Random Forests and Ferns

David Capel

With annotations by Bob!
The Multi-class Classification Problem

Training: Labelled exemplars representing multiple classes

<table>
<thead>
<tr>
<th>Handwritten digits</th>
<th>Planes</th>
<th>Faces</th>
<th>Cars</th>
<th>Cats</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><img src="image1.png" alt="Plane" /></td>
<td><img src="image2.png" alt="Face" /></td>
<td><img src="image3.png" alt="Car" /></td>
<td><img src="image4.png" alt="Cat" /></td>
</tr>
<tr>
<td>1</td>
<td><img src="image1.png" alt="Plane" /></td>
<td><img src="image2.png" alt="Face" /></td>
<td><img src="image3.png" alt="Car" /></td>
<td><img src="image4.png" alt="Cat" /></td>
</tr>
<tr>
<td>2</td>
<td><img src="image1.png" alt="Plane" /></td>
<td><img src="image2.png" alt="Face" /></td>
<td><img src="image3.png" alt="Car" /></td>
<td><img src="image4.png" alt="Cat" /></td>
</tr>
<tr>
<td>3</td>
<td><img src="image1.png" alt="Plane" /></td>
<td><img src="image2.png" alt="Face" /></td>
<td><img src="image3.png" alt="Car" /></td>
<td><img src="image4.png" alt="Cat" /></td>
</tr>
<tr>
<td>4</td>
<td><img src="image1.png" alt="Plane" /></td>
<td><img src="image2.png" alt="Face" /></td>
<td><img src="image3.png" alt="Car" /></td>
<td><img src="image4.png" alt="Cat" /></td>
</tr>
<tr>
<td>5</td>
<td><img src="image1.png" alt="Plane" /></td>
<td><img src="image2.png" alt="Face" /></td>
<td><img src="image3.png" alt="Car" /></td>
<td><img src="image4.png" alt="Cat" /></td>
</tr>
<tr>
<td>6</td>
<td><img src="image1.png" alt="Plane" /></td>
<td><img src="image2.png" alt="Face" /></td>
<td><img src="image3.png" alt="Car" /></td>
<td><img src="image4.png" alt="Cat" /></td>
</tr>
<tr>
<td>7</td>
<td><img src="image1.png" alt="Plane" /></td>
<td><img src="image2.png" alt="Face" /></td>
<td><img src="image3.png" alt="Car" /></td>
<td><img src="image4.png" alt="Cat" /></td>
</tr>
<tr>
<td>8</td>
<td><img src="image1.png" alt="Plane" /></td>
<td><img src="image2.png" alt="Face" /></td>
<td><img src="image3.png" alt="Car" /></td>
<td><img src="image4.png" alt="Cat" /></td>
</tr>
<tr>
<td>9</td>
<td><img src="image1.png" alt="Plane" /></td>
<td><img src="image2.png" alt="Face" /></td>
<td><img src="image3.png" alt="Car" /></td>
<td><img src="image4.png" alt="Cat" /></td>
</tr>
</tbody>
</table>

Classifying: to which class does this new example belong?

![New Example](image5.png)
The Multi-class Classification Problem

- A classifier is a mapping $H$ from feature vectors $f$ to discrete class labels $C$

$$f = (f_1, f_2, ..., f_N)$$
$$C \in \{c_1, c_2, ..., c_K\}$$
$$H : f \rightarrow C$$

- Numerous choices of feature space $F$ are possible, e.g.

  - Raw pixel values
  - Texton histograms
  - Color histograms
  - Oriented filter banks
The Multi-class Classification Problem

- We have a (large) database of labelled exemplars

\[ D^m = (\mathbf{f}^m, C^m) \quad \text{for } m = 1 \ldots M \]

- Problem: Given such training data, learn the mapping \( \mathbf{H} \)

\[ \mathbf{H} : \mathbf{f} \rightarrow C \]

- Even better: learn the posterior distribution over class label conditioned on the features:

\[ P(C = c_k | f_1, f_2, \ldots, f_N) \]

..and obtain classifier \( \mathbf{H} \) as the mode of the posterior:

\[ H(\mathbf{f}) = \arg\max_k P(C = c_k | f_1, f_2, \ldots, f_N) \]
The Multi-class Classification Problem

- We have a (large) database of labelled exemplars

\[ D^m = (f^m, C^m) \quad \text{for } m = 1 \ldots M \]

- **Problem**: Given such training data, learn the mapping \( H \)

\[ H : f \rightarrow C \]

- **Even better**: learn the posterior distribution over class label conditioned on the features:

\[ P(C = c_k | f_1, f_2, \ldots, f_N) \]

..and obtain classifier \( H \) as the mode of the posterior:

\[ H(f) = \arg\max_k P(C = c_k | f_1, f_2, \ldots, f_N) \]
How do we represent and learn the mapping?

There are numerous ways to build multi-class classifiers. See, for example, Bishop’s PRML book.

- Direct, non-parametric learning of class posterior (Bag of Words)
- K-Nearest Neighbours
- Fisher Linear Discriminants
- Relevance Vector Machines
- Multi-class SVMs

Direct, non-parametric

$k$-Nearest Neighbours
Binary Decision Trees

- Decision trees classify features by a series of Yes/No questions
- At each node, the feature space is split according to the outcome of some binary decision criterion
- The leaves are labelled with the class \( C \) corresponding the feature reached via that path through the tree
Binary Decision Trees

- Decision trees classify features by a series of Yes/No questions
- At each node, the feature space is split according to the outcome of some binary decision criterion
- The leaves are labelled with the class $C$ corresponding to the feature reached via that path through the tree
Binary Decision Trees

- Decision trees classify features by a series of Yes/No questions
- At each node, the feature space is split according to the outcome of some binary decision criterion
- The leaves are labelled with the class $C$ corresponding the feature reached via that path through the tree
Binary Decision Trees

- Decision trees classify features by a series of Yes/No questions
- At each node, the feature space is split according to the outcome of some binary decision criterion
- The leaves are labelled with the class $\mathbf{C}$ corresponding the feature reached via that path through the tree
Binary Decision Trees: Training

- To train, recursively partition the training data into subsets according to some Yes/No tests on the feature vectors.
- Partitioning continues until each subset contains features of a single class.
Binary Decision Trees: Training

- To train, recursively partition the training data into subsets according to some Yes/No tests on the feature vectors
- Partitioning continues until each subset contains features of a single class
Binary Decision Trees: Training

• To train, recursively partition the training data into subsets according to some Yes/No tests on the feature vectors
• Partitioning continues until each subset contains features of a single class
Binary Decision Trees: Training

- To train, recursively partition the training data into subsets according to some Yes/No tests on the feature vectors.
- Partitioning continues until each subset contains features of a single class.
Binary Decision Trees: Training

- To train, recursively partition the training data into subsets according to some Yes/No tests on the feature vectors.
- Partitioning continues until each subset contains features of a single class.
Common decision rules

• **Axis-aligned splitting**
  Threshold on a single feature at each node
  Very fast to evaluate
  \[ f_n > T \]

• **General plane splitting**
  Threshold on a linear combination of features
  More expensive but produces smaller trees
  \[ w^T f > T \]
Note, different choices of partitions could lead to simpler trees.
Note, different ordering / choices of partitions could lead to simpler trees.
Choosing the right split

- The split location may be chosen to optimize some measure of classification performance of the child subsets.
- Encourage child subsets with lower entropy (confusion).

\[ \text{fraction of each class } p_k \quad f_n > T ? \]
Choosing the right split

• The split location may be chosen to optimize some measure of classification performance of the child subsets
• Encourage child subsets with lower entropy (confusion)

\[
\text{fraction of each class } p_k \quad \text{\( f_n > T \) ?} 
\]

• Gini impurity

\[
I_G = \sum_k p_k^{\text{left}} (1 - p_k^{\text{left}}) + \sum_k p_k^{\text{right}} (1 - p_k^{\text{right}})
\]
Choosing the right split

- The split location may be chosen to optimize some measure of classification performance of the child subsets
- Encourage child subsets with lower entropy (confusion)

\[
\text{fraction of each class } p_k \quad f_n > T ?
\]

- Gini impurity

\[
I_G = \sum_k p_k^{\text{left}} (1 - p_k^{\text{left}}) + \sum_k p_k^{\text{right}} (1 - p_k^{\text{right}})
\]

- Information gain (entropy reduction)

\[
I_E = - \sum_k p_k^{\text{left}} \log p_k^{\text{left}} - \sum_k p_k^{\text{right}} \log p_k^{\text{right}} - p \log p
\]
Pros and Cons of Decision Trees

**Advantages**

- Training can be fast and easy to implement
- Easily handles a large number of input variables

**Drawbacks**

- Clearly, it is always possible to construct a decision tree that scores 100% classification accuracy on the *training* set
- But they tend to over-fit and do not generalize very well
- Hence, performance on the *testing* set will be far less impressive

*So, what can be done to overcome these problems?*
Random Forests

Breiman, Leo (2001) "Random Forests" Machine Learning

Basic idea

• Somehow introduce randomness into the tree-learning process
• Build multiple, independent trees based on the training set
• When classifying an input, each tree votes for a class label
• The forest output is the consensus of all the tree votes
• *If the trees really are independent, the performance should improve with more trees*

Related concept: Ensemble learning
How to introduce randomness?

- **Bagging**
  Generate randomized training sets by sampling with replacement from the full training set (bootstrap sampling)

<table>
<thead>
<tr>
<th>Full training set</th>
<th>D_1</th>
<th>D_2</th>
<th>D_3</th>
<th>D_4</th>
<th>D_5</th>
<th>D_6</th>
<th>D_7</th>
<th>D_8</th>
<th>D_9</th>
<th>D_{10}</th>
<th>D_{11}</th>
<th>D_{12}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random “bag”</td>
<td>D_4</td>
<td>D_9</td>
<td>D_3</td>
<td>D_4</td>
<td>D_{12}</td>
<td>D_{10}</td>
<td>D_7</td>
<td>D_3</td>
<td>D_1</td>
<td>D_6</td>
<td>D_1</td>
<td></td>
</tr>
</tbody>
</table>

- **Feature subset selection**
  Choose different random subsets of the full feature vector to generate each tree

<table>
<thead>
<tr>
<th>Full feature vector</th>
<th>f_1</th>
<th>f_2</th>
<th>f_3</th>
<th>f_4</th>
<th>f_5</th>
<th>f_6</th>
<th>f_7</th>
<th>f_8</th>
<th>f_9</th>
<th>f_{10}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature subset</td>
<td>f_4</td>
<td>f_6</td>
<td>f_7</td>
<td>f_9</td>
<td>f_{10}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Related idea: choose one feature at random to threshold on at each node in the tree.
Each tree votes for which class. Combining those votes and normalizing gives a posterior on class label.
Performance comparison

- Results from Ho 1995  *(This is an old idea, by the way)*
- Handwritten digits (10 classes)
- 20x20 pixel binary images
- 60000 training, 10000 testing samples

- Feature set $\mathbf{f_1}$: raw pixel values (400 features)
- Feature set $\mathbf{f_2}$: $\mathbf{f_1}$ plus gradients (852 features in total)

- Uses full training set for every tree (no bagging)
- Compares different features subset sizes (100 and 200 resp)
Performance comparison

![Graph showing performance comparison with different methods and parameters.](Image)
Revision: Naive Bayes Classifiers

• We would like to evaluate the posterior probability over class label

\[
\arg\max_k P(C_k|f_1, f_2, ..., f_N)
\]

• Bayes’ rules tells us that this is equivalent to

\[
\arg\max_k P(f_1, f_2, ..., f_N|C_k)P(C_k) \quad \text{(likelihood \times prior)}
\]

• But learning the joint likelihood distributions over all features is most likely intractable!

• Naive Bayes makes the simplifying assumption that features are conditionally independent given the class label

\[
P(f_1, f_2, ..., f_N|C_k) = \prod_{i=1}^{N} P(f_i|C_k)
\]
Revision: Naive Bayes Classifiers

\[
\text{Class}(f) \equiv \arg \max_k P(C_k) \prod_{n=1}^{N} P(f_n|C_k)
\]

- This independence assumption is usually false!
- The resulting approximation tends to grossly underestimate the true posterior probabilities

However ...
- It is usually easy to learn the 1-d conditional densities \(P(f_i|C_k)\)
- It often works rather well in practice!
- \textit{Can we do better without sacrificing simplicity?}
Ferns: "Semi-Naive" Bayes


• Group of features into L small sets of size S (called Ferns)
  \[ F_l = \{ f_{l,1}, f_{l,2}, \ldots, f_{l,S} \} \]
  where features \( f_n \) are the outcome of a binary test on the input vector, such that \( f_n : \{0,1\} \)

• Assume **groups** are conditionally independent, hence
  \[ P(f_1, f_2, \ldots, f_N | C_k) = \prod_{l=1}^{L} P(F_l | C_k) \]

• Learn the class-conditional distributions for each group and apply Bayes rule to obtain posterior,
  \[ \text{Class}(\mathbf{f}) \equiv \arg\max_k P(C_k) \prod_{l=1}^{L} P(F_l | C_k) \]
Naive vs Semi-Naive Bayes

- Full joint class-conditional distribution: \( P(f_1, f_2, ..., f_N | C_k) \)
  Intractable to estimate

- Naive approximation: \( P(f_1, f_2, ..., f_N | C_k) = \prod_{i=1}^{N} P(f_i | C_k) \)
  Too simplistic
  Poor approximation to true posterior

- Semi-Naive: \( P(f_1, f_2, ..., f_N | C_k) = \prod_{l=1}^{L} P(F_l | C_k) \)
  Balance of complexity and tractability
  Trade complexity/performance by choice of Fern size (S), NumFerns (L)
So how does a single Fern work?

- A Fern applies a series of $S$ binary tests to the input vector $\mathbf{1}$
  
  e.g. relative intensities of a pair of pixels:
  
  $f_1(\mathbf{1}) = I(x_a, y_a) > I(x_b, y_b)$ -> true
  
  $f_2(\mathbf{1}) = I(x_c, y_c) > I(x_d, y_d)$ -> false

- This gives an $S$-digit binary code for the feature which may be interpreted as an integer in the range $[0 \ldots 2^S-1]$

- It’s really just a simple hashing scheme that drops input vectors into one of $2^S$ “buckets”
So how does a single Fern work?

- The output of a fern when applied to a large number of input vectors of the same class is a **multinomial distribution**
Training a Fern

- Apply the fern to each labelled training example \( D_m = (l_m, c_m) \) and compute its output \( F(D_m) \)
- Learn multinomial densities \( p(F|C_k) \) as histograms of fern output for each class
Classifying using a single Fern

- Given a test input, simply apply the fern and “look-up” the posterior distribution over class label

Test input $I$  

Apply fern $F(I)=[00011]$  

$F(I)=3$  

$p(F|C_0)$  

$p(F|C_1)$  

$p(F|C_2)$  

$p(F|C_K)$  

Normalize distribution  

$C_1$  

$C_2$  

$C_3$  

$C_K$  

Class posterior

$$p(C_k|F) = \frac{p(F|C_k)}{\sum_k p(F|C_k)}$$

Note: This is assuming equal class priors.
Adding randomness: an ensemble of ferns

- A single fern does not give great classification performance
- **But ..** we can build an ensemble of “independent” ferns by randomly choosing different subsets of features

  e.g. \( F_1 = \{f_2, f_7, f_{22}, f_5, f_9\} \)
  \( F_2 = \{f_4, f_1, f_{11}, f_8, f_3\} \)
  \( F_3 = \{f_6, f_{31}, f_{28}, f_{11}, f_2\} \)

- Finally, combine their outputs using Semi-Naive Bayes:

  \[
  \text{Class}(f) \equiv \arg\max_k P(C_k) \prod_{l=1}^{L} P(F_l|C_k)
  \]
Classifying using Random Ferns

- Having randomly selected and trained a collection of ferns, classifying new inputs involves only simple look-up operations.
One small subtlety...

- Even for moderate size ferns, the output range can be large
  e.g. fern size $S=10$, output range = $[0 \ldots 2^{10}] = [0 \ldots 1024]$
- Even with large training set, many "buckets" may see no samples

These zero-probabilities will be problematic for our Semi-Naive Bayes!

- So .. assume a Dirichlet prior on the distributions $P(F|C_k)$
- Assigns a baseline low probability to all possible fern outputs:

\[
p(F = z|C_k) = \frac{N(F = z|C_k) + 1}{\sum_{z=0}^{2^S-1} (N(F = z|C_k) + 1)}
\]

where $N(F=z|C_k)$ is the number of times we observe fern output equal to $z$ in the training set for class $C_k$
Comparing Forests and Ferns

**Forests**
- Decision trees directly learn the posterior $P(C_k|F)$
- Applies different sequence of tests in each child node
- Training time grows exponentially with tree depth
- Combine tree hypotheses by averaging

**Ferns**
- Learn class-conditional distributions $P(F|C_k)$
- Applies the same sequence of tests to every input vector
- Training time grows linearly with fern size $S$
- Combine hypothesis using Bayes rule (multiplication)

Note: Fern can be interpreted as a decision tree with the same feature used for all decisions at one level of the tree.
Application: Fast Keypoint Matching

- Ozuysal et al. use Random Ferns for keypoint recognition
- Similar to typical application of SIFT descriptors
- Very robust to affine viewing deformations
- Very fast to evaluate (13.5 microsec per keypoint)
In my opinion, for some applications, (e.g. pedestrian/face tracking) the extreme rotations are overkill.

Training

- Each keypoint to be recognized is a separate class
- Training sets are generated by synthesizing random affine deformations of the image patch (10000 samples)

![Synthesized viewpoint variation](image)

- Features are pairwise intensity comparisons: $f_n(I) = I(x_a,y_a) > I(x_b,y_b)$
- Fern size $S=10$ (randomly selected pixel pairs)
- Ensembles of 5 to 50 ferns are tested
Classification Rate vs Number of Ferns

- 300 classes were used
- Also compares to Random Trees of equivalent size
Classification Rate vs Number of Classes (Keypoints)

- 30 random ferns of size 10 were used
Drawback: Memory requirements

- **Fern classifiers can be very memory hungry, e.g.**

  Fern size = 11
  Number of ferns = 50
  Number of classes = 1000

  \[
  \text{RAM} = 2^{\text{fern size}} \times \text{sizeof(float)} \times \text{NumFerns} \times \text{NumClasses}
  \]
  \[
  = 2^{11} \times 4 \times 50 \times 1000
  \]
  \[
  = 2048 \times 4 \times 50 \times 1000
  \]
  \[
  = 400 \text{ MBytes!}
  \]
Conclusions?

- Random Ferns are easy to understand and easy to train
- Very fast to perform classification once trained
- Provide a probabilistic output (how accurate though?)
- Appear to outperform Random Forests
- Can be very memory hungry!