Ensemble Learning: Bagging & Boosting
Outline

- Ensemble learning to fight bias-variance dilemma
- The **variance** and bootstrap sampling
- **Bagging**
- **Random forests**
- Weak learners and **bias** reduction
- **AdaBoost** algorithm
- Bagging vs Random Forests versus Boosting
Recommended reading

Chapter 13

Chapter 10 & Chapter 15
Ensemble learning

The ultimate goal of any learning algorithm is to make accurate predictions on test data or generalize well (low-bias and low-variance).

Source: http://scott.fortmann-roe.com/docs/BiasVariance.html
The ultimate goal of any learning algorithm is to make accurate predictions on test data or generalize well (low-bias and low-variance).

**Ensemble methods** use multiple learning algorithms to obtain better **predictive performance** than could be obtained from any of the constituent learning algorithms alone!

This is accomplished by either **reducing** the **variance**, **or bias**, or even both!

Source: [http://scott.fortmann-roe.com/docs/BiasVariance.html](http://scott.fortmann-roe.com/docs/BiasVariance.html)
Reducing variance

Variance of a model: how much learned model changes by slightly changing training data! More complex the model space, higher the variance would be!

Each model is trained on a slightly different training set (e.g., replacing a handful of training data with new samples). Variance among models is caused by change in the training data.

Higher the model complexity, larger the variance would be (recall the approximation-estimation (or bias-variance) decomposition.)
Reducing variance

A simple idea to reduce variance: average many noisy but approximately unbiased models/predictions, and hence reduce the variance.
Reducing variance

- But how we can get models to average them?

- Idea: train a single type of classifier (e.g., decision tree) on multiple data sets and average.

- The question is: where do these multiple data sets come from, since we are only given one at training time?

- One option is to fragment your original data set (caveat: every model is trained on only a very small part of the entire data set and is likely to perform poorly)
Bagging

化肥 A better solution is to use bootstrap resampling.

化肥 The basic idea is to **randomly draw datasets with replacement** from the training data (**bootstrap samples**), each sample the same size as the original training set.

化肥 Build a separate prediction model using each **bootstrap** training set. Then we (combine) average the resulting predictions:

- **Regression**: average predictions

- **Classification**: pick the majority of label among all predictions of models
Bagging

A better solution is to use bootstrap resampling.

The basic idea is to randomly draw datasets with replacement from the training data (bootstrap samples), each sample the same size as the original training set.

Build a separate prediction model using each bootstrap training set. Then we (combine) average the resulting predictions.

Bagging = Bootstrap Aggregating

Bootstrap aggregation or bagging is a general-purpose procedure for reducing the variance of a statistical learning method.
Bagging: bootstrap samples

Training sample

\[ S = \{ (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \} \]

Bootstrap samples: each bootstrap sample is independently generated from original data with sampling with replacement with same size \( n \).
Bagging: fitting

\[ S = \{ (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \} \]

We fit a model (e.g., decision tree) to different Bootstrap samples.
Bagging: fitting

\[ S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \]

In generalized bagging, you can use different learners on different samples!
Bagging: final model

\[ S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \]

The final model is aggregation of all models (voting for classification or averaging for regression)

\[ f(x) = \frac{1}{M} \sum_{i=1}^{M} f_i(x) \]
How different are bootstrap samples?

\[ \mathbb{P}[\text{training example } (x_i, y_i) \in \text{ bootstrap } b] \]
How different are bootstrap samples?

\[ P[\text{training example } (x_i, y_i) \in \text{ bootstrap } b] \]

\[ = 1 - \left( 1 - \frac{1}{n} \right)^n \approx 1 - e^{-1} = 0.632 \]

Each bootstrapped dataset includes roughly 63% unique samples from original dataset!
Random forests

- Bagging seems to work especially well for high-variance, low-bias procedures, such as trees.

- Bagging has a problem: The trees produced by different Bootstrap samples can be very similar/correlated.

- Random forests provide an improvement over bagged trees by way of a small tweak that *de-correlates* the models (e.g., trees).
Bagging with de-correlation

- Let assume we learn $M$ models on bootstrapped training samples and average their predictions.

- Assume the variance of each model is $\sigma^2$.

