STA141C: Big Data & High Performance Statistical Computing
Lecture 10: Other Classification and Regression Algorithms

Cho-Jui Hsieh
UC Davis

May 18, 2017
What is Machine Learning?

Training Data
(documents, images, …)

Test Data
(documents, images, …)

- Train and test data are usually assumed to be iid samples from the same distribution
Training

Training Data
(documents, images, ...)

Learning Algorithm

Model (decision function)
(hyperplane, decision tree, ...)

Test Data
(documents, images, ...)

- Linear SVM/regression: Linear hyperplane
- Kernel SVM/regression: Nonlinear hyperplane
- Decision tree, random forest
- Nearest Neighbor
- ...
Learn a model that best explains the observed data as well as generalizes to unseen data.
Example: Ridge Regression

- **Training phase:**
  - Given a set of training data \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\)
  - Learn the model by solving
    \[
    w^* = \arg \min_w \|Xw - y\|^2 + \lambda \|w\|^2
    \]

- **Testing phase:**
  - Testing data: \(x_1, \ldots, x_m\) (different from training data)
  - Prediction: \(\hat{y}_1 = (w^*)^T x_1, \ldots, \hat{y}_m = (w^*)^T x_m\)
  - Evaluation: Mean Square Error
    \[
    \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2
    \]
Example: How to choose $\lambda$?

How to estimate the “prediction accuracy” of $\lambda$ without seeing testing data?

- **Method 1: Cross validation**
  - Split training data into $k$-fold:
    $$S_1, S_2, \ldots, S_k$$
  - For each $i = 1, \cdots, k$,
    - Obtain model $w_i$ using $S_1, \cdots, S_{i-1}, S_{i+1}, \cdots, S_k$ as training (excluding $S_i$)
    - Evaluate $w_i$ using $S_i$ (get $\text{MSE}_i$)
  - The MSE of the model using a specific $\lambda$ is estimated by
    $$\frac{1}{k} \sum_{i=1}^{k} \text{MSE}_i$$
Example: How to choose $\lambda$?

How to estimate the “prediction accuracy” of $\lambda$ without seeing testing data?

- Method 2: Using a validation set
  - Randomly split training data into $S_{\text{train}}$ and $S_{\text{validation}}$
    - (e.g., 80% training and 20% validation)
  - Obtain model $w$ using $S_{\text{train}}$
  - Evaluate $w$ using $S_{\text{validation}}$
Binary vs Multiclass classification
Given training data $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$

- **Regression**: $y_i \in \mathbb{R}$
- **Binary classification**: $y_i \in \{+1, -1\}$
- **Multiclass classification**: $y_i \in \{1, 2, \ldots, L\}$ (totally $L$ labels)
  - Each sample belongs to one class
- **Multilabel classification**: $y_i \in \{0, 1\}^L$
  - Each sample can be long to multiple classes
Multiclass Learning

- $n$ data points, $L$ labels, $d$ features
- Input: training data $\{x_i, y_i\}_{i=1}^n$:
  - Each $x_i$ is a $d$ dimensional feature vector
  - Each $y_i \in \{1, \ldots, L\}$ is the corresponding label
  - Each training data belongs to one category
- Goal: find a function to predict the correct label

$$f(x) \approx y$$
Multi-label Problems

- $n$ data points, $L$ labels, $d$ features
- Input: training data $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^n$:
  - Each $\mathbf{x}_i$ is a $d$ dimensional feature vector
  - Each $\mathbf{y}_i \in \{0, 1\}^L$ is a label vector (or $Y_i \in \{1, 2, \ldots, L\}$)
    - Example: $\mathbf{y}_i = [0, 0, 1, 0, 0, 1, 1]$ (or $Y_i = \{3, 6, 7\}$)
  - Each training data can belong to multiple categories
- Goal: Given a testing sample $\mathbf{x}$, predict the correct labels

