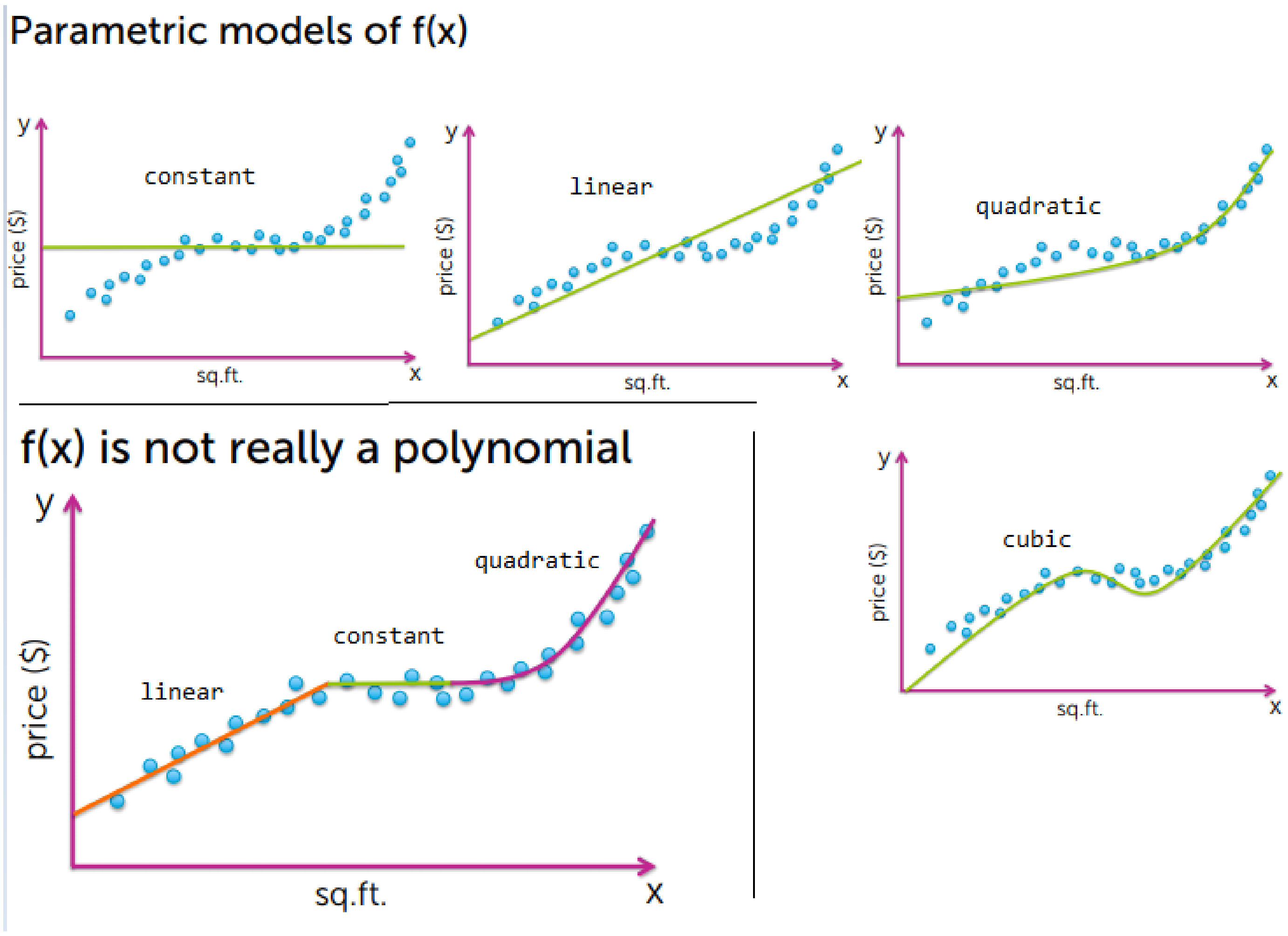
Nearest Neighbors & Kernel Regression

Motivating Local fits

Limitations of parametric regression

- So far dealt with models that have a specified set of features.
- This results in a model with fixed flexibility.
- The approaches going to be dealt in this module are termed - nonparametric approaches -> **K-Nearest Neighbours** & **Kernel Regression**.
- Provide more model flexibility. Require enough data to fit the approach.

Fitting the model 'globally' v/s 'locally'

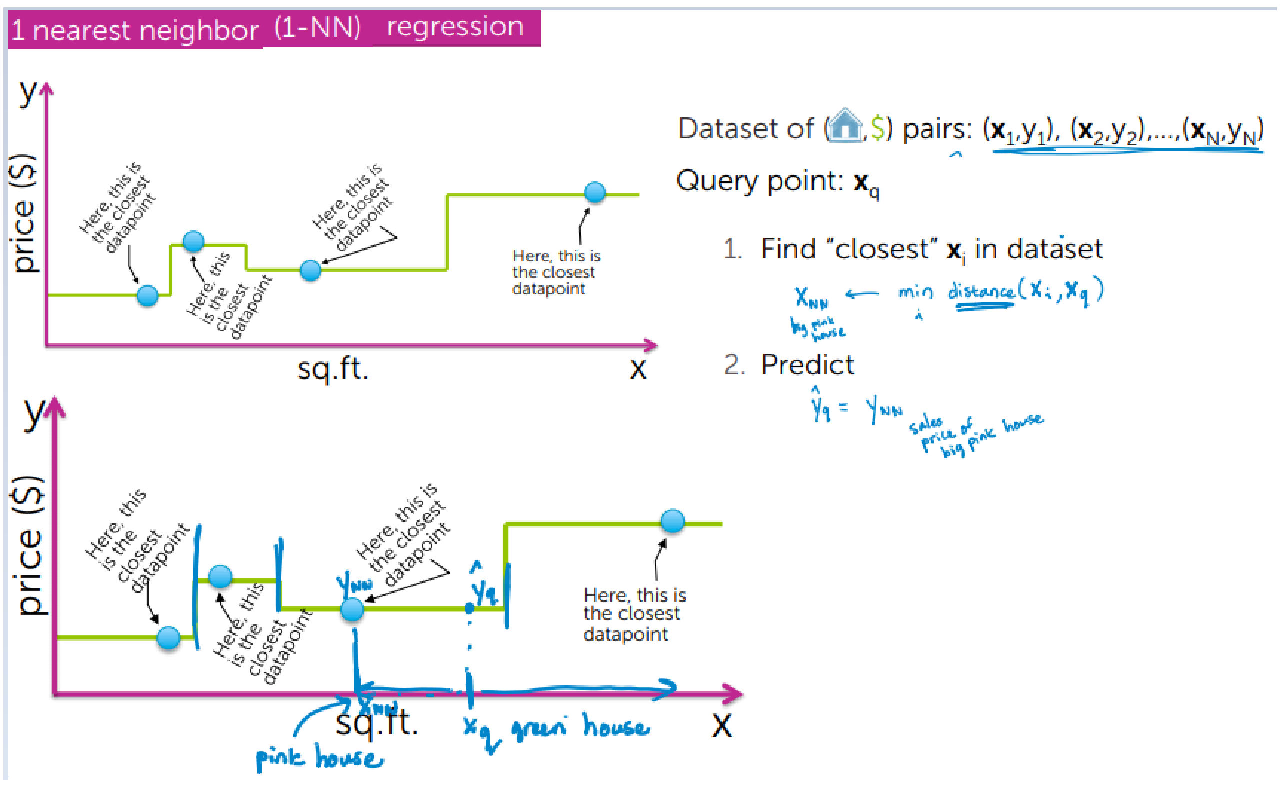


- This allows flexibility in the model - local structures. -> linear, constant, quadratic;
- But shouldn't have to infer 'structural breaks'.
- Alternative simple option -> assume we have plenty of data.

Nearest neighbor regression

- Have some set of data points.

- While predicting values at any point in the input space, need to look for the closest observation that we have and what its outputted value is and predict the your value is equally to that(closest) y/output value.
- Green line -> fit for the data.Local fits are defined around each observation.



