Ensemble learning

AI+X
INSTITUTE FOR ARTIFICIAL INTELLIGENCE

Ensemble learning

- Combining multiple models
  - The basic idea
- Bagging
  - Bias-variance decomposition, bagging with costs
- Randomization
  - Random forests, rotation forests
- Boosting
  - AdaBoost, the power of boosting
- Additive regression
  - Numeric prediction, additive logistic regression
- Interpretable ensembles
  - Option trees, alternating decision trees, logistic model trees
- Stacking

Combining multiple models

- Basic idea:
  build different “experts”, let them vote
- Advantage:
  - often improves predictive performance
- Disadvantage:
  - usually produces output that is very hard to analyze
  - but: there are approaches that aim to produce a single comprehensible structure

Snapshot Learning

- Useful with deep learned models when you do not have enough data for a validation set
  - Also, if you have lots of data and want to use multiple models without taking the time to train each model (e.g. if they take a week to train)
- Take a deep learned model trained for N epochs. Choose to save the weights at every g epochs from start_epoch. This will enable you to have k=(N – start_epoch)/g deep neural network classifiers whose predictions can be voted
- Original paper shows Cyclic learning rate works best for snapshot learning. Like all else, depends on data

Snapshot Learning

- Why would this work? Assuming you have reasonable models for each of the k weight sets, they likely make different errors. A vote may get problem examples correct
  - Consider that the models may struggle to get the boundary correct for some examples (training) that will cause the same issue in testing
- How do you choose k and which epochs should their weights reflect?
  - If you have validation data, you can choose every g epochs starting on say $epoch_i$ based on a point where the validation loss is not wildly changing
  - No validation data, when train data loss has stabilized a little

Bagging

- Combining predictions by voting/averaging
  - Each model receives equal weight
- "Idealized" version:
  - Sample several training sets of size *n*
    (instead of just having one training set of size *n*)
  - Build a classifier for each training set
  - Combine the classifiers' predictions
- Learning scheme is *unstable* **->**
  almost always improves performance
  - Unstable learner: small change in training data can make big change in model (e.g., when learning decision trees)

Bias-variance decomposition

- The *bias-variance decomposition* is used to analyze how much restriction to a single training set affects performance
- Assume we have the idealized ensemble classifier discussed on the previous slide
- We can decompose the expected error of any individual ensemble member as follows:
  - *Bias* = expected error of the ensemble classifier on new data
  - *Variance* = component of the expected error due to the particular training set being used to built our classifier
  - Total expected error = bias + variance
- Note (A): we assume noise inherent in the data is part of the bias component as it cannot normally be measured
- Note (B): multiple versions of this decomposition exist for zero-one loss but the basic idea is always the same

More on bagging

- Idealized version of bagging improves performance because it eliminates or reduces the *variance* component of the error
  - Note: in some pathological hypothetical situations the overall error may increase when zero-one loss is used (i.e., there is negative “variance”)
    - The bias-variance decomposition was originally only known for numeric prediction with squared error where the error *never* increases
- Problem: we only have one dataset!
- Solution: generate new datasets of size *n* by sampling from the original dataset *with replacement*
- This is what *bagging* does and even though the datasets are all dependent, bagging often reduces variance, and, thus, error
  - Can be applied to numeric prediction and classification
  - Can help a lot if the data is noisy
  - Usually, the more classifiers the better, with diminishing returns

Bagging classifiers

Model generation

```
Let n be the number of instances in the training data
For each of t iterations:
Sample n instances from training set
   (with replacement)
Apply learning algorithm to the sample
Store resulting model
```

Classification

```
For each of the t models:
Predict class of instance using model
Return class that is predicted most often
```

Bagging with costs

- Bagging unpruned decision trees is known to produce good probability estimates
  - Where, instead of voting, the individual classifiers' probability estimates are averaged
  - Note: this can also improve the zero-one loss
- Can use this with the minimum-expected cost approach for learning problems with costs
  - Note that the minimum-expected cost approach requires accurate probabilities to work well
- Problem: ensemble classifier is not interpretable
  - *MetaCost* re-labels the training data using bagging with costs and then builds a single tree from this data

