Lecture 13:

- $k$-nn (cont)
- Classifier evaluation (intro)

Announcements:

- HW-4 out, due Tuesday
- Project part 1 due
- Exam 1 next Thurs
- Quiz 6 Tues ($k$-nn, classifier eval from today)
The Basic \( k \)-NN algorithm

<table>
<thead>
<tr>
<th>Algorithm: ( k )-NN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> training set, instance to classify, ( k ), class column, distance columns</td>
</tr>
<tr>
<td><strong>Result:</strong> predicted class label for ( i )</td>
</tr>
</tbody>
</table>

1 \begin{align*}
\text{row-dists} &= [] \\
\text{for} \ & \text{row in training set} \ & \text{do} \\
\text{distance} &= \text{dist}(\text{row}, \text{instance}) \\
\text{row-dists} &= \text{row-dists} + [(\text{distance}, \text{row})] \\
\text{top-k} &= \text{get-top-k}(\text{row-dists}, k) \\
\text{label} &= \text{select-label}(\text{top-k}) \\
\text{return} & \text{label}
\end{align*} \\

Some Notes

Q: What happens if there are ties in the top-\( k \) distances?

- do top-\( k \) distances (instead of instances)
- randomly select from ties
- ignore the ties (skip them altogether)

Nearest doesn’t imply near

- top-\( k \) instances might not be that close to the instance being classified
- especially true as the number of attributes (“dimensions”) increases
- again, have to use common sense and an understanding of the dataset

This is an example of the “\textbf{curse of dimensionality}”

- as number of dimensions (features) increases ...
- can mean fewer instances “matching” on all dimensions
- the data becomes more sparse
Efficiency issues

Q: Is $k$-NN efficient? Can you find any efficiency issues?

• given a training set with $D$ instances and $k = 1$
• $O(D)$ comparisons needed to classify a given instance
• in other words, have to look through *entire* dataset!

Q: Can you think of any ways to improve the efficiency?

1. Use search trees
   • presort and arrange instances into a search tree
   • can reduce comparisons to $O(\log D)$
   • $kd$-trees can help here

2. Check against each training instance in parallel
   • gives $O(1)$ comparisons ... but likely not practical!
   • can still split up into smaller “chunks” of instances
   • can also use search tree

3. Editing/Pruning
   • remove “redundant” training instances (e.g., close with same label)
Training and Testing: ... for estimating classifier (algorithm) performance

Building a classifier starts with a learning (training) phase

- based on predefined set of examples – the training set

The classifier is then evaluated for predictive performance

- based on another set of examples – the testing set
- we use the actual labels (ground truth) of the examples to test the predictions

In general, we want to be careful to avoid overfitting

- encoding particular characteristics/anomalies of training set in classifier
- i.e., the model is highly specialized to the training data set
- also underfitting: too simple of a model (e.g., linear instead of polynomial)

We’ll discuss different ways to select training and testing sets

1. Holdout method
2. Random subsampling
3. $k$-fold cross validation and variants
4. Bootstrap method
Note: Some things to keep in mind ...

- Our goal is to determine how good an algorithm is for a classification problem
- ... or to try to develop a good (specialized) algorithm for the problem
- This is different than building a model for deployment
- Some ML approaches also build/update models via testing (e.g., neural nets)
- The classification approaches we’ll look at don’t “learn from their mistakes”
- But still learn (a model) from data

(1). Holdout

- randomly divide data set into a training and test set
- lots of options, e.g., partition evenly, $\frac{2}{3}$ to $\frac{1}{3}$ (2:1) training to test set, etc.
- uses random selection without replacement ... instances selected once

Q: How can we randomly select $n$ instances w/out replacement from a data table?

- easiest approach is to copy the table
- then randomly select a row and remove it from the table copy
- repeat $n$ times

(2). Random Subsampling

- repeat the holdout method $k$ times
- performance estimate is the average of the performance of each iteration