Going nonparametric:
Nearest neighbor methods for regression and classification

Locality sensitive hashing for approximate NN search...cont’d
Complexity of brute-force search

Given a query point, scan through each point
- $O(N)$ distance computations per 1-NN query!
- $O(N \log k)$ per $k$-NN query!

What if $N$ is huge???
(and many queries)

Moving away from exact NN search

• Approximate neighbor finding...
  - Don’t find exact neighbor, but that’s okay for many applications
    
    Out of millions of articles, do we need the closest article or just one that’s pretty similar?
    Do we even fully trust our measure of similarity???

• Focus on methods that provide good probabilistic guarantees on approximation
Using score for NN search

**2D Data** | **Sign(Score)** | **Bin index**
---|---|---
$x_1 = [0, 5]$ | -1 | 0
$x_2 = [1, 3]$ | -1 | 0
$x_3 = [3, 0]$ | 1 | 1
... | ... | ...

Only search here for queries with Score($x$)$>0$

Only search here for queries with Score($x$)$<0$

candidate neighbors if Score($x$)$<0$

**Query point $x$**

**1.0 #awesome - 1.5 #awful = 0**

**Using score for NN search**

**2D Data** | **Sign(Score)** | **Bin index**
---|---|---
$x_1 = [0, 5]$ | -1 | 0
$x_2 = [1, 3]$ | -1 | 0
$x_3 = [3, 0]$ | 1 | 1
... | ... | ...

**Bin**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>List containing indices of datapoints:</td>
<td>{1,2,4,7,...}</td>
</tr>
</tbody>
</table>

**HASH TABLE**

search for NN amongst this set
Three potential issues with simple approach

1. **Challenging to find good line**
2. **Poor quality solution:**
   - Points close together get split into separate bins
3. **Large computational cost:**
   - Bins might contain many points, so still searching over large set for each NN query

How to define the line?

**Crazy idea:**
Define line randomly!
How bad can a random line be?

**Goal:** If \( x,y \) are close (according to cosine similarity), want binned values to be the same.

- Bins are the same
- Bins are different

If \( \theta_{xy} \) is small \((x,y \text{ close})\), unlikely to be placed into separate bins

Three potential issues with simple approach

1. **Challenging to find good line**
2. **Poor quality solution:**
   - Points close together get split into separate bins
3. **Large computational cost:**
   - Bins might contain many points, so still searching over large set for each NN query

<table>
<thead>
<tr>
<th>Bin</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>List containing indices of datapoints:</td>
<td>( {1,2,4,7,\ldots} )</td>
<td>( {3,5,6,8,\ldots} )</td>
</tr>
</tbody>
</table>
Improving efficiency:
Reducing # points examined per query

Reducing search cost through **more bins**
Using score for NN search

<table>
<thead>
<tr>
<th>2D Data</th>
<th>Sign (Score$_1$)</th>
<th>Bin 1 index</th>
<th>Sign (Score$_2$)</th>
<th>Bin 2 index</th>
<th>Sign (Score$_3$)</th>
<th>Bin 3 index</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 = [0, 5]$</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>$x_2 = [1, 3]$</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>$x_3 = [3, 0]$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Data indices: \{1,2\} \rightarrow \{4,8,11\} \rightarrow \{7,9,10\} \rightarrow \{3,5,6\}

Improving search quality by searching neighboring bins

Query point here, but is NN? Not necessarily

Even worse than before...Each line can split pts. Sacrificing accuracy for speed
Improving search quality by searching neighboring bins

<table>
<thead>
<tr>
<th>Bin</th>
<th>[0 0 0] = 0</th>
<th>[0 0 1] = 1</th>
<th>[0 1 0] = 2</th>
<th>[0 1 1] = 3</th>
<th>[1 0 0] = 4</th>
<th>[1 0 1] = 5</th>
<th>[1 1 0] = 6</th>
<th>[1 1 1] = 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data indices:</td>
<td>(1,2)</td>
<td>--</td>
<td>(4,8,11)</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>(7,9,10)</td>
<td>(3,5,6)</td>
</tr>
</tbody>
</table>

Next closest bins (flip 1 bit)

Further bin (flip 2 bits)
Improving search quality by searching neighboring bins

<table>
<thead>
<tr>
<th>Bin</th>
<th>[0 0 0]</th>
<th>[0 0 1]</th>
<th>[0 1 0]</th>
<th>[0 1 1]</th>
<th>[1 0 0]</th>
<th>[1 0 1]</th>
<th>[1 1 0]</th>
<th>[1 1 1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>indices</td>
<td>(1,2)</td>
<td>--</td>
<td>(4,8,11)</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>(7,9,10)</td>
<td>(3,5,6)</td>
</tr>
</tbody>
</table>

Quality of retrieved NN can only improve with searching more bins

**Algorithm:**
Continue searching until computational budget is reached or quality of NN good enough

LSH recap

- Draw $h$ random lines
- Compute “score” for each point under each line and translate to binary index
- Use $h$-bit binary vector per data point as bin index
- Create hash table
- For each query point $x$, search $\text{bin}(x)$, then neighboring bins until time limit
Moving to higher dimensions $d$

Draw random planes

Score($x$) = $v_1$ #awesome + $v_2$ #awful + $v_3$ #great
Cost of binning points in d-dim

\[ \text{Score}(x) = v_1 \#\text{awesome} + v_2 \#\text{awful} + v_3 \#\text{great} \]

Per data point, need \( d \) multiplies to determine bin index per plane

One-time cost offset if many queries of fixed dataset

Summary for retrieval using nearest neighbors and locality sensitive hashing
What you can do now...

- Implement nearest neighbor search for retrieval tasks
- Contrast document representations (e.g., raw word counts, tf-idf,...)
  - Emphasize important words using tf-idf
- Contrast methods for measuring similarity between two documents
  - Euclidean vs. weighted Euclidean
  - Cosine similarity vs. similarity via unnormalized inner product
- Describe complexity of brute force search
- Implement LSH for approximate nearest neighbor search

Motivating NN methods for regression: Fit globally vs. fit locally
Parametric models of $f(x)$

$y$ vs. price ($) vs. sq.ft.

Parametric models of $f(x)$

$y$ vs. price ($) vs. sq.ft.
Parametric models of $f(x)$

$y$

price ($) vs sq.ft.

$x$

Parametric models of $f(x)$

$y$

price ($) vs sq.ft.

$x$
f(x) is not really a polynomial

What alternative do we have?

If we:
- Want to allow flexibility in f(x) having local structure
- Don’t want to infer “structural breaks”

What’s a simple option we have?
- Assuming we have plenty of data...
Simplest approach: Nearest neighbor \textit{regression}

Fit locally to each data point

Predicted value = “closest” $y_i$

1 nearest neighbor (1-NN) regression
What people do naturally...

Real estate agent assesses value by finding sale of most similar house

$ = ???$

$ = 850k$

1-NN regression more formally

Dataset of $(\text{sq.ft.}, \text{price})$ pairs: $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$

Query point: $x_q$

1. Find “closest” $x_i$ in dataset

2. Predict
Visualizing 1-NN in multiple dimensions

Voronoi tessellation (or diagram):
- Divide space into N regions, each containing 1 datapoint
- Defined such that any \( x \) in region is “closest” to region’s datapoint

Don’t explicitly form!

Different distance metrics lead to different predictive surfaces

Euclidean distance

Manhattan distance
Can 1-NN be used for classification?

Yes!!

Just predict class of neighbor

1-NN algorithm
Performing 1-NN search

- **Query house:**

- **Dataset:**

- **Specify:** Distance metric
- **Output:** Most similar house

1-NN algorithm

Initialize $\text{Dist2NN} = \infty$, $\hat{i} = \emptyset$

For $i=1,2,...,N$

Compute: $\delta = \text{distance}(\hat{x}_i, q)$

If $\delta < \text{Dist2NN}$

set $\hat{i} = i$

set $\text{Dist2NN} = \delta$

Return **most similar house**
1-NN in practice

Fit looks good for data dense in x and low noise

Sensitive to regions with little data

Not great at interpolating over large regions...
Also sensitive to noise in data

Fits can look quite wild... Overfitting?

k-Nearest neighbors
Get more “comps”

More reliable estimate if you base estimate off of a larger set of comparable homes

$k$-NN regression more formally

Dataset of $(\mathbf{x}, \$)$ pairs: $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$

Query point: $x_q$

1. Find $k$ closest $x_i$ in dataset

2. Predict
Performing k-NN search

• Query house:

• Dataset:

• **Specify:** Distance metric
• **Output:** Most similar houses

---

**k-NN algorithm**

Initialize $\text{Dist2kNN} = \text{sort}(\delta_1, ..., \delta_k)$

For $i = k+1, ..., N$

Compute: $\delta = \text{distance}(\text{house}_i, \text{query house}_q)$

If $\delta < \text{Dist2kNN}[k]$

find $j$ such that $\delta > \text{Dist2kNN}[j-1]$ but $\delta < \text{Dist2kNN}[j]$

remove furthest house and shift queue:

$\text{Dist2kNN}[j+1:k] = \text{Dist2kNN}[j:k-1]$

set $\text{Dist2kNN}[j] = \delta$ and $\text{house}[j] = \text{house}_i$

Return $k$ most similar houses
k-NN in practice

Much more reasonable fit in the presence of noise

Boundary & sparse region issues

Discontinuities! Neighbor either in or out
Issues with discontinuities

Overall predictive accuracy might be okay, but...

For example, in housing application:
- If you are a buyer or seller, this matters
- Can be a jump in estimated value of house going just from 2640 sq.ft. to 2641 sq.ft.
- Don't really believe this type of fit

Weighted k-nearest neighbors
Weighted k-NN

Weigh more similar houses more than those less similar in list of k-NN

Predict:

\[ \hat{y}_q = \frac{c_{q\text{NN}1}y_{\text{NN}1} + c_{q\text{NN}2}y_{\text{NN}2} + c_{q\text{NN}3}y_{\text{NN}3} + \ldots + c_{q\text{NN}k}y_{\text{NN}k}}{\sum_{j=1}^{k} c_{q\text{NN}j}} \]

How to define weights?

Want weight \( c_{q\text{NN}j} \) to be small when \( \text{distance}(x_{\text{NN}j}, x_q) \) large

and \( c_{q\text{NN}j} \) to be large when \( \text{distance}(x_{\text{NN}j}, x_q) \) small
Kernel weights for $d=1$

Define: $c_{qNNj} = \text{Kernel}_\lambda(|x_{NNj} - x_q|)$

Gaussian kernel:
$\text{Kernel}_\lambda(|x_i - x_q|) = \exp\left(-\frac{(x_i - x_q)^2}{\lambda}\right)$

Note: never exactly 0!

Kernel weights for $d \geq 1$

Define: $c_{qNNj} = \text{Kernel}_\lambda(\text{distance}(x_{NNj}, x_q))$
Kernel regression

Weighted k-NN

Weigh more similar houses more than those less similar in list of k-NN

Predict:

\[ \hat{y}_q = \frac{c_{qNN1}y_{NN1} + c_{qNN2}y_{NN2} + c_{qNN3}y_{NN3} + \ldots + c_{qNNk}y_{NNk}}{\sum_{j=1}^{k} c_{qNNj}} \]
Kernel regression

Instead of just weighting NN, weight all points

Predict:

\[
\hat{y}_q = \frac{\sum_{i=1}^{N} c_q y_i}{\sum_{i=1}^{N} c_i} = \frac{\sum_{i=1}^{N} \text{Kernel}_\lambda(\text{distance}(x_i, x_q)) \cdot y_i}{\sum_{i=1}^{N} \text{Kernel}_\lambda(\text{distance}(x_i, x_q))}
\]

Nadaraya-Watson kernel weighted average

Kernel regression in practice

Kernel has bounded support...
Only subset of data needed to compute local fit
Choice of bandwidth $\lambda$

Often, choice of kernel matters much less than choice of $\lambda$

Choosing $\lambda$ (or k in k-NN)

How to choose? Same story as always...

Cross Validation
Formalizing the idea of local fits

Contrasting with global average

A **globally constant fit** weights all points equally

\[ \hat{y}_q = \frac{1}{N} \sum_{i=1}^{N} c y_i \]

equal weight on each datapoint
Contrasting with global average

Kernel regression leads to **locally constant fit**
- slowly add in some points and let others gradually die off

\[
\hat{y}_q = \frac{\sum_{i=1}^{N} \text{Kernel}_\lambda(\text{distance}(x_i, x_q)) \cdot y_i}{\sum_{i=1}^{N} \text{Kernel}_\lambda(\text{distance}(x_i, x_q))}
\]

Local linear regression

So far, fitting **constant function locally** at each point
→ “locally weighted averages”

Can instead fit a **line or polynomial locally** at each point
→ “locally weighted linear regression”
Local regression rules of thumb

- Local linear fit reduces bias at boundaries with minimum increase in variance
- Local quadratic fit doesn’t help at boundaries and increases variance, but does help capture curvature in the interior
- With sufficient data, local polynomials of odd degree dominate those of even degree

Recommended default choice: local linear regression

Discussion on k-NN and kernel regression
Nonparametric approaches

k-NN and kernel regression are examples of nonparametric regression

General goals of nonparametrics:
- Flexibility
- Make few assumptions about f(x)
- Complexity can grow with the number of observations N

Lots of other choices:
- Splines, trees, locally weighted structured regression models...

Limiting behavior of NN: Noiseless setting (ε_i=0)

In the limit of getting an infinite amount of noiseless data, the MSE of 1-NN fit goes to 0
Limiting behavior of NN: **Noiseless** setting ($\varepsilon_i=0$)

In the limit of getting an infinite amount of noiseless data, the MSE of 1-NN fit goes to 0.

![Graph showing 1-NN fit and Quadratic fit](image)

Not true for parametric models!

---

**Error vs. amount of data**

For a fixed model complexity:

- With few data points, fixed complexity model can fit three points reasonably well.
- In the limit, true error = training error.
- In the limit, model will flatten out to show how well model can fit true relationship.
- Bias + Noise.

![Graph showing error vs. amount of data](image)
Limiting behavior of NN: Noisy data setting

In the limit of getting an infinite amount of data, the MSE of NN fit goes to 0 if k grows, too

NN and kernel methods for large d or small N

NN and kernel methods work well when the data cover the space, but...
- the more dimensions d you have, the more points N you need to cover the space
- need $N = O(exp(d))$ data points for good performance

This is where parametric models become useful...
Complexity of NN search

Naïve approach: **Brute force search**
- Given a query point $x_q$
- Scan through each point $x_1, x_2, ..., x_N$
- $O(N)$ distance computations per 1-NN query!
- $O(N \log k)$ per k-NN query!

What if $N$ is huge???
(and many queries)

- KD-trees!
- Locality-sensitive hashing, etc.

---

**k-NN for classification**
Spam filtering example

Input: \( x \)  
Output: \( y \)  
Text of email, sender, IP,...

Using k-NN for classification

Space of labeled emails (not spam vs. spam), organized by similarity of text

not spam vs. spam: decide via majority vote of k-NN
Using k-NN for classification

Space of labeled emails (not spam vs. spam), organized by similarity of text

query email

not spam vs. spam

decide via majority vote of k-NN

Summary for nearest neighbor and kernel regression
What you can do now...

• Motivate the use of nearest neighbor (NN) regression
• Perform k-NN regression
• Analyze computational costs of these algorithms
• Discuss sensitivity of NN to lack of data, dimensionality, and noise
• Perform weighted k-NN and define weights using a kernel
• Define and implement kernel regression
• Describe the effect of varying the kernel bandwidth $\lambda$ or # of nearest neighbors k
• Select $\lambda$ or k using cross validation
• Compare and contrast kernel regression with a global average fit
• Define what makes an approach nonparametric and why NN and kernel regression are considered nonparametric methods
• Analyze the limiting behavior of NN regression
• Use NN for classification