CS 2750: Machine Learning

K-NN (cont'd) + Review

Prof. Adriana Kovashka
University of Pittsburgh
February 3, 2016

Plan for today

- A few announcements
- Wrap-up K-NN
- Quizzes back + linear algebra and PCA review
- Next time: Other review + start on linear regression

Announcements

- HW2 released – due 2/24
  - Should take a lot less time
  - Overview
- Reminder: project proposal due 2/17
  - See course website for a project can be + ideas
  - “In the proposal, describe what techniques and data you plan to use, and what existing work there is on the subject.”
- Notes from board from Monday uploaded

Announcements

- Just for this week, the TA's office hours will be on Wednesday, 2-6pm

Machine Learning vs Computer Vision

- I spent 20 minutes on computer vision features
  - You will learn soon enough that how you compute your feature representation is important in machine learning
- Everything else I've shown you or that you've had to do in homework has been machine learning *applied to computer vision*
  - The goal of Part III in HW1 was to get you comfortable with the idea that each dimension in a data representation captures the result of some separate "test" applied to the data point

Machine Learning vs Computer Vision

- Half of computer vision today is *applied machine learning*
- The other half has to do with image processing, and you've seen none of that
  - See the slides for my undergraduate computer vision class

Machine Learning vs Computer Vision

- Many machine learning researchers today do computer vision as well, so these topics are not entirely disjoint
  - Look up the recent publications of Kevin Murphy (Google), the author of “Machine Learning: A Probabilistic Perspective”

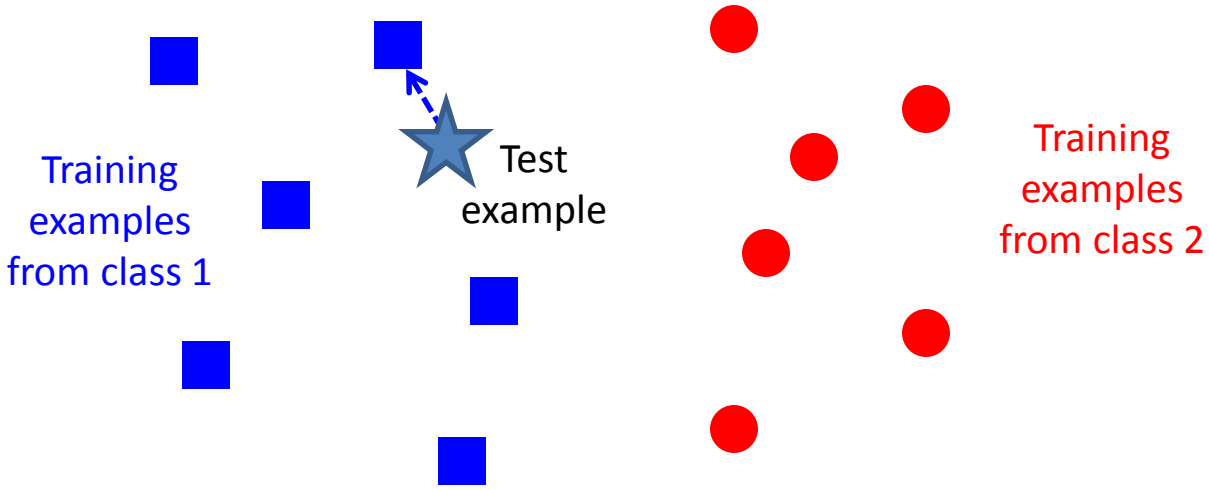
Machine Learning vs Computer Vision

- In conclusion: I will try to also use data other than images in the examples that I give, but I trust that you will have the maturity to not think that I'm teaching you computer vision just because I include examples with images ☺

Last time: Supervised Learning Part I

- Basic formulation of the simplest classifier: K Nearest Neighbors
- Example uses
- Generalizing the distance metric and weighting neighbors differently
- Problems:
  - The curse of dimensionality
  - Picking K
  - Approximation strategies

1-Nearest Neighbor Classifier



f(**x**) = label of the training example nearest to **x**

- All we need is a distance function for our inputs
- No training required!