- If the predictions are IID then the variance of average prediction becomes:
  $$\frac{\sigma^2}{M}$$

- However, if the predictions are simply ID (identically distributed but not necessarily independent, i.e., might be correlated) with positive pairwise correlation $\rho$, then the variance of average is:
  $$\rho \sigma^2 + \frac{1 - \rho}{M} \sigma^2$$

- As $M$ increases, the second term disappears, but the first remains, and hence the size of correlation of pairs of models limits the benefits of averaging! So we need a way to reduce the correlation between the models!
Random forests

- As in bagging, we build a number of decision trees on bootstrapped training samples.

- But when building these decision trees, each time a split in a tree is considered, a random sample of $p < d$ features (predictors) is chosen as split candidates from the full set of all $d$ features.

- The split is allowed to use only one of those $p$ features. A fresh sample of features is taken at each split, and typically we choose $p = \sqrt{d}$. 
The rationale

In building a random forest, at each split in the tree, the algorithm is not even allowed to consider a majority of the available features. What’s the rationale behind this?

Suppose that there is one very strong feature in the data set, along with a number of other moderately strong features.

Then in the collection of bagged trees, most or all of the trees will use this strong feature in the top split.

All of the bagged trees will look quite similar to each other and the predictions from the bagged trees will be highly correlated.

Averaging many highly correlated quantities does not lead to as large of a reduction in variance as averaging many uncorrelated quantities.

In particular, this means that bagging will not lead to a substantial reduction in variance over a single tree in this setting.
The rationale

Random forests overcome this problem by forcing each split to consider only a subset of the features.

Therefore, on average \((d - p)/d\) of the splits will not even consider the strong feature, and so other features will have more of a chance.

We can think of this process as de-correlating the trees, thereby making the average of the resulting trees less variable and hence more reliable.

If a random forest is built where \(p = d\), then bagging is a special case of random forests.
Can we make dumb learners smart?

Boosting is the process of taking a crummy learning algorithm (technically called a weak learner) and turning it into a great learning algorithm (technically, a strong learner).
Imagine the situation where you want to build an email filter that can distinguish spam from non-spam.

The general way we would approach this problem in ML/PR follows the same scheme we have for the other topics:

1. **Gathering** as many examples as possible of both spam and non-spam emails.

2. **Train** a classifier using these examples and their labels.

3. Take the learned classifier, or prediction rule, and use it to filter your mail.

4. The goal is to train a classifier that makes the most accurate predictions possible on new test examples.
Imagine the situation where you want to build an email filter that can distinguish spam from non-spam.

But, building a highly accurate classifier is a difficult task (you still get spam, right?!)

We could probably come up with many quick rules of thumb. These could be only moderately accurate [“if the subject line contains ‘buy now’ then classify as spam.”]

This certainly doesn’t cover all spams, but it will be significantly better than random guessing.
A decision stump (a.ka., shallow decision tree) is a machine learning model consisting of a one-level decision tree. That is, it is a decision tree with one internal node (the root) which is immediately connected to the terminal nodes (its leaves). A decision stump makes a prediction based on the value of just a single input feature.

From decisions stumps to accurate rules

- Hard to find single highly accurate prediction rule.
- Easy to find “rules of thumb” that are “often” correct.
From decisions stumps to accurate rules

- Hard to find single highly accurate prediction rule.
- Easy to find “rules of thumb” that are “often” correct.

A decision stump (a.ka., shallow decision tree) is a machine learning model consisting of a one-level decision tree. That is, it is a decision tree with one internal node (the root) which is immediately connected to the terminal nodes (its leaves). A decision stump makes a prediction based on the value of just a single input feature.

Boosting: It is based on the observation that finding many rough rules of thumb can be a lot easier than finding a single, highly accurate classifier.
Fighting the bias-variance tradeoff

Boosting is also aimed at overcoming the bias-variance dilemma:

Simple (a.k.a. **weak learners**) models such as decision stumps:

😊 Are **GOOD**: low variance, don’t usually overfit, easy to learn!

😔 Are **BAD**: high bias, cannot learn very hard problems!

**Question**: how can we boost many weak learners and build a high accuracy learner? Not always, but often yes!
Boosting refers to a general and provably effective method of producing a very accurate classifier by combining rough and moderately inaccurate rules of thumb (slightly better than random guess).

Instead of learning a single (weak) classifier, learn many weak classifiers that are good at different parts of the input space!

