}{\partial \theta_j} \end{bmatrix}$$

- ***Gradient descent*** nudges θ against the gradient (the direction that reduces MSE):

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \text{MSE}$$

  - η = learning rate

- keep nudging until convergence.

Test-Train Split

- ML models can achieve arbitrarily high accuracy in-sample, so **performance should be evaluated out-of-sample**.
- Test-train split involves **randomly dividing the data** into training and testing sets (e.g., 80%/20% or 70%/30%).
- The training set is used to train the model, and the testing set is used to evaluate its performance.
- Advantages: quick evaluation of the model's performance and useful for small to medium-sized datasets
- Disadvantages: high variance due to the randomness of the split not suitable for very large datasets.

Cross-Validation

- The data is divided into **k equal-sized folds**
- Each time, one fold is used as the testing set, and the remaining k-1 folds are used as the training set.
- The model's performance is averaged across the k iterations to get an overall evaluation.
- Advantages: a more reliable estimate of the model's performance and reduces the variance due to randomness by using different combinations of training and testing sets.
- Disadvantages: slightly more complex to implement compared to test-train split and computationally expensive for very large datasets.

Cross-Validation for hyperparameter tuning

- Within the training set:
  - Use cross-validation with grid search to get model performance metrics across subsets of data using different **hyperparameter specs**.
  - Find the best hyperparameters for out-of-fold prediction in the training set.
- Then evaluate model performance in the test set using these hyperparameters.

“Rookie data science mistakes invalidates a dozen medical studies”

When Gilles Vandewiele noticed a large number of studies reporting near-perfect accuracy in predicting whether would-be mothers will undergo premature delivery, his jaw dropped. This was huge.

“Rookie data science mistakes invalidates a dozen medical studies”

When Gilles Vandewiele noticed a large number of studies reporting near-perfect accuracy in predicting whether would-be mothers will undergo premature delivery, his jaw dropped. This was huge.

Now, it seemed that with the help of artificial intelligence, researchers had managed to solve the puzzle. Swept with excitement, Vandewiele, a Ph.D. candidate in machine learning at Ghent University, recruited his peers and set out to replicate the mind-boggling results. Little did he know they were about to embark on a journey of total scientific annihilation, resulting in the invalidation of almost a dozen peer-reviewed articles.

"Rookie data science mistakes invalidates a dozen medical studies"

When Gilles Vandewiele noticed a large number of studies reporting near-perfect accuracy in predicting whether would-be mothers will undergo premature delivery, his jaw dropped. This was huge.

> Now, it seemed that with the help of artificial intelligence, researchers had managed to solve the puzzle. Swept with excitement, Vandewiele, a Ph.D. candidate in machine learning at Ghent University, recruited his peers and set out to replicate the mind-boggling results. Little did he know they were about to embark on a journey of total scientific annihilation, resulting in the invalidation of almost a dozen peer-reviewed articles.

To their amazement, Vandewiele's team discovered that the authors of the too-good-to-be-true studies were performing oversampling *before* splitting the dataset into two. Because the split was done randomly, this had the devastating side-effect of the same datapoints ending up in both the training and testing set. In effect, the models were being shown the questions they were to be assessed on well before the exam! No wonder their results were suspiciously great.

Bias-Variance Trade-off

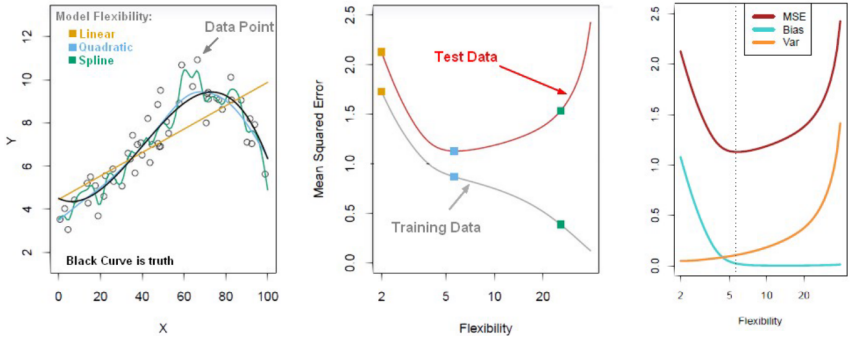
There is often a **trade-off** between **bias** and **variance**:

- **Bias** refers to the **error** that is introduced by approximating a real-world problem with a **simplified model**.
- **Variance** refers to the amount by which the **model would change** if we trained it on a different set of training data.

The goal is to find a model that generalizes well to new data.

- A model with **high bias and low variance** tends to **underfit** the training data and may not generalize well.
- A model with **low bias and high variance** tends to **overfit** the training data and may not generalize well.

Bias-Variance Trade-off



- As the complexity of the model increases, the bias tends to decrease and the variance tends to increase.
- The challenge is to find the right balance between bias and variance in order to achieve good generalization performance.

Regression models $\leftrightarrow$ Continuous outcome

- If the outcome is continuous (e.g., Y = tax revenues collected, or criminal sentence imposed in months of prison) use regression model:
- Problems with OLS:
  - tends to over-fit training data.
  - cannot handle multicollinearity.
- Regularization helps fix this

Regularization

- Minimizing the loss L directly usually results in over-fitting. It is standard to add regularization:

$$\hat{\boldsymbol{\theta}} = \arg\min_{\theta} \frac{1}{n_D} \sum_{i=1}^{n_D} L(h(\boldsymbol{x}_i; \theta), \boldsymbol{y}_i) + \lambda R(\theta)$$

  - $R(\theta)$ is a "regularization function" or "regularizer", designed to reduce over-fitting.
  - λ is a hyperparameter where higher values increase regularization.

Regularization- Ridge and LASSO

- "Ridge" and "Lasso" penalize larger coefficients, shrinking them toward zero:
- Ridge (or L2) penalty:

$$R_2 = \|\theta\|_2^2 = \sum_{j=1}^{n_x} (\theta_j)^2$$

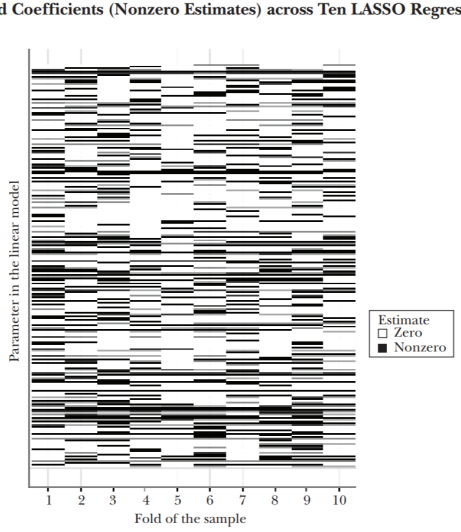
  - also helps select between collinear predictors.
- Lasso (or L1) penalty:

$$R_1 = \|\theta\|_1 = \sum_{j=1}^{n_x} |\theta_j|$$

  - also performs feature selection and outputs a sparse model.
- Lasso can push coefficient to 0, Ridge would keep all coefficients

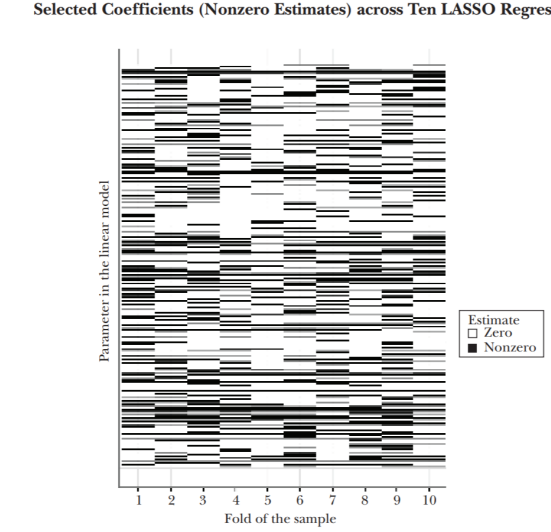
Does lasso pick the "true" model?