Dataset of (🏠,$) pairs: $(\mathbf{x}_1,y_1), (\mathbf{x}_2,y_2),\ldots,(\mathbf{x}_N,y_N)$

Query point: $\mathbf{x}_q$

1. Find "closest" $\mathbf{x}_i$ in dataset

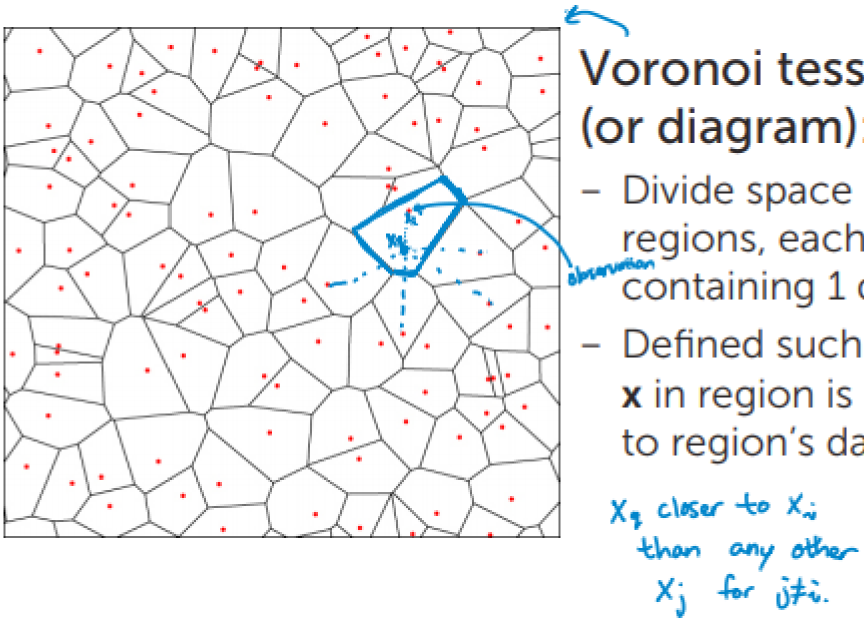
   $x_{NN} \leftarrow \min_i \text{distance}(x_i, x_q)$ big pink house

2. Predict

   $\hat{y}_q = y_{NN}$ sales price of big pink house

Visualizing 1-NN in multiple dimension

- **Voronoi tesselation** - Picture is 2-D, idea for many dimensions.
- Divide the input space into regions.
- Each region has one observation/data-point (marked in red-dot);
- Any point define in the region will be clossed to the observation defined in that point.



Voronoi tesselation (or diagram):

- Divide space into N regions, each containing 1 datapoint
- Defined such that any $\mathbf{x}$ in region is "closest" to region's datapoint

x_q closer to x_i than any other x_j for $j \neq i$.

Distance metrics

- In 1D, just Euclidean distance: **distance(x(j),x(q)) = |x(j) - (q)|**.
- In multiple dimensions:
- can define many interesting distance functions.

- most straighforwardly, might want to weight different dimensions differently.

Weighting house inputs

- Some inputs are more relevant than others.
- #bathrooms, #bedrooms, sqft living, year built features more relevant than -> #floors, year renovated, waterfront.

For multiple dimensions -> Scaled Euclidean distance

- The distance is taken between **vector of input - x(j)** and **vector of query input - x(q)**.
- Then component-wise take their squared difference.
- Scale the values by some number. (a1 -> scaling on the 1st input, aD -> scaling on the Dth input); - emphasis on the weight of different inputs.
- Sum this over all the dimensions.
- Take the square root of the above term.
  - **In case all a1 ... aD all have the same value, same scaling weight -> means all inputs have same weight = 1 and it reduces from Scaled Euclidean distance -> Euclidean distance**.

Other examples of distance metrics:

- Mahalanobis
- Rank-based
- Correlation-based
- Cosine similarity
- Manhattan
- Hamming ...

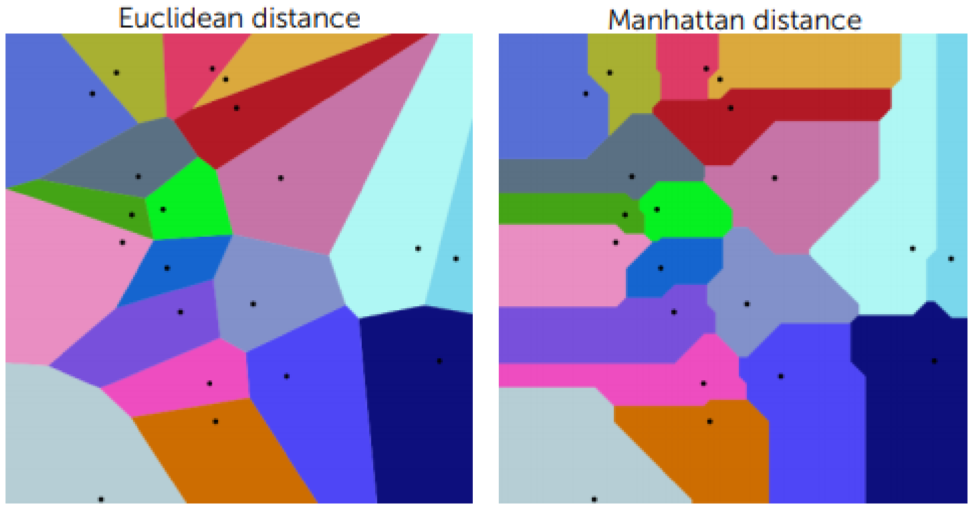
Euclidean distance: $\boxed{\text{distance}(x_j, x_q) = |x_j - x_q|}$

Scaled Euclidean distance

$$\text{distance}(\mathbf{x}_j, \mathbf{x}_q) = \sqrt{a_1(\mathbf{x}_j[1]-\mathbf{x}_q[1])^2 + \ldots + a_d(\mathbf{x}_j[d]-\mathbf{x}_q[d])^2}$$

weight on each input
(defining relative importance)

Different distance metrics lead to different predictive surfaces



1-Nearest neighbor algorithm

Performing 1-NN search

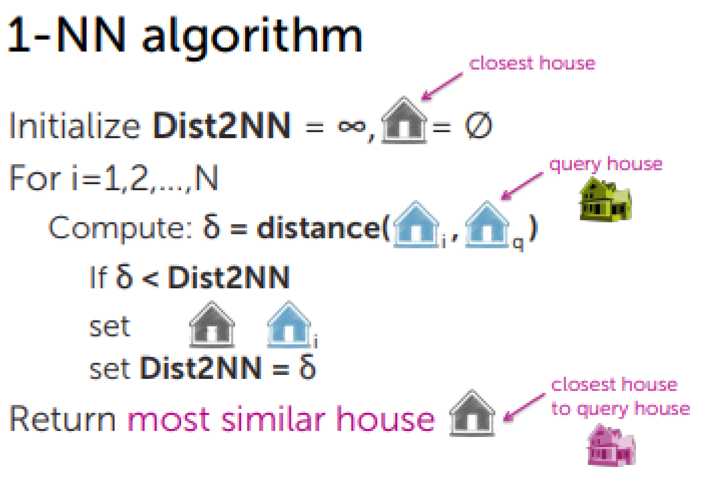
- Step 1: Query (The house to evaluate the price)
- Step 2: Dataset (Provide the training dataset)
- Step 3: Specify the Distance metrics (Euclidean, hamming, manhattan,etc)
- Step 4: Output (Similar houses)

1-NN algorithm

- Step 1: Intialize Dist2NN (Distance to nearest neighbout) = infinity, Closest datapoint(house) = null set;
- Step 2 : For all observations
  - compute the distance-**d'** for the particular observation w.r.t query datapoint.
  - If **d'** > Dist2NN

    ```
    # set Closest datapoint = particular observation.
    # set Dist2NN = d'
    ```

- Final step : Return most similar datapoints / houses.



