Predictive Analytics Overview

Robert T. Bauer

Why Predictive Analytics?

Predictive Analytics is driving innovation throughout IT.

So at least one good reason to learn about predictive analytics is that it will help you in your career or perhaps help you start a new one.

It is fun and interesting.

It can be very deep mathematically, with lots of opportunity to learn new things.

There are plenty of tools available. Almost anyone can get some sort of result quickly. You'll have to know something, though, to get good results.

Lastly, in my opinion, guessing the future is far more useful than rationalizing the past.

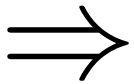
Robert - Bio

- Debugging is twice as hard as writing the code in the first place. Therefore if you write the code as cleverly as possible, you are by definition, not smart enough to debug it.
- It's easy to mix up the simplicity that comes from a deep analysis with a simple-minded analysis that leads to hopeless complexities. Finding the key simplicities is a deep problem not often resolved by a simple-minded approach; but these simplicities, found before the detailed programming, spell the difference between a program completed quickly and cleanly and one difficult to finish and even more difficult to debug.

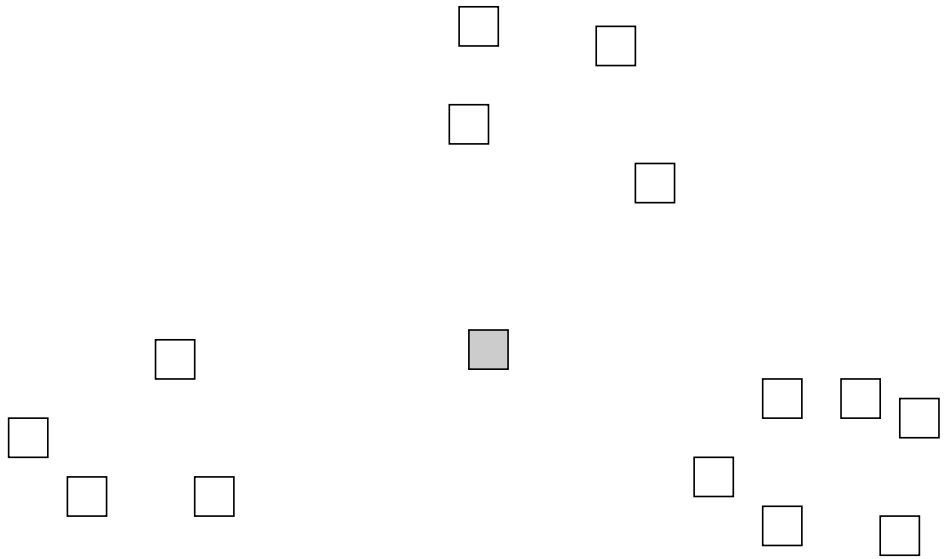
My motivation to understand predictive analytics:

Large-scale IT operations include incident management. The main function of incident management is to restore service. Systems on the threshold of failing very often return to normal without any intervention. If an analytic engine could accurately advise incident management that conditions being observed will return to normal without intervention, we can avoid significant cost.

Outline



Nearest Neighbor
Classification & Approximation



Background

KNN is the simplest machine learning technique. It is called *instance*-based or *lazy* because all classification work is done when a query instance is presented.

"Classification" is overloaded. It means approximating the value of real or discrete-valued functions (statisticians say *regression*). It also means determining the class of an object — its class being the one most common within its K nearest neighbors.

To approximate a real-valued function, the computed value is taken as the average of the K nearest neighbors; often this average is weighted by the distance, so that "nearer" values contribute more to the computation than "farther" values.

Definitions

All data instances correspond to points in $\mathbb{R}^n$.

Nearest neighbors are defined in terms of Euclidean distance:

$$\delta(x, y) \stackrel{\text{def}}{=} \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Let $\mathcal{T}$ be a set of m data instances.

Let $\bar{\mathcal{T}} = \{t \in \mathbb{R}^n \wedge t \notin \mathcal{T}\}$.

Let f be a function defined over $\mathcal{T}$. Its *sort* determines whether we intend classification or regression.

Classification - Problem Statement

Let P be a partition of $\mathcal{T}$ – recall that a partition of T is a set of nonempty subsets such that every element $t \in T$ is in exactly one of these subsets.

We'll call each subset $P' \in P$ a *class* and call $f : \mathcal{T} \to P$ the classification function; more generally, it is called the learned hypothesis.

Given a data instance, q (called the query instance), we wish to determine to which class it belongs. When $q \in \mathcal{T}$, f is sufficient. When $q \in \bar{\mathcal{T}}$, we need to approximate f.

Intuitively, we'd like to approximate f with $\hat{f}$ such that if $q \in \bar{\mathcal{T}}$ is most "like" $p \in \mathcal{T}$ then $\hat{f}(q) = f(p)$.

Classification - Learning

"Most like" is defined in terms of Euclidean distance:

$$q \in \bar{\mathcal{T}} \text{ is "most like" } t \in \mathcal{T}, \text{ if } \forall u \in \mathcal{T}\ \delta(q,u) \geq \delta(q,t)$$

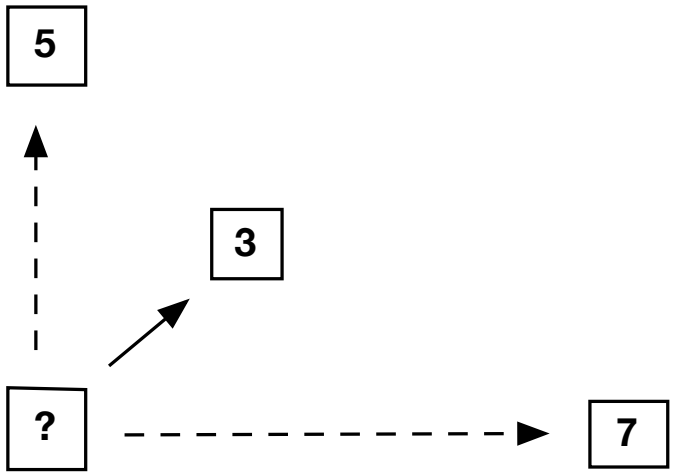
When $K = 1$,we classify q as t is classified $(f(q) = f(t))$

For $K > 1$, we classify as the majority of the K-nearest neighbors

```
type CLass = String
type Vec = [Num]
type CVec = [(CLass, Vec)]
distance :: Vec-> Vec-> Num
classify :: Vec -> CVec -> Int -> CLass
classify (Vec q) (CVec c) (Int k) =
   vote $ take k $ sort $ map (distance q) c
```

But, it's better to use an R Tree or similar spatial data structure.

Approximation - Intuition



Approximation - Learning

As before, "most like" is defined in terms of Euclidean distance. When $K = 1$, we approximate $f(q)$ as $f(t)$ where t is "most like" q.

When $K > 1$, we approximate $f(q)$ as the average or distance weighted average[1].

```
type FVal = Num
type Vec = [Num]
type CVec = [(FVal,Vec)] (* extensive representation *)
distance :: Vec -> Vec -> Num
approx :: Vec -> CVec -> Int -> Num
approx (Vec q) (CVec c) (Int k)  =
  avg $ take k $ sort $ map (distance q) c
```

[1] Radial bias and locally weighted regression are used as well.

KNN - Summary

- Simple to implement – when "similarity" is simple[2]
- Use spatial data structures to improve performance
- Large dimensions (more than 20) are problematic
- Classification inductive bias - data instances near each other are similar to each other
- Approximation inductive bias - ?? Let's discuss

[2] Categorical, biological, string and other data makes similarity hard; seeTanimoto, Jaccard, Dice, Sorenson and other distance/similarity measures.