Lasso prediction of house prices with 150 variables – which variables are "selected" (non-zero coefficients) by lasso, in ten models trained on separate data subsamples (Mullainathan and Spiess 2017):



Does lasso pick the "true" model?

Lasso prediction of house prices with 150 variables – which variables are "selected" (non-zero coefficients) by lasso, in ten models trained on separate data subsamples (Mullainathan and Spiess 2017):



- The set of lasso-selected variables changes across folds in the data
- $\rightarrow$ Lasso does not pick the "correct" predictors.
  - It just learns the correct $\hat{h}(X)$
  - when predictors are correlated with each other, they are substitutable.

Elastic Net = Lasso + Ridge

The Elastic Net cost function is:

$$\begin{aligned} L(\theta) &= \text{MSE}(\theta) + \lambda_1 R_1 + \lambda_2 R_2 \\ &= \text{MSE}(\theta) + \lambda_1 \sum_{j=1}^{n_x} |\theta_j| + \lambda_2 \sum_{j=1}^{n_x} (\theta_j)^2 \end{aligned}$$

- λ_1, λ_2 = strength of L1 (Lasso) penalty and L2 (Ridge) penalty, respectively.

Elastic Net = Lasso + Ridge

The Elastic Net cost function is:

$$\begin{aligned} L(\theta) &= \text{MSE}(\theta) + \lambda_1 R_1 + \lambda_2 R_2 \\ &= \text{MSE}(\theta) + \lambda_1 \sum_{j=1}^{n_x} |\theta_j| + \lambda_2 \sum_{j=1}^{n_x} (\theta_j)^2 \end{aligned}$$

- λ_1, λ_2 = strength of L1 (Lasso) penalty and L2 (Ridge) penalty, respectively.

In scikit-learn, e-net penalties are parametrized as "`alpha`" = total penalty, and "`l1_ratio`" = proportion of penalty to L1.

```
from sklearn.linear_model import ElasticNet
enet = ElasticNet(alpha=2.0, l1_ratio = .75) # L1 = 1.5, L2 = 0.5
enet.fit(X,y)
```

Model Evaluation in Test Set - Regression

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$\text{RMSE} = \sqrt{\text{MSE}}$$

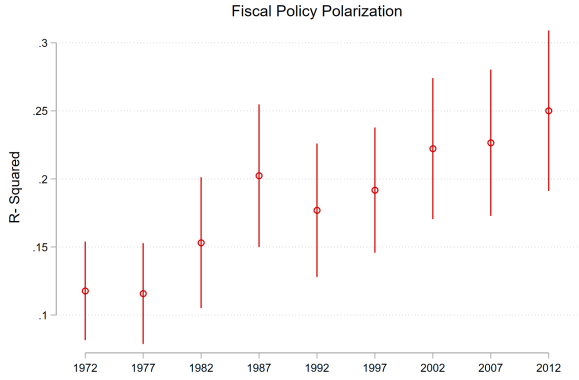
$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

- mean squared error (MSE)
- R-squared (same ranking as MSE, but units are more interpretable)
- mean absolute error (MAE) is less sensitive to outliers.

Example

Fiscal Policy Polarization - Ash and Galletta 2023



Notes: Each marker is the mean of the R-squares of 1000 predictions with different random splits using the modal hyperparameters selected with an initial five-fold cross-validation. The lines are the 95% confidence interval (i.e., 2× SD).

- **Supervised ML** to predict counties' political preferences from their local public budget
  - Outcome to predict: **democratic candidate results in presidential election** (every four years from 1972 to 2012)
  - Predictors: **577 budget variables** (every five years from 1972 to 2012)
- Hyperparameters are $\alpha = 0.1$ and L1 ratio$= 0.5$

Binary Outcome $\leftrightarrow$ Binary Classification

- Binary classifiers try to match a boolean outcome $y \in \{0, 1\}$.
  - The standard approach is to apply a transformation (e.g. sigmoid/logit) to normalize $\hat{y} \in [0, 1]$.
  - Prediction rule is 0 for $\hat{y} < .5$ and 1 otherwise

Binary Outcome $\leftrightarrow$ Binary Classification

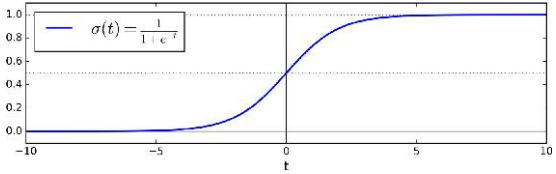
- Binary classifiers try to match a boolean outcome $y \in \{0, 1\}$.
  - The standard approach is to apply a transformation (e.g. sigmoid/logit) to normalize $\hat{y} \in [0, 1]$.
  - Prediction rule is 0 for $\hat{y} < .5$ and 1 otherwise
- The binary cross-entropy (or log loss) is:

$$L(\theta) = \underbrace{-\frac{1}{n_D}}_{\text{negative}} \sum_{i=1}^{n_D} [\underbrace{y_i}_{y_i=1} \underbrace{\log(\hat{y}_i)}_{\text{log prob} y_i=1} + \underbrace{(1-y_i)}_{y_i=0} \underbrace{\log(1-\hat{y}_i)}_{\text{log prob} y_i=0}]$$

Logistic regression

- In **logistic regression** we use a sigmoid transformation:.

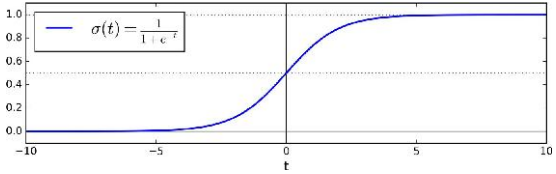
$$\hat{y} = \text{sigmoid}(\boldsymbol{x} \cdot \theta) = \frac{1}{1 + \exp(-\boldsymbol{x} \cdot \theta)}$$



Logistic regression

- In **logistic regression** we use a sigmoid transformation:.

$$\hat{y} = \text{sigmoid}(\boldsymbol{x} \cdot \theta) = \frac{1}{1 + \exp(-\boldsymbol{x} \cdot \theta)}$$



- Plugging into the binary-cross entropy loss gives the logistic regression cost objective:

$$\min_{\theta} \sum_{i=1}^{n_D} -y_i \log(\text{sigmoid}(\boldsymbol{x}_i \cdot \theta)) - [1 - y_i] \log(1 - \text{sigmoid}(\boldsymbol{x}_i \cdot \theta))$$

  - does not have a closed form solution, but it is convex (guaranteeing that gradient descent will find the global minimum).