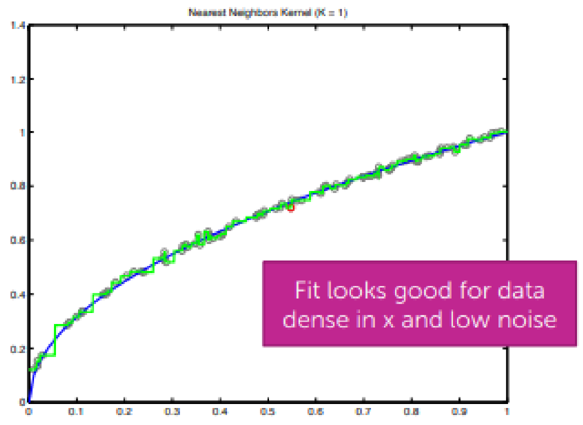
1-NN algorithm

```
Initialize Dist2NN = ∞,  = ∅
For i=1,2,...,N
    Compute: δ = distance( i ,  q )
        If δ < Dist2NN
        set    i
        set Dist2NN = δ
Return most similar house 
```

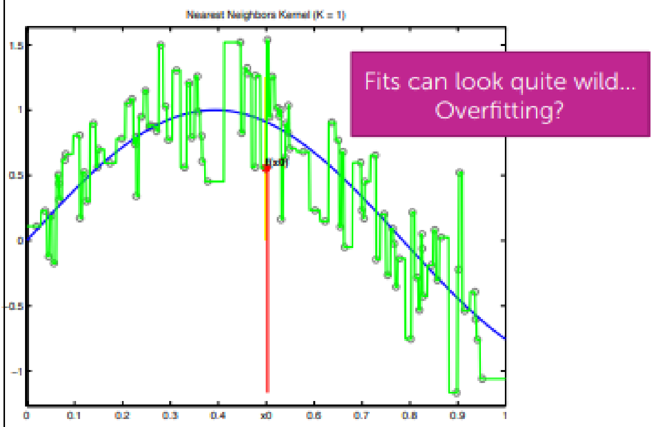
closest house
query house
closest house to query house

1-NN in practice:

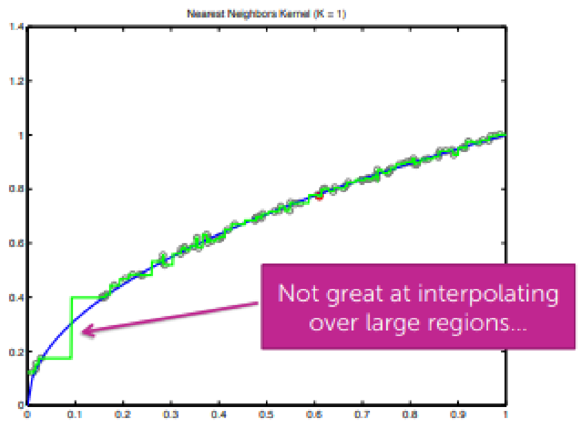
- It works well, predicts well for a dense set of data.
- Works poorly in case of - of regions with little data.
- It is also sensitive to noise in data.



sensitive to noise in data



Sensitive to regions with little data



k-Nearest neighbors

- Dataset of all data-point pairs (house, price) pairs : (x1,y1), (x2,y2), ... , (xN, yN);
- Query point : x(q)
  - Find the closest x(i) in the dataset.

    ```
    * (Xnn1, Xnn2, ..., Xnnk) -> such that for any Xi not in the nearest
     neighbour set, distance(Xi,Xq) >= distance (Xnnk, Xq);
    ```

  - Predict : Take the average over all predicted output. y-hat(q) = 1/k (Ynn1 + Ynn2 + ... + Ynnk) = 1/k Σ(j=1 to k) Ynnj;

Performing k-NN search

- Query datapoint (house)
- Dataset : training set.
- Specify : Distance metrics.
- Output : Most similar houses.

k-NN algorithm

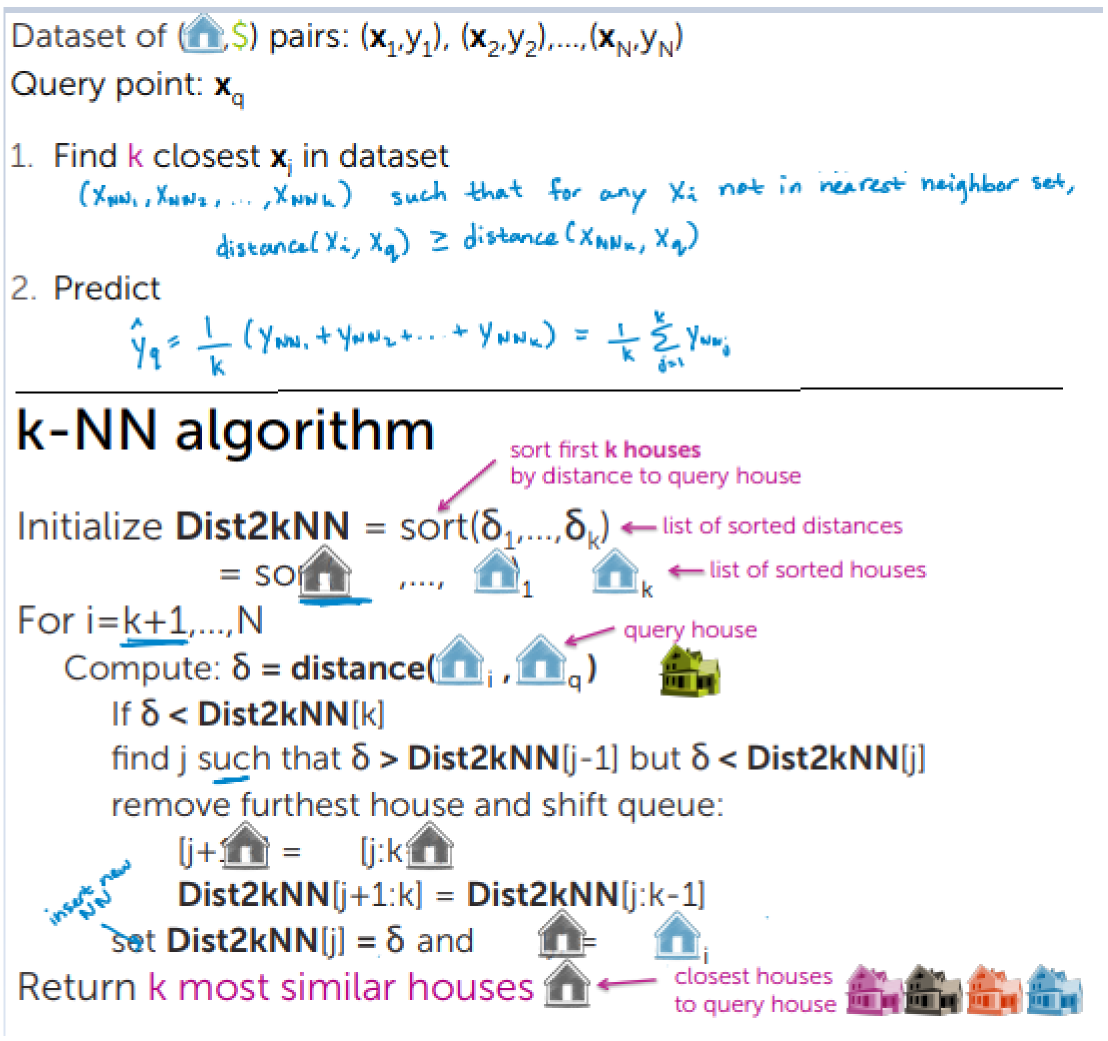
- Step 1 : Initialize Dist2kNN = sort(d1,....,dk) - sort the first k houses by distance to query house. Sort the list of houses.
- Step 2 : For observations i from k+1 -> N (Since 1-> k are used in the initialization)
  - 2.2 Compute distance d' = distance(observation, query)
  - 2.3 if d' < Dist2kNN[k]

- 2.4 find j such that Dist2kNN[j] > d' > Dist2kNN[j-1] ; (Need to insert the new d' among the selected distance in the sorted order).
- 2.5 remove the furthest house and shift queue:

```
[j+1:k] house = [j,k-1] house
Dist2kNN [j+1:k] = Dist2kNN [j:k-1]
```

- 2.6 (insert new NN) Dist2kNN[j] = d' and house/closest = house(i),

* Step Final : Return k most similar houses.
* Prediction : Take the average over all the estimated outputs.