Output class: (weighted) vote of each classifier

- Classifiers that are most “sure” will vote with more conviction
- Classifiers will be most “sure” about a particular part of the space
- On average, do better than single classifier!
Adaptive boosting

Idea: given a weak learner, run it multiple times on (re-weighted) training data, then let learned classifiers vote.
Adaptive boosting

**Idea:** given a *weak learner*, run it multiple times on *(re-weighted)* training data, then let learned classifiers vote

---

**AdaBoost Algorithm**

On each iteration $k = 1, 2, \ldots, K$ (number of weak learners):

- Weight each training examples by how incorrectly it was classified (i.e., hardness of classifying example)
- Learn a weak classifier on weighted training data $f_k$
- Compute the strength of this classifier $\alpha_k$

Return final classifier $f(x) = \text{sign} \left( \sum_{k=1}^{K} \alpha_k f_k(x) \right)$

---

**Practically useful**

**Theoretically interesting**

Fit many large or small trees to *re-weighted* versions of the training data. Classify by *weighted majority vote*. 
How should the weights of training examples be chosen/adjusted each round?

How we learn a week classifier (e.g., decision tree, SVM, etc) when there is a weight for each training example?

How to decide the strength of each weak classifier?

How should the weak classifiers be combined into a single classifier?

How many weak classifiers should we learn?
AdaBoost Algorithm

**Inputs:** training samples + $K$ (number of weak learners) + a weak learner

1. Initialize the weights $w^{(0)} = \left[ \frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n} \right]$

2. for $k = 1, 2, \ldots, K$ do:

3. Learn a weak learner $f_k \leftarrow \text{WeakLearner} \left( \mathcal{S}, w^{(k-1)} \right)$

4. Compute the error $\epsilon_k = \sum_{i=1}^{n} w_i^{(k-1)} \mathbb{I} [y_i \neq f_k(x_i)]$

5. Compute the strength $\alpha_k = \frac{1}{2} \log \left( \frac{1 - \epsilon_k}{\epsilon_k} \right)$

6. Adjust the weights $w_i^{(k)} = \frac{1}{Z} w_i^{(k-1)} \exp \left( -\alpha_k y_i f_k(x_i) \right)$

Return $(\alpha_1, f_1), (\alpha_2, f_2), \ldots, (\alpha_K, f_K)$
Toy example (decision stumps)

weak classifiers: vertical or horizontal half-planes

\[ S \]

i.e., \textbf{WeakLearner} \((S, w)\) module returns either a \textbf{vertical} or \textbf{horizontal} line that minimizes \textbf{weighted} loss on training data!
Round nulla

**weak classifiers:**
*vertical* or *horizontal* half-planes

At the beginning, all the training data share the same weight (1/10 here)

\[ w^{(0)} = \left[ \frac{1}{10}, \frac{1}{10}, \ldots, \frac{1}{10} \right] \]

i.e., \( \text{WeakLearner} (S, w) \) module returns either a *vertical* or *horizontal* line that minimizes *weighted* loss on training data!
We learn a vertical predictor on $(S, w^{(0)})$. It makes mistake on three + examples! We increase the weight of these three data and decrease the weight of rest! (telling the next learner to mostly focus on these three). The new weights vector is $w^{(1)}$. 

The size of label shows its current weight!
We learn another vertical predictor on $(S, w^{(1)})$. It makes mistake on three examples! We increase the weight of these two data and decrease the weight of rest! The new weights vector is $w^{(2)}$.

$$
\begin{aligned}
\epsilon_2 &= 0.21 \\
\alpha_2 &= 0.65
\end{aligned}
$$
Finally, we learn a horizontal predictor on \((S, w^{(2)})\) and compute its weighted error on training data to calculate its weight!

\[ f_1 \quad f_2 \quad f_3 \]

\[ \epsilon_3 = 0.14 \]
\[ \alpha_3 = 0.92 \]
Final classifier

The **final classifier** is the weighted combination of learned classifiers: \( f(x) = \text{sign} \left( \sum_{k=1}^{K} \alpha_k f_k(x) \right) \)

\[
\begin{align*}
\mathbf{f} &= \text{sign} \left( \begin{pmatrix}
0.42 \\
+ 0.65 \\
+ 0.92
\end{pmatrix}
\right)
\end{align*}
\]

This is how the decision boundary of final combined classifier looks like!
### AdaBoost Algorithm

**Inputs:** training samples $S$, $K$ (number of learners) + a weak learning algorithm

1: Initialize the weights $w^{(0)} = \left[\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right]

2: for $k = 1, 2, \ldots, K$ do:

3: Learn a weak learner $f_k \leftarrow \text{WeakLearner} \left( S, w^{(k-1)} \right)$

4: Compute the error $\epsilon_k = \sum_{i=1}^{n} w_i^{(k-1)} \mathbb{I}[y_i \neq f_k(x_i)]$

5: Compute the strength $\alpha_k = \frac{1}{2} \log \left( \frac{1 - \epsilon_k}{\epsilon_k} \right)$

6: Adjust the weights $w_i^{(k)} = \frac{1}{Z} w_i^{(k-1)} \exp (-\alpha_k y_i f_k(x_i))$

Return $(\alpha_1, f_1), (\alpha_2, f_2), \ldots, (\alpha_K, f_K)$
AdaBoost Algorithm

**Inputs:** training samples $S$, $K$ (number of learners) + a weak learning algorithm

1: Initialize the weights $\mathbf{w}^{(0)} = \left[ \frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n} \right]$

2: for $k = 1, 2, \ldots, K$ do:

3: Learn a weak learner $f_k \leftarrow W_{\mathbf{w}^{(k-1)}} \left( \mathbf{w}^{(k-1)} \right)$

4: Compute the error $\alpha_k = \frac{1}{2} \log \left( \frac{1}{\mathbf{w}^{(k-1)}_i} \exp (-\alpha_k y_i f_k(x_i)) \right)$

5: Adjust the weights $w_i^{(k)} = \frac{1}{Z} w_i^{(k-1)} \exp (-\alpha_k y_i f_k(x_i))$

Return $(\alpha_1, f_1), (\alpha_2, f_2), \ldots, (\alpha_K, f_K)$

At the beginning, all training examples are considered equally hard. Do not mistaken weights with parameters in SVM or Regression!
AdaBoost Algorithm

**Inputs:** training samples $S$, $K$ (number of learners) + a weak learning algorithm

1: Initialize the weights $\mathbf{w}^{(0)} = \left[ \frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n} \right]$

2: for $k = 1, 2, \ldots, K$ do:

3: Learn a weak learner $f_k \leftarrow \text{WeakLearner} \left( S, \mathbf{w}^{(k-1)} \right)$

4: Compute the error $E_k = \frac{\sum_{i=1}^{n} w_i^{(k-1)} \mathbb{1}[y_i \neq f_k(x_i)]}{\sum_{i=1}^{n} w_i^{(k-1)}}$

5: Compute the strength $S_k = \frac{1}{2} \log \left( \frac{1 - E_k}{E_k} \right)$

6: Adjust the weights $w_i^{(k)} = \frac{1}{Z} w_i^{(k-1)} \exp \left( -S_k y_i f_k(x_i) \right)$

Return $(\alpha_1, f_1), (\alpha_2, f_2), \ldots, (\alpha_K, f_K)$
AdaBoost Algorithm

**Inputs:** training samples $S$, $K$ (number of learners) + a weak learning algorithm

1. Initialize the weights $w^{(0)} = \left[ \frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n} \right]

2. for $k = 1, 2, \ldots, K$ do:

3. Learn a weak learner $f_k \leftarrow \text{WeakLearner} \left( S, w^{(k-1)} \right)$

4. Compute the error $\epsilon_k = \sum_{i=1}^{n} w_i^{(k-1)} \mathbb{I} [y_i \neq f_k(x_i)]$

5. Compute the strength $S_k = \frac{1}{2} \log \frac{1}{\epsilon_k (1-\epsilon_k)}$

6. Adjust the weights $w_i^{(k)} = w_i^{(k-1)} \exp (-S_k y_i f_k(x_i))$

Return $(\alpha_1, f_1), (\alpha_2, f_2), \ldots$
AdaBoost Algorithm

**Inputs:** training samples \( S \), \( K \) (number of learners) + a weak learning algorithm

1: Initialize the weights \( w^{(0)} = \left[ \frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n} \right] \)

2: for \( k = 1, 2, \ldots, K \) do:

3: Learn a weak learner \( f_k \leftarrow \text{WeakLearner} \left( S, w^{(k-1)} \right) \)

4: Compute the error \( \epsilon_k = \sum_{i=1}^{n} w_i^{(k-1)} \mathbb{I} \left[ y_i \neq f_k(x_i) \right] \)

5: Compute the strength \( \alpha_k = \frac{1}{2 \ln(1 - \epsilon_k)} \)

6: Adjust the weights \( w_i^{(k)} = w_i^{(k-1)} \exp \left( \epsilon_k \right) \)

Return \((\alpha_1, f_1), (\alpha_2, f_2), \ldots, (\alpha_K, f_K)\)
AdaBoost Algorithm

**Inputs:** training samples $S$, $K$ (number of learners) + a weak learning algorithm

1: Initialize the weights $\mathbf{w}^{(0)} = \left[\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right]$  
2: for $k = 1, 2, \ldots, K$ do

3: Learn a weak learner $f_k(x_i) = S_{(\mathbf{w}^{(k-1)}, \mathbf{w}^{(k-1)})}$

4: Compute the error $\epsilon_k = \sum_{i=1}^{n} \mathbf{w}_i^{(k-1)} \mathbb{I} [y_i \neq f_k(x_i)]$

5: Compute the strength $\alpha_k = \frac{1}{2} \log \left(\frac{1 - \epsilon_k}{\epsilon_k}\right)$

6: Adjust the weights $w_i^{(k)} = \frac{1}{Z} w_i^{(k-1)} \exp\left(-\alpha_k y_i f_k(x_i)\right)$

Return $(\alpha_1, f_1), (\alpha_2, f_2), \ldots, (\alpha_K, f_K)$
AdaBoost Algorithm

**Inputs:** training samples $S$, $K$ (number of learners) + a weak learning algorithm

1: Initialize the weights $\mathbf{w}^{(0)} = \left[ \frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n} \right]$

2: for $k = 1, 2, \ldots, K$ do:

3: Learn a weak learner $f_k$.

4: Compute the error $\epsilon_k$.

5: Compute the strength $\alpha_k = \frac{1}{2} \log \left( \frac{1 - \epsilon_k}{\epsilon_k} \right)$.

6: Adjust the weights $w_i^{(k)} = \frac{1}{Z} w_i^{(k-1)} \exp \left( -\alpha_k y_i f_k(x_i) \right)$.

Return $\left( \alpha_1, f_1 \right), \left( \alpha_2, f_2 \right), \ldots, \left( \alpha_K, f_K \right)$.
AdaBoost Algorithm

**Inputs:** training samples $\mathcal{S}$, $K$ (number of learners) + a weak learning algorithm

1: Initialize the weights $\mathbf{w}^{(0)} = \left[ \frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n} \right]$

2: for $k = 1, 2, \ldots, K$ do:

3: Learn a weak learner $f_k \leftarrow \text{WeakLearner} \left( \mathcal{S}, \mathbf{w}^{(k-1)} \right)$

4: Compute the error $\epsilon_k = \sum_{i=1}^{n} \mathbf{w}^{(k-1)} \mathbb{I} [ y_i \neq f_k(\mathbf{x}_i) ]$

5: Compute the strength $s_k = \frac{1}{2} \log \frac{1}{\epsilon_k}$

6: Adjust the weights $\mathbf{w}^{(k)} = \mathbf{w}^{(k-1)} \exp \left( -s_k y_i f_k(\mathbf{x}_i) \right)$

Return $\left( \alpha_1, f_1 \right), \left( \alpha_2, f_2 \right), \ldots, \left( \alpha_K, f_K \right)$
AdaBoost with cost-sensitive classification

**Question:** How to learn a classifier on a weighted training data

- The error is not uniform for all training examples
- The higher weight means, larger loss for making mistake
- The smaller weight means, smaller loss for making mistake

A simple solution is to change the training objective and make it weighted (i.e., **penalize different levels of loss for different mistakes**)

\[
\arg\min_f \sum_{i=1}^n w_i \ell(f; (x_i, y_i))
\]

Known as **cost-sensitive classification**
At each stage, we generate a new training data by sampling each example by its weight (examples with larger weight get higher chance to be included).

An alternative solution is to sample a new training data based on weight of individual examples (can easily use existing algorithms without any change!).
Need to decide the rounds of boosting (# of weak learners in final model) by cross-validation

![Graph showing percent error vs. number of weak learners](image_url)
AdaBoost and # of weak classifiers II

Early stopping: compute the error on validation set after every round of boosting, and stop training as soon as validation error starts increasing!
Bagging versus Boosting

Ensembling

- Boosting
  - Can overfit
  - Reduces bias & variance
  - Sequential learning
    - Handles overfitting
    - Reduces variance
    - Independent learning

- Bagging

- AdaBoost
Bagging versus Boosting

Bagging (parallel)

Boosting (adaptive and sequential)
Gradient Boosting = Gradient Descent + Boosting

- In AdaBoost, “shortcomings” are identified by high-weight data points.

- In Gradient Boosting, “shortcomings” are identified by gradients.

Gradient boosting

When squared loss is used for regression, the gradients are equal to residuals:

\[ f_1(x) \approx y \]
\[ f_2(x) \approx y - f_1(x) \]
\[ f_3(x) \approx y - f_1(x) - f_2(x) \]

The training labels at each round is the residual accumulated so far

\[ S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \]

We feed the residual errors as new labels!
Gradient boosting

<table>
<thead>
<tr>
<th>Round</th>
<th>Residual</th>
<th>Total Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="image1" alt="Residual" /></td>
<td><img src="image2" alt="Total Prediction" /></td>
</tr>
<tr>
<td>2</td>
<td><img src="image3" alt="Residual" /></td>
<td><img src="image4" alt="Total Prediction" /></td>
</tr>
<tr>
<td>5</td>
<td><img src="image5" alt="Residual" /></td>
<td><img src="image6" alt="Total Prediction" /></td>
</tr>
<tr>
<td>9</td>
<td><img src="image7" alt="Residual" /></td>
<td><img src="image8" alt="Total Prediction" /></td>
</tr>
</tbody>
</table>
Machine Learning Challenge Winning Solutions

XGBoost is extensively used by machine learning practitioners to create state of the art data science solutions, this is a list of machine learning winning solutions with XGBoost. Please send pull requests if you find ones that are missing here.

- Maksims Volkovs, Guangwei Yu and Torrii Poutanen, 1st place of the 2017 ACM RecSys challenge. Link to paper.
- Vlad Sandulescu, Mihal Chiru, 1st place of the KDD Cup 2016 competition. Link to the arxiv paper.
- Marius Michailidis, Mathias Müller and HJ van Veen, 1st place of the Dato Truely Native? competition. Link to the Kaggle interview.
- Vlad Mironov, Alexander Guschin, 1st place of the CERN LHCb experiment Flavour of Physics competition. Link to the Kaggle interview.
- Josef Stanek, 3rd place of the CERN LHCb experiment Flavour of Physics competition. Link to the Kaggle interview.
- Mario Filho, José Fagl, Lucas, Gilberto, 1st place of the Caterpillar Tube Pricing competition. Link to the Kaggle interview.
- Qingchen Wang, 1st place of the Liberty Mutual Property Inspection. Link to the Kaggle interview.
- Chenglong Chen, 1st place of the Crowdflower Search Results Relevance. Link to the winning solution.
- Alexandre Barachant (“Cat”) and Rafał Cyroń (“Dog”), 1st place of the Grasp-and-Lift EEG Detection. Link to the Kaggle interview.
- Haila Yang, 2nd place of the Recruit Coupon Purchase Prediction Challenge. Link to the Kaggle interview.
- Owen Zhang, 1st place of the Avito Context Ad Clicks competition. Link to the Kaggle interview.
- Kelichi Kuroyanagi, 2nd place of the Airbnb New User Bookings. Link to the Kaggle interview.
- Marius Michailidis, Mathias Müller and Ning Sui, 1st place Homesite Quote Conversion. Link to the Kaggle interview.

Very successful in many competitions:

conda install -c conda-forge xgboost

from xgboost import XGBClassifier

xgb = XGBClassifier()
xgb.fit(X_train, y_train)
xgb.score(X_test, y_test))
Bagging versus Boosting

Spam Data

Test Error

Number of Trees

0
500
1000
1500
2000
2500
0.040
0.045
0.050
0.055
0.060
0.065
0.070

Bagging
Random Forest
Gradient Boosting (5 Node)
• Decision trees can be simple, but often produce noisy (bushy) or weak (stunted) classifiers.

• **Bagging** (Breiman, 1996): Fit many large trees to bootstrap-resampled versions of the training data, and classify by majority vote.

• **Boosting** (Freund & Shapire, 1996): Fit many large or small trees to re-weighted versions of the training data. Classify by weighted majority vote.

• **Random Forests** (Breiman 1999): Fancier version of bagging.

• In general

   **Boosting** > **Random Forests** > **Bagging** > **Single Tree**

   [where ‘>‘ means performs better]
When to use tree-based ensemble models

- Model non-linear relationships
- Doesn’t care about scaling, no need for feature engineering
- Single tree: very interpretable (if small)
- Random forests very robust, good benchmark
- Gradient boosting often best performance with careful tuning