Logistic regression

- The gradient for one data point is

$$\frac{\partial L(\theta)}{\partial \theta_j} = \underbrace{(\text{sigmoid}(\boldsymbol{x}_i \cdot \theta) - y_i)}_{\text{error for obs } i} \underbrace{x_i^j}_{\text{input } j}$$

Logistic regression

- The gradient for one data point is

$$\frac{\partial L(\theta)}{\partial \theta_j} = \underbrace{(\text{sigmoid}(\boldsymbol{x}_i \cdot \theta) - y_i)}_{\text{error for obs } i} \underbrace{x_i^j}_{\text{input } j}$$

- Like linear regression, logistic regression can be regularized with L1 or L2 penalties.

```
from sklearn.linear_model import LogisticRegression
logit = LogisticRegression(penalty='l2', C = 2.0) # lambda = 1/2
logit.fit(X,y)
```

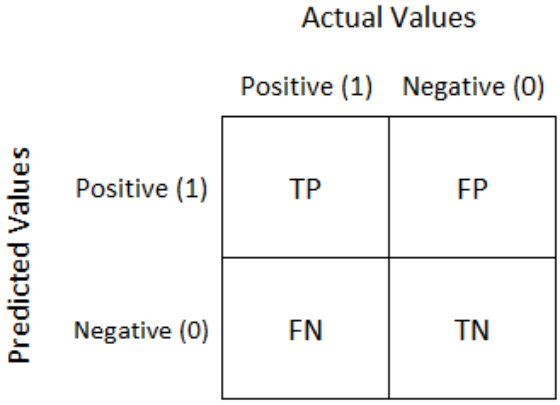
Model Evaluation in Test Set - Classification

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

$$Precision = \frac{TP}{TP+FP}$$

$$Recall = \frac{TP}{TP+FN}$$

$$F1 = 2 \times \frac{Precision \times Recall}{Precision+Recall}$$



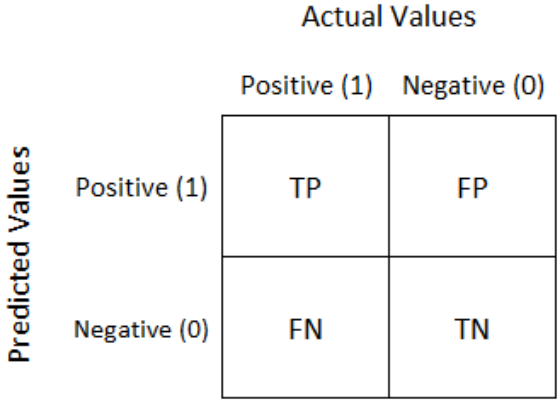
Model Evaluation in Test Set - Classification

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

$$Precision = \frac{TP}{TP+FP}$$

$$Recall = \frac{TP}{TP+FN}$$

$$F1 = 2 \times \frac{Precision \times Recall}{Precision+Recall}$$



- Judicial system: better to have low recall + high precision (many actual-guilty go free, but has very few actual-innocent put in jail)
- Bomb detection: better to have high recall + low precision when (many false alarms, but minimize the number of misses (high recall).

Balanced Accuracy and F1 Score

- If labels are (almost) balanced, then accuracy (share correct predictions) is a decent metric.
  - If not (say 90% in one category), accuracy will be uninformative/misleading.

Balanced Accuracy and F1 Score

- If labels are (almost) balanced, then accuracy (share correct predictions) is a decent metric.
  - If not (say 90% in one category), accuracy will be uninformative/misleading.
- A standard metric in this case is **balanced accuracy** = the average recall in both classes:

$$\text{Balanced Accuracy} = \frac{1}{2}\left(\frac{\text{TP}}{\text{TP} + \text{FN}} + \frac{\text{TN}}{\text{TN} + \text{FP}}\right)$$

  - $\rightarrow$ equal to accuracy when classes are balanced, or performance is the same across classes.

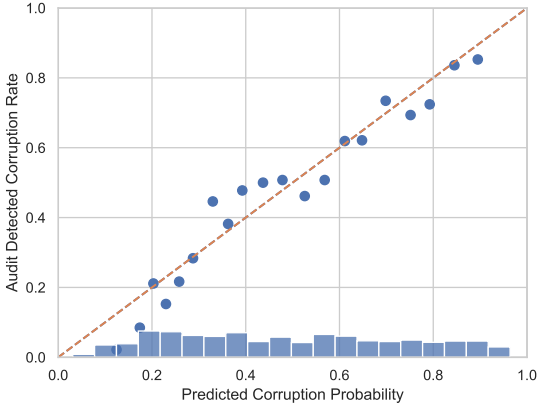
Balanced Accuracy and F1 Score

- Another standard metric is F_1 score = the harmonic mean of precision and recall:

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

  - penalizes both false positives and false negatives. still ignores true negatives.

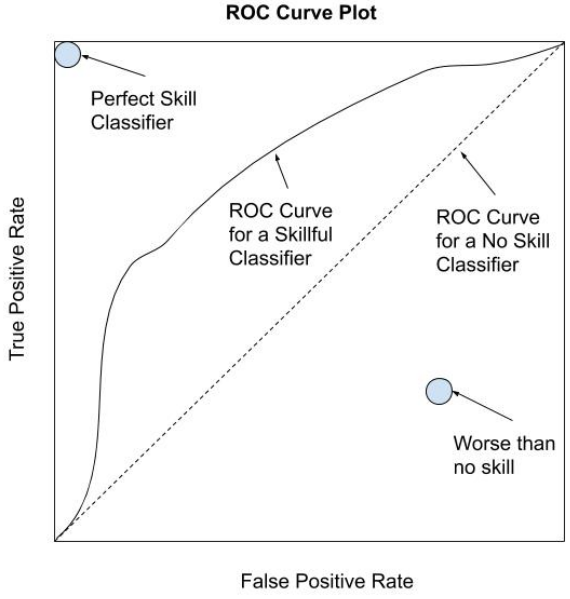
Evaluating Classification Models: Calibration Curves



- Plotting the binned fraction in a category (Y axis) against the predicted probability in a category (X axis):
- Provides evidence of whether the classifer is replicating the conditional distribution of the outcome.

```
from seaborn import regplot
regplot(y_test, y_pred, x_bins=20)
```

ROC (receiver operating characteristic) Curve and AUC



AUC = area under the curve

- provides an aggregate measure of performance across all possible classification thresholds.

Interpreting AUC:

- = probability that the model (correctly) ranks a random positive example more highly than a random negative example.

Example

Media Slant is Contagious - Widmer, Galletta and Ash 2021

- News show transcripts for FNC, CNN, and MSNBC (40,000 episodes)
- 5,000-bigram dictionary, we select the 2,000 most predictive features

$$\widehat{FNC}_m = \Pr[FNC_m = 1|B_m] = \frac{1}{1 + \exp(-\psi' B_m)}$$

$$J(\psi) = -\frac{1}{M^*} \sum_{m=1}^{M^*} \left(FNC_m \log(\widehat{FNC}_m) + (1 - FNC_m) \log(1 - \widehat{FNC}_m) \right) + \lambda |\psi|_2$$

Example

Media Slant is Contagious - Widmer, Galletta and Ash 2021

- News show transcripts for FNC, CNN, and MSNBC (40,000 episodes)
- 5,000-bigram dictionary, we select the 2,000 most predictive features

$$\widehat{FNC}_m = \Pr[FNC_m = 1|B_m] = \frac{1}{1+\exp(-\psi' B_m)}$$

$$J(\psi) = -\frac{1}{M^*}\sum_{m=1}^{M^*}\Big(FNC_m \log(\widehat{FNC}_m) + (1-FNC_m)\log(1-\widehat{FNC}_m)\Big) + \lambda|\psi|_2$$

	Predicted CNN	**Predicted FNC**
Actual CNN	38.3% (235K)	11.7% (72K)
Actual FNC	15.0% (92K)	35.0% (215K)

Multi-Class Models

Many interesting machine learning problems involve multiple un-ordered categories:

- categorizing a case by area of law.
- predicting the political party of a speaker in a multi-party system.
- predicting authorship from documents
- image classification or object detection

Multiple Classes: Setup

- The outcome is $y_i \in \{1, ..., k, ..., n_y\}$ output classes, which can also be represented as a one-hot vector

$$\boldsymbol{y}_i = \{1[y_i = 1], ..., 1[y_i = n_y]\}$$

Multiple Classes: Setup

- The outcome is $y_i \in \{1, ..., k, ..., n_y\}$ output classes, which can also be represented as a one-hot vector

$$\boldsymbol{y}_i = \{1[y_i = 1], ..., 1[y_i = n_y]\}$$

- We want to learn a vector function

$$\boldsymbol{y} = \boldsymbol{h}(\boldsymbol{x}, \theta)$$

outputing a vector of probabilities across outcomes as a function of the inputs:

$$\hat{\boldsymbol{y}} = \{\hat{y}^1, ..., \hat{y}^{n_y}\}, \hat{y}^k \in [0, 1] \ \forall k$$

  - for prediction, select the highest-probability class:

$$\tilde{y} = \arg\max_k \hat{y}_{[k]}$$

Categorical Cross Entropy

- The standard loss function in multinomial classification is **categorical cross entropy**:

$$L(\theta) = -\sum_{k=1}^{n_y} \boldsymbol{y}_{[k]} \log(\hat{\boldsymbol{y}}_{[k]}(\boldsymbol{x}, \theta))$$

  - measures dissimilarity between the true label distribution $\boldsymbol{y}$ and the predicted label distribution $\hat{\boldsymbol{y}}$.