Dataset of (🏠,$) pairs: $(\mathbf{x}_1,y_1)$, $(\mathbf{x}_2,y_2)$,…,$(\mathbf{x}_N,y_N)$

Query point: $\mathbf{x}_q$

1. Find k closest $\mathbf{x}_i$ in dataset

   $(x_{NN_1}, x_{NN_2}, \dots, x_{NN_k})$ such that for any x_i not in nearest neighbor set, $\text{distance}(x_i, x_q) \geq \text{distance}(x_{NN_k}, x_q)$

2. Predict

   $$\hat{y}_q = \frac{1}{k}(y_{NN_1} + y_{NN_2} + \dots + y_{NN_k}) = \frac{1}{k}\sum_{j=1}^{k} y_{NN_j}$$

k-NN algorithm

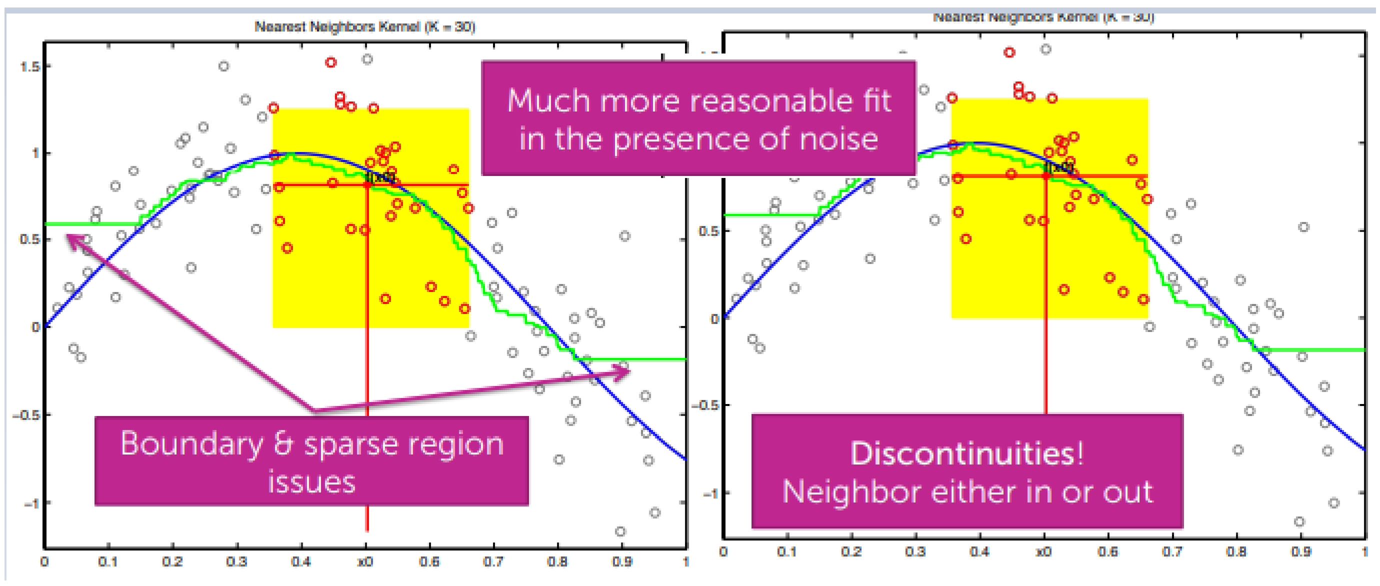
```
Initialize Dist2kNN = sort(δ_1,…,δ_k)   ← list of sorted distances      (sort first k houses by distance to query house)
                    = sort(🏠,…, 🏠_1 … 🏠_k)   ← list of sorted houses
For i=k+1,…,N
    Compute: δ = distance(🏠_i, 🏠_q)   ← query house
        If δ < Dist2kNN[k]
        find j such that δ > Dist2kNN[j-1] but δ < Dist2kNN[j]
        remove furthest house and shift queue:
            [j+1:k] 🏠 = [j:k-1] 🏠
            Dist2kNN[j+1:k] = Dist2kNN[j:k-1]
        set Dist2kNN[j] = δ and 🏠[j] = 🏠_i      (insert new NN)
Return k most similar houses 🏠   ← closest houses to query house
```

k-NN in practice

- Yellow box -> nearest neighbours for a specific point, the cross in red. The nearest neighbours to cross point are marked in red against the grey points.
- Averaging all the red points / nearest neighbors will provide the green line at that point.
- This process can be repeated for all the inputs - and the average of all the nearest neighbors will produce the green line as shown in the diagram.
- It is a good fit even in the presence of noise, unlike the 1-NN.
- **Issues :**

1. Boundary and sparse regions the fit is not perfect or does not model the dataset. The reason in particular at the boundary is due to the constant fits since the nearest neighbors are exactly the same set of points for all the other different input points.At the boundary, for close points the nearest neighbours tend to remain the same expect the query point. Hence the constant fit.

2. Discontinuity -> The fit on the model - green line - is not a clean curve, but is interspersed with jumps along its length.The reason for these jumps is the fact that as you shift from one input to the next input, a nearest neighbor is either completely in or out of the window. therefore the nearest neighbor changes and results in a jump in the predicted value.



Weighted k-nearest neighbors

- It weighs more similar points more than those less similar in the list of k-NN. It down-weights the points farther away from the query point/target point.
- The predicted value is the sum of the product of the weight of the point and the prediction at the point divided by weights for all the k terms. Average of the value of the predictions and their respective magnitude.

How the weights are defined?

- In case the distance(X(NNj), Xq) is large - then CqNNj (weight) is to be small.
- In case the distance(X(NNj), Xq) is small - then CqNNj (weight) is to be large. (More similarity);
- **CqNNj (weight) = 1/ distance(X(NNj), Xq)**.

Kernel weights

Isotropic kernels - these kernels are a function of the distance between any point and the target point/query.

- The kernel showns how the weights are gonna decay as a function of the distance between a given point and the query point.
- "lamda" parameter - defines how quickly the fuction decays as a function of the distance.
- For the Gaussian kernel the value never goes to 0.

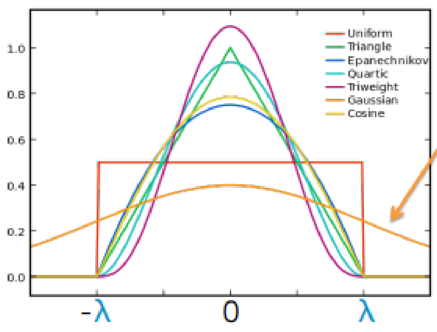
Weighted k-NN

Predict:

weights on NN

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

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



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

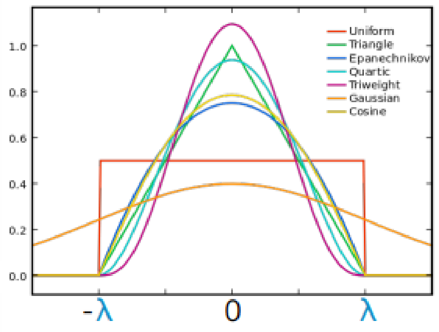
Note: never exactly 0!

Simple method:

$$c_{qNN_j} = \frac{1}{\text{distance}(x_j, x_q)}$$

Kernel weights for d≥1

Define: $c_{qNNj} = \text{Kernel}_\lambda(\text{distance}(\mathbf{x}_{NNj}, \mathbf{x}_q))$



Kernel Regression

- In weighed k-NN , only the nearest neighbors were weighed.
- In **Kernel regression** all points are weighed in the dataset.
- Prediction y-hat = sum of all kernel distance between xi and xq * prediction i divided by the sum of all kernel distances.

Kernel regression in practice

- The yellow curved region - represents the kernel choosen. - e.g **Epanechnikov kernel**.
- The are some observations highlighted in red for a given target point in this case (x,0).
- In k-NN there are fixed number of neighbors. Here instead a kernel is choosen and a **bandwidth** is given - **lambda**. In this bandwidth/lambda region the observations are included. Outside the fit the observations are completely discarded.
- At every target point (x,0) in the input space, the observation is weighed by the kernel. So the weighed average is computed -> **predicted value**.
- The green line is the resultant fit. It is a more smoother curve than a standard k-NN.

Impending questions with the kernel regession:

1. Which kernel is to be employed?
2. For a given kernel, what should the choice of bandwidth/lambda be?

Choice of bandwidth - lambda

1. lambda = 0.04 - small, fit model - overfit; -> Low bias & high variance. (changing the observation - change the fit);
2. lambda = 0.2 - moderate, reasonable fit - bias-variance trade-off.
3. lambda = 0.4 - high, fit model - oversmoothed fit; -> Low variance & high bias.

