Intro to non-parametric machine learning models

Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

We've been studying parametric models

- *Parametric models* have a finite number of parameters like linear model (β_i's), multinomial Naïve Bayes (P(*c*), P(*w*|*c*))
- 😜 *Nonparametric models* have huge numbers of parameters (world's worst name)
- Random Forests (RF) and gradient boosting machines are nonparametric

General advice for choosing a model

- If you know that the relationship between X and y is linear, use a linear model; or, if you need an extreme compression of the training data down to a few coefficients
- If you know the relationship is nicely summarized by conditional probabilities, Naïve Bayes approach is a good one
- For unstructured data such as images or signals, use deep learning neural networks (large number of parameters)
- For structured data like database tables or Excel spreadsheets, use decision tree-based methods: *Random Forests* (RF) or *Gradient Boosting Machines* (GBM)

Some basic modeling advice

- That “choosing model” advice is solid in practice and reduces the number of models you need to study and understand (ignore SVM, …)
- Remember: good features matter way more than the model
- Pick a decent model and then focus on feature engineering
- Know the strengths/weaknesses of your model; e.g., random forests don’t extrapolate outside of support region but parametric models tend to
- Compare your model to a weaker model
  - Sometimes simpler model (e.g., linear model) just as good
  - Gives a good lower bound on accuracy
  - Helps identify bugs in your code; e.g., when weaker model is better

Reinventing machine learning models

- Let's imagine creating model to predict SF rent prices
- What features, training data do we need? What's X and y?
- Goal: generalize from training data
- How do people do it manually?
  Find a few comparable apts and then predict average price
- That's called a *k-Nearest-Neighbor* (kNN) model & is pretty good!
  (more on this shortly)

Features				Target
bedrooms	**bathrooms**	**latitude**	**longitude**	**price**
3	1.5	40.7145	-73.9425	3000
2	1.0	40.7947	-73.9667	5465
1	1.0	40.7388	-74.0018	2850
1	1.0	40.7539	-73.9677	3275
4	1.0	40.8241	-73.9493	3350

See https://mlbook.explained.ai/intro.html

Starting with extreme models

- Recall our goal: to build an accurate model without being overly specific to training data

dict

- What if we simply memorized the training set? How could we use such a dictionary method to make predictions?

mean

- The other extreme would be to compute the average rent price from all apt data, ignoring all features, and make that our sole prediction
- How would you describe the differences / tradeoffs between them?
  - Dictionary has no bias (very accurate) but is not general (only works for training data)
  - Average has big bias but is very general (applies to any apartment)
- *Bias-generality tradeoff (Often called Bias-variance trade-off)*

overfitting = not general

Dealing with uncertainty in target (prices)

- Aside from overfitting, what's wrong with the dictionary method?
- It can't handle multiple prices for identical apt feature vectors
- But, prices fluctuate from noise, errors, or exogenous features like square footage, view, proximity to BART, etc…
- Which/what price should our decent model return for data below?
- Merge identical records, recording mean(y) for prototype

	bedrooms	bathrooms	latitude	longitude	price
1470	0	1.0000	40.7073	-73.9664	2650
36509	0	1.0000	40.7073	-73.9664	2850
39241	0	1.0000	40.7073	-73.9664	2950
46405	0	1.0000	40.7073	-73.9664	2850

Dealing with inexact feature matches

- Dictionaries are super rigid: they can't deal with mismatched keys
- Feature vectors not found in the training data dictionary will get a "key not found" error, rather than a prediction!
- How can we predict prices for inexact matches?
- Scan all apartment records, find the closest match
- Or, find the closest *k* matches and predict the average price (this is what real estate agents do; they are called "comps")

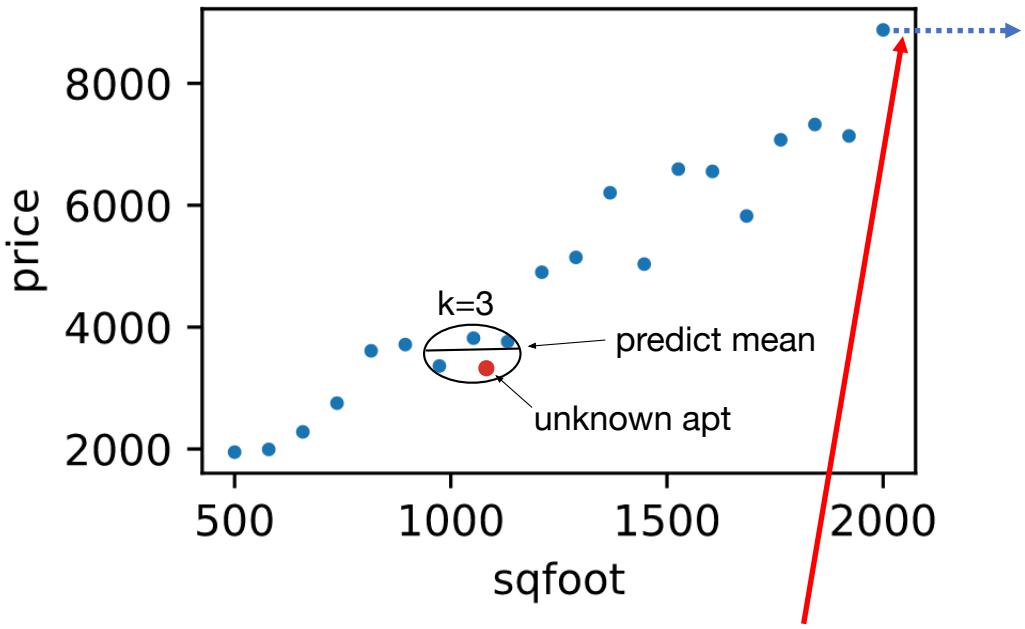
Detour: *k*-nearest neighbors (*kNN*)