Randomization and random forests

- Can randomize learning algorithm instead of input
- Some algorithms already have a random component: e.g., initial weights in a neural net
- Most algorithms can be randomized, e.g., greedy algorithms:
  - Pick *N* options at random from the full set of options, then choose the best of those *N* choices
  - E.g.: attribute selection in decision trees
- More generally applicable than bagging: e.g., we can use random subsets (of attributes) in a nearest-neighbor classifier
  - Bagging does not work with stable classifiers such as nearest neighbor classifiers
- Can be combined with bagging
  - When using decision trees, this yields the heavily used *random forests* method for building ensemble classifiers

Random Forests

- Use bagging for each tree
- Randomly select k < total features/attributes
  - Select best feature for the random subset of k features
- No pruning
- Build 200-1000 trees. Vote their output to obtain a class prediction

Rotation forests: motivation

- Bagging creates ensembles of accurate classifiers with relatively low diversity
  - Bootstrap sampling creates training sets with a distribution that resembles the original data
- Randomness in the learning algorithm increases diversity but sacrifices accuracy of individual ensemble members
  - This is why random forests normally require hundreds or thousands of ensemble members to achieve their best performance
- So-called *rotation forests* have the goal of creating accurate **and** diverse ensemble members

Rotation forests

- Combine random attribute sets, bagging and principal components to generate an ensemble of decision trees
- An iteration of the algorithm for creating rotation forests, building *k* ensemble members, involves
  - Randomly dividing the input attributes into *k* disjoint subsets
  - Applying PCA to each of the *k* subsets in turn
  - Learning a decision tree from the *k* sets of PCA directions
- Further increases in diversity can be achieved by creating a bootstrap sample in each iteration before applying PCA
- Performance of this method compares favorably to that of random forests on many practical datasets

Boosting

- Bagging can easily be parallelized because ensemble members are created independently
- Boosting is an alternative approach
- Also uses voting/averaging
- But: weights models according to performance
- Iterative: new models are influenced by performance of previously built ones
  - Encourage new model to become an “expert” for instances misclassified by earlier models
  - Intuitive justification: models should be experts that complement each other
- Many variants of boosting exist, we cover a couple

Boosting using AdaBoost.M1

Model generation

```
Assign equal weight to each training instance
For t iterations:
  Apply learning algorithm to weighted dataset,
  store resulting model
  Compute model's error e on weighted dataset
  If e = 0 or e ≥ 0.5:
    Terminate model generation
  For each instance in dataset:
    If classified correctly by model:
      Multiply instance's weight by e/(1-e)
  Normalize weight of all instances
```

Classification

```
Assign weight = 0 to all classes
For each of the t (or less) models:
For the class this model predicts
  add -log e/(1-e) to this class's weight
Return class with highest weight
```

Comments on AdaBoost.M1

- Boosting needs weights … but
- can adapt learning algorithm ... or
- can apply boosting *without* weights:
  - Resample data with probability determined by weights
  - Disadvantage: not all instances are used
  - Advantage: if error > 0.5, can resample again
- The AdaBoost.M1 boosting algorithm stems from work in *computational learning theory*
- Theoretical result:
  - Training error decreases exponentially as iterations are performed
- Other theoretical results:
  - Works well if base classifiers are not too complex and
  - their error does not become too large too quickly as more iterations are performed

More comments on boosting

- Continue boosting after training error = 0?
- Puzzling fact: generalization error continues to decrease!
  - Seems to contradict Occam's Razor
- Possible explanation:
  consider *margin* (confidence), not just error
  - A possible definition of *margin*: difference between estimated probability for true class and nearest other class (between –1 and 1)
  - Margin continues to increase with more iterations
- AdaBoost.M1 works well with so-called *weak* learners; only condition: error does not exceed 0.5
  - Example of weak learner: decision stump
- In practice, boosting sometimes overfits if too many iterations are performed (in contrast to bagging)

Additive regression

- Using statistical terminology, boosting is a greedy algorithm for fitting an *additive model*
- More specifically, it implements *forward stagewise additive modeling*
- Forward stagewise additive modeling for numeric prediction:
  - Build standard regression model (e.g., regression tree)
  - Gather residuals, learn model predicting *residuals* (e.g. another regression tree), and repeat
- To predict, simply sum up individual predictions from all regression models