Categorical Cross Entropy

- The standard loss function in multinomial classification is **categorical cross entropy**:

$$L(\theta) = -\sum_{k=1}^{n_y} \boldsymbol{y}_{[k]} \log(\hat{\boldsymbol{y}}_{[k]}(\boldsymbol{x}, \theta))$$

  - measures dissimilarity between the true label distribution $\boldsymbol{y}$ and the predicted label distribution $\hat{\boldsymbol{y}}$.
- Since there is just one true class ($y = 1$ for one class k^*, and zero for others), simplifies to

$$L(\theta) = -\log(\hat{\boldsymbol{y}}_{[k^*]}(\boldsymbol{x}, \theta))$$

  - Rewards putting higher probability on the true class, ignores distribution of probabilities on other classes.
  - function is convex $\rightarrow$ gradient descent will find the optimum.

Multinomial Logistic Regression

Multinomial logistic regression computes probabilities for each class k using the softmax transformation

$$\hat{y}_k(\boldsymbol{x}_i) = \Pr(y_i = k) = \frac{\exp(\theta_k' \boldsymbol{x}_i)}{\sum_{l=1}^{n_y} \exp(\theta_l' \boldsymbol{x}_i)}$$

- softmax is the multiclass generalization of sigmoid. can then interpret $\hat{y}$ as probabilities.
- n_x features and n_y output classes $\rightarrow$ there is a $n_y \times n_x$ parameter matrix Θ, where the parameters for each class θ_k are stored as rows.
- the prediction $y_i \in \{1, ..., n_y\}$ is determined by the highest-probability category.

Regularized Multinomial Logistic

- The L2-penalized logistic regression loss (the default in sklearn) is

$$\mathcal{L}(\theta) = -\frac{1}{n_D}\sum_{i=1}^{n_D}\log\frac{\exp(\theta'_{k^*}\boldsymbol{x}_i)}{\sum_{l=1}^{n_y}\exp(\theta'_l\boldsymbol{x}_i)} + \lambda\sum_{j=1}^{n_x}\sum_{k=1}^{n_y}(\theta_{[j,k]})^2$$

  - λ = strength of L2 penalty (could also add lasso penalty)
  - as before, predictors should be scaled to the same variance.
  - if y is categorical, sklearn automatically uses multinomial logistic

```
from sklearn.linear_model import LogisticRegression
logit = LogisticRegression()
logit.fit(X,y)
```

Multi-Class Confusion Matrix

		Predicted Class		
		Class A	Class B	Class C
3***True Class**	Class A	Correct A	A, classed as B	A, classed as C
	Class B	B, classed as A	Correct B	B, classed as C
	Class C	C, classed as A	C, classed as B	Correct C

- More generally, can have a confusion matrix M with items M_{ij} (row i, column j).

Multi-Class Performance Metrics

Confusion matrix M with items M_{ij} (row i, column j).

$$\text{Precision for } k = \frac{\text{True Positives for } k}{\text{True Positives for k} + \text{False Positives for } k} = \frac{M_{kk}}{\sum_l M_{lk}}$$

$$\text{Recall for } k = \frac{\text{True Positives for } k}{\text{True Positives for } k + \text{False Negatives for } k} = \frac{M_{kk}}{\sum_l M_{kl}}$$

$$F_1(k) = 2 \times \frac{\text{precision}(k) \times \text{recall}(k)}{\text{precision}(k) + \text{recall}(k)}$$

- in sklearn, syntax is the same as binary case.

Metrics for whole model

$$\text{Balanced Accuracy} = \frac{1}{n_y} \sum_k \text{Recall for } k$$

Metrics for whole model

$$\text{Balanced Accuracy} = \frac{1}{n_y} \sum_k \text{Recall for } k$$

- **Macro-averaging**: average of the per-class precision, recall, and F1, e.g.

$$F_1 = \frac{1}{n_y} \sum_{k=1}^{n_y} F_1(k)$$

  - weights all classes equally

Metrics for whole model

$$\text{Balanced Accuracy} = \frac{1}{n_y} \sum_k \text{Recall for } k$$

- **Macro-averaging**: average of the per-class precision, recall, and F1, e.g.

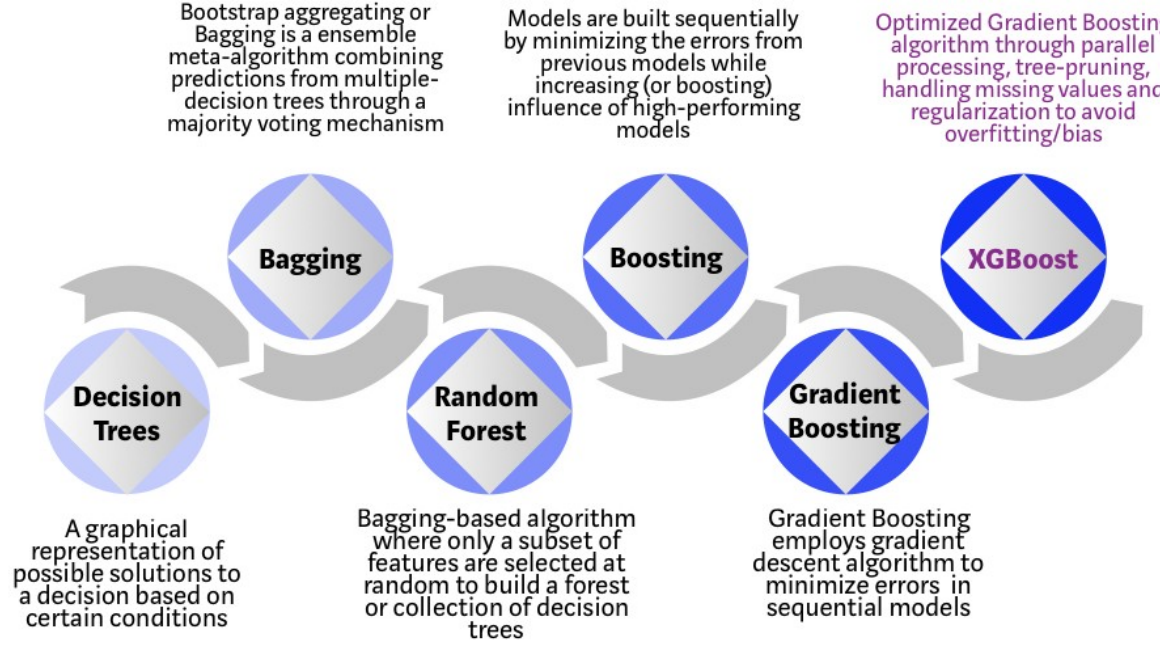
$$F_1 = \frac{1}{n_y} \sum_{k=1}^{n_y} F_1(k)$$

  - weights all classes equally
- Both of these approaches up-weight frequent classes:
  - **Micro-averaging**: Compute model-level sums for true positives, false positives, and false negatives; compute precision/recall from model sums.
  - "**Weighted**": computed like macro-averaging, but classes are weighted by true frequency.