- kNN is less often used in practice, but it's part of your education to understand how they work
- Regressor: get *k* observations closest to unknown *x* using Euclidean distance then predict average *y* from those *k*
- Classifier: get *k* observations closest to unknown *x* using Euclidean distance then predict most common class from those *k*
- Finding closest observations for large *N* can be slow
- Simple: there is no training process but must choose suitable *k*
- Best if we normalize data due to Euclidean distances used
- Requires distance metric, which is problematic for categoricals

See p14 in ESLII book

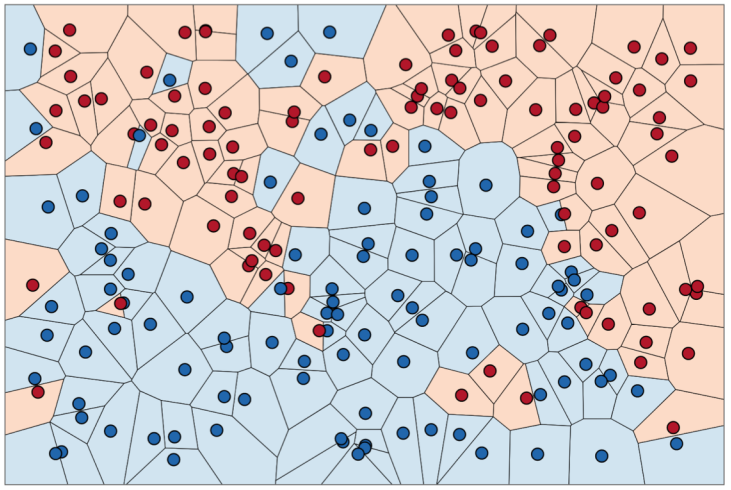
Sample kNN models

1D k=3 regressor



Note: kNN regressors can't extrapolate

2D k=1 classifier



Classifier image credit:
http://scott.fortmann-roe.com/docs/BiasVariance.html

Feature space partitioning

- To avoid inefficiency and distance metric requirement of kNN, we can partition feature space into rectangular hypervolumes
- Each hypervolume would represent a prototypical apartment with similar features
- Predictions come from average *y* (regressor) or most common class (classifier) in hypervolume
- Note similarities with kNN; hypervolumes chosen by partitioning rather than Euclidean distance
- No distance computation means:
  - No need to normalize data
  - Can easily partition (nominal/ordinal) categorical variables

Example partitioning rules

- The goal is to split each feature into as many ranges as necessary to get accuracy but w/o creating so many tight regions we kill generality by overfitting to training data
- Rules for partitioning might look like:

```
if bedrooms==1 and bathrooms==1.0 and \
   latitude>=40.6661 and latitude<=40.6663 and \
   longitude>=-73.9882 and longitude<=-73.9402:
       price = 2143 # average of 2200,2100,2100,2200,2100,1800,1800
if bedrooms==2 and bathrooms==1.0 and \
   latitude>=40.6661 and latitude<=40.6663 and \
   longitude>=-73.9882 and longitude<=-73.9402:
       price = 2462 # average of 2500,2500,2500,2350
```

Predict by testing rules until we find a match and get a price

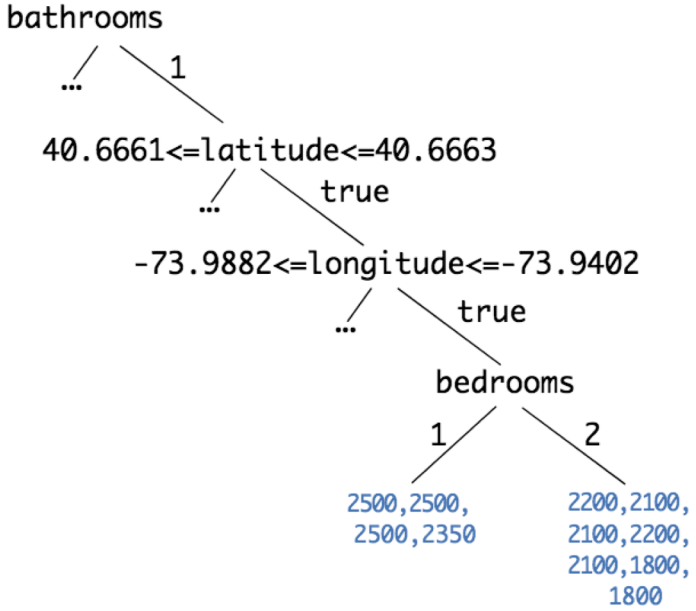
Partitioning rules prediction efficiency