Comments on additive regression

- Additive regression greedily minimizes squared error of ensemble if base learner minimizes squared error
- Note that it does not make sense to use additive regression with standard multiple linear regression
  - Why? Sum of linear regression models is a linear regression model and linear regression already minimizes squared error
- But: can use forward stagewise additive modeling with *simple* linear regression to implement multiple linear regression
  - Idea: build simple (i.e., one-attribute) linear regression models in each iteration of additive regression, pick attribute that yields lowest error
  - Use cross-validation to decide when to stop performing iterations
  - Automatically performs attribute selection!
- A trick to combat overfitting in additive regression: shrink predictions of base models by multiplying with pos. constant < 1
  - Caveat: need to start additive regression with initial model that predicts the mean, in order to shrink towards the mean, not 0

Additive logistic regression

- Can apply additive regression in conjunction with the logit transformation to get an algorithm for classification
  - More precisely, an algorithm for class probability estimation
  - Probability estimation problem is transformed into a regression problem
  - Regression scheme is used as base learner (e.g., regression tree learner)
- Implemented using forward stagewise algorithm: at each stage, add base model that maximizes the probability of the data
- We consider two-class classification in the following
- If f_j is the jth regression model, and **a** is an instance, the ensemble predicts probability

$$p(1|\mathbf{a}) = \frac{1}{1 + e^{-\sum f_j(\mathbf{a})}}$$

for the first class (compare to logistic regression model)

LogitBoost

Model generation

```
For j = 1 to t iterations:
  For each instance a[i]:
    Set the target value for the regression to
      z[i] = (y[i] – p(1|a[i])) / [p(1|a[i]) × (1-p(1|a[i])]
    Set the weight of instance a[i] to p(1|a[i]) × (1-p(1|a[i])
  Fit a regression model f[j] to the data with class
    values z[i] and weights w[i]
```

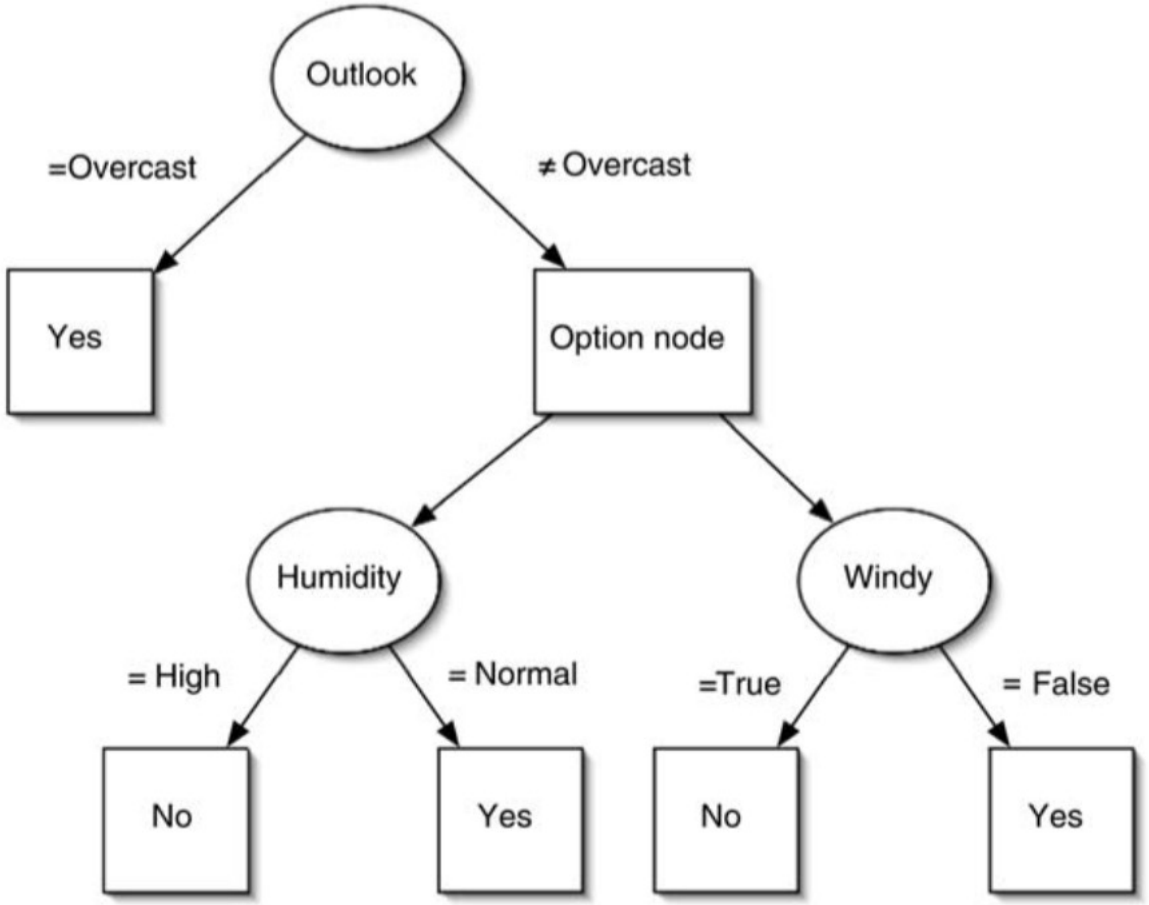
Classification

```
Predict 1st class if p(1 | a) > 0.5, otherwise predict 2nd class
```