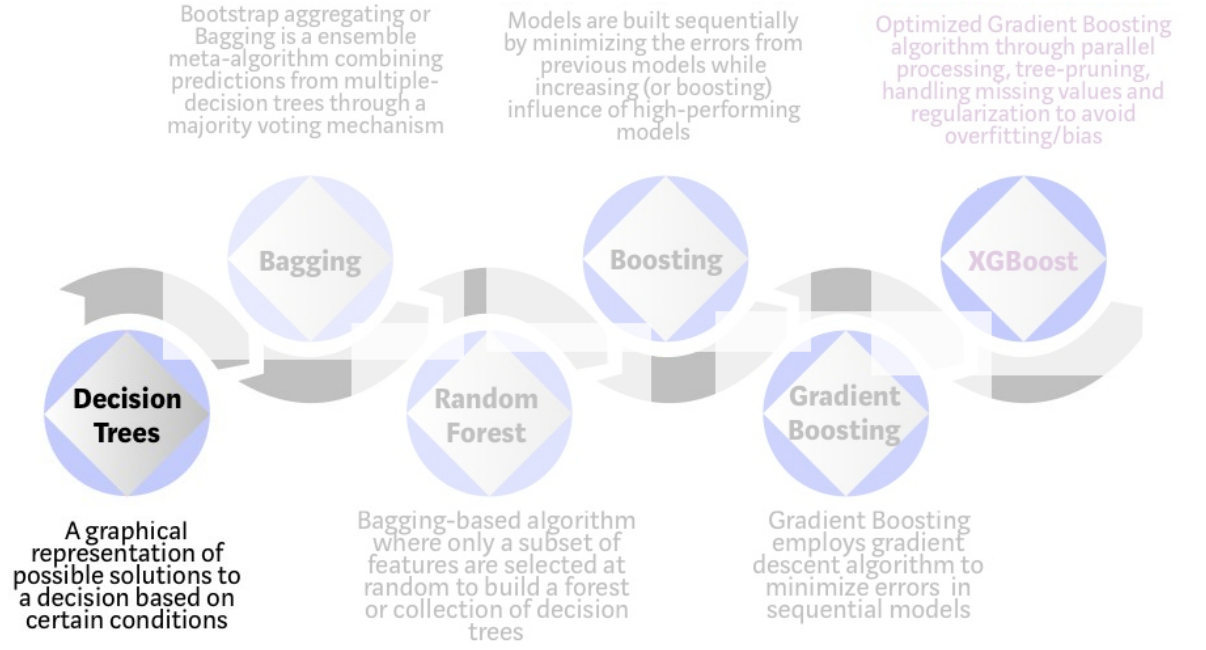
Ensemble Learning

- Ensemble Learning is a machine learning technique that combines multiple models to improve the accuracy and stability of a prediction.
- The main idea behind ensemble learning is to build a group of models that are diverse and complementary in terms of their strengths and weaknesses.
- The models are then combined using some aggregation method to make the final prediction.
- Among the most common types of ensemble learning techniques, there are Bagging, Boosting, Random Forest and Gradient Boosting

Ensemble Learning with XGBoost



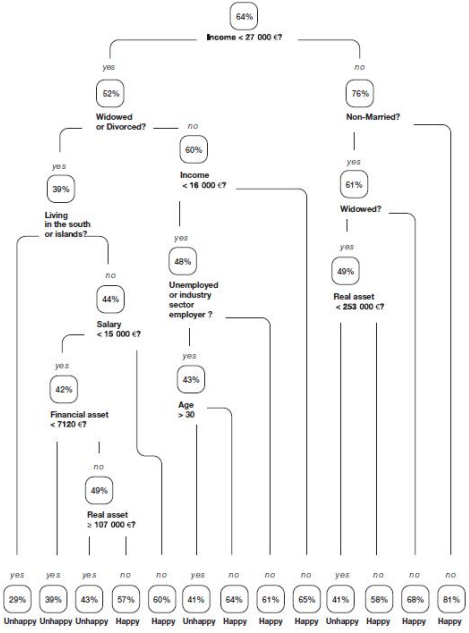
XGBoost Ingredients: Decision Trees



Decision Trees

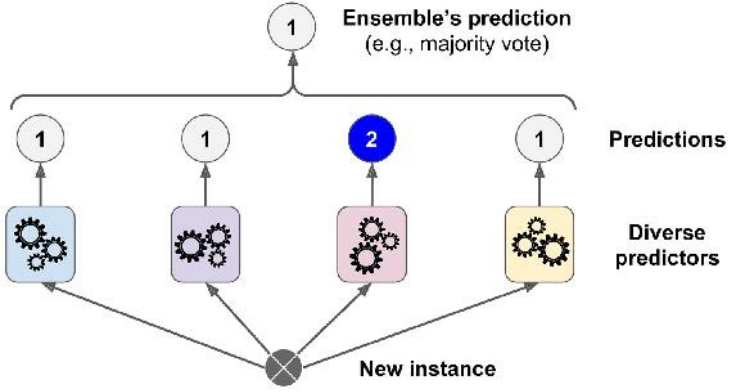
- Decision trees learn a series of binary splits in the data based on hard thresholds.
- The partitioning is done recursively by selecting the feature that best splits the data according to some criterion (e.g., information gain (entropy), Gini index (purity)).
- Can have additional splits as you move through the tree.
- fast and interpretable, but performance is often poor.

Example - On the determinants of happiness - Galletta 2016



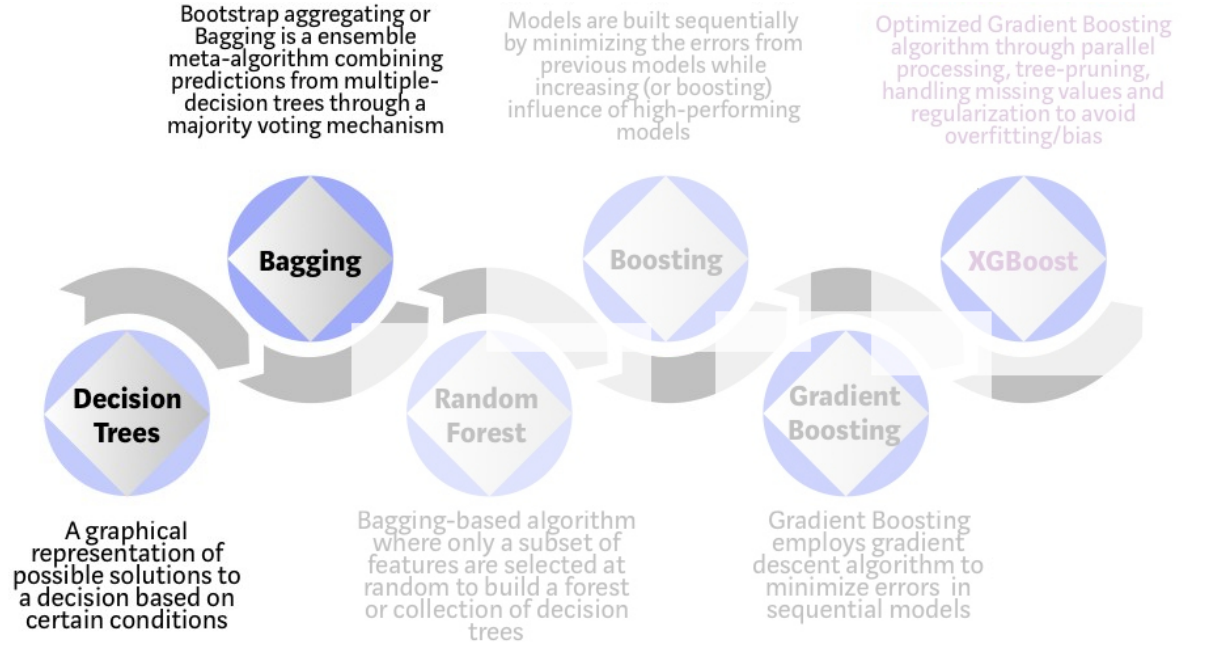
- Sample of 50,000 Italians surveyed by the Bank of Italy
- Apply a CART to predict happiness based on socio-demographic characteristics
- The least happy: poor, divorced/widowed and living in the south
- The happiest: rich and married