Naive Bayesian Classification

Basic Probability Formulas

- Covering Rule: $P(A) = P(A|B) + P(A|\bar{B})$
- Product Rule:
  $P(A \wedge B) = P(A|B) \times P(B) = P(B|A) \times P(A)$
- Sum Rule: $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$
- Bayes Theorem: $P(A|B) = \frac{P(A) \times P(B|A)}{P(B)}$
- Total Probability: If $A_1, \ldots, A_n$ are mutually exclusive

$$\sum_{i=1}^{n} P(A_i) = 1 \Rightarrow P(B) = \sum_{i=1}^{n} P(B|A_i) \times P(A_i)$$

Connecting Data and Hypothesis

$$P(H, D) = P(H|D) \times P(D)$$

$$P(D, H) = P(D|H) \times P(H)$$

$$P(H, D) = P(D, H) \Rightarrow P(H|D) \times P(D) = P(D|H) \times P(H)$$

$P(H|D) = P(D|H) \times P(H) \times P(D)^{-1}$ $P(D)$ is a "constant" ?

$$P(H|D) \propto P(D|H) \times P(H)$$

$$P(D|H) = P(H|D) \times P(D) \times P(H)^{-1}$$

Naive Bayes - Foundation

Consider:

$$P(C \mid F_1, F_2, \ldots, F_n)$$

Where C is the classification domain ($C = \{\text{good, bad}\}$)

F_i are the "features" (e.g., response time, number of errors, etc.)

Applying Bayes Theorem:

$$P(C \mid F_1, \ldots, F_n) = \frac{P(C) \times P(F_1,\ldots,F_n \mid C)}{P(F_1,\ldots,F_n)}$$

Joint Probability

$p(C, F_1, \ldots, F_n)$

rewritten:

$$p(C, F_1, \ldots, F_n) \propto p(C) \times p(F_1, \ldots, F_n \mid C)$$

$$p(C, F_1, \ldots, F_n) \propto p(C) \times p(F_1|C) \times p(F_2, \ldots, F_n \mid C, F_1)$$

$$p(C, F_1, \ldots, F_n) \propto$$
$$p(C) \times p(F_1|C) \times p(F_2|C, F_1) \times p(F_3, \ldots, F_n \mid C, F_1, F_2)$$

"Naive" assumption: Each F_i is conditionally independent of every F_j for $j \neq i$. So

$$p(F_i|C, F_j) = p(F_i|C)$$

Giving:

$$p(C|F_1, \ldots, F_n) \propto p(C) \prod_{i=1}^{n} p(F_i|C)$$

Naive Bayes Classifier

Choose c_j (i.e., $c_{nb} = c_j$) such that

$$p(c_j) \times \prod_i p(F_i|c_j)$$

is maximum - i.e.,

$$classify(f_1, \ldots, f_n) = \underset{c}{\text{argmax}}\ p(C = c) \prod_i^n p(F_i = f_i|C = c)$$

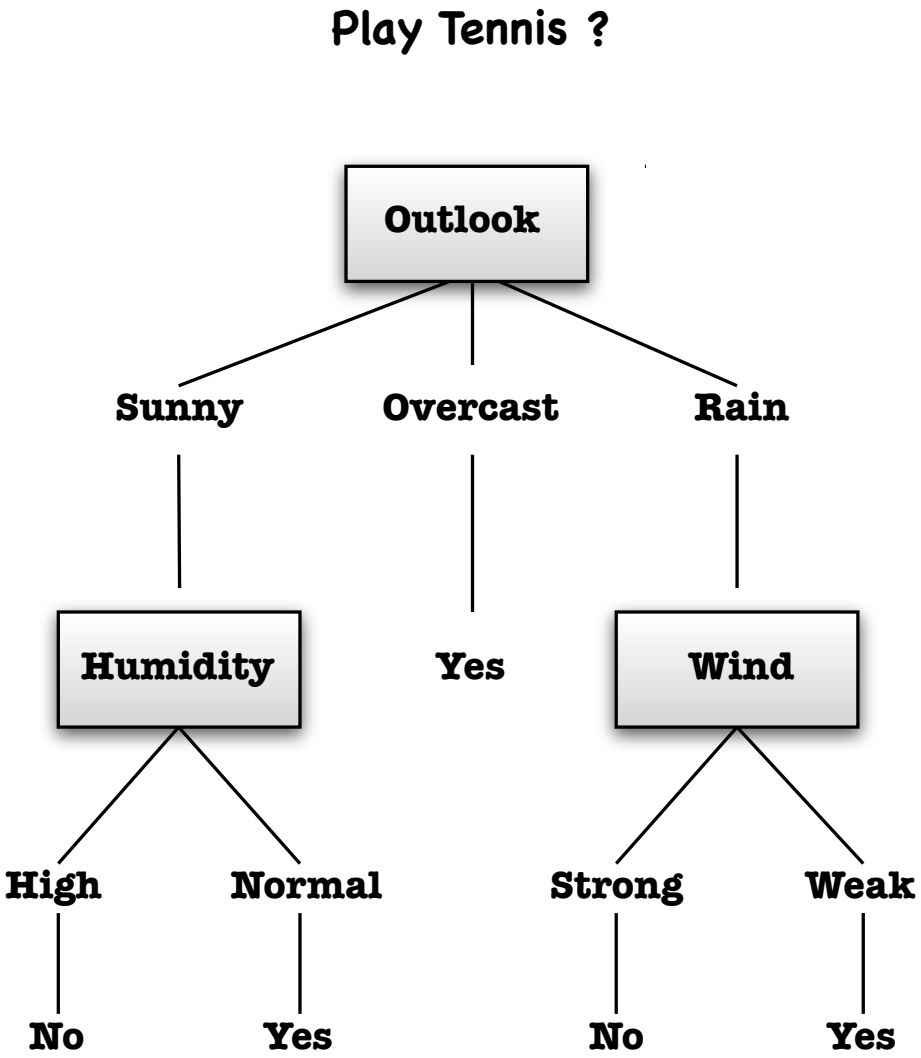
How might this be implemented?

