Introduction to Machine Learning
NPFL 054

http://ufal.mff.cuni.cz/course/npfl054

Barbora Hladká
hladka@ufal.mff.cuni.cz

Martin Holub
holub@ufal.mff.cuni.cz

Charles University in Prague,
Faculty of Mathematics and Physics,
Institute of Formal and Applied Linguistics

Ensemble learning methods

Outline

- Combining classifiers into ensembles – general scheme
- Generating random samples by bootstrapping
- Bagging vs. boosting
- Bagging – example classifier
- Random Forests
- AdaBoost

Ensemble classifiers – a motivation exercise

Consider the following task – we have a binary classification problem and a number of predictors, each with error less than 0.5. Will the resulting majority voting ensemble have an error lower than the single classifers?

Ensemble classifiers – a motivation exercise

Consider the following task – we have a binary classification problem and a number of predictors, each with error less than 0.5. Will the resulting majority voting ensemble have an error lower than the single classifers?

– Depends on the *accuracy* and the *diversity* of the base learners!

Ensemble classifiers – a motivation exercise

Consider the following task – we have a binary classification problem and a number of predictors, each with error less than 0.5. Will the resulting majority voting ensemble have an error lower than the single classifers?

– Depends on the *accuracy* and the *diversity* of the base learners!

Particular settings – assume that you have

- 21 classifiers
- each with error $p = 0.3$
- their outputs are *statistically independent*

Compute the error of the ensemble under these conditions!

General scheme of combining classifiers

Resampling approach

- Distribute the training data into K portions
- Run the learning process to get K different models
- Collect the output of the K models use a combining function to get a final output value

Bootstrapping principle

- New data sets $Data_1, \ldots, Data_K$ are drawn from $Data$ with replacement, each of the same size as the original $Data$, i.e. n.
- In the i-th step of the iteration, $Data_i$ is used as a training set, while the examples $\{\mathbf{x} \mid \mathbf{x} \in Data \land \mathbf{x} \notin Data_i\}$ form the test set.

Bootstrapping principle

- New data sets $Data_1, \ldots, Data_K$ are drawn from $Data$ with replacement, each of the same size as the original $Data$, i.e. n.

- In the i-th step of the iteration, $Data_i$ is used as a training set, while the examples $\{\mathbf{x} \mid \mathbf{x} \in Data \wedge \mathbf{x} \notin Data_i\}$ form the test set.

- The probability that we pick an instance is $1/n$, and the probability that we do not pick an instance is $1 - 1/n$. The probability that we do not pick it after n draws is $(1 - 1/n)^n \approx e^{-1} \doteq 0.368$.

- It means that for training the system will not use 36.8 % of the data, and the error estimate will be pessimistic. So the solution is to repeat the process many times.

Same algorithm, different classifiers

Combining classifiers to improve the performance

Bootstrapping methods – key ideas

- combining the classification results from different classifiers to produce the final output
- using (un)weighted voting
- different training data
- different features
- different values of the relevant paramaters
- performance: complementarity $\longrightarrow$ potential improvement

Two fundamental approaches

- **Bagging** works by taking a bootstrap sample from the training set
- **Boosting** works by changing the weights on the training set

Bagging and boosting — the difference

- **Bagging**: each predictor is trained independently
- **Boosting**: each predictor is built on the top of previous predictors trained
  – Like bagging, boosting is also a voting method. In contrast to bagging, boosting actively tries to generate complementary learners by training the next learner on the mistakes of the previous learners.

Combining multiple learners

- the more **complementary** the learners are, the more useful their combining is
- the simpliest way to combine multiple learners is **voting**
- in **weighted voting** the voters (= base-learners) can have different weights

Unstable learning

- learning algorithm is called unstable if small changes in the training set cause large differences in generated models
- typical unstable algorithm is the decision trees learning
- bagging or boosting techniques are a natural remedy for unstable algorithms

Bagging

- Bagging is a voting method that uses slightly different training sets (generated by bootstrap) to make different base-learners. Generating complementary base-learners is left to chance and to unstability of the learning method.
- Generally, bagging can be combined with any approach to learning.