<table>
<thead>
<tr>
<th>Document 1</th>
<th>{Sports, Politics}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document 2</td>
<td>{Science, Politics}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Document n</td>
<td>{Environment}</td>
</tr>
</tbody>
</table>
Multiclass: each row of $L$ has exact one “1”
Multilabel: each row of $L$ can have multiple ones
Measure the accuracy (multi-class)

- Let \( \{x_i, y_i\}_{i=1}^{m} \) be a set of testing data for multi-class problems.
- Let \( z_1, \ldots, z_m \) be the prediction for each testing data.
- Accuracy: \( (I(\cdot) \) is the indicator function)

\[
\frac{1}{m} \sum_{i=1}^{m} I(y_i = z_i)
\]
Measure the accuracy (multi-class)

- Let \( \{x_i, y_i\}_{i=1}^m \) be a set of testing data for multi-class problems
- Let \( z_1, \ldots, z_m \) be the prediction for each testing data
- Accuracy: \( (I(\cdot) \) is the indicator function) 
  \[
  \frac{1}{m} \sum_{i=1}^{m} I(y_i = z_i)
  \]

- If the algorithm outputs a set of \( k \) potential labels for each sample:
  \[ Z_1, Z_2, \ldots, Z_m \]
  Each \( Z_i \) is a set of \( k \) labels
- Precision@\( k \):
  \[
  \frac{1}{m} \sum_{i=1}^{m} I(y_i \in Z_i)
  \]
Traditional Approach

- Some algorithms can be naturally extended to multi-class classification
  - K-NN, Decision trees, ...
Traditional Approach

- Some algorithms can be naturally extended to multi-class classification
  K-NN, Decision trees, . . .
- For algorithms that only work for binary classification
  transform multi-class or multi-label problems to multiple binary classification problems
- Two approaches:
  - One versus All (OVA)
  - One versus One (OVO)
One Versus All (OVA)

- Multi-class/multi-label problems with $L$ categories
- Build $L$ different binary classifiers
- For the $t$-th classifier:
  - Positive samples: all the points in class $t$ ($\{x_i : t \in y_i\}$)
  - Negative samples: all the points not in class $t$ ($\{x_i : t \notin y_i\}$)
  - $f_t(x)$: the decision value for the $t$-th classifier
    (larger $f_t \Rightarrow$ higher probability that $x$ in class $t$)
- Prediction:
  $$f(x) = \arg \max_t f_t(x)$$
- Example: using SVM to train each binary classifier.
One Versus One (OVO)

- Multi-class/multi-label problems with $L$ categories
- Build $L(L - 1)$ different binary classifiers
- For the $(s, t)$-th classifier:
  - Positive samples: all the points in class $s$ ($\{x_i : s \in y_i\}$)
  - Negative samples: all the points in class $t$ ($\{x_i : t \in y_i\}$)
  - $f_{s,t}(x)$: the decision value for this classifier
    - (larger $f_{s,t}(x)$ $\Rightarrow$ label $s$ has higher probability than label $t$)
  - $f_{t,s}(x) = -f_{s,t}(x)$
- Prediction:
  \[ f(x) = \arg \max_s \left( \sum_t f_{s,t}(x) \right) \]
- Example: using SVM to train each binary classifier.
Linear Model vs Nonlinear Model
Linear Model vs Nonlinear Model

- Linear models:
  - Prediction by \( \hat{y} = w^T x_i + b \)
  - Usually can be learned efficiently
  - Usually not over-fitting:
    - training error is closer to testing error
  - Examples: Linear logistic regression, linear SVM, ridge regression, ⋯
Linear Model vs Nonlinear Model

- **Linear models:**
  - Prediction by $\hat{y} = \mathbf{w}^T \mathbf{x}_i + b$
  - Usually can be learned efficiently
  - Usually not over-fitting:
    - training error is closer to testing error
  - Examples: Linear logistic regression, linear SVM, ridge regression, ···