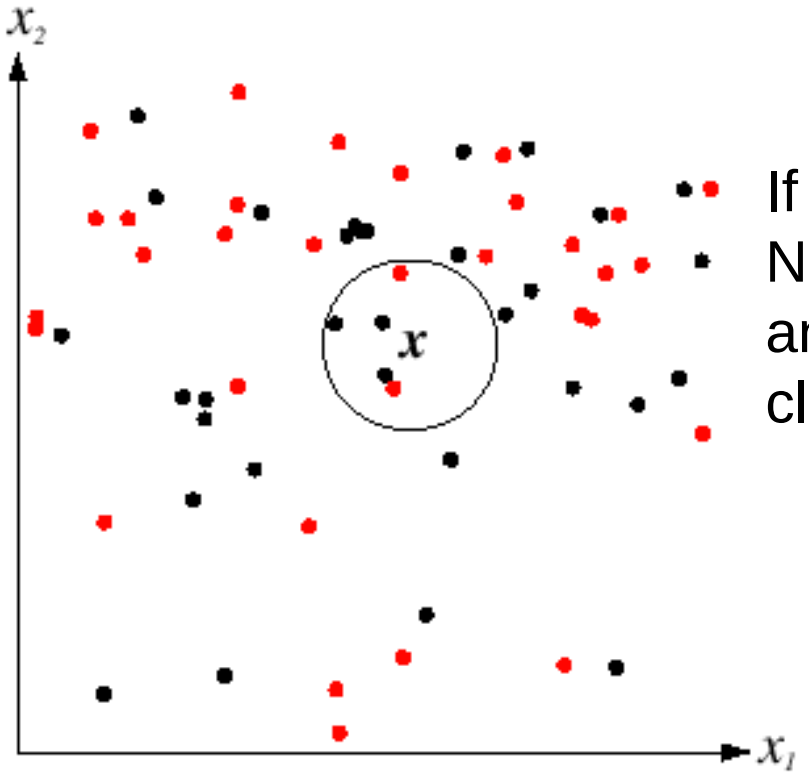
Slide credit: Lana Lazebnik

K-Nearest Neighbors Classifier

- For a new point, find the *k* closest points from training data (e.g. *k*=5)
- Labels of the *k* points "vote" to classify

Black = negative

Red = positive



If query lands here, the 5 NN consist of 3 negatives and 2 positives, so we classify it as negative.

Slide credit: David Lowe

k-Nearest Neighbor

Four things make a memory based learner:

- *A distance metric*
  - **Euclidean (and others)**

- *How many nearby neighbors to look at?*
  - **k**

- *A weighting function (optional)*
  - **Not used**

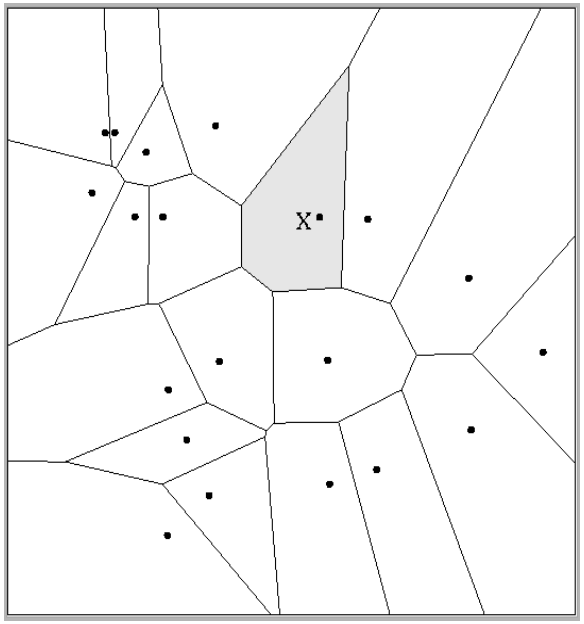
- *How to fit with the local points?*
  - **Just predict the average output among the nearest neighbors**

Slide credit: Carlos Guestrin

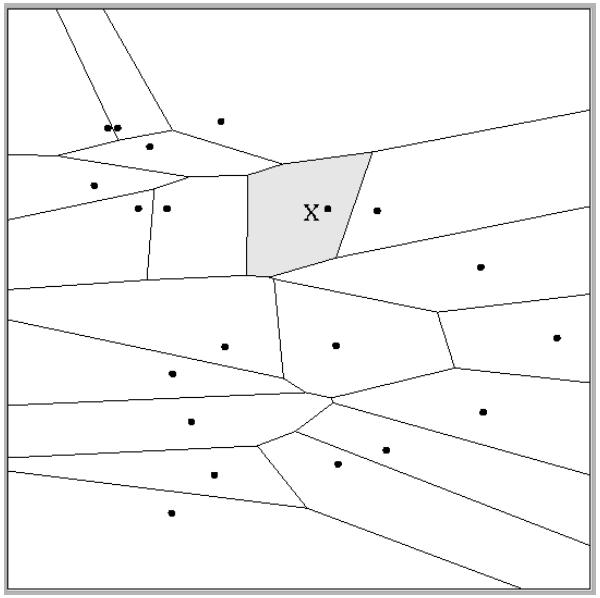
Multivariate distance metrics

Suppose the input vectors $\mathbf{x}_1$, $\mathbf{x}_2$, …$\mathbf{x}_N$ are two dimensional:

$\mathbf{x}_1 = (x_{11} , x_{12})$, $\mathbf{x}_2 = (x_{21} , x_{22})$, … , $\mathbf{x}_N = (x_{N1} , x_{N2})$.



$Dist(\mathbf{x}_i,\mathbf{x}_j) = (x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2$



$Dist(\mathbf{x}_i,\mathbf{x}_j) = (x_{i1} - x_{j1})^2 + (3x_{i2} - 3x_{j2})^2$

The relative scalings in the distance metric affect region shapes

Slide credit: Carlos Guestrin

Another generalization: Weighted K-NNs

- Neighbors weighted differently:

- Extremes
  - Bandwidth = infinity: prediction is dataset average
  - Bandwidth = zero: prediction becomes 1-NN

Kernel Regression/Classification

Four things make a memory based learner:

- *A distance metric*
  - **Euclidean (and others)**

- *How many nearby neighbors to look at?*
  - **All of them**

- *A weighting function (optional)*
  - $w_i = exp(-d(x_i, query)^2 / \sigma^2)$
  - Nearby points to the query are weighted strongly, far points weakly. The σ parameter is the **Kernel Width**.

- *How to fit with the local points?*
  - **Predict the weighted average of the outputs**

Slide credit: Carlos Guestrin

Problems with Instance-Based Learning

- Too many features?
  - Doesn't work well if large number of irrelevant features, distances overwhelmed by noisy features
  - Distances become meaningless in high dimensions (the **curse of dimensionality**)

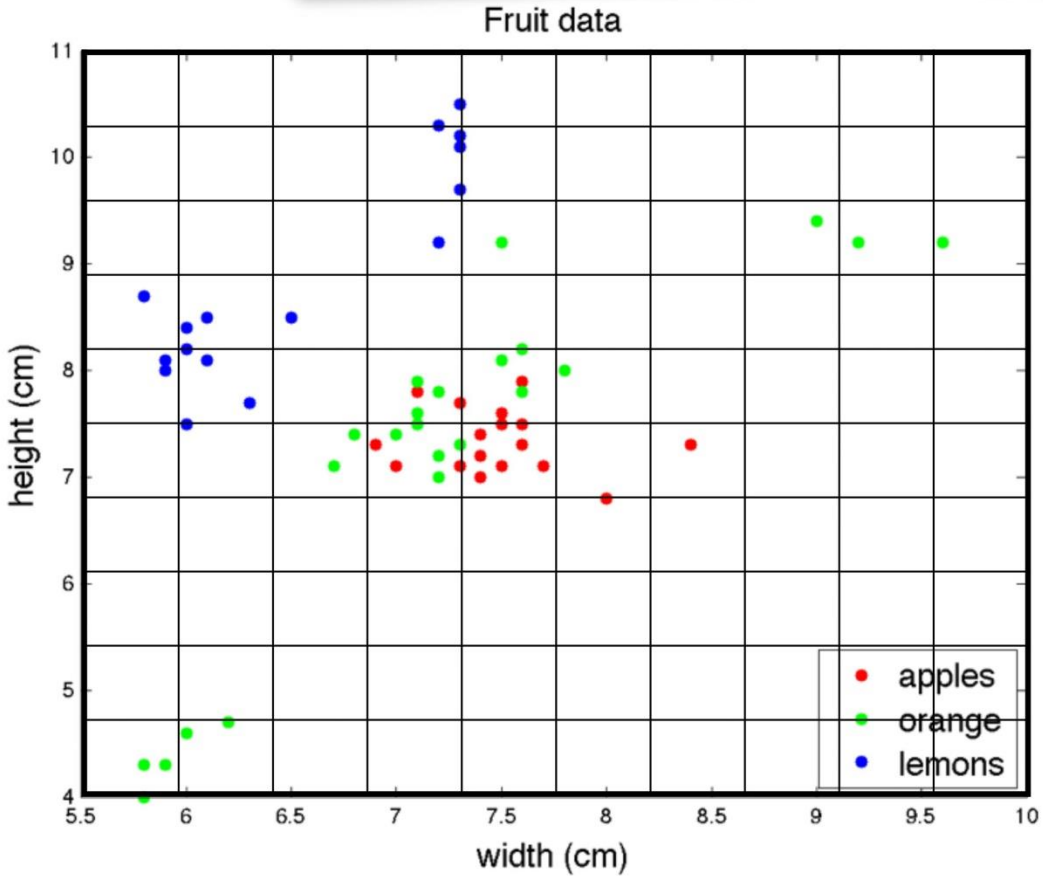
- What is the impact of the **value of K**?

- Expensive
  - No learning: most real work done during testing
  - For every test sample, must search through all dataset – very slow!
  - **Must use tricks like approximate nearest neighbor search**
  - Need to store all training data

Adapted from Dhruv Batra

Curse of Dimensionality

How many neighborhoods are there?



#bins = 10x10

d = 2

#bins = 10^{d}

d = 1000

Atoms in the universe

~ 10^{80}

Curse of Dimensionality

- Consider: Sphere of radius 1 in d-dims
- Consider: an outer ε-shell in this sphere
- What is $\frac{shell\ volume}{sphere\ volume}$?

Slide credit: Dhruv Batra

Curse of Dimensionality