- Greedily maximizes probability if base learner minimizes squared error
- Difference from AdaBoost.M1: optimizes probability/likelihood instead of a special loss function called *exponential loss*
- Can be extended to multi-class problems
- Overfitting avoidance: shrinking and cross-validation-based selection of the number of iterations apply

Option trees

- Ensembles are not easily interpretable
- Can we generate a single model?
  - One possibility: “cloning” the ensemble by using large amounts of artificial data that is labeled by the ensemble
  - Another possibility: generating a single structure that represents an ensemble in a compact fashion
- *Option tree*: decision tree with option nodes
  - Idea: follow all possible branches at option node
  - Predictions from different branches are merged using voting or by averaging probability estimates

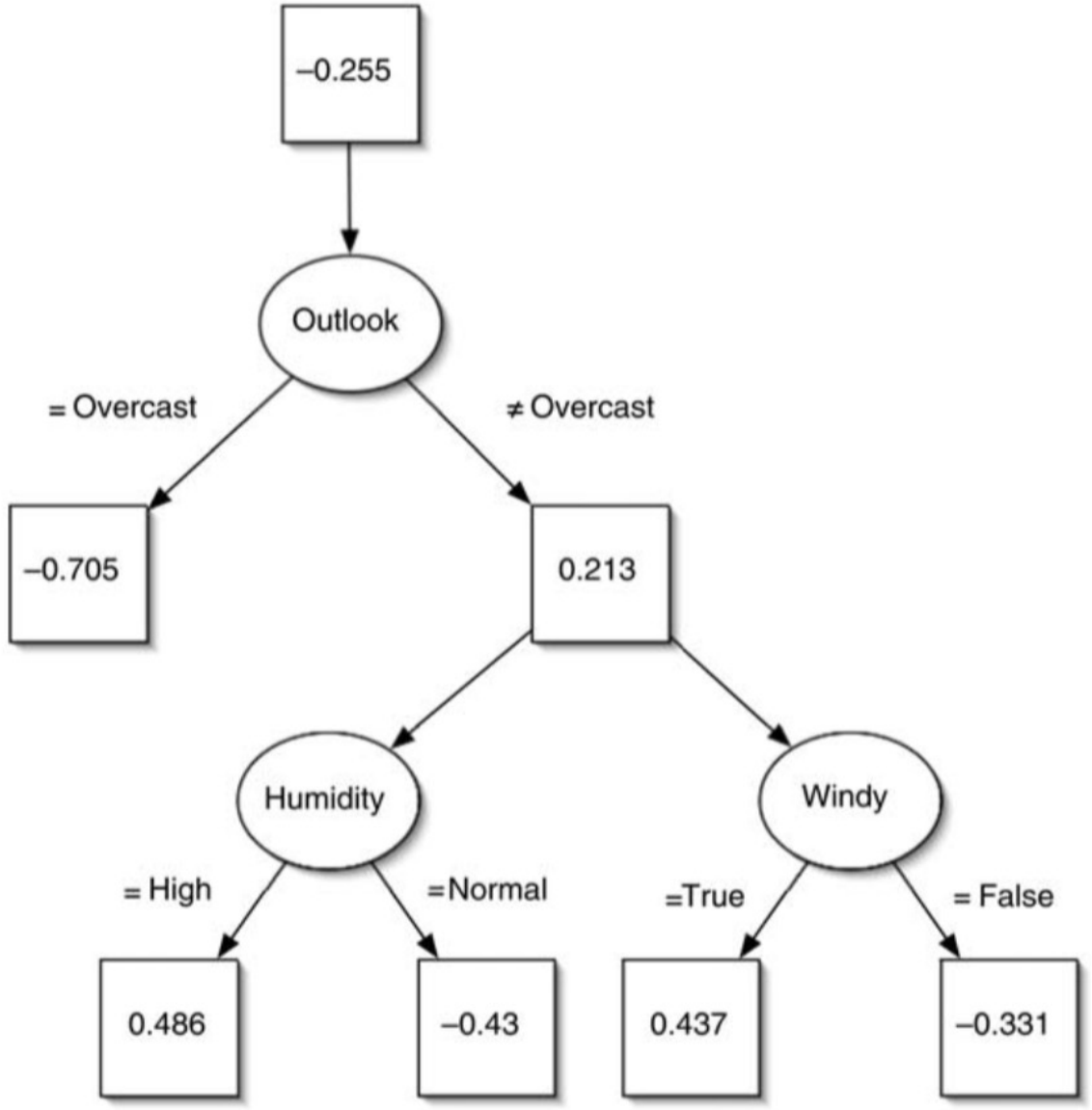


- Can be learned by modifying a standard decision tree learner:
  - Create option node if there are several equally promising splits (within a user-specified interval)
  - When pruning, error at option node is average error of options

Alternating decision trees

- Can also grow an option tree by incrementally adding nodes to it using a boosting algorithm
- The resulting structure is called an *alternating decision tree*, with *splitter nodes* and *prediction* nodes
  - Prediction nodes are leaf nodes if no splitter nodes have been added to them yet
  - Standard alternating tree applies to 2-class problems but the algorithm can be extended to multi-class problems
  - To obtain a prediction from an alternating tree, filter the instance down all applicable branches and sum the predictions
  - Predictions from all relevant predictions nodes need to be used, whether those nodes are leaves or not
  - Predict one class or the other depending on whether the sum is positive or negative

Example tree



Growing alternating trees

- An alternating tree is grown using a boosting algorithm, e.g., the LogitBoost algorithm described earlier:
  - Assume that the base learner used for boosting produces a single conjunctive if-then rule in each boosting iteration
    (an if-then rule for least-squares regression if LogitBoost is used)
  - Each rule could simply be added into the current alternating tree, including the numeric prediction obtained from the rule
  - Problem: tree would grow very large very quickly
  - Solution: base learner should only consider candidate regression rules that extend existing branches in the alternating tree
  - An extension of a branch adds a splitter node and two prediction nodes (assuming binary splits)
  - The standard approach chooses the best extension among all possible extensions applicable to the tree, according to the loss function used
  - More efficient heuristics can be employed instead

Logistic model trees

- Alternating decision trees may still be difficult to interpret
  - The number of prediction nodes that need to be considered for any individual test instance increases exponentially with the depth of tree in the worst case
- But: can also use boosting to build decision trees with linear models at the leaves (trees without options)
  - These trees are often more accurate than standard decision trees but remain easily interpretable because they lack options
- Algorithm for building *logistic model trees* using LogitBoost:
  - Run LogitBoost with simple linear regression as the base learner (choosing the best attribute for linear regression in each iteration)
  - Interrupt boosting when the cross-validated accuracy of the additive model no longer increases
  - Once that happens, split the data (e.g., as in the C4.5 decision tree learner) and resume boosting in the subsets of data that are generated by the split
  - This generates a decision tree with logistic regression models at the leaves
  - Additional overfitting avoidance: prune tree using cross-validation-based cost-complexity pruning strategy from CART tree learner