- **Nonlinear models:**
  - Prediction by a nonlinear function $\hat{y} = f(\mathbf{x}_i)$ for some nonlinear $f(\cdot)$
  - Much slower than linear models
  - Sometimes can achieve better accuracy
  - But easy to over-fit (training error $\ll$ testing error)
Nonlinear Models: Nearest Neighbor

- Training: No need for training
- Prediction:
  - Given the testing point $\bar{x}$, find the $k$ nearest points in the training data.
  - Predict the class by majority voting
- Weakness: long prediction time.
  Can be sped up by KDTree/BallTree.
Nonlinear Models: Nearest Neighbor

$k$ (number of neighbors) is an important tuning parameter.
Nonlinear Models: Nearest Neighbor

```python
>>> from sklearn.neighbors import NearestNeighbors
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> nbrs = NearestNeighbors(n_neighbors=2, algorithm='ball_tree').fit(X)
>>> distances, indices = nbrs.kneighbors(X)
>>> indices
array([[0, 1],
       [1, 0],
       [2, 1], ...])
>>> distances
array([[ 0. , 1. ],
       [ 0. , 1. ],
       [ 0. , 1.41421356], ...])
```
Tree-based Algorithms
Tree-based Algorithms

- Decision Tree
- Random Forest
- Gradient Boosted Decision Tree

(All of them can be used for classification and regression)
Decision Tree

Predicting the survival of passengers on the Titanic.

Predict the majority class for each leaf node.
Decision trees are among the most widely used non-linear methods.

Play tennis or not

- **Outlook**
  - Sunny
  - Rain
    - Overcast
      - Yes
      - No
    - Rain
      - Yes
      - No
- **Humidity**
  - High
  - Normal
    - Yes
    - No
- **Wind**
  - Strong
    - Yes
    - No
  - Weak
    - Yes

- Fast prediction: $O(p)$ (the average depth of the tree, usually less than 10)
- Small model size $O(2^p)$ (each node only needs two variables).
Splitting the node

- ID3, CART, ...

  Split the node to maximize the entropy

- Let $S$ be the set of data points in a node and $c = 1, \ldots, C$ are the labels:

  \[
  \text{Entropy} : H(S) = - \sum_{c=1}^{C} p(c) \log p(c),
  \]

  where $p(c)$ is the proportion of the data belonging to class $c$.

  - Entropy=0 if all samples are in the same class
  - Entropy is large if $p(1) = \cdots = p(C)$

- The “information gain” of the split $S = S_1 \cup \cdots \cup S_T$:

  \[
  H(S) - \sum_{t} \frac{|S_t|}{|S|} H(S_t)
  \]
Regression Tree

Predict average outcome for each leaf node.

\[ X_1 > 2.5 \]

- \[ X_5 > -4 \]
  - \[ y_1=1, y_4=3 \] (predict 2)
  - \[ y_2=1, y_5=2 \] (predict 1.5)

- \[ X_7 > 1.4 \]
  - \[ y_3=4 \] (predict 1.5)
  - \[ y_6=5, y_7=7, y_8=8 \] (predict 6.66)
Regression Tree

- Commonly used in Gradient Boosted Decision Tree (will see later)
- The quality of partition $S = S_1 \cup S_2$ can be computed by the objective function:

$$
\sum_{i \in S_1} (y_i - y^{(1)})^2 + \sum_{i \in S_2} (y_i - y^{(2)})^2,
$$

where $y^{(1)} = \frac{1}{|S_1|} \sum_{i \in S_1} y_i$, $y^{(2)} = \frac{1}{|S_2|} \sum_{i \in S_2} y_i$.
Splitting the node

- Test all the features \(\{1, \cdots, d\}\) and all the potential cutting values, and find the \((\text{feature}, \text{value})\) pair that maximize information gain.
- This is the computational bottleneck in training decision trees.
- Various approaches:
  - Linear scan (XGBoost)
  - Histogram-based splitting (XGBoost, LightGBM)
Parameters

- Maximum depth: (usually $\sim 10$)
- Minimum number of nodes in each node: (10, 50, 100)
- Regularization parameter (avoid over-fitting)
Random Forest
Random Forest

- Random Forest (Bootstrap ensemble for decision trees):
  - Create $T$ trees
  - Learn each tree using a subsampled dataset $S_i$ and subsampled feature set $D_i$
  - Prediction: Average the results from all the $T$ trees

- Benefit:
  - Avoid over-fitting
  - Improve stability and accuracy

- Good software available:
  - R: “randomForest” package
  - Python: Scikit Learn
Random Forest

Embarrassingly parallel
Gradient Boosted Decision Tree
Boosted Decision Tree

- Goal: minimizing a loss function $\ell(y, F(x))$ using boosting method.
- Gradient boosting considers estimating $F$ in an additive form:

$$F^* = \arg\min_F \sum_{i=1}^{n} \ell(y_i, F(x_i)) \quad \text{with} \quad F(x) = \sum_{m=1}^{T} f_m(x)$$
Boosted Decision Tree

- Goal: minimizing a loss function $\ell(y, F(x))$ using boosting method.
- Gradient boosting considers estimating $F$ in an additive form:

$$F^* = \arg\min_F \sum_{i=1}^{n} \ell(y_i, F(x_i)) \quad \text{with} \quad F(x) = \sum_{m=1}^{T} f_m(x)$$

- Direct loss minimization: at each stage $m$, find the best function to minimize objective function:
  - solve $\theta_m = \arg\min_{\theta} \sum_{i=1}^{N} \ell(y_i, F_{m-1}(x_i) + f_m(x_i, \theta))$
  - update $F_m(x) \leftarrow F_{m-1}(x) + f_m(x, \theta_m)$
- $F_m(x) = \sum_{j=1}^{m} f_j(x, \theta_j)$ is the prediction of $x$ after $m$ iterations.

Two problems:
- Hard to implement for general loss
- Tend to overfit training data
Boosted Decision Tree

- Goal: minimizing a loss function $\ell(y, F(x))$ using boosting method.
- Gradient boosting considers estimating $F$ in an additive form:

$$F^* = \arg\min_F \sum_{i=1}^{n} \ell(y_i, F(x_i)) \quad \text{with} \quad F(x) = \sum_{m=1}^{T} f_m(x)$$

- Direct loss minimization: at each stage $m$, find the best function to minimize objective function:
  - solve $\theta_m = \arg\min_{\theta} \sum_{i=1}^{N} \ell(y_i, F_{m-1}(x_i) + f_m(x_i, \theta))$
  - update $F_m(x) \leftarrow F_{m-1}(x) + f_m(x, \theta_m)$

- $F_m(x) = \sum_{j=1}^{m} f_j(x, \theta_j)$ is the prediction of $x$ after $m$ iterations.
- Two problems:
  - Hard to implement for general loss
  - Tend to overfit training data
Approximate the current loss function by a quadratic approximation:

\[
\sum_{i=1}^{n} \ell_i(\hat{y}_i + f_m(x_i)) \approx \sum_{i=1}^{n} \left( \ell_i(\hat{y}_i) + g_i f_m(x_i) + \frac{1}{2} h_i f_m(x_i)^2 \right)
\]

\[
= \sum_{i=1}^{n} \frac{h_i}{2} \| f_m(x_i) - g_i/h_i \|^2 + \text{constant}
\]

where \( g_i = \partial_{\hat{y}_i} \ell_i(\hat{y}_i) \) is gradient,

\( h_i = \partial^2_{\hat{y}_i} \ell_i(\hat{y}_i) \) is second order derivative
Gradient Boosted Decision Tree

- Finding $f_m(x, \theta_m)$ by minimizing the loss function:

$$
\arg\min_{f_m} \sum_{i=1}^{N} \left[ f_m(x_i, \theta) - g_i/h_i \right]^2 + R(f_m)
$$

- Reduce the training of any loss function to regression tree (just need to compute $g_i$ for different functions)
- $h_i = \alpha$ (fixed step size) for original GBDT.
- XGboost shows computing second order derivative yields better performance
Gradient Boosted Decision Tree

- Finding $f_m(x, \theta_m)$ by minimizing the loss function:

$$\argmin_{f_m} \sum_{i=1}^{N} \left[ f_m(x_i, \theta) - g_i/h_i \right]^2 + R(f_m)$$

- Reduce the training of any loss function to regression tree (just need to compute $g_i$ for different functions)
- $h_i = \alpha$ (fixed step size) for original GBDT.
- XGboost shows computing second order derivative yields better performance

**Algorithm:**
- Computing the current gradient for each $\hat{y}_i$.
- Building a base learner (decision tree) to fit the gradient.
- Updating current prediction $\hat{y}_i = F_m(x_i)$ for all $i$. 
Gradient Boosted Decision Trees (GBDT)

Key idea:
- Each base learner is a decision tree
- Each regression tree approximates the functional gradient $\frac{\partial \ell}{\partial F}$
Gradient Boosted Decision Trees (GBDT)

Key idea:
- Each base learner is a decision tree
- Each regression tree approximates the functional gradient \( \frac{\partial \ell}{\partial F} \)

\[
F_{m-1}(x_i) = \sum_{j=1}^{m-1} f_j(x_i) \\
g_m(x_i) = \frac{\partial \ell(y_i, F(x_i))}{\partial F(x_i)} \bigg|_{F(x_i)=F_{m-1}(x_i)}
\]
Gradient Boosted Decision Trees (GBDT)

Key idea:
- Each base learner is a decision tree
- Each regression tree approximates the functional gradient \( \frac{\partial \ell}{\partial F} \)

\[
F_{m-1}(x_i) = \sum_{j=1}^{m-1} f_j(x_i) \quad \quad g_m(x_i) = \left. \frac{\partial \ell(y_i, F(x_i))}{\partial F(x_i)} \right|_{F(x_i)=F_{m-1}(x_i)}
\]
Gradient Boosted Decision Trees (GBDT)

Key idea:
- Each base learner is a decision tree
- Each regression tree approximates the functional gradient $\frac{\partial \ell}{\partial F}$

\[
F_{m-1}(x_i) = \sum_{j=1}^{m-1} f_j(x_i) \quad g_m(x_i) = \frac{\partial \ell(y_i, F(x_i))}{\partial F(x_i)} \bigg|_{F(x_i) = F_{m-1}(x_i)}
\]
Gradient Boosted Decision Trees (GBDT)

- **Key idea:**
  - Each base learner is a decision tree
  - Each regression tree approximates the functional gradient $\frac{\partial \ell}{\partial f}$

$$(x, g_1) \quad \text{update} \quad F(x_i) \quad f_1(x)$$

$$(x, g_2) \quad \text{update} \quad F(x_i) \quad f_2(x)$$

$$(x, g_T) \quad \text{update} \quad F(x_i) \quad f_T(x)$$

**Final prediction**

$$F(x_i) = \sum_{j=1}^{T} f_j(x_i)$$
GBDT Software

- **XGBoost**: [https://github.com/dmlc/xgboost](https://github.com/dmlc/xgboost)
  - Support regression, classification, ranking, ...
  - Used by the winner of many Kaggle competitions.

- **LightGBM**: [https://github.com/Microsoft/LightGBM](https://github.com/Microsoft/LightGBM)
  - Released recently.
  - Use histogram-based splitting (they claimed faster than XGBoost)
  - A good GPU implementation (developed by our TA)
Coming up

- Clustering

Questions?