Ensemble of Learners

PROF XIAOHUI XIE
SPRING 2019

CS 273P Machine Learning and Data Mining
Ensemble methods

• Why learn one classifier when you can learn many?

• Ensemble: combine many predictors
  – (Weighted) combinations of predictors
  – May be same type of learner or different

“Who wants to be a millionaire?”

Various options for getting help:
Simple ensembles

- “Committees”
  - Unweighted average / majority vote

- Weighted averages
  - Up-weight “better” predictors
  - Ex: Classes: +1, -1, weights alpha:
    \[ \hat{y}_1 = f_1(x_1, x_2, \ldots) \]
    \[ \hat{y}_2 = f_2(x_1, x_2, \ldots) \]
    \[ \Rightarrow \hat{y}_e = \text{sign} \left( \sum \alpha_i \hat{y}_i \right) \]
    ...

“Stacked” ensembles

• Train a “predictor of predictors”
  • Treat individual predictors as features

\[
\hat{y}_1 = f_1(x_1, x_2, \ldots)
\]
\[
\hat{y}_2 = f_2(x_1, x_2, \ldots) \quad \Rightarrow \quad \hat{y}_e = f_e(\hat{y}_1, \hat{y}_2, \ldots)
\]

... 

• Similar to multi-layer perceptron idea

• Special case: binary, \( f_e \) linear \( \Rightarrow \) weighted vote

• Can train stacked learner \( f_e \) on validation data
  • Avoids giving high weight to overfit models
Mixtures of experts

- Can make weights depend on $x$
  - Weight $\alpha_z(x)$ indicates “expertise”
  - Combine using weighted average (or even just pick largest)

Example

Weighted average:

$$f(x; \omega, \theta) = \sum_z \alpha_z(x; \omega) \cdot f_z(x; \theta_z)$$

Weights: (multi) logistic regression

$$\alpha_z(x; \omega) = \frac{\exp(x \cdot \omega^z)}{\sum_c \exp(x \cdot \omega^c)}$$

If loss, learners, weights are all differentiable, can train jointly…

Mixture of three linear predictor experts
Machine Learning

Ensembles: Bagging

Ensembles: Gradient Boosting

Ensembles: Ada Boost
Ensemble methods

- Why learn one classifier when you can learn many?
  - “Committee”: learn K classifiers, average their predictions

- “Bagging” = bootstrap aggregation
  - Learn many classifiers, each with only part of the data
  - Combine through model averaging

- Remember overfitting: “memorize” the data
  - Used test data to see if we had gone too far
  - Cross-validation
    - Make many splits of the data for train & test
    - Each of these defines a classifier
    - Typically, we use these to check for overfitting
    - Could we instead combine them to produce a better classifier?
Bagging

• Bootstrap
  – Create a random subset of data by sampling
  – Draw \( m' \) of the \( m \) samples, with replacement
    • Some data left out; some data repeated several times

• Bagging
  – Repeat \( K \) times
    • Create a training set of \( m' \leq m \) examples
    • Train a classifier on the random training set
  – To test, run each trained classifier
    • Each classifier votes on the output, take majority
    • For regression: each regressor predicts, take average

• Notes:
  – Some complexity control: harder for each to memorize data
  – Doesn’t work for linear models (average of linear functions is linear function), but perceptrons OK (linear + threshold = nonlinear)
Data we observe

We only see a little bit of data

Can decompose error into two parts

- **Bias** – error due to model choice
  - Can our model represent the true best predictor?
  - Gets better with more complexity

- **Variance** – randomness due to data size
  - Better w/ more data, worse w/ complexity

\[
y(x) = \theta_0 + \theta_1 x + \nu
\]

\[
\hat{y}(x) = \hat{\theta}_0 + \hat{\theta}_1 x
\]

“The world”

Predictive Error

Error on test data

(High bias)

(High variance)
Bagged decision trees

- Randomly resample data
- Learn a decision tree for each
  - No max depth = very flexible class of functions
  - Learner is low bias, but high variance

Sampling:
simulates “equally likely”
data sets we could have
observed instead, &
their classifiers
Bagged decision trees

- Average over collection
  - Classification: majority vote

- Reduces memorization effect
  - Not every predictor sees each data point
  - Lowers effective “complexity” of the overall average
  - Usually, better generalization performance
  - Intuition: reduces variance while keeping bias low

Avg of 5 trees

Avg of 25 trees

Avg of 100 trees

Full data set
% Train on data set X, Y
[N,D] = size(X);
Classifiers = cell(1,Nbag); % Allocate space
for i=1:Nbag
    ind = ceil( N*rand(Nuse, 1) );  % Bootstrap sample data
    Xi = X(ind, :); Yi = Y(ind, :);  % Select those indices
    Classifiers{i} = Train_Classifier(Xi, Yi);  % Train
end;

# test on data Xtest
[Ntest,D] = size(Xtest);
predict = zeros(Ntest,Nbag); % Allocate space
for i=1:Nbag,
    % Apply each classifier
    predict(:,i)=Apply_Classifier( Xtest, Classifiers{i});
end;
predict = (mean(predict,2) > 1.5);  % Vote on output (if classes 1 vs 2)
Bagging in Python

```python
# Load data set X, Y for training the ensemble...
m,n = X.shape
classifiers = [ None ] * nBag  # Allocate space for learners
for i in range(nBag):
    ind = np.floor( m * np.random.rand(nUse) ).astype(int)  # Bootstrap sample a data set:
    Xi, Yi = X[ind,:], Y[ind]  # select the data at those indices
classifiers[i] = ml.MyClassifier(Xi, Yi)  # Train a model on data Xi, Yi

# test on data Xtest
mTest = Xtest.shape[0]
predict = np.zeros( (mTest, nBag) )  # Allocate space for predictions from each model
for i in range(nBag):
    predict[:,i] = classifiers[i].predict(Xtest)  # Apply each classifier

# Make overall prediction by majority vote
predict = np.mean(predict, axis=1) > 0  # if +1 vs -1
```
Random forests

• Bagging applied to decision trees

• Problem
  – With lots of data, we usually learn the same classifier
  – Averaging over these doesn’t help!

• Introduce extra variation in learner
  – At each step of training, only allow a (random) subset of features
  – Enforces diversity (“best” feature not available)
  – Keeps bias low (every feature available eventually)
  – Average over these learners (majority vote)

# in FindBestSplit(X,Y):
   for each of a subset of features
      for each possible split
         Score the split (e.g. information gain)
         Pick the feature & split with the best score
      Recurse on left & right splits
Microsoft Kinect Pose Estimation

Shotton et al., PAMI 2012
Summary

• Ensembles: collections of predictors
  – Combine predictions to improve performance

• Bagging
  – “Bootstrap aggregation”
  – Reduces complexity of a model class prone to overfit
  – In practice
    • Resample the data many times
    • For each, generate a predictor on that resampling
  – Plays on bias / variance trade off
  – Price: more computation per prediction
Machine Learning

Ensembles: Bagging

Ensembles: Gradient Boosting

Ensembles: Ada Boost
Ensembles

- Weighted combinations of predictors
- “Committee” decisions
  - Trivial example
  - Equal weights (majority vote / unweighted average)
  - Might want to weight unevenly – up-weight better predictors

- Boosting
  - Focus new learners on examples that others get wrong
  - Train learners sequentially
  - Errors of early predictions indicate the “hard” examples
  - Focus later predictions on getting these examples right
  - Combine the whole set in the end
  - Convert many “weak” learners into a complex predictor
Gradient boosting

- Learn a regression predictor
- Compute the error residual
- Learn to predict the residual

Learn a simple predictor…
\[ f_1(x^{(i)}) \approx y^{(i)} \]

Then try to correct its errors
\[ \epsilon^{(i)} = y^{(i)} - f_1(x^{(i)}) \]

\[ f_2(x^{(i)}) \approx \epsilon^{(i)} \]
Gradient boosting

- Learn a regression predictor
- Compute the error residual
- Learn to predict the residual

\[ f_1(x^{(i)}) \approx y^{(i)} \]
\[ \epsilon^{(i)} = y^{(i)} - f_1(x^{(i)}) \]
\[ f_2(x^{(i)}) \approx \epsilon^{(i)} \]