Stacking

- Question: how to build a *heterogeneous* ensemble consisting of different types of models (e.g., decision tree and neural network)
  - Problem: models can be vastly different in accuracy
- Idea: to combine predictions of base learners, do *not* just vote, instead, use *meta learner*
  - In stacking, the base learners are also called *level-0 models*
  - Meta learner is called *level-1 model*
  - Predictions of base learners are input to meta learner
- Base learners are usually different learning schemes
- Caveat: cannot use predictions on training data to generate data for level-1 model!
  - Instead use scheme based on cross-validation

Generating the level-1 training data

- Training data for level-1 model contains predictions of level-0 models as attributes; class attribute remains the same
- Problem: we cannot use the level-0 models predictions on their *training* data to obtain attribute values for the level-1 data
  - Assume we have a perfect rote learner as one of the level-0 learner
  - Then, the level-1 learner will learn to simply predict this level-0's learners predictions, rendering the ensemble pointless
- To solve this, we generate the level-1 training data by running a *cross-validation* for each of the level-0 algorithms
  - Then, the predictions (and actual class values) obtained for the *test instances* encountered during the cross-validation are collected
  - This pooled data obtained from the cross-validation for each level-0 model is used to train the level-1 model
- If validation data is available, it can be used for level-1 model training

More on stacking

- Stacking is hard to analyze theoretically: “black magic”
- If the base learners can output class probabilities, use those as input to meta learner instead of plain classifications
  - Makes more information available to the level-1 learner
- Important question: which algorithm to use as the meta learner (aka level-1 learner)?
  - In principle, any learning scheme
  - In practice, prefer “relatively global, smooth” models because
  - base learners do most of the work and
  - this reduces the risk of overfitting
- Note that stacking can be trivially applied to numeric prediction too

Ensemble Methods Comparison

Our experimental evaluation:

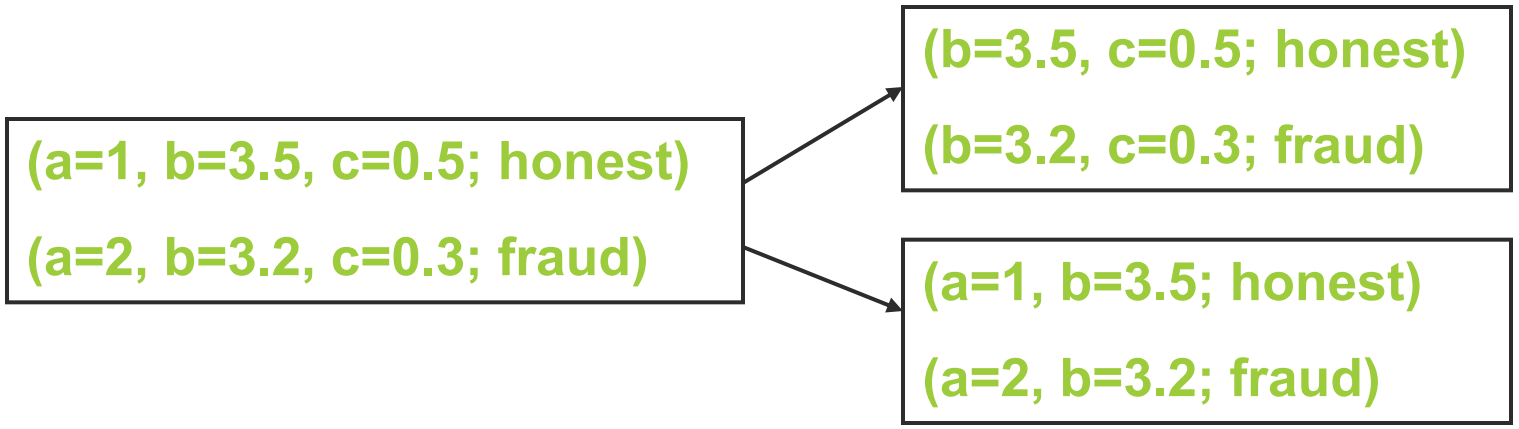
- **Five well-known ensemble methods.**
- **57 public-domain datasets.**
- **5 x 2-fold cross-val and F-test.**
  (5 x 2-fold rather than 10-fold because …)
- **Friedman-Holm test based on ranks.**

Do fancier methods improve on bagging?

