Combining k-nearest neighbor algorithm with other ML algorithms, curse of dimensionality, feature selection

Relevant Readings: None

CS495 - Machine Learning, Fall 2009

Combining k-nearest neighbors with other ML algs

- Pure k-nearest neighbors takes a majority vote of the k closest training instances

Combining k-nearest neighbors with other ML algs

- Pure k-nearest neighbors takes a majority vote of the k closest training instances
- Why use a majority vote?

Combining k-nearest neighbors with other ML algs

- Pure k-nearest neighbors takes a majority vote of the k closest training instances
- Why use a majority vote?
- A majority vote in itself might be thought of as a very primitive learning algorithm

Combining k-nearest neighbors with other ML algs

- Pure k-nearest neighbors takes a majority vote of the k closest training instances
- Why use a majority vote?
- A majority vote in itself might be thought of as a very primitive learning algorithm
- Perhaps, then, we should apply another supervised learning algorithm to the k nearest neighbors to predict future instances (neural networks, decision trees, etc.)

Combining k-nearest neighbors with other ML algs

- Pure k-nearest neighbors takes a majority vote of the k closest training instances
- Why use a majority vote?
- A majority vote in itself might be thought of as a very primitive learning algorithm
- Perhaps, then, we should apply another supervised learning algorithm to the k nearest neighbors to predict future instances (neural networks, decision trees, etc.)
- What is the effect if we try this with a kernel-based version of k-nearest neighbors?

Combining k-nearest neighbors with other ML algs

- Pure k-nearest neighbors takes a majority vote of the k closest training instances
- Why use a majority vote?
- A majority vote in itself might be thought of as a very primitive learning algorithm
- Perhaps, then, we should apply another supervised learning algorithm to the k nearest neighbors to predict future instances (neural networks, decision trees, etc.)
- What is the effect if we try this with a kernel-based version of k-nearest neighbors?

The Curse of Dimensionality

- If data instances have many features, then algorithms may tend to overfit the data by picking up on spurious correlations

The Curse of Dimensionality

- If data instances have many features, then algorithms may tend to overfit the data by picking up on spurious correlations
- This is known as the *Curse of Dimensionality* (too many dimensions is bad)

The Curse of Dimensionality

- If data instances have many features, then algorithms may tend to overfit the data by picking up on spurious correlations
- This is known as the *Curse of Dimensionality* (too many dimensions is bad)

Feature selection

- One way to combat the curse of dimensionality is by throwing some features out

Feature selection

- One way to combat the curse of dimensionality is by throwing some features out
- Question: How do we decide which ones to throw away?

Feature selection

- One way to combat the curse of dimensionality is by throwing some features out
- Question: How do we decide which ones to throw away?
- Answer: Use a tuning set on different combinations of features to find out what works best

Feature selection

- One way to combat the curse of dimensionality is by throwing some features out
- Question: How do we decide which ones to throw away?
- Answer: Use a tuning set on different combinations of features to find out what works best
- How long does it take to try all possibilities for f features?

Feature selection

- One way to combat the curse of dimensionality is by throwing some features out
- Question: How do we decide which ones to throw away?
- Answer: Use a tuning set on different combinations of features to find out what works best
- How long does it take to try all possibilities for f features?
  - 2^f trials (way too much time)

Feature selection

- One way to combat the curse of dimensionality is by throwing some features out
- Question: How do we decide which ones to throw away?
- Answer: Use a tuning set on different combinations of features to find out what works best
- How long does it take to try all possibilities for f features?
  - 2^f trials (way too much time)
- Instead, we can use hill-climbing heuristics such as *forward feature selection* and *backward feature selection*

Feature selection

- One way to combat the curse of dimensionality is by throwing some features out
- Question: How do we decide which ones to throw away?
- Answer: Use a tuning set on different combinations of features to find out what works best
- How long does it take to try all possibilities for f features?
  - 2^f trials (way too much time)
- Instead, we can use hill-climbing heuristics such as *forward feature selection* and *backward feature selection*