Combining gives a better predictor…
\[ f_1(x^{(i)}) + f_2(x^{(i)}) \approx y^{(i)} \]

Can try to correct its errors also, & repeat
\[ \epsilon_2^{(i)} = y^{(i)} - f_1(x^{(i)}) - f_2(x^{(i)}) \]
Gradient boosting

- Learn sequence of predictors
- Sum of predictions is increasingly accurate
- Predictive function is increasingly complex

\[ y^{(i)} \approx \sum_{z} f_{z}(x^{(z)}) \]
Gradient boosting

- Make a set of predictions $\hat{y}[i]$

- The “error” in our predictions is $J(y, \hat{y})$
  - For MSE: $J(.) = \sum (y[i] - \hat{y}[i])^2$

- We can “adjust” $\hat{y}$ to try to reduce the error
  - $\hat{y}[i] = \hat{y}[i] + \alpha f[i]$
  - $f[i] \triangleq rJ(y, \hat{y}) = (y[i]-\hat{y}[i])$ for MSE

- Each learner is estimating the gradient of the loss function
- Gradient descent: take sequence of steps to reduce $J$
  - Sum of predictors, weighted by step size $\alpha$
Gradient boosting in Matlab

```matlab
% Data set X, Y
mu = mean(Y); % Often start with constant "mean" predictor
dY = Y - mu; % subtract this prediction away
For k=1:Nboost,
    Learner{ } = Train_Regressor(X,dY);
    alpha(k) = 1; % alpha: a "learning rate" or "step size"
    % smaller alphas need to use more classifiers, but tend to
    % predict better given enough of them

    % compute the residual given our new prediction
    dY = dY - alpha(k) * predict(Learner{k}, X)
end;

% Test data Xtest
test_data = size(Xtest);
predict = zeros(Ntest,1) + mu; % Allocate space & add mean
For k=1:Nboost, % Predict with each learner
    predict = predict + alpha(k)*predict(Learner{k}, Xtest);
end;
```
Gradient boosting in Python

# Load data set X, Y …
learner = [None] * nBoost  # storage for ensemble of models
alpha = [1.0] * nBoost    # and weights of each learner
mu = Y.mean()             # often start with constant ”mean” predictor
dY = Y – mu                # subtract this prediction away
for k in range(nBoost):
    learner[k] = ml.MyRegressor( X, dY )  # regress to predict residual dY using X
    alpha[k] = 1.0                   # alpha: ”learning rate” or ”step size”
    # smaller alphas need to use more classifiers, but may predict better given enough of them
    # compute the residual given our new prediction:
    dY = dY – alpha[k] * learner[k].predict(X)

# test on data Xtest
mTest = Xtest.shape[0]
predict = np.zeros( (mTest,) ) + mu       # Allocate space for predictions & add 1st (mean)
for k in range(nBoost):
    predict += alpha[k] * learner[k].predict(Xtest)  # Apply predictor of next residual & accum
Summary

• Ensemble methods
  – Combine multiple classifiers to make “better” one
  – Committees, average predictions
  – Can use weighted combinations
  – Can use same or different classifiers

• Gradient Boosting
  – Use a simple regression model to start
  – Subsequent models predict the error residual of the previous predictions
  – Overall prediction given by a weighted sum of the collection
Machine Learning

- Ensembles: Bagging
- Ensembles: Gradient Boosting
- Ensembles: Ada Boost
Ensembles

• Weighted combinations of classifiers
• “Committee” decisions
  – Trivial example
  – Equal weights (majority vote)
  – Might want to weight unevenly – up-weight good experts

• Boosting
  – Focus new experts on examples that others get wrong
  – Train experts sequentially
  – Errors of early experts indicate the “hard” examples
  – Focus later classifiers on getting these examples right
  – Combine the whole set in the end
  – Convert many “weak” learners into a complex classifier
Boosting example

Original data set, $D_1$

Update weights, $D_2$

Update weights, $D_3$

Trained classifier

Trained classifier

Trained classifier

Classes $+1, -1$
Minimizing weighted error

- So far we’ve mostly minimized unweighted error
- Minimizing weighted error is no harder:

Unweighted average loss:

\[ J(\theta) = \frac{1}{m} \sum_i J_i(\theta, x^{(i)}) \]

Weighted average loss:

\[ J(\theta) = \sum_i w_i J_i(\theta, x^{(i)}) \]

For any loss (logistic MSE, hinge, …)

\[ J(\theta, x^{(i)}) = (\sigma(\theta x^{(i)}) - y^{(i)})^2 \]

\[ J(\theta, x^{(i)}) = \max[0, 1 - y^{(i)} \theta x^{(i)}] \]

To learn decision trees, find splits to optimize weighted impurity scores:
- \( p(+1) = \) total weight of data with class +1
- \( p(-1) = \) total weight of data with class -1  \( \Rightarrow H(p) = \) impurity
Boosting example

Weight each classifier and combine them:

\[ \begin{align*}
0.33 \times \text{Classifier 1} & \quad + \quad 0.57 \times \text{Classifier 2} & \quad + \quad 0.42 \times \text{Classifier 3} \\
\end{align*} \]

Combined classifier

1-node decision trees
“decision stumps”
very simple classifiers
AdaBoost = “adaptive boosting”

- Pseudocode for AdaBoost

```python
# Load data set X, Y … ; Y assumed +1 / -1
for i in range(nBoost):
    learner[i] = ml.MyClassifier( X, Y, weights=wts )  # train a weighted classifier
    Yhat = learner[i].predict(X)
    e = wts.dot( Y != Yhat )  # compute weighted error rate
    alpha[i] = 0.5 * np.log( (1-e)/e )
    wts *= np.exp( -alpha[i] * Y * Yhat )  # update weights
    wts /= wts.sum()  # and normalize them

# Final classifier:
predict = np.zeros( (mTest,) )
for i in range(nBoost):
    predict += alpha[i] * learner[i].predict(Xtest)  # compute contribution of each model
predict = np.sign(predict)  # and convert to +1 / -1 decision
```

- Notes
  - e > .5 means classifier is not better than random guessing
  - Y * Yhat > 0 if Y == Yhat, and weights decrease
  - Otherwise, they increase
AdaBoost theory

- Minimizing classification error was difficult
  - For logistic regression, we minimized MSE or NLL instead
  - Idea: low MSE $\Rightarrow$ low classification error
- Example of a surrogate loss function
- AdaBoost also corresponds to a surrogate loss function

$$
C_{ada} = \sum_{i} \exp[-y^{(i)} f(x^{i})]
$$

- Prediction is $\hat{y} = \text{sign}( f(x) )$
  - If same as $y$, loss < 1; if different, loss > 1; at boundary, loss=1
- This loss function is smooth & convex (easier to optimize)
AdaBoost example: Viola-Jones

- Viola-Jones face detection algorithm
- Combine lots of very weak classifiers
  - Decision stumps = threshold on a single feature
- Define lots and lots of features
- Use AdaBoost to find good features
  - And weights for combining as well
Haar wavelet features

• Four basic types.
  – They are easy to calculate.
  – The white areas are subtracted from the black ones.
  – A special representation of the sample called the **integral image** makes feature extraction faster.
Training a face detector

• Wavelets give ~100k features
• Each feature is one possible classifier
• To train: iterate from 1:T
  – Train a classifier on each feature using weights
  – Choose the best one, find errors and re-weight

• This can take a long time… (lots of classifiers)
  – One way to speed up is to not train very well…
  – Rely on adaboost to fix “even weaker” classifier

• Lots of other tricks in “real” Viola-Jones
  – Cascade of decisions instead of weighted combo
  – Apply at multiple image scales
  – Work to make computationally efficient
Summary

• **Ensemble methods**
  – Combine multiple classifiers to make “better” one
  – Committees, majority vote
  – Weighted combinations
  – Can use same or different classifiers

• **Boosting**
  – Train sequentially; later predictors focus on mistakes by earlier

• **Boosting for classification (e.g., AdaBoost)**
  – Use results of earlier classifiers to know what to work on
  – Weight “hard” examples so we focus on them more
  – Example: Viola-Jones for face detection