Bagging – algorithm

Bootstrap **AGG**regat**ING**

1. for $i \leftarrow 1$ to K do
2. $Train_i \leftarrow \text{bootstrap}(Data)$
3. $h_i \leftarrow \text{TrainPredictor}(Train_i)$

Combining function

- **Classification:** $h_{final}(\mathbf{x}) = \text{MajorityVote}(\text{h}_1(\mathbf{x}), \text{h}_2(\mathbf{x}), \ldots, \text{h}_\text{K}(\mathbf{x}))$
- **Regression:** $h_{final}(\mathbf{x}) = \text{Mean}(\text{h}_1(\mathbf{x}), \text{h}_2(\mathbf{x}), \ldots, \text{h}_\text{K}(\mathbf{x}))$

Random Forests

- an ensemble method based on decision trees and bagging
- builds a number of random decision trees and then uses voting
- introduced by L. Breiman (2001), then developed by L. Breiman and A. Cutler
- very good (state-of-the-art) prediction performance
- a nice page with description
  `www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm`
- important: Random Forests helps to
  - avoid overfitting (by random sampling the training data set)
  - select important/useful features (by random sampling the feature set)

Building Random Forests

The algorithm for building a tree in the ensemble

1. Having a training set of the size n, sample n cases at random - but with replacement, and use the sample to build a decision tree.
2. If there are M input features, choose a less number $m \ll M$ (fixed for the whole procedure). When building the tree, at each node m variables are selected at random out of the M and the best split on these m features is used to split the node.
3. Each tree is grown to the largest extent possible. There is no pruning.

The more trees in the ensemble, the better.
There is no risk of overfitting!

Regularized Random Forests

- a recent extension of the original Random Forest
  – introduced by Houtao Deng and George Runger (2012)
- produces a compact feature subset
- provides an effective and efficient feature selection solution for many practical problems
- overcomes the weak spot of the ordinary RF: Random Forest importance score is biased toward the variables having more (categorical) values
- a useful page: `https://sites.google.com/site/houtaodeng/rrf`
  – a presentation
  – a sample code
  – links to papers
  – a brief explanation of the difference between RRF and guided RRF

R packages for Random Forests

- **randomForest**: Breiman and Cutler's random forests for classification and regression
  – Classification and regression based on a forest of trees using random inputs.
- **RRF**: Regularized Random Forest
  – Feature Selection with Regularized Random Forest. This package is based on the 'randomForest' package by Andy Liaw. The key difference is the RRF function that builds a regularized random forest.
  – `http://cran.r-project.org/web/packages/RRF/index.html`
- **party**: A Laboratory for Recursive Partytioning
  – a computational toolbox for recursive partitioning
  – `cforest()` provides an implementation of Breiman's random forests
  – extensible functionality for visualizing tree-structured regression models is available

Boosting

Motivation

- I want to write a program that will accurately predict the winner of a tennis tournament based on the information like number of tournaments recently won by each player.
- I have not much experience so I ask a highly successfull expert gambler to explain his betting strategy. In general, he is not able to explain a grand set of rules for predicting a winner. However, when he is provided with the data for a particular tournament, the expert has no problem to come up with a "rule of thumb" like *Bet on the player who has recently won the most matches*.
- Such a rule of thumb is obviously rough and inaccurate, we can expect to provide predictions that are at least a little bit better than random guessing.

Boosting

Motivation

- How to extract rules of thumb from expert that will be the most useful?
- How to combine moderately accurate rules of thumb into a single highly accurate prediction rule?

Basic idea

- Boosting is a method that produces a very accurate predictor by combininig rough and moderately accurate predictors.
- It is based on the observation that finding many rough predictors (rules of thumb) can be easier than finding a single, highly accurate predictor.

Boosting — Adaboost (Adaptive Boosting)

AdaBoost is a boosting method that repeatedly calls a given weak learner, each time with different distribution over the training data. Then we combine these weak learners into a strong learner.

- originally proposed by Freund and Schapire (1996)
- nice presentation including theoretical details and a demonstration available at `http://cmp.felk.cvut.cz/~sochmj1/adaboost_talk.pdf`

Boosting — Adaboost (Adaptive Boosting)

Key questions

- How to choose the distribution?
- How to combine the weak predictors into a single predictor?
- How many weak predictors should be trained?

Schapire's strategy: Change the distribution over the examples in each iteration, feed the resulting sample into the weak learner, and then combine the resulting hypotheses into a voting ensemble, which, in the end, would have a boosted prediction accuracy.

