Lecture 11 - Modelling with Decision Trees

Summary

Decision trees provide a representation for classifiers that are easily understood by humans. At each node of the tree, the decision to go to the left or right branch is based on a simple predicate formed from a single feature of the data. From the data-mining perspective, decision trees not only allow us to make classifiers, but also allow us to see which features are most important. In this lecture, we'll explore how to learn decision trees from data and explore the situations in which they should be used.

Key points

What are Decision Trees?

- Flowchart-like structure
    - Each internal node represents a "test" on a feature
    - Each branch represents an outcome of the test
    - Each leaf node represents a class label
    - Path from root to leaf represents a conjunction of tests that lead to a class label
- In "Decision Tree Analysis", the trees explicitly represent decisions and decision making.
    - Often used in operational research for "decision analysis"
    - Typically trees hand crafted
- In "**Decision Tree Learning**", the tree describes data.
    - Can be used as an input to decision making
    - Uses machine learning to learn trees that can be used as predictive models
        - Both classification and regression are possible

Why model with Decision Trees?

- Key feature is **Interpretability**
    - Decision trees make the reasoning process explicit
- Lots of real-world uses:
    - Medicine
    - Finance
    - Astronomy
    - e.g. areas where there is a domain expert involved
        - Often tree is created automatically and expert will use it to understand key factors and refine to match their own beliefs (**domain knowledge**)

Learning Trees

- Lots of different algorithms (largely originating from research in the 1980s):
    - ID3 (Iterative Dichotomiser 3)

- C4.5 (ID3's successor)
- CART (Classification And Regression Trees) - kind of a catch-all term referring to many of the different algorithms
- CHAID (CHi-squared Automatic Interaction Detector)

- CART is based on a conceptually simple idea
  - Recursively split dataset by choosing a feature and a value to branch
    - For numeric features splits can be `feature >= value`
    - For categorical features split can be `feature == value`
  - Key is choosing the optimal split
    - For the tree to be useful we ideally want it to **separate** the classes as effectively and efficiently as possible. Specifically we want a split to minimise the amount of **mixing** of different classes in its two children
    - Need measures of amount of mixing (**impurity measures**)
      - **Gini Impurity**
        - a measure of how often a randomly chosen item from the set would be incorrectly labelled if it were randomly labelled according to the distribution of labels in the subset:

          $$I_G(f) = \sum_{i=1}^{m} f_i(1 - f_i) = \sum_{i \neq k} f_i f_k$$

          - if all items have the same label, Gini impurity is 0
          - if there are 4 labels with equal likelihood, Gini impurity is 0.75
      - **Entropy**
        - Measure the amount of disorder in the set:

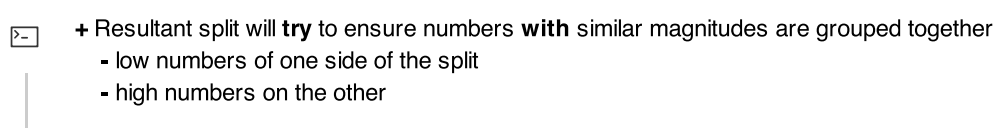
          $$I_E(f) = -\sum_{i=1}^{m} f_i \log_2 f_i$$

          - if all items have the same label, entropy is 0 bits
          - if there are 4 labels with equal likelihood, entropy is 2 bits
  - Splitting a node:
    - Given a node, N, in the tree with n items
      - Compute its impurity I(N)
      - Search for the predicate that splits the data in such that it maximises the improvement in impurity: `I(N) - ((nL/n)I(L) + (nR/n)I(R))` where `nL` and `nR` are the number of items that would fall into the left and right branches and `I(L)` and `I(R)` are the impurity of the left and right subsets formed from the branches
        - If we're using entropy, then we're trying to maximise the **information gain** from the split
  - Stopping splitting
    - Stop when the node is *completely pure* or can not be split further
      - Note: might be identical data instances with different classes
      - Leaf node is either
        - labelled with the majority class of the children data items
        - labelled with all classes, together with counts of items
  - Making classifications of new instances is simple
    - Given a data item, walk down the tree by comparing the predicate at each node against the item's features
    - When you hit a leaf node, the label of that node is the predicted class of the item
      - or the **most likely** label of that leaf