Voting Classifiers



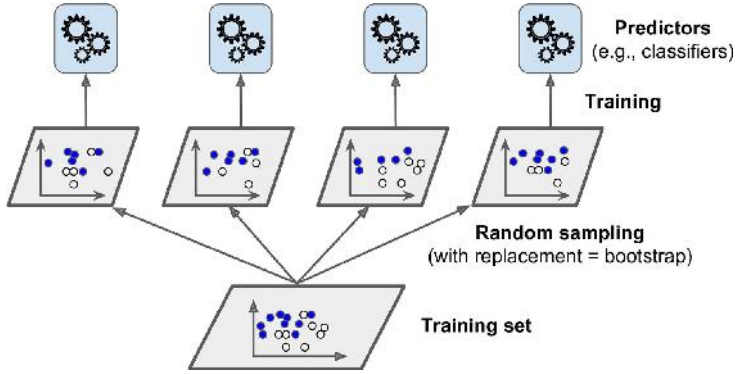
- voting classifiers (ensembles of different models that vote on the prediction) generally out-perform the best classifier in the ensemble.
  - More diverse algorithms will make different types of errors, and improve your ensemble's robustness.

XGBoost Ingredients: Bagging



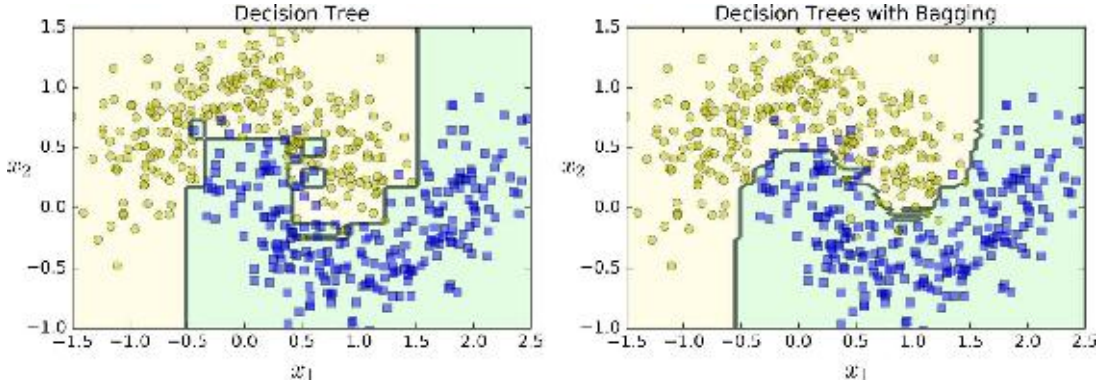
Bagging (Bootstrap Aggregation)

- Rather than use the same data on different classifiers, one can use different subsets of the data on the same classifier:



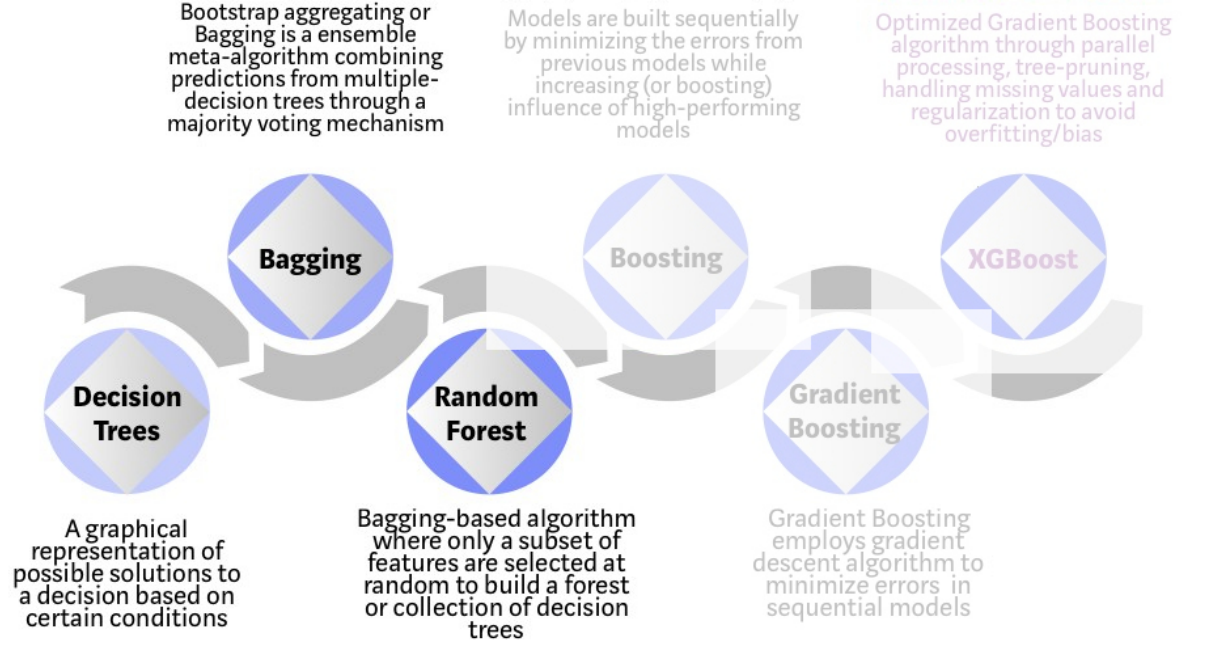
  - can also use different subsets of features across subclassifiers.

Bagging Benefits



- A bootstraped ensemble generally has a similar bias but lower variance than a single predictor trained on all the data.
- Predictors can be trained in parallel using separate CPU cores.

XGBoost Ingredients: Random Forests



Random Forests

Random Forests are optimized ensembles of bootstrapped decision trees:

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier()
rfc.fit(X,y)
```

Random Forests

Random Forests are optimized ensembles of bootstrapped decision trees:

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier()
rfc.fit(X,y)
```

1. Randomly selects a subset of the instances from the training data to create a new dataset for each tree (similar to bagging).