- Unlike a dictionary, partitioning rules automatically:
  - handle multiple identical apartments with different prices
  - can make predictions for previously unseen feature combinations
- The number of feature ranges or "splits" tested by the model are a kind of a bias-generality "knob" we can turn up or down
- Potentially very slow walking through a large number of partitioning rules so factor / nest the IF-rules to avoid redundant tests

```
if bathrooms==1.0:
    if latitude>=40.6661 and latitude<=40.6663 and \
       longitude>=-73.9882 and longitude<=-73.9402:
        if bedrooms==1: price = 2143
        elif bedrooms==2: price = 2462
    if latitude>=40.6661 and latitude<=40.6663 and …
```

Encoding partitioning rules as a tree

- Can encode those nested rules as tree data structure
- Each node performs feature comparison, leaves make predictions
- Leaves contain prices for all apts fitting criteria on path from root down to that leaf
- Leaves form rectangular feature-hypervolume
- These are called *decision trees*
- By testing same feature many times, can carve up feature space arbitrarily tightly
- Training finds feature & value to test in each decision node (and when to stop splitting feature space)



The problem with decision trees

- Decision trees overfit like crazy to the training data
- By default, they split feature space until each leaf has a single observation (apartment in this case); that is precise like dictionary but does not generalize very well
- We can control overfitting partially by requiring a minimum number of observations for leaf
- A single-node decision tree degenerates to an extreme model predicting mean

Randomness is your friend

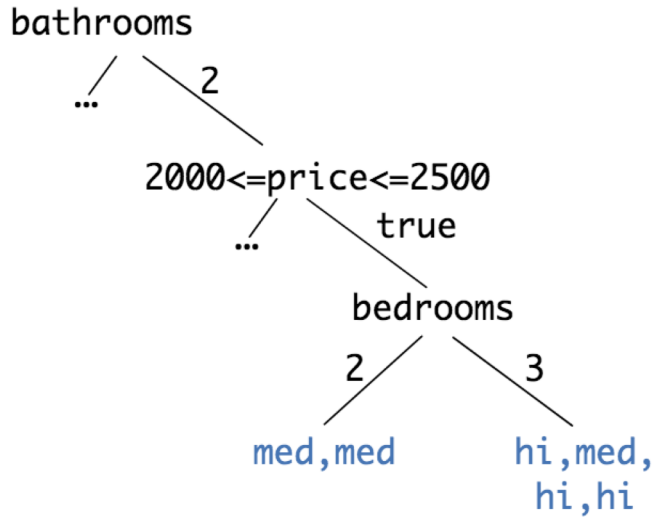
- To prevent overfitting, weaken decision tree by showing it a random subset of the training data (*bootstrapping*)
- *Bootstrapping*: from *n* records, randomly select *n* w/replacement
- To go further, degrade training so that it always forgets that some features exist when making splitting decisions
- Such individual decision trees are weaker and less accurate than regular decision trees but are more general

Random Forest (RF) regressors

- To compensate for weaker learners, create lots of them
- Take the average of their predictions to get overall prediction
- This is called *ensemble learning* and is excellent technique to increase accuracy without a tendency to overfit
- RFs are crowd-sourcers; analogous to group of real estate agents looking for comparable apartments, and cooperating to estimate apartment price
- During training, agents independently select and visit apts
- Randomize to avoid visiting, say, only 1-bedroom apts
- Agents find different apt subsets with some overlap

RF classifiers

- RFs can predict classes too but take a majority vote among the decision tree classifiers, rather than predicting average y value
- Each decision tree classifier leaf predicts the most common category from observations in that leaf
- Example classifier: predicting website interest in apartments (low, medium, hi)
- (full lecture on RFs soon)



Key takeaways

- Parametric vs nonparametric models (few vs many parameters)
- Default model choice for structured data: RF or GBM
- Feature engineering much more important than the model
- Bias-generality tradeoff
- kNN: Find k comparable feature vectors and then predict average y (regressor) or most common y (classifier); tesselates feature space
- Decision trees: partition feature space into rectangular hypervolumes; predict average/most common y in volume
- Random Forest: collection of decision trees trained on subset of training data and sometimes ignoring features; avg or majority vote among trees