Forward feature selection

- Start with no features

Forward feature selection

- Start with no features
- Whichever feature maximizes accuracy on the tuning set by adding it in, add that feature in

Forward feature selection

- Start with no features
- Whichever feature maximizes accuracy on the tuning set by adding it in, add that feature in
- Repeat until no remaining feature would increase accuracy on the tuning set by adding it in

Forward feature selection

- Start with no features
- Whichever feature maximizes accuracy on the tuning set by adding it in, add that feature in
- Repeat until no remaining feature would increase accuracy on the tuning set by adding it in
- This takes $O(f^2)$ trials, where f is the number of features

Forward feature selection

- Start with no features
- Whichever feature maximizes accuracy on the tuning set by adding it in, add that feature in
- Repeat until no remaining feature would increase accuracy on the tuning set by adding it in
- This takes $O(f^2)$ trials, where f is the number of features
  - Much more efficient than 2^f trials

Forward feature selection

- Start with no features
- Whichever feature maximizes accuracy on the tuning set by adding it in, add that feature in
- Repeat until no remaining feature would increase accuracy on the tuning set by adding it in
- This takes $O(f^2)$ trials, where f is the number of features
  - Much more efficient than 2^f trials

Backward feature selection

- Start with all features

Backward feature selection

- Start with all features
- Whichever feature maximizes accuracy on the tuning set by removing it, take that feature out

Backward feature selection

- Start with all features
- Whichever feature maximizes accuracy on the tuning set by removing it, take that feature out
- Repeat until we cannot increase accuracy on the tuning set by removing another feature

Backward feature selection

- Start with all features
- Whichever feature maximizes accuracy on the tuning set by removing it, take that feature out
- Repeat until we cannot increase accuracy on the tuning set by removing another feature
- Again, this takes $O(f^2)$ trials, where f is the number of features

Backward feature selection

- Start with all features
- Whichever feature maximizes accuracy on the tuning set by removing it, take that feature out
- Repeat until we cannot increase accuracy on the tuning set by removing another feature
- Again, this takes $O(f^2)$ trials, where f is the number of features

Wrapper design pattern

- The idea is this: write an object that contains an instance of some other object, for the purpose of changing that object's interface or changing the way it acts

Wrapper design pattern

- The idea is this: write an object that contains an instance of some other object, for the purpose of changing that object's interface or changing the way it acts
- Suppose we have some existing learning algorithm implemented as an object (say, k-nearest neighbor).

Wrapper design pattern

- The idea is this: write an object that contains an instance of some other object, for the purpose of changing that object's interface or changing the way it acts
- Suppose we have some existing learning algorithm implemented as an object (say, k-nearest neighbor).
- We can write a *wrapper* around the learning algorithm to do feature selection

Wrapper design pattern

- The idea is this: write an object that contains an instance of some other object, for the purpose of changing that object's interface or changing the way it acts
- Suppose we have some existing learning algorithm implemented as an object (say, k-nearest neighbor).
- We can write a *wrapper* around the learning algorithm to do feature selection
- Then the end result is an object that adds the functionality of feature selection to whatever the learning algorithm is

Wrapper design pattern

- The idea is this: write an object that contains an instance of some other object, for the purpose of changing that object's interface or changing the way it acts
- Suppose we have some existing learning algorithm implemented as an object (say, k-nearest neighbor).
- We can write a *wrapper* around the learning algorithm to do feature selection
- Then the end result is an object that adds the functionality of feature selection to whatever the learning algorithm is
- Using interfaces or inheritence, you can set things up so that feature selection can be freely applied to any learning algorithm you have already written

Wrapper design pattern

- The idea is this: write an object that contains an instance of some other object, for the purpose of changing that object's interface or changing the way it acts
- Suppose we have some existing learning algorithm implemented as an object (say, k-nearest neighbor).
- We can write a *wrapper* around the learning algorithm to do feature selection
- Then the end result is an object that adds the functionality of feature selection to whatever the learning algorithm is
- Using interfaces or inheritence, you can set things up so that feature selection can be freely applied to any learning algorithm you have already written