Random Forests

Random Forests are optimized ensembles of bootstrapped decision trees:

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier()
rfc.fit(X,y)
```

1. Randomly selects a subset of the instances from the training data to create a new dataset for each tree (similar to bagging).
2. At each tree split, a random sample of features is drawn, only those features are considered for splitting.

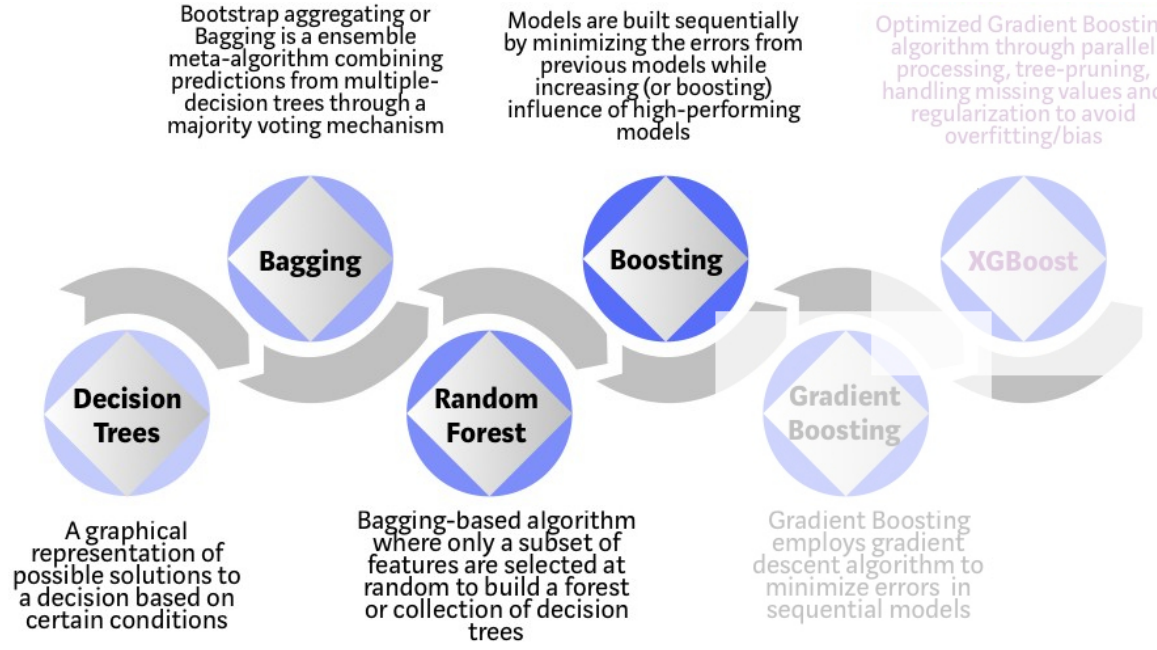
Random Forests

Random Forests are optimized ensembles of bootstrapped decision trees:

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier()
rfc.fit(X,y)
```

1. Randomly selects a subset of the instances from the training data to create a new dataset for each tree (similar to bagging).
2. At each tree split, a random sample of features is drawn, only those features are considered for splitting.
3. For each tree, error rate is computed using data outside its bootstrap sample.

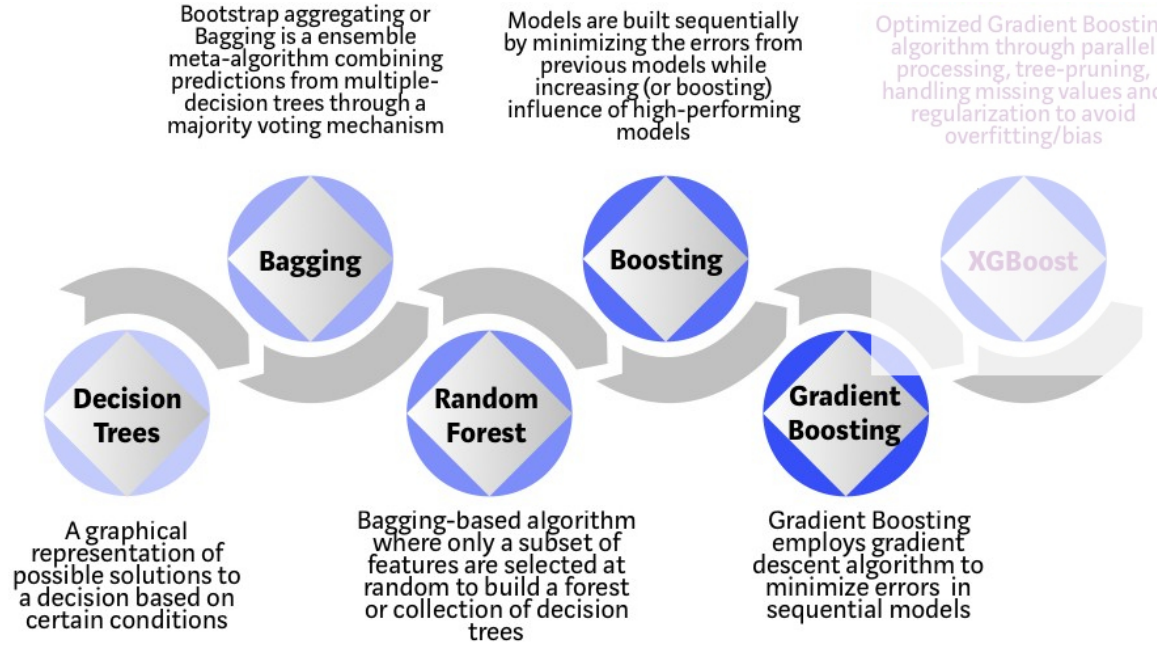
XGBoost Ingredients: Boosting



XGBoost Ingredients: Boosting

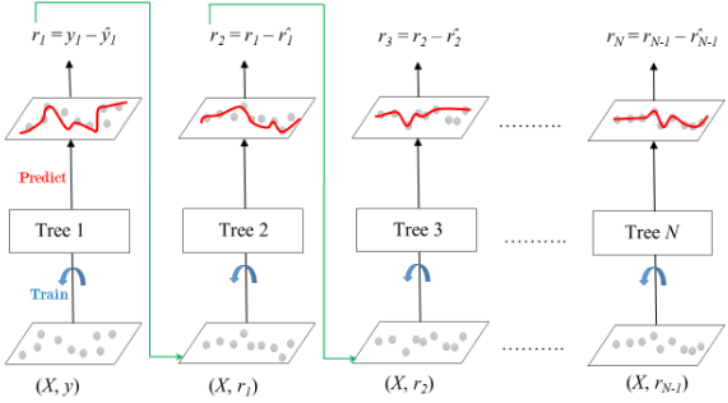
- Boosting is a popular ensemble learning technique that combines multiple weak models to create a strong model.
- The weak models are trained iteratively, with each subsequent model focusing on the misclassified instances from the previous model.
- The final prediction is made by combining the predictions of all the weak models using a weighted majority vote.

XGBoost Ingredients: Gradient Boosting

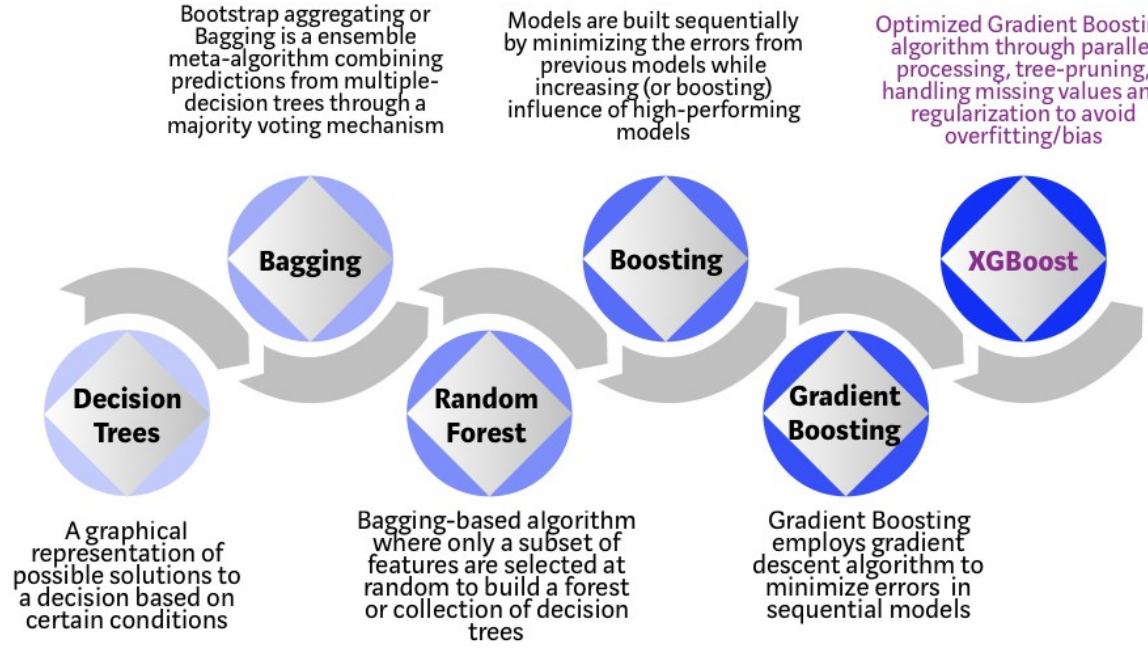


Gradient Boosting Machines

- Gradient Boosting uses gradient descent to optimize the loss function of the weak models.
- In Gradient Boosting, each weak model is trained to minimize the residual error between the predicted and actual values, with the subsequent model focusing on the residual errors of the previous model.