- logarithms "convert" products to sums
- map-reduce

Naive Bayes - Summary

- Simple to understand and use
- Requires *counting* to determine prior and likelihood
- Inductive Bias - maximal conditional independence
- Tolerates noise
- Easy to overfit

ID3 Decision Tree



Information Theory

Let X be a set of $m > 1$ objects and let each object be available in $n > 1$ configurations. An example of such a set is Apple's iPod offerings: A buyer chooses a specific size and a color.

Let $I(x)$ be the amount of information associated with choosing $x \in X$. In general, the amount of information is related to the probability of choosing x: $I(x_i) = f(p(x_i))$. If there is no particular preference, $p(x_i) = \frac{1}{|X|}$.

The amount of information of selecting an object and then selecting the configuration is:

$$I(mn) = f(\frac{1}{m}) + f(\frac{1}{n})$$

The amount of information of jointly selecting the object and configuration:

$$I(mn) = f(\frac{1}{m}) \times f(\frac{1}{n})$$

Information Theory - continued

We are looking for a function, f, that has the property:

$$f(\frac{1}{m}) + f(\frac{1}{n}) = f(\frac{1}{m}) \times f(\frac{1}{n})$$

An obvious choice is logarithm, since:

$$\log(m) + log(n) = log(mn)$$

A (perhaps) less obvious choice is the number of factors to decompose $\frac{1}{x}$. For example, let $m = 8$, $n = 18$.

$m = 2 \times 2 \times 2 - - - - - - - - - - - f(m) = 3$
$n = 2 \times 3 \times 3 - - - - - - - - - - - f(n) = 3$
$mn = 2 \times 2 \times 2 \times 2 \times 3 \times 3 - - - - f(mn) = f(m) + f(n)$

We'll use logarithms!

In practice, we want to know the *information* content over a set of values. We compute this as the *expected* value:

$$H(x) = -\sum_i p(x_i) \log p(x_i)$$

Binary Entropy

Binary entropy is defined as the entropy of a Bernoulli process with probability of success p.

Let $X = 1$ be a *success* and $X = 0$ be a *failure*. These two events are mutually exclusive and exhaustive.

From above, we have $p(X = 1) = p$ and $p(X = 0) = 1 - p$ giving entropy:

$$H(X) = -p \log p - (1 - p) \log(1 - p)$$

Note: $p = 0 \Rightarrow H = 0$ and also $p = 1 \Rightarrow H = 0$. When $p = 0.5$, H is maximum.

Entropy "measures" uncertainty. When *success* or *failure* is certain, the entropy is zero; entropy is most when we are least certain of the result and we are least certain of the result when the outcome possibilities are equally likely.

Decision Tree Algorithm 1

We'll describe the ID3 algorithm[3] assuming that the target concept is boolean and all attributes are categorical.

```
ID3(Examples, Target_Attribute, Attributes)
 n = mknode()
 if Examples.All('Yes')
  n.leaf= 'Yes', return n
 if Examples.All('No')
  n.leaf= 'No', return n
 if Attributes.Empty
  n.leaf= Examples.MostCommonTargetAttributeValue()
  return n

 // otherwise, node "roots" a sub-tree
```

[3] Quinlan, 1986

Decision Tree Algorithm 2

```
// build root of sub-tree
a = BestClassifyAttribute(Examples,
        Target_Attribute, Attributes)
n.attr = a
n.branches[] = a.values() // Wind = {Strong, Weak}
for i in n.branches[]
 Example = Examples.FilterAttributeValue(a, i)
 if Example.Empty
  i.leaf = Examples.MostCommonTargetValue()
 else
  i.node = ID3(
                Example,
                Target_Attribute,
                Attributes - {a})
```

Which Attribute is Best Classifier? - 1

Entropy of S where target attribute has c distinct values:

$$H(S) = \sum_{i}^{c} -p_i \ log \ p_i$$

When target classification is boolean:

$$H(S) = -p_{\text{pos}} \ log \ p_{\text{pos}} - p_{\text{neg}} \ log \ p_{\text{neg}}$$

p_{pos} is the proportion of positive examples in S. p_{neg} is the proportion of negative examples.

Which Attribute is Best Classifier? - 2

Entropy "measures" uncertainty; the best attribute is the one that reduces entropy the most. Or, equivalently, choose the attribute that increases "information" the most.

$$\text{gain}(S, A) = H(S) - \sum_{v} \frac{|S_v|}{|S|} H(S_v)$$

Where v are the "values" (e.g., *Weak*,*Strong*) of A and S_v refers to S "restricted" to entities whose attribute A's value is v.

Which Attribute is Best Classifier? - 3

```
BestClassifyAttribute(E, t, A) :
 best_gain = 0
 best_attr = None
 for a in A:
  gain  = getGain(E, t, a)
  if  gain >= best_gain :
   best_gain = gain
   best_attr = a

 return best_attr
```

Which Attribute is Best Classifier - 4

```
getGain(E, t, a):
 value_frequency = {}
 hsv = 0
 for e in E:   // each "row"
  if value_frequency.has_key(e[a]):
   value_freq[e[a]] += 1
  else
   value_freq[e[a]] = 1 //  adds e[a] as key

 for v in value_frequency.keys():
  value_probability =
    value_freq[v] / sum(value_freq.values())
  sv = [r for r in E if r[a] == v]
  hsv += value_probability * entropy(sv,t)
```

Which Attribute is Best Classifier - 5

```
entropy(Example, target_attribute):
 val_frequency = {}
 example_entropy = 0

 // calc frequency of each value
 for r in example:  // each row
  if(val_frequency.has_key(target_attribute):
   val_frequency[r[target_attribute]] + = 1
  else:
   val_frequency[r[target_attribute]] = 1

 // H(S) = sum (-p_i log p_i)
 for f in val_frequency.values():
  example_entropy +=
      (-f/len(example)) * log(f/len(data))
```

Decision Tree Issues

Overfitting - Tree is too deep; data is noisy

To reduce problems with overfitting:

- Separate test data to evaluate "post pruning"
- Statistical test to either expand or prune a node
- Minimum Description Length principle

Information gain measure favors attributes with many values. There are many measures, such as Split Information:

$$\text{SplitInformation}(S, A) = -\sum_{i}^{c} \frac{|S_i|}{|S|} \; log \; \frac{|S_i|}{|S|}$$

which is the entropy of S with respect to A whereas previously we considered the entropy of S with respect to the target attribute.

ID3 Decision Tree - Summary

- Practical!!!!!
- Prefers smaller trees over longer ones
- Overfitting is a real concern
- Many "improvements" over basic ID3 are available

Closing Thoughts

We looked, in depth, at three different techniques for classification/approximation

- K-Nearest Neighbor
- Naive Bayes
- ID3 Decision Tree

We've reviewed the algorithms and some basic code. We've also discussed various issues with each approach.

I hope you understand much better how these classifiers work and if so disposed, have confidence that with moderate effort you can implement your own.

Discussion

Q & A