- Unfortunately the CART approach suffers from a potential problem - it has a tendency to over-fit to the data, leading to poor generalisation against new instances
    - Tends to create trees that are too complex
    - One solution to overfitting is to grow the tree fully, and then **prune** it back
        - Pruning should reduce the size of the tree
            - often without reducing predictive accuracy as measured using a *validation set*
        - Many different techniques
            - usually varying with respect to measurement used to optimise performance
    - **Reduced Error Pruning**
        - Starting at the leaves, each node is replaced with its most popular class.
        - If the prediction accuracy (tested on the validation set) is not affected then the change is kept.
            - Naive, but both simple and fast
    - Pruning based on Entropy
        - Check pairs of leaf nodes that have a common parent to see if merging them would increase entropy by less than a threshold.
            - If so, then merge nodes
        - Doesn't require additional data
- Classification algorithm can be modified to work with missing data (unknown values)
    - Follow both branches when you hit a predicate against an unknown feature
    - Weight each branch based on fraction of counts of items in order to compute which result to prefer

Dealing with numerical outcomes: Regression trees

- Can we modify how we build trees so that rather than performing classification they predict numerical outcomes and thus perform **regression**
    - Could use the same approach as for classification, but...
        - each numeric outcome would be considered to be a single class, with no regard for ordering or similarity
    - Use variance instead (e.g. best feature-value is the one that maximises variance-gain)



Problems with CART-like trees

- Learning optimal tree is known to be NP-complete!
- Generalisation/overfitting
    - hence need to prune (or use a different algorithm [with its own problems])
- Information gain is biased by features with more categories
    - Splits are performed in an axis-aligned manner...
- Might not effectively scale to large numbers of classes
- Problems learning from features that interact

Ensemble methods

- Rather than learning a single tree, learn lots of trees and combine them

- **Bagging: Bootstrap aggregating**
  - **Uniformly** sample initial dataset **with replacement** into m subsets
  - train a classifier/regressor (e.g. decision tree) for each subset
  - To perform classification apply each classifier and choose by voting (i.e. take mode)
    - For regression take mean
  - Leads to better performance – decreases variance without increasing bias
- **Boosting**
  - Can a set of weak learners create a single strong learner?
  - Learn a sequence of weak trees (i.e. fixed size trees)
    - **Gradient Tree Boosting**
- **Random Forests**
  - Applying bagging
    - but for each subset when learning the tree choose the split searching over a random sample the features rather than all of them
  - Improves bagging by **reducing overfitting**

Further Reading

- Chapter 7 of "Programming Collective Intelligence" gives a good overview of classification and regression trees with lots of examples
- Wikipedia is a good starting point to reading about general ideas and finding links to the original research papers:
  - https://en.wikipedia.org/wiki/Random_forest (https://en.wikipedia.org/wiki/Random_forest)
  - https://en.wikipedia.org/wiki/Decision_tree_learning (https://en.wikipedia.org/wiki/Decision_tree_learning)
  - https://en.wikipedia.org/wiki/Pruning_(decision_trees (https://en.wikipedia.org/wiki/Pruning_(decision_trees))
  - https://en.wikipedia.org/wiki/Gradient_boosting (https://en.wikipedia.org/wiki/Gradient_boosting)

Practical exercises

- Have a play with the demos in the slides and make sure the results you get match those you compute by hand on the same data.
- The demos in the slides have a function called `mdclassify` – this works in the same way the the `classify` function, but deals with missing data (represented by a Python `None` in the input). Read the code and have a play to understand how this works.