Kernel regression Nadaraya-Watson kernel weighted average

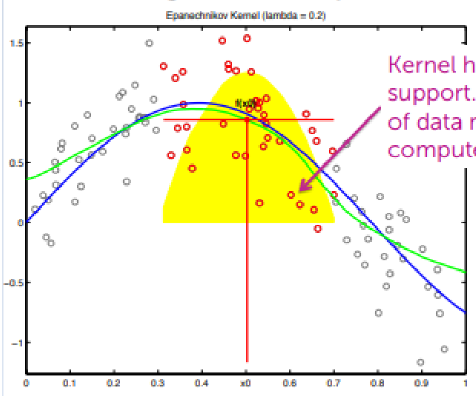
Instead of just weighting NN, weight *all* points

Predict:

weight on each datapoint

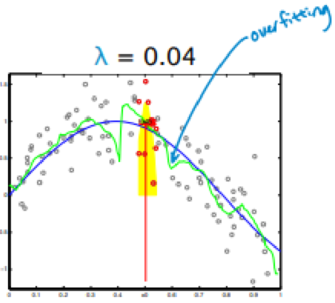
$$\hat{y}_q = \frac{\sum_{i=1}^{N} c_{qi} y_i}{\sum_{i=1}^{N} c_{qi}} = \frac{\sum_{i=1}^{N} \text{Kernel}_\lambda(\text{distance}(\mathbf{x}_i,\mathbf{x}_q)) * y_i}{\sum_{i=1}^{N} \text{Kernel}_\lambda(\text{distance}(\mathbf{x}_i,\mathbf{x}_q))}$$

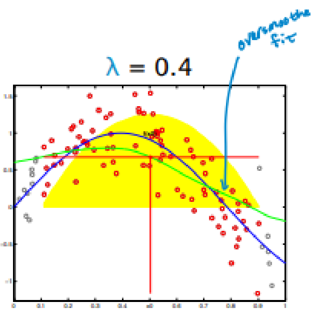
Kernel regression in practice

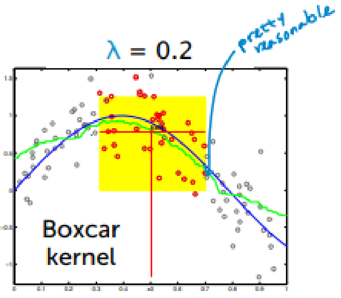


Choice of bandwidth λ

Often, choice of kernel matters much less than choice of λ







Choice lambda (or k in k-NN)

- Cross Validation.
- Or use the validation set in case of insufficient data.

Global fits of parametric models vs. local fits of kernel regression

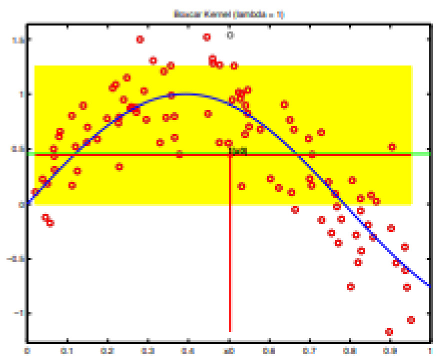
Global average : A global fit constant function weights all points equally.

- take all the observations -> add them together -> take the average.
- summing the weighted set of observations - where the weights are exactly the same oneach of the data points and then dividing by the total sum of the weights.
- Equal weights on each datapoints.
- It is almost like kernel regression - but all the observations are included and with equal weight on every observation. Large bandwidth parameter.

A globally constant fit weights all points equally

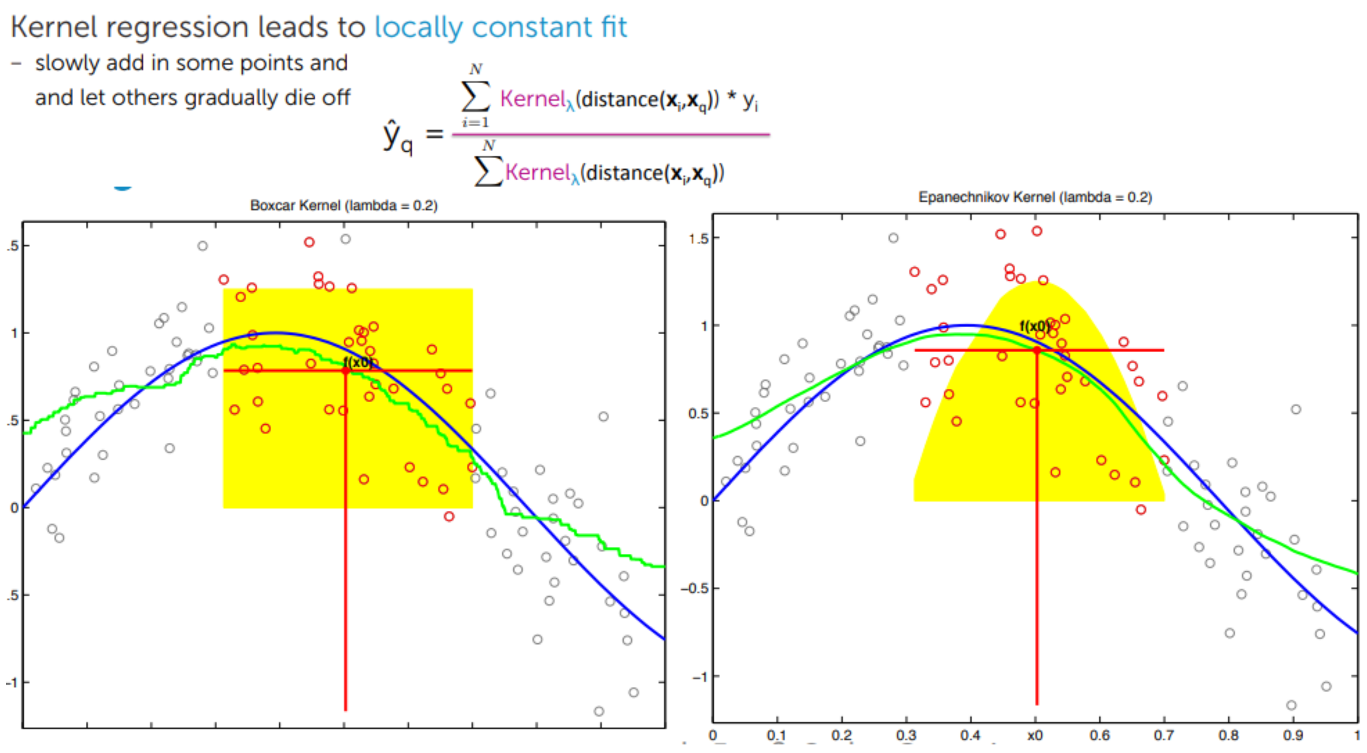
equal weight on each datapoint

$$\hat{y}_q = \frac{1}{N}\sum_{i=1}^{N} y_i = \frac{\sum_{i=1}^{N} c\, y_i}{\sum_{i=1}^{N} c}$$



Local average : Kernel Regression leads to local fit.

1. Box-car kernel - Here the observation within the boxcar kernel are weighed equall and help determine the target value at that point. A set of target points are constructed in this manner individually. This leads to the green fitted line. Constant fit is obtained at the target fit.
2. Epanechnikov kernel - This has weights decaying over the fixed region. It down-weights observations that are further from the target point and emphasizes more heavily the observations more closely to our target point.



Kernel regression leads to locally constant fit

- slowly add in some points and and let others gradually die off

$$\hat{y}_q = \frac{\sum_{i=1}^{N} \text{Kernel}_\lambda(\text{distance}(\mathbf{x}_i,\mathbf{x}_q)) * y_i}{\sum_{i=1}^{N} \text{Kernel}_\lambda(\text{distance}(\mathbf{x}_i,\mathbf{x}_q))}$$

It is a weighted global average but locally defined within the kernel window/bandwidth.

Local linear regression :

- Thus far discussed fitting **constant function** locally at each point. This is "locally weighted **averages**".
- Can instead fit a **line or polyniomial** locally at each point. This is "locally weighted **linear regression**".

Local regression rules:

1. **Local linear fit** - reduces bias at the boundaries and with minimum increase in variance.
2. **Local quadratic fit** - it doesn't help at the boundaries and increases variance. It helps capture curvature in the interior.
3. With sufficient data, local polynomials of odd degree dominate those of even degree.

Recommended -> local linear regression.

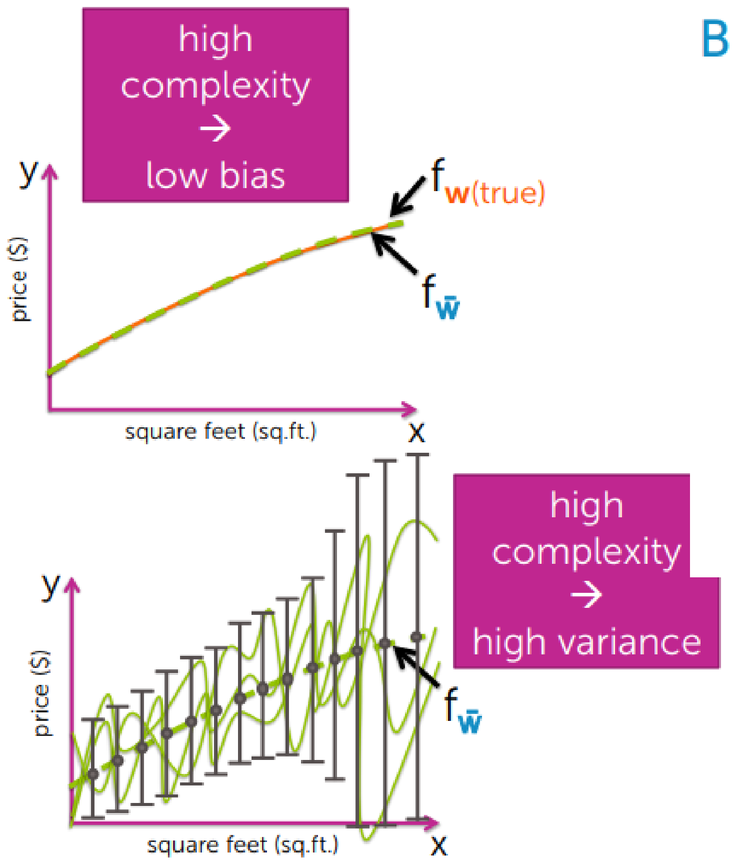
Nonparametric approaches

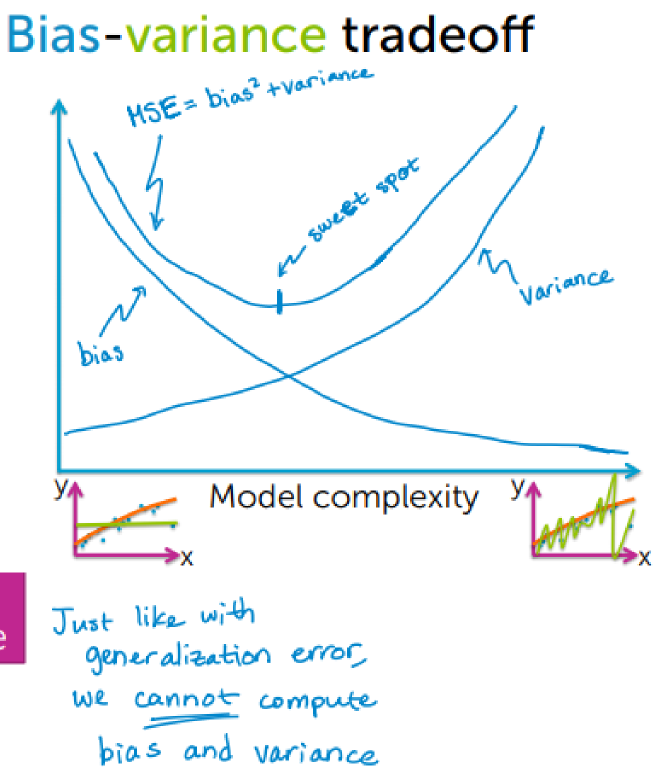
- **k-NN** and **kernel regression** are examples of 'nonparametric' approaches.
- General goals of nonparametrics:
  - Flexibility.
  - Make few assumptions about f(x).
  - **Complexity can grow with the number of observations N**.

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

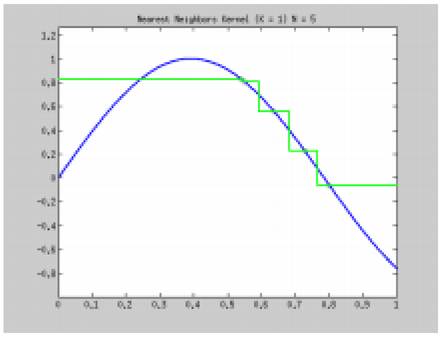
Limiting behavior of NN: (Noiseless setting (epsilon - i = 0))

- In the limit of getting an infinite amount of noiseless data, the **MSE of 1-NN fit goes to 0**.
- MSE - Mean Squared Error => Sum of Bias square and variance.
- MSE = bias^2+variance;



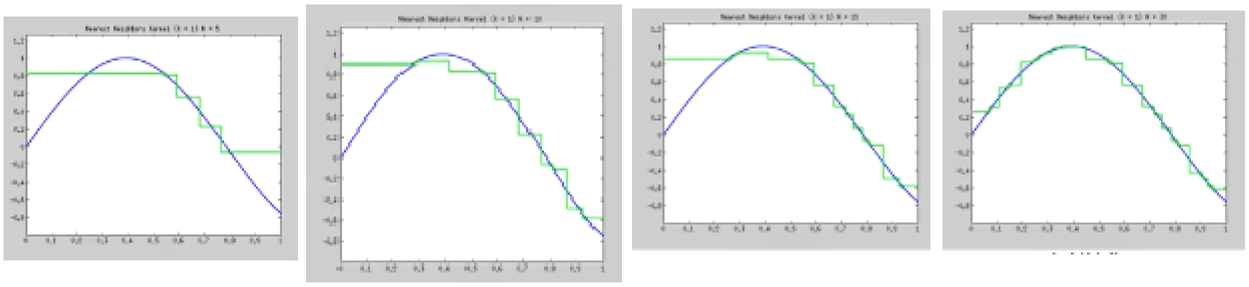


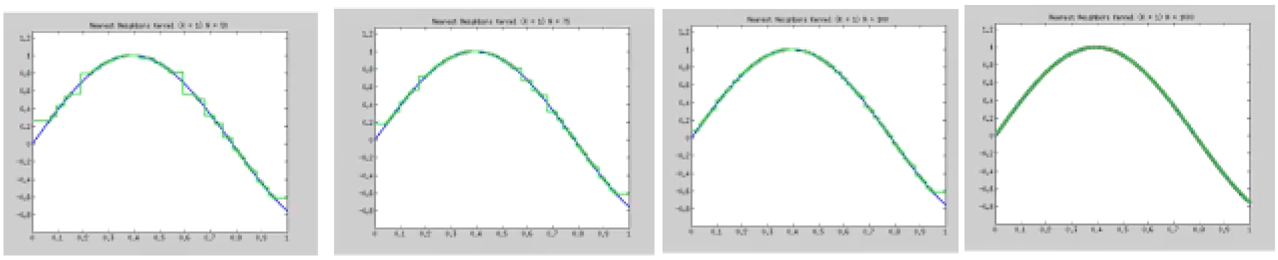
For 1-nn fit, as the amount of data increases, the fit is in perfect sync with the true function. (Non-parametric model)



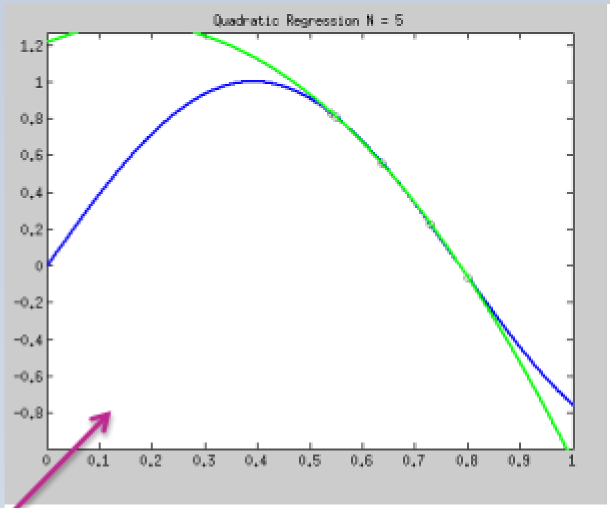
1-NN fit

The blue line is the true curve,the green line is the NN fit based on observations that are on the blue function.
The fit changes with more and more data and gets closer to the true function.
The bias and variance are minimize as the amount of data inreases.