Ensemble Methods Comparison

Random subspaces.

- **Randomly select N_F of possible features.**
- **Create classifier using selected features.**
- **Repeat N_C times.**



Ensemble Methods Comparison

Bagging

- Randomly select examples with replacement to make up the training data.
- Usually, you create a bag that is the same size as the original data.
- Typically, 100 or more bagged classifiers are created.

Ensemble Methods Comparison

Random trees.

- **During the tree-building process:**
  - **Find the best N_S splits at each node.**
  - **Randomly select one of these N_S tests.**
- **Repeat to create N_C trees.**

Issues: continuous / discrete features?

Ensemble Methods Comparison

Random forests.

- **Bag to select data for creating a tree.**
- **In creating a tree:**
  - **At each node, randomly select N_F features.**
  - **Select the best test among these features.**
- **Repeat to create N_C trees.**

Ensemble Methods Comparison

Boosting

- Focus on misclassified examples by weighting them more.
- Either use integer weights with repeated examples or incorporate weights into the learning algorithm.
- AdaBoost.M1 works as shown on the next slide.

AdaBoost.M1

- Assign equal weight to each training instance.
- For each of t iterations
  - Apply Learning algorithm to weighted dataset and store resulting model.
  - Compute error e of model on weighted dataset and store error.
  - If e equal to 0 or e >= 0.5:
    - Terminate model generation.
  - For Each instance in dataset:
    - If instance classified correctly by model:
      - Multiply weight of instance by e/(1-e).
  - Normalize weight of all instances.

AdaBoost.M1- Clasification

- Assign weight of 0 to all classes.
- For each of the t (or less) models:
  - Add –log(e/(1-e)) to weight of class predicted by model.
- Return class with highest weight

Ensemble Methods Comparison

Experimental comparison.

- 1,000-classifier ensemble by each method.
- Boosting also evaluated at 50 classifiers.
- Accuracy of others compared to bagging.
- Compare statistically significant wins and losses in accuracy out of 57 datasets.
- Use Bonferroni correction, F-Test with 5x2 fold cross validation.

Ensemble Methods Comparison

- Rank the classifiers from 1 for the most accurate on a data set to 8 for the least accurate.
- Two tie at 3, get a rank of 3.5.
- Apply the nonparametric Freidman test to see if there are differences with many classifiers and many data sets.

Ensemble Methods Comparison

- If Freidman test indicates there is a difference, the Holm test can be used.
- The Holm test allows the comparison of one classifier (bagging) against the rest by differences in rank.
- This approach does not have a problem with overlapping training sets.

Ensemble Methods Comparison

(statistical significance at 0.05 level)

	Win	Loss	"Tie"	Average Rank
Random Forests-2	5	2	50	3.32
Random Forests-lg	6	0	51	3.7
Random Trees	2	4	51	4.53
Random Subspaces	5	9	43	5.39
Boosting (50)	6	0	51	5.15
Boosting (1000)	8	0	49	3.34
Bagging	-	-	-	6.06

Ensemble Methods Comparison

Conclusions.

- **Boosting and RF-lg improve on accuracy of bagging in about 10% of datasets.**
- **Boosting appears to benefit from larger ensemble sizes than once thought.**
- ***Friedman-Holm tells us only boosting-50 and random subspaces fail to improve on bagging.***
- **Methods to automatically choose ensemble size may be important topic to develop.**

Ensemble Methods Comparison

Conclusions.

- **While most approaches are not much more accurate than bagging, they are consistently more accurate.**

You have reached the end of the lecture.

Reference:

I. H. Witten, E. Frank,M. A. Hall and C. J. Pal(2016).*Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann