

<https://www.cs.ubc.ca/~fwood/CS340/>

## Lecture XVI - Feature selection

February 10th, 2020

<https://www.cs.ubc.ca/~fwood/CS340/lectures/L16.pdf>

### Feature Selection

To discuss feature selection we recall the motivating example for decision tree: The food allergy example.

Egg	Milk	Fish	Wheat	Shellfish	Peanuts	...	Sick?
0	0.7	0	0.3	0	0		1
0.3	0.7	0	0.6	0	0.01		1
0	0	0	0.8	0	0		0
0.3	0.7	1.2	0	0.10	0.01		1

Here instead of predicting "sick" we want to do feature selection. This means that we want to discover *which* foods are "relevant" in predicting *sick*.

The general feature selection problem is outlined as follows:

$$X = \begin{bmatrix} & & & & & & \end{bmatrix} \quad y = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

(A blue oval highlights a column in matrix X, with an arrow pointing to the label "feature 'j'" written above it.)

**Find the features (columns) of  $X$  that are important in predicting  $y$**

- "What are the relevant factors?"
- "Which basis functions should I use among these choices?"
- "What types of new data should I collect?"
- "How can I speed up computation?"

This is one of the most important problems in ML/statistics, but it is very messy:

- For now, we'll say a feature is "relevant" if it helps *predict  $y_i$  from  $x_i$* .

### "Association" Approach

A simple/common way to do feature selection is the following:

- For each feature 'j', compute correlation between feature values  $x_j$  and  $y$ . We say that  $j$  is relevant if correlation is above 0.9 or below  $-0.9$ .
- This turns feature selection into hypothesis testing for each feature.

However this usually gives unsatisfactory results as it *ignores variable interactions*:

- **Includes irrelevant variables:** “Taco Tuesdays”.
  - If tacos make you sick, and you often eat tacos on Tuesdays, it will say “Tuesday” is relevant.
- **Excludes relevant variables:** “Diet Coke + Mentos Eruption”.
  - Diet coke and Mentos don’t make you sick on their own, but *together* they make you sick.

## ”Regression Weight” Approach

This is a simple and common approach to feature selection. We :

- Fit regression weights  $w$  based on all features (maybe with least squares).
- Take all features  $j$  where weight  $|w_j|$  is greater than a threshold.

For example if you fit a least squares model with 5 features and get:

$$w = \begin{bmatrix} 0.01 \\ -0.2 \\ 10 \\ -3 \\ 0.0001 \end{bmatrix}$$

We observe that:

- Feature 3 looks the most relevant.
- Feature 4 also looks relevant.
- Feature 5 seems irrelevant.

This could:

- Recognize that “Tuesday” is irrelevant.
  - If you get enough data, and you sometimes eat tacos on other days. (And the relationship is actually linear.)
- Recognize that “Diet Coke” and “Mentos” are relevant.
  - Assuming this combination occurs enough times in the data.

**However:** this problem has **major problems with collinearity**:

- If the “Tuesday” variable always equals the “taco” variable, it **could say that Tuesdays are relevant but tacos are not.**

$$\hat{y}_i = w_1 * \text{taco} + w_2 * \text{Tuesday} = 0 * \text{taco} + (w_1 + w_2) * \text{Tuesday}$$

- If you have two copies of an irrelevant feature, **it could take both irrelevant copies.**

$$\hat{y}_i = 0 * \text{irrelevant} + 0 * \text{irrelevant} = 10000 * \text{irrelevant} + (-10000) * \text{irrelevant}$$

## Search and Score Methods

The most common feature selection method is [search and score](#). It focuses around:

1. Define [score function](#)  $f(S)$  that measures quality of a set of features  $S$ .
2. Now [search](#) for the variables  $S$  with the best score

Example with 3 features:

- Compute “score” of using feature 1.
- Compute “score” of using feature 2.
- Compute “score” of using feature 3.
- Compute “score” of using features  $\{1, 2\}$ .
- Compute “score” of using features  $\{2, 3\}$ .
- Compute “score” of using features  $\{1, 3\}$ .
- Compute “score” of using features  $\{1, 2, 3\}$ .
- Compute “score” of using feature  $\{\}$
- Return the set of features  $S$  with the best “score”.

### Which Score function?

Firstly, the [score can't be the training error](#):

- because training error goes down as you add features, so will [select all features](#).

Therefore a more logical score function would be the [validation error](#).

- [“Find the set of features that gives the lowest validation error.”](#)
- To minimize test error, this is what we want.

However there are problems due to the large [number of sets of variables](#):

- If we have  $d$  variables, there are  $2^d$  [sets](#) of variables.
- [Optimization bias](#) is high: we're optimizing over  $2^d$  models (not 10).
- Prone to [false positives](#): irrelevant variables will sometimes help by chance.

### Number of Features” Penalties

To reduce false positives, we can again use complexity penalties:

$$\text{score}(S) = \frac{1}{2} \sum_{i=1}^n (w_S^T x_{iS} - y_i)^2 + \text{size}(S)$$

- E.g., we could use squared error and number of non-zeroes.
- We're using  $x_{iS}$  as the features  $S$  of example  $x_i$ .

Here we have that if two  $S$  have similar error, this prefers the smaller set:

- It prefers removing feature 3 instead of having  $w_3 = 0.00001$ .

Instead of “size(S)”, we usually write this using the “L0-norm”

## L0-Norm and “Number of Features We Use”

In linear models, setting  $w_j = 0$  is the same as removing feature  $j$ :

$$\begin{aligned}\hat{y}_i &= w_1 x_{i1} + w_2 x_{i2} + w_3 x_{i3} + \dots + w_d x_{id} \\ &\quad \downarrow \text{set } w_2 = 0 \\ \hat{y}_i &= w_1 x_{i1} + 0 + w_3 x_{i3} + \dots + w_d x_{id} \\ &\quad \underbrace{\hspace{1cm}}_{\text{ignore } x_{i2}}\end{aligned}$$

The L0-Norm is the number of non-zero values ( $\|w\|_0 = \text{size}(S)$ )

## L0-penalty: optimization

L0-norm penalty for feature selection is:

$$f(w) = \underbrace{\frac{1}{2} \|Xw - y\|^2}_{\text{training error}} + \underbrace{\lambda \|w\|_0}_{\text{degrees of freedom 'k'}}$$

Suppose we want to use this to evaluate the features  $S = \{1, 2\}$ :

- First fit the  $w$  just using features 1 and 2.
- Now compute the training error with this  $w$  and features 1 and 2.
- Add  $2\lambda$  to the training error to get the score.

We repeat this with other choices of  $S$  to find the best features.

The above equation balances between training error and number of features we use.:

- With  $\lambda = 0$ , we get least squares with all features.
- With  $\lambda = \text{inf}$ , we must set  $w = 0$  and not use any features.
- With other  $\lambda$ , balances between training error and number of non-zeroes.
  - Larger  $\lambda$  puts more emphasis on having zeroes in  $w$  (more features removed).
  - Different values give AIC, BIC, and so on.

## Forward Selection (Greedy Search Heuristic)

In search and score, it's also just hard to search for the best  $S$ , since there are  $2^d$  possible sets.

A common greedy approach is **forward selection**:

1. Compute score if we use no features
2. Try adding "taco", "milk", "egg", and so on (computing score for each)
3. Add "milk" because it got the best score.

4. Try  $\{milk, taco\}$ ,  $\{milk, egg\}$  and so on, computing score of each variable with milk
5. Add "egg" because it got the best score combined with "milk"
6. Try  $\{milk, egg, taco\}$ ,  $\{milk, egg, pizza\}$

Formally, the **forward selection** search algorithm is as follows:

1. Start with an **empty set** of features,  $S = [ ]$ .
2. For each possible feature  $j$ 
  - **Compute the scores of features in  $S$  combined with feature  $j$**
3. Find the  $j$  that has the highest score when added to  $S$ .
4. Check if  $S \cup j$  improves on the best score found so far.
5. Add  $j$  to  $S$  and go back to Step 2.
  - A variation is to **stop if no  $j$  improves the score** over just using  $S$ .

This method is **not guaranteed to find the best set**, but **reduces many problems**:

- Considers  $O(d^2)$  models: cheaper, overfits less, has fewer false positives

## In summary

We set out to choose the relevant features:

$$X = \begin{bmatrix} \text{ } & \text{ } \end{bmatrix} \quad y = \begin{bmatrix} \text{ } \end{bmatrix}$$

(The second column of X is circled in blue, with an arrow pointing to the word "relevant" written in blue.)

The most common approach is **search and score**:

- Define "score" and "search" for features with best score.

But it's **hard to define the "score" and it's hard to "search"**.

- So we often use greedy methods like **forward selection**.

Methods work ok on "toy" data, but are **frustrating on real data**.

- Different methods may return very different results.
- Defining whether a feature is "relevant" is complicated and ambiguous.

The Advice is choosing the relevant variables is:

- Try the association approach.
- Try forward selection with different values of  $\lambda$ .
- Try out a few other feature selection methods too.

**Then:**

- **Discuss the results** with the domain expert.
  - They probably have an idea of why some variables might be relevant.
- **Don't be over confident**
  - These methods are probably not discovering how the world truly works.
  - "The algorithm has found that these variables are helpful in predicting  $y_i$ ."
    - \* Then a warning that these models are not perfect at finding relevant variables.

## Lecture XVII - Regularization

February 12th, 2020

<https://www.cs.ubc.ca/~fwood/CS340/lectures/L17.pdf>

### “Feature” Selection vs. “Model” Selection?

**Model selection:** “which model should I use?”

- KNN vs. decision tree, depth of decision tree, **degree of polynomial basis**.

**Feature selection:** “which features should I use?”

- Using feature 10 or not, **using  $x_i^2$  as part of basis**.

These two tasks are **highly-related**:

- It’s a different “model” if we add  $x_i^2$  to linear regression.
- But the  $x_i^2$  term is just a “feature” that could be “selected” or not.
- Usually, “feature selection” means choosing from some “original” features.
  - You could say that “feature” selection is a special case of “model” selection.

### Can it help prediction to throw features away?

Yes, because **linear regression can over-fit** with large  $d$  (Even though it’s “just” a hyper-plane.)

Consider using  $d = n$ , with completely random features.

- With high probability, you will be able to **get a training error of 0**.
- But the features were random, this is **completely overfitting**.

You could view **“number of features” as a hyper-parameter**:

- Model gets more complex as you add more features.

### Controlling Complexity

We’ve said that **complicated models tend to overfit more**. But what if we need a complicated model?

Usually **“true” mapping from  $x_i$  to  $y_i$  is complex**.

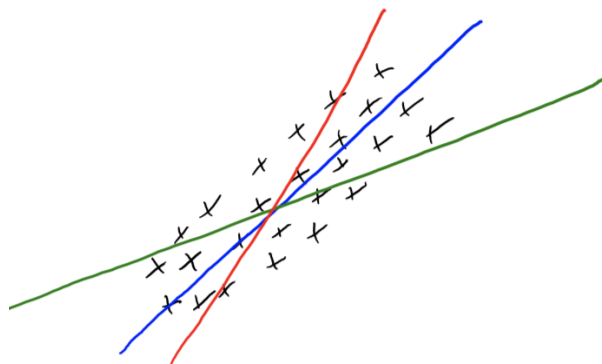
- Might need high-degree polynomial.
- Might need to combine many features, and don’t know “relevant” ones

But complex models can overfit, so what do we do???

Our main tools are:

1. **Model averaging**: average over multiple models to decrease variance
2. **Regularization**: add a **penalty on the complexity** of the model.

Consider the following dataset and 3 linear regression models:



Which line should we choose?

If we are forced to choose between **red** and **green** assuming they have the same training error, we **should pick green**

- Since slope is smaller, **small change in  $x_i$  has a smaller change in prediction  $y_i$** .
  - Green line's predictions are **less sensitive to having  $w$  exactly right**.
- Since green  $w$  is less sensitive to data, test error might be lower.

## L2-regularization

### Motivation

The standard strategy for **regularization** is **L2-regularization**.

$$f(w) = \frac{1}{2} \sum_{i=1}^n (w^T x_i - y_i)^2 + \frac{\lambda}{2} \sum_{j=1}^d w_j^2 \quad \text{or} \quad f(w) = \frac{1}{2} \|Xw - y\|^2 + \frac{\lambda}{2} \|w\|^2$$

The intuition is that **large slopes  $w_j$**  tend to lead to overfitting. Consequently the objective **balances getting low error vs. having small slopes  $w_j$** :

- “You can increase the training error if it makes ‘w’ much smaller.”
- This nearly-always reduces overfitting.
- The regularization parameter  $\lambda > 0$  controls “strength” of regularization.
  - Large  $\lambda$  puts large penalty on slopes.

In terms of the fundamental trade-off:

- Regularization **increases training error**.
- Regularization **decreases approximation error**.

Therefore now the question is: How do we choose  $\lambda$ ?

- Theory: as  $n$  grows  $\lambda$  should be in the range  $O(1)$  to  $(\sqrt{n})$ .
- Practice: optimize **validation set** or **cross-validation error**
  - **This almost always decreases the test error**.

**L2-Regularization “Shrinking” Example**

Solution to a “least squares with L2-regularization” for different  $\lambda$ :

$\lambda$	$w_1$	$w_2$	$w_3$	$w_4$	$w_5$	$\ Xw - y\ ^2$	$\ w\ ^2$
0	-1.88	1.29	-2.63	1.78	-0.63	285.64	15.68
1	-1.88	1.28	-2.62	1.78	-0.64	285.64	15.62
4	-1.87	1.28	-2.59	1.77	-0.66	285.64	15.43
16	-1.84	1.27	-2.50	1.73	-0.73	285.71	14.76
64	-1.74	1.23	-2.22	1.59	-0.90	286.47	12.77
256	-1.43	1.08	-1.70	1.18	-1.05	292.60	8.60
1024	-0.87	0.73	-1.03	0.57	-0.81	321.29	3.33
4096	-0.35	0.31	-0.42	0.18	-0.36	374.27	0.56

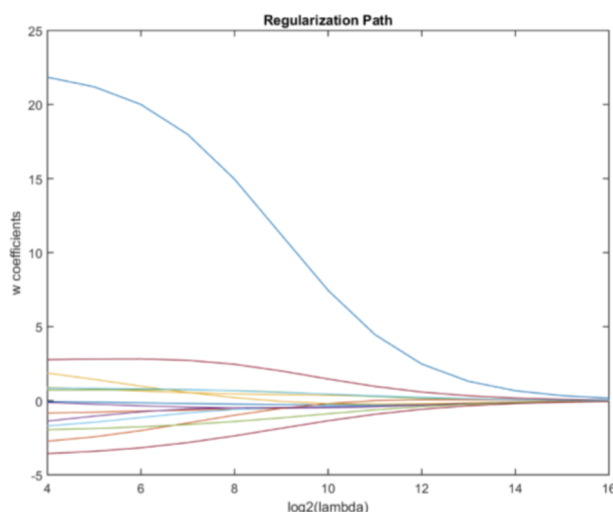
We obtain least squares when  $\lambda = 0$ , but we can achieve similar training error with smaller  $\|w\|$

$\|Xw - y\|$  increases with  $\lambda$ , and  $\|w\|$  decreases with  $\lambda$ .

- Though individual  $w_j$  can increase or decrease with lambda.
- Because we use the L2-norm, the large ones decrease the most.

**Regularization Path**

Regularization path is a plot of the optimal weights  $w_j$  as  $\lambda$  varies.



Starts with least squares with  $\lambda = 0$ , and  $w_j$  converge to 0 as  $\lambda$  grows.

**L2-regularization and the normal equations**

When using the L2-regularized squared error, we can solve for  $\nabla f(w) = 0$ . We have that:

- Loss before:  $f(w) = \frac{1}{2} \|Xw - y\|^2$
- Loss after:  $f(w) = \frac{1}{2} \|Xw - y\|^2 + \frac{\lambda}{2} \|w\|^2$

We also have that:

- Gradient before:  $\nabla f(w) = X^T Xw - X^T y$



- Gradient after:  $\nabla f(w) = X^T X w - X^T y + \lambda w$

And:

- Linear system before:  $X^T X w = X^T y$
- Linear system after:  $(X^T X + \lambda I)w = X^T y$

But unlike  $X^T X$ , the matrix  $(X^T X + \lambda I)$  is always invertible. Then we multiply by its inverse to get  $w = (X^T X + \lambda I)^{-1}(X^T y)$

## Gradient Descent for L2-Regularized Least Squares

The L2-regularized least squares objective and gradient is:

$$f(w) = \frac{1}{2} \|Xw - y\|^2 + \frac{\lambda}{2} \|w\|^2$$
$$\nabla f(w) = X^T(Xw - y) + \lambda w$$

The gradient descent iterations for L2-regularized least squares is:

$$w^{t+1} = w^t - \alpha^t \underbrace{(X^T(Xw^t - y) + \lambda w^t)}_{\nabla f(w^t)}$$

Here the cost of a gradient descent iteration is still  $O(nd)$ . We can also show that the number of iterations decreases as  $\lambda$  increases (which is not an obvious proof).

### Digression: Why use L2-Regularization?

In general it "Almost always decreases the test error". However here are 6 more reasons:

1. Solution to  $w$  is **unique**.
2.  $X^T X$  **does not have to be invertible**
3. **Less sensitive** in changes to  $X$  or  $y$ .
4. Gradient descent **converges faster** (bigger  $\lambda$  means fewer iterations).
5. *Stein's paradox*: if  $d \geq 3$ , 'shrinking' **moves us closer to 'true'  $w$** .
6. Worst case: just set  $\lambda$  small and get the same performance.

## Standardizing Features

### Features with Different Scales

Let's consider continuous features with different scales:

Egg (#)	Milk (mL)	Fish (g)	Pasta (cups)
0	250	0	1
1	250	200	1
0	0	0	0.5
2	250	150	0

Should we convert them to a certain normalized/standardized unit?

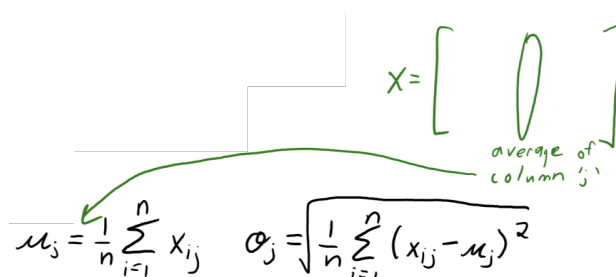
- It **doesn't matter for decision trees or Naïve Bayes**, since they only look at one feature at a time.
- It also **doesn't matter for Least squares**, since unit conversion doesn't affect the model.

However:

- It **matters for K-Nearest Neighbors**, since distance will be affected more by large features than small features
- It **matters for regularized least squares**, since penalizing  $w_j^2$  means different things if features  $j$  are on different scales.

Consequently it is common to **standardize continuous features**:

- For each feature:
  - Compute the mean and standard deviation:



$$X = \begin{bmatrix} & & & & 0 \\ & & & & \vdots \\ & & & & 0 \end{bmatrix}$$

average of column  $j$

$$\mu_j = \frac{1}{n} \sum_{i=1}^n x_{ij} \quad \sigma_j = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_{ij} - \mu_j)^2}$$

- Subtract mean and divide by standard deviation ("z-score")

Replace  $x_{ij}$  with  $\frac{x_{ij} - \mu_j}{\sigma_j}$

- Now changes in  $w_j$  have the same effect for any feature  $j$

**How should we standardize test data?**

- **The wrong approach is** to use the mean and standard deviation of the test data
- The training and test mean and standard deviation might be very different.
- The **right approach is to use the mean and standard deviation of the training data**

**If we're doing 10-fold cross validation:**

- Compute  $\mu_j$  and  $\sigma_j$  based on the 9 training folds (e.g.: average over 9/10s of the data)
- Standardize the remaining "validation" fold with this training  $\mu_j$  and  $\sigma_j$
- Re-standardize for the different folds

**Standardizing Target**

In regression we sometimes **standardize the targets  $y_i$** . This puts the targets on the same standard scale as the standardized features:

Replace  $y_i$  with  $\frac{y_i - \mu_y}{\sigma_y}$

With a standardized target, setting  $w = 0$  predicts average  $y_i$ , which means that higher regularization lets us predict closer to the average value.

But it is important to remember to **standardize test data with the training sets!**

Other common approaches to transformations are the use of exponents or logarithms:

Use  $\log(y_i)$  or  $\exp(\gamma y_i)$

## Lecture XVIII - More Regularization

February 14th, 2020

<https://www.cs.ubc.ca/~fwood/CS340/lectures/L18.pdf>

### Parametric vs. Non-Parametric Transforms

So far we've been using linear models with **polynomial bases**

$$y_i = w_0 \boxed{\text{---}} + w_1 \boxed{\text{---}} + w_2 \boxed{\text{---}} + w_3 \boxed{\text{---}} + w_4 \boxed{\text{---}}$$

$|$                        $x_{ii}$                        $(x_{ii})^2$                        $(x_{ii})^3$                        $(x_{ii})^4$

But polynomials are not the only possible bases:

- Exponentials, logarithms, trigonometric functions, etc
- The **right basis** will vastly improve performance
- If we use the wrong basis, the accuracy will be limited even with a large amount of data
- But the **right bases might not be obvious!**

An alternative is **non-parametric bases**:

- The size of the basis (number of features) **grown with  $n$**
- Model gets more complicated as you get more data
- It can model complicated functions where you don't know the right basis (*with enough data*)
- A classic example is **Gaussian RBF's** ("Gaussian" = "Normal Distribution")

### Gaussian RBF's: A sum of "bumps"

$$y_i = w_0 \boxed{\text{---}} + w_1 \boxed{\text{---}} + w_2 \boxed{\text{---}} + w_3 \boxed{\text{---}} + w_4 \boxed{\text{---}}$$

Polynomial basis represents function as sum of global polynomials.

$$y_i = w_0 \boxed{\text{---}} + w_1 \boxed{\text{---}} + w_2 \boxed{\text{---}} + w_3 \boxed{\text{---}} + w_4 \boxed{\text{---}}$$

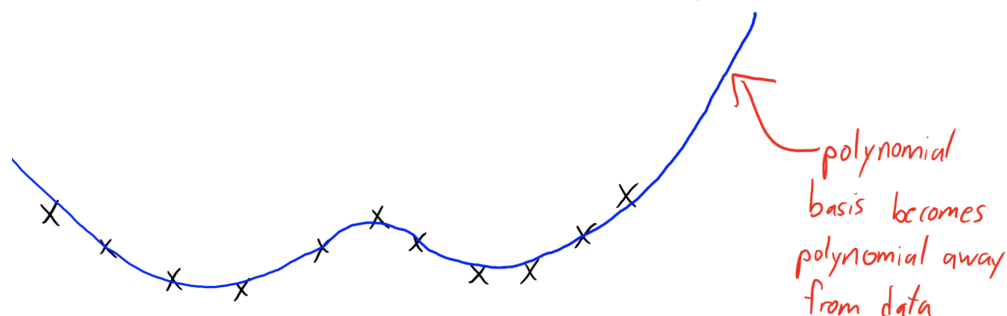
Gaussian RBFs represent function as sum of local "bumps"

Gaussian RBFs are **universal approximators** (compact subsets of  $\mathbb{R}^d$ ):

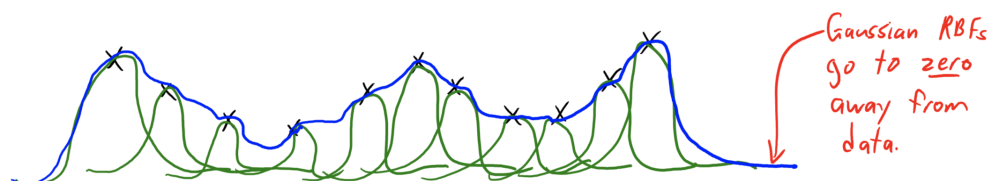
- Enough bumps can **approximate any continuous function** to an arbitrary precision
- This means we can **achieve optimal test error** as  $n$  goes to infinity

Comparison:

- *Polynomial Fit:*



- *Constructing a function from bumps ("smooth histogram")*



### Gaussian RBFs parameters

Some obvious **questions** are:

1. How many bumps should we use?
2. Should the bumps be centered?
3. How far up should the bumps go?
4. How wide should the bumps be?

The usual **answers** are:

1. We use  $n$  bumps (non-parametric basis)
2. Each bump is centered on one training example  $x_i$
3. Fitting regression weights  $w$  gives us the heights (and signs)
4. The width is a hyper parameter (narrow bumps = complicated model).

### Gaussian RBFs: Formal Details

What is a **radial basis functions** (RBFs)?

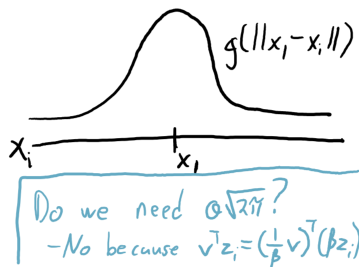
- A set of non-parametric bases that depend on distances to training points

Replace  $x_i = (\underbrace{x_{i1}, x_{i2}, \dots, x_{in}}_{\text{'d' features}})$  with  $z_i = (\underbrace{g(\|x_i - x_1\|), g(\|x_i - x_2\|), \dots, g(\|x_i - x_n\|)}_{\text{'h' features}})$

- We have  $n$  features with feature  $j$  depending on its distance to example  $i$
- The most common  $g$  is the Gaussian RBF:

$$g(\varepsilon) = \exp\left(-\frac{\varepsilon^2}{2\sigma^2}\right)$$

- The variance  $\sigma^2$  is a hyper-parameter controlling the "width". This affects fundamental trade-off (and we set it using a validation set).



Replace  $X = \begin{bmatrix} \vdots \end{bmatrix}_n$  by  $Z = \begin{bmatrix} g(\|x_1 - x_1\|) & g(\|x_1 - x_2\|) & \dots & g(\|x_1 - x_n\|) \\ g(\|x_2 - x_1\|) & g(\|x_2 - x_2\|) & \dots & g(\|x_2 - x_n\|) \\ \vdots & \vdots & \ddots & \vdots \\ g(\|x_n - x_1\|) & g(\|x_n - x_2\|) & \dots & g(\|x_n - x_n\|) \end{bmatrix}_n$

To make predictions on  $\tilde{X} = \begin{bmatrix} \vdots \end{bmatrix}_t$  use  $\tilde{Z} = \begin{bmatrix} g(\|\tilde{x}_1 - x_1\|) \\ \vdots \\ g(\|\tilde{x}_t - x_n\|) \end{bmatrix}_t$

Number of "features" is number of training examples

### Pseudocode

Here is the pseudo code for construction RBFs given data  $X$  and a hyper-parameter  $\sigma$

```

Z = zeros(n,n)
for i1 in 1:n :
    for i2 in 1:n :
        Z[i1, i2] = exp(-norm(X[i1,:] - X[i2,:])^2 / (2 * sig^2))

```

With the test data  $\tilde{X}$ : form  $\tilde{Z}$  based on distances to training examples.

And if we wish to do least squares for different values of the hyper-parameter  $\sigma$ , then we could add a bias, and a linear basis:

$$Z = \begin{bmatrix} 1 - x_1 - g(\|x_1 - x_1\|) & \dots & g(\|x_1 - x_n\|) \\ 1 - x_2 - g(\|x_2 - x_1\|) & \dots & g(\|x_2 - x_n\|) \\ \vdots & \ddots & \vdots \\ 1 - x_n - g(\|x_n - x_1\|) & \dots & g(\|x_n - x_n\|) \end{bmatrix}$$

$\underbrace{\hspace{10em}}_1$ 
 $\underbrace{\hspace{10em}}_n$

This reverts to linear regression, instead of 0 away from the data.

## RBFs, Regularization and Validation

We make predictions using rbf's as follows:

$$\begin{aligned} \hat{y}_i &= w_1 \exp\left(-\frac{\|x_i - x_1\|^2}{2\sigma^2}\right) + w_2 \exp\left(-\frac{\|x_i - x_2\|^2}{2\sigma^2}\right) + \dots + w_n \exp\left(-\frac{\|x_i - x_n\|^2}{2\sigma^2}\right) \\ &= \sum_{j=1}^n w_j \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right) \end{aligned}$$

Here:

- The flexible basis can model any continuous function.
- But with  $n$  data points RBFs have  $n$  basis functions.

How can we avoid over-fitting with this huge number of features?

We regularize  $w$  and use validation error to choose  $\sigma$  and  $\lambda$

This turns out to be a model that is hard to beat:

- RBF basis with L2-regularization and cross validation to choose  $\sigma$  and  $\lambda$
- Flexible non-parametric basis, magic of regularization, and tuning for test error.

```
for each value of sig and lamd:
    Compute Z on training data (and sig)
    Compute best v: v = (Z.T@Z + lamd*I)^(-1) @ Z.T@y
    Compute Z_tilde on validation data (using train data distances)
```