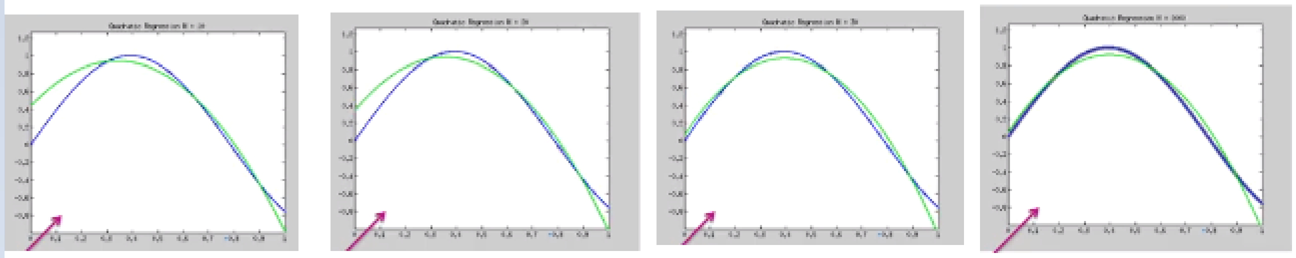
Parametric model - quadratic fit -> the fit never exactly sync with the true curve/ function even with infint amount of data.



Quadratic fit

Not true for parametric models!

The blue line is the true curve,the green line is the NN fit based on observations that are on the blue function.
The fit changes with the amount of data.
But the quadratic fit will always have certain amount of 'bias'. Due to which it will never exactly fit the true curve.
This is a ***Parametric model***.



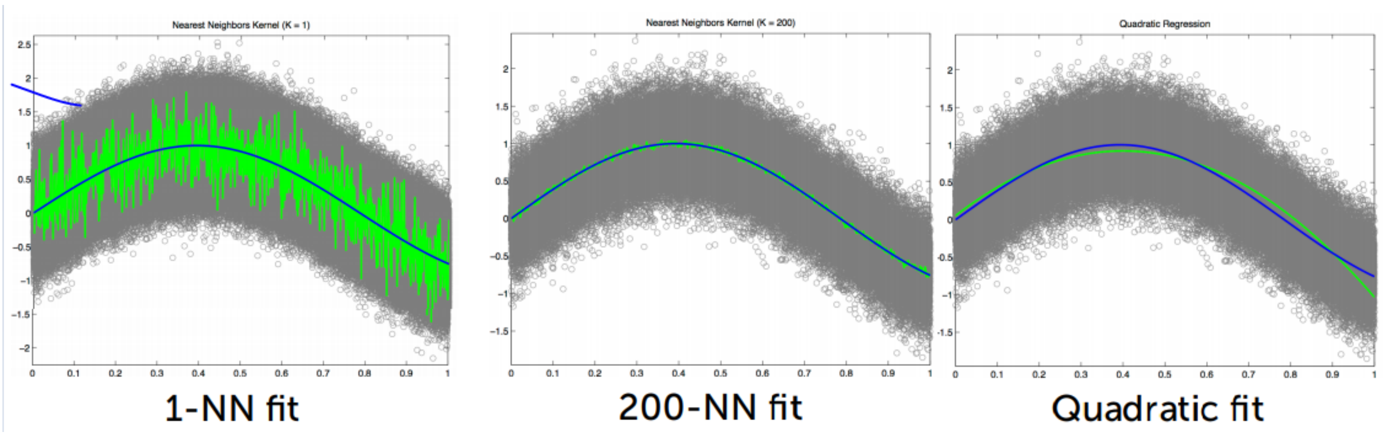
Error vs. amount of data.

1.With noise

- For parametrized model, the true error decrease with increase in data to a certain extent and thereafter it remains constant. The training error increases with increase in amount of data to a certain extent and thereafter remains constant.
- This constant limit -> sum of **bias** and **noise**.

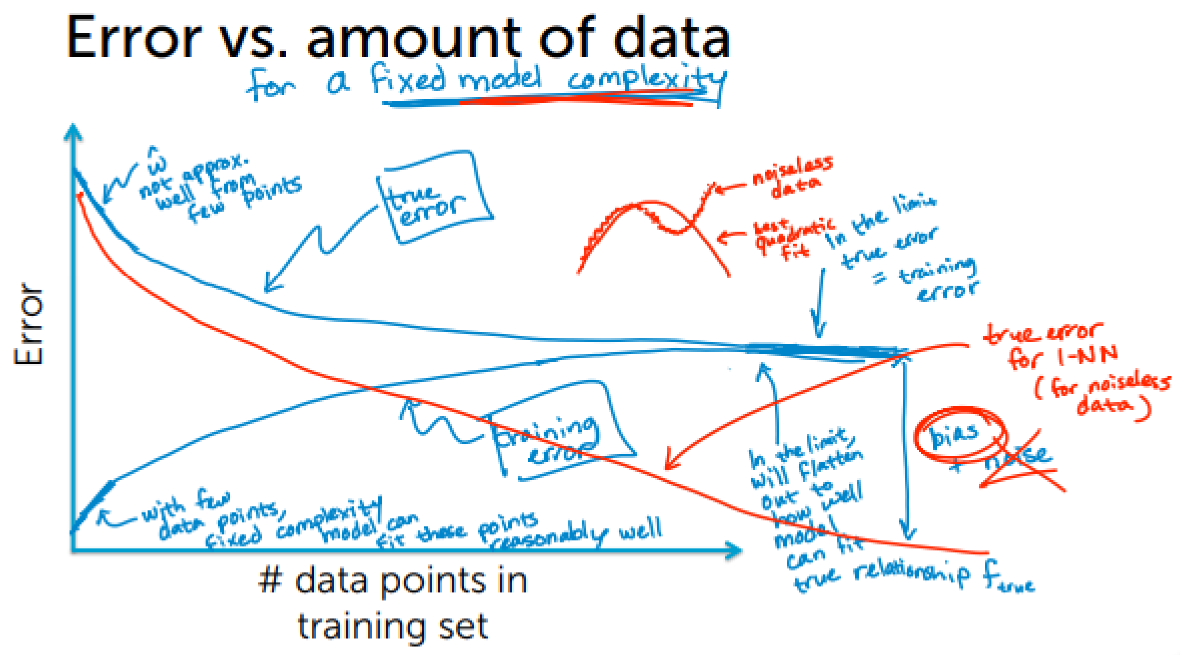
In the limit of getting an infinte amount of data, the MSE(Mean Square Error) of NN fit goes to 0 if k grows to infintity.

```
* The parametric model - quadratic fit, will always have bias, hence will never tr
uly fit the true function.
* non-parametric model -> 1-NN is highly senitive to noise and hence fails to prov
ide an efficient fit.
* non-parametric model with k (high value -> 200) provides a better fit.
```



2. Without noise.

- As the amount of data increases.
- For non-parametric model - k-nn, kernel regression the bias goes to 0, noise = 0, and the fit model is in perfect fit to the true curve. As shown by the **true error for 1-NN (for noiseless data)**.
- For parametric model (e.g = quadratic fit), even as the amount of data increases, and noise = 0 , the fit model is not in perfect sync with the true function due to **bias**.



NN and kernel methods for 'large d' or 'small N'

- NN and kernel regression methods work well when the data covers the space.
  - the more dimensions d you have, the more points N are required to cover the space.
  - need **N = O(exp(d))** data points for good performance.
  - Computationally intensive.
  - These cases require parametric models.

Complexity of NN search :

Naive approach: Brute force search

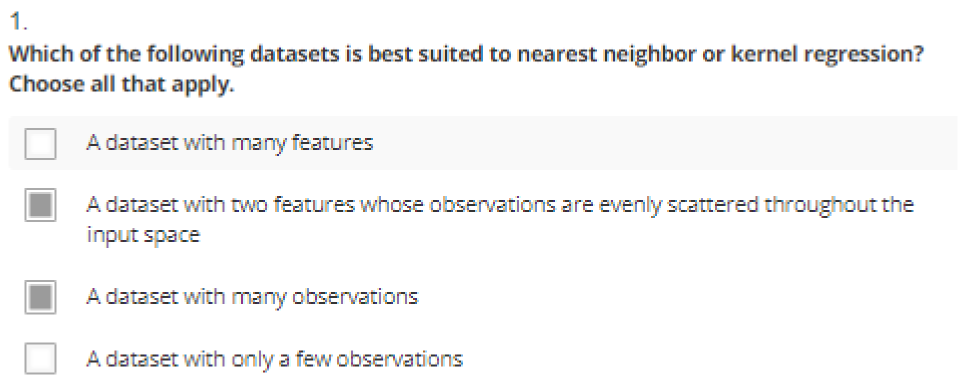
```
- Given a query point x(q)
- Scan through each point x1,x2,x3,...,xN;
- O(N) distance computation per 1-NN query.
- o(Nlogk) pe k-NN query.
```

- If N is large then - Computationally intensive.

Using k-NN for classification.