Binary classification and AdaBoost

- We explain the notion of boosting using binary classification with the training data

$$Data = \{\langle \mathbf{x}_i, y_i \rangle : \mathbf{x}_i \in X, y_i \in Y, Y = \{-1, +1\}, i = 1, \dots, n\}$$

.

- We need to define distribution $\mathcal{D}$ over *Data* such that $\sum_{i=1}^{n} \mathcal{D}_i = 1.$

- A weak classifier $h_t : X \to Y$ has the property

$$\mathrm{error}_{\mathcal{D}}(h_t) < 1/2.$$

AdaBoost

- Initialize $\mathcal{D}_1(i) = 1/n$
- At each step t
  - Learn h_t using $\mathcal{D}_t$: find the weak classifier h_t with the minimum weighted sample error $\mathrm{error}_{\mathcal{D}_t}(h_t) = \sum_{i=1}^{n} \mathcal{D}_t(i)\,\delta(h(\mathbf{x}_i) \neq y_i)$
  - Set weight α_t of h_t based on the sample error

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \mathrm{error}_{\mathcal{D}_t}(h_t)}{\mathrm{error}_{\mathcal{D}_t}(h_t)} \right)$$

  - Update the distribution (Z_t is a normalization factor)

$$\mathcal{D}_{t+1} = \frac{1}{Z_t} \mathcal{D}_t \, \mathrm{e}^{-\alpha_t y_i h_t(\mathbf{x}_i)}$$

  - Stop when impossible to find a weak classifier being better than chance
- Output the final classifier $h_{final}(\mathbf{x}) = \mathrm{sign} \sum_{i=1}^{T} \alpha_i h_i(\mathbf{x})$

AdaBoost

- constructing $\mathcal{D}_t$:
  - On each round, the weights of incorrectly classified instances are increased so that the algorithm is forced to focus on the hard training examples.
  - $\mathcal{D}_1(i) = 1/n$
  - given $\mathcal{D}_t$ and h_t (i.e. update $\mathcal{D}_t$):

$$\mathcal{D}_{t+1}(i) = \frac{\mathcal{D}_t(i)}{Z_t} \cdot \begin{cases} e^{-\alpha_t} & \text{if} \quad y_i = h_t(x_i) \\ e^{\alpha_t} & \text{if} \quad y_i \neq h_t(x_i) \end{cases} = \frac{\mathcal{D}_t(i)}{Z_t} \, \mathrm{e}^{-\alpha_t y_i h_t(x_i)},$$

    where Z_t is normalization constant $Z_t = \sum_i \mathcal{D}_t(i) \, \mathrm{e}^{-\alpha_t y_i h_t(x_i)}$
  - α_t measures the importance that is assigned to h_t

AdaBoost

Weights

- $error_{\mathcal{D}_t}(h_t) < \frac{1}{2} \Rightarrow \alpha_t > 0$
- the smaller the error, the bigger the weight of the weak learner
- The bigger the weight, the more impact on the strong classifier:
  $error_{\mathcal{D}_t}(h_1) < error_{\mathcal{D}_t}(h_2) \Rightarrow \alpha_1 > \alpha_2$
- $\mathcal{D}_{t+1} = \dfrac{1}{Z_t}\mathcal{D}_t\, \mathrm{e}^{-\alpha_t y_i h_t(\mathbf{x}_i)}$
  The weights of correctly classified instances are reduced while weights of misclassified instances are increased.

AdaBoost.M1 — multiclass problem

Multiclass problem

- Assume classification task where $Y = \{y_1, \ldots, y_k\}$

$$h_t : X \rightarrow Y,$$

$$\mathcal{D}_{t+1}(i) = \frac{\mathcal{D}_t(i)}{Z_t} \cdot \begin{cases} e^{-\alpha_t} & \text{if} \quad y_i = h_t(\mathbf{x_i}) \\ e^{\alpha_t} & \text{if} \quad y_i \neq h_t(\mathbf{x_i}) \end{cases}$$

$$h_{final}(\mathbf{x}) = argmax_{y \in Y} \sum_{\{y \mid h_t(\mathbf{x}) = y\}} \alpha_t.$$

We can prove same bound on the error **if** $\forall t : \epsilon_t \leq \frac{1}{2}$