XGBoost


Bootstrap aggregating or Bagging is a ensemble meta-algorithm combining predictions from multiple-decision trees through a majority voting mechanism
Models are built sequentially by minimizing the errors from previous models while increasing (or boosting) influence of high-performing models
Optimized Gradient Boosting algorithm through parallel processing, tree-pruning, handling missing values and regularization to avoid overfitting/bias
Bagging
Boosting
XGBoost
Decision Trees
Random Forest
Gradient Boosting
A graphical representation of possible solutions to a decision based on certain conditions
Bagging-based algorithm where only a subset of features are selected at random to build a forest or collection of decision trees
Gradient Boosting employs gradient descent algorithm to minimize errors in sequential models

XGBoost

- XGBoost (Extreme Gradient Boosting) is an optimized implementation of Gradient Boosting that uses a combination of advanced regularization techniques and system optimization to improve performance and speed.
- XGBoost has several advantages over traditional Gradient Boosting, including:
  - Improved performance due to parallelization and distributed computing.
  - Regularization techniques such as L1 and L2 regularization, and tree pruning to prevent overfitting.
  - Handling of missing data and automatic handling of categorical features.
  - Built-in cross-validation and early stopping to improve generalization.

XGBoost

- Feurer et al (2018) find that XGBoost beats a sophisticated AutoML procedure with grid search over 15 classifiers and 18 data preprocessors.
- A good starting point for any machine learning task:
  - easy to use
  - actively developed
  - efficient / parallelizable
  - provides model explanations
  - takes sparse matrices as input

Complicated in Theory, Easy in practice

```
from xgboost import XGBClassifier
model = XGBClassifier()

model.fit(X_train, y_train,
          early_stopping_rounds=10,
          eval_metric="logloss",
          eval_set=[(X_eval, y_eval)]
          )

y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
```

Example

Predicting corruption - Ash, Galletta and Giommoni 2021

- Can we predict political corruption?
- Focus on Brazilian Municipality
  - In Brazilian municipalities, we have information on fiscal corruption from random audits.
  - We train a machine learning algorithm to detect corruption in held-out data using budget data.

Example

Predicting corruption - Ash, Galletta and Giommoni 2021

Brollo, Nannicini, Perotti, and Tabellini (2013) provide corruption audit data from 1481 Brazilian municipalities

- We use the measure of **serious corruption**
- About half (47%) of audits reveal serious corruption.

The annual municipality budget is available from the ministry of finance online database:

- We collected/cleaned data for 2001 through 2012 and made them comparable across years.
- In total, we have 797 variables (Revenue 250, Expenditure 334, Assets 100, Liabilities 79)

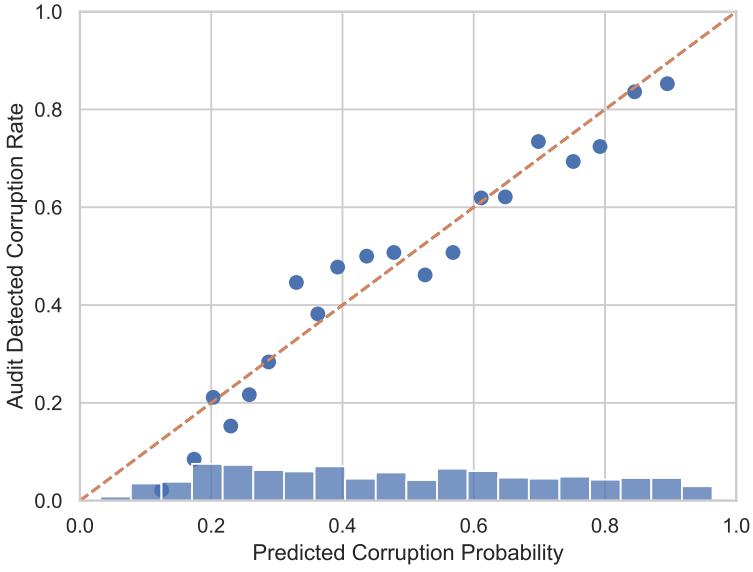
Example

Predicting corruption - Ash, Galletta and Giommoni 2021

1. We randomly split the sample of audited municipalities into five different sets
2. We train five separate models using each time four different subsets (80% of the sample) and take the tuned models to get performance metrics in the test set (the remaining 20 % of the sample, which also rotates).
3. Each time we tuned hyperparameters in the training set using five-fold cross-validation (e.g., max depth of trees and learning rate)
   - Use early stopping to avoid over-fitting.
4. Take the tuned model and get performance metrics in the test set

Example

Predicting corruption - Ash, Galletta and Giommoni 2021



Example

Predicting corruption - Ash, Galletta and Giommoni 2021

Table 1: Out-of-Sample Metrics for Predicting Corruption

		Optimal (5-fold NCV)				Bootstrap	
	Guessing (1)	OLS (2)	Lasso (3)	Logistic (4)	XGBoost (5)	XGBoost (6)	XGBoost (7)
Accuracy	0.580	0.502 [0.413-0.566]	0.500 [0.481-0.527]	0.575 [0.465-0.619]	0.720 [0.699-0.739]	0.700 (0.013)	0.683 (0.011)
AUC-ROC		0.519 [0.473-0.569]	0.486 [0.429-0.535]	0.559 [0.475-0.600]	0.781 [0.759-0.811]	0.755 (0.014)	0.736 (0.011)
F1	0.000	0.507 [0.300-0.582]	0.452 [0.252-0.527]	0.477 [0.306-0.550]	0.627 [0.578-0.657]	0.612 (0.018)	0.592 (0.015)

Example

Predicting corruption - Ash, Galletta and Giommoni 2021

Table 2: Most important budget features for Corruption Prediction

				Perturbation Response		
N.	Category	Macro Category	Weight	Mean	Min	Max
1	Spending in agriculture	Expenditure	114	0.010	-0.25	0.58
2	Tax on agricultural territorial property (ITR) (compartecipation)	Revenue	96	0.022	-0.24	0.45
3	Tax on export of industrialized products (IPI) (compartecipation)	Revenue	93	0.023	-0.42	0.53
4	Spending in transportation	Expenditure	92	0.008	-0.22	0.43
5	Taxes	Revenue	82	0.011	-0.41	0.65
6	Motor vehicle property tax (IPVA) (compartecipation)	Revenue	80	0.001	-0.34	0.45
7	Tax on real estate transactions (ITB)	Revenue	76	0.026	-0.20	0.50
8	Cash	Assets	75	0.010	-0.23	0.43
9	Income Tax (IRRF)	Revenue	73	0.003	-0.21	0.26
10	Tax on real estate (IPTU)	Revenue	73	0.024	-0.26	0.41