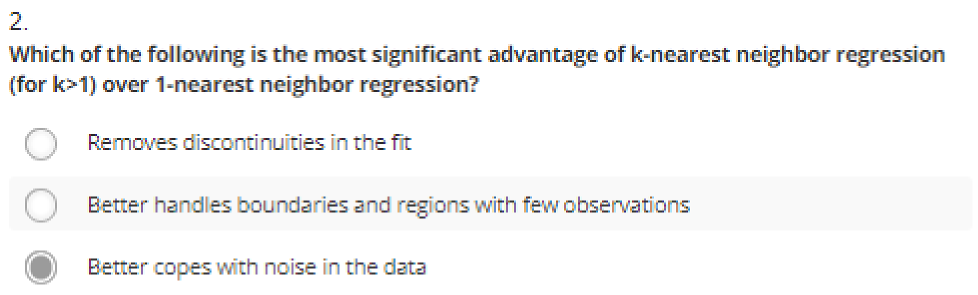
- Spam filtering example.
- Given input x -> email, ip adress.
- output y -> spam / not spam;
- query email - need to classify.
- algorithm - decide via majority votes of k-NN.

Quiz


1.
Which of the following datasets is best suited to nearest neighbor or kernel regression? Choose all that apply.

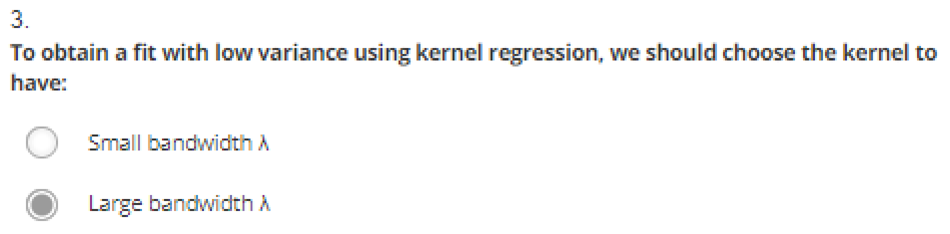
- A dataset with many features
- [✓ A dataset with two features whose observations are evenly scattered throughout the input space]
- [✓ A dataset with many observations]
- A dataset with only a few observations

- Best suited dataset for nonparametric regression -> NN, kernel
  - Few features
  - Many observations.


2.
Which of the following is the most significant advantage of k-nearest neighbor regression (for k>1) over 1-nearest neighbor regression?

- Removes discontinuities in the fit
- Better handles boundaries and regions with few observations
- [✓ Better copes with noise in the data]

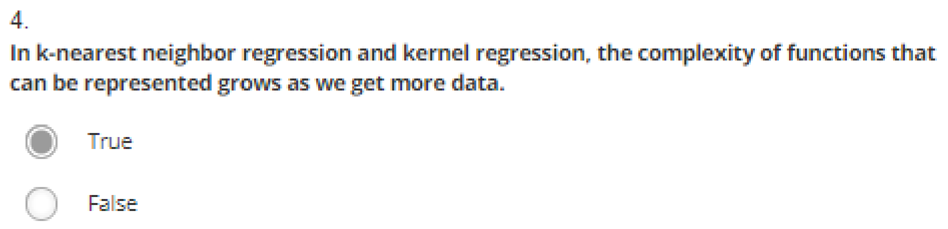
- 1-NN -> highly sensitive to noises.
- k-NN -> copes better with noises.


3.
To obtain a fit with low variance using kernel regression, we should choose the kernel to have:

- Small bandwidth λ
- [✓ Large bandwidth λ]

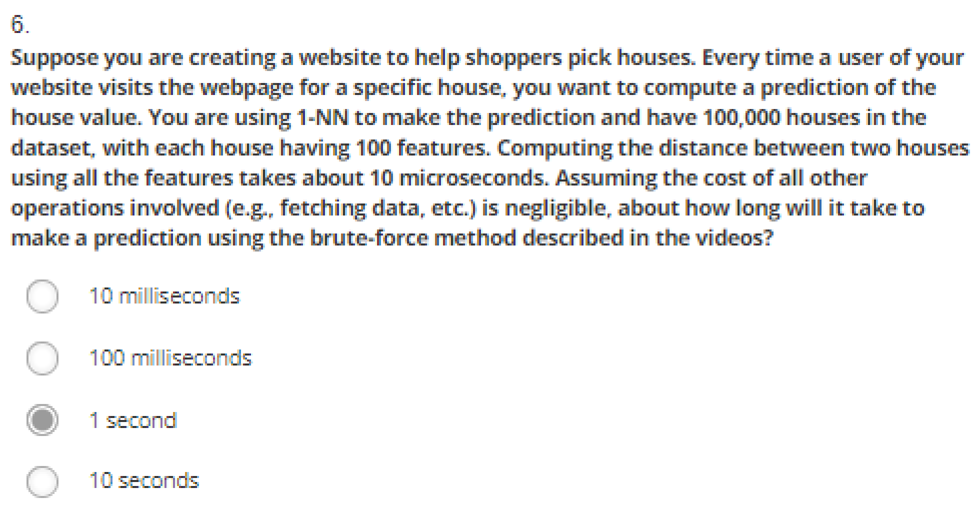
Choice of bandwidth - lambda

1. lambda = 0.04 - small, fit model - overfit; -> Low bias & high variance. (changing the observation - change the fit);
2. lambda = 0.2 - moderate, reasonable fit - bias-variance trade-off.
3. lambda = 0.4 - high, fit model - oversmoothed fit; -> Low variance & high bias.


4.
In k-nearest neighbor regression and kernel regression, the complexity of functions that can be represented grows as we get more data.

- [✓ True]
- False

- In case of parametric regression -> quadratic fit -> it will never converge to the true solution since 'bias' will exist.
- In case of 1-NN it will converge to the true curve with lare amount of data.

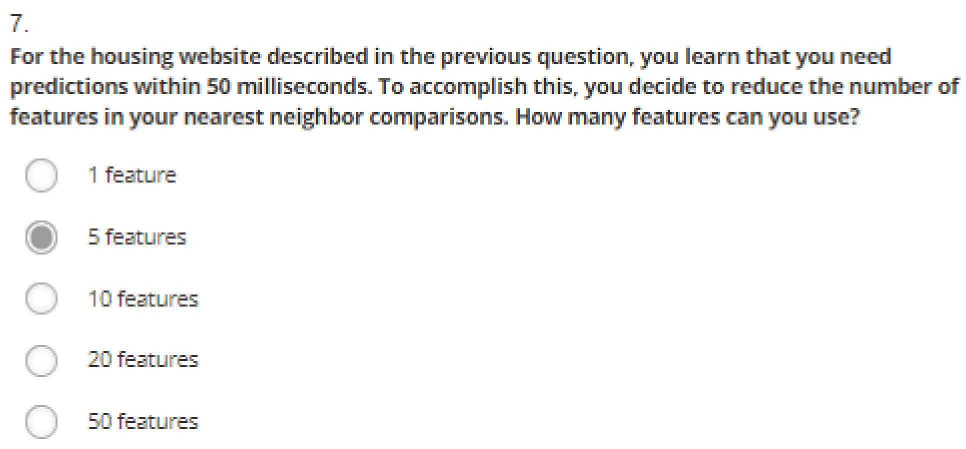


6.

Suppose you are creating a website to help shoppers pick houses. Every time a user of your website visits the webpage for a specific house, you want to compute a prediction of the house value. You are using 1-NN to make the prediction and have 100,000 houses in the dataset, with each house having 100 features. Computing the distance between two houses using all the features takes about 10 microseconds. Assuming the cost of all other operations involved (e.g., fetching data, etc.) is negligible, about how long will it take to make a prediction using the brute-force method described in the videos?

- 10 milliseconds
- 100 milliseconds
- [✓ 1 second]
- 10 seconds

Answer : O(N) distance computation per 1-nn query, so the time make the prediction = 100,000 houses * 10 microseconds = 1 second.



7.

For the housing website described in the previous question, you learn that you need predictions within 50 milliseconds. To accomplish this, you decide to reduce the number of features in your nearest neighbor comparisons. How many features can you use?

- 1 feature
- [✓ 5 features]
- 10 features
- 20 features
- 50 features

-**Answer** : 100,000 houses *X = 50 milliseconds; =>X = 50* 10^-8;

```
                         # features     time to predict
          # from above       100          10 microseconds
          # now               Y           50 * 10^-8 seconds

       Y = 5 features;
```

```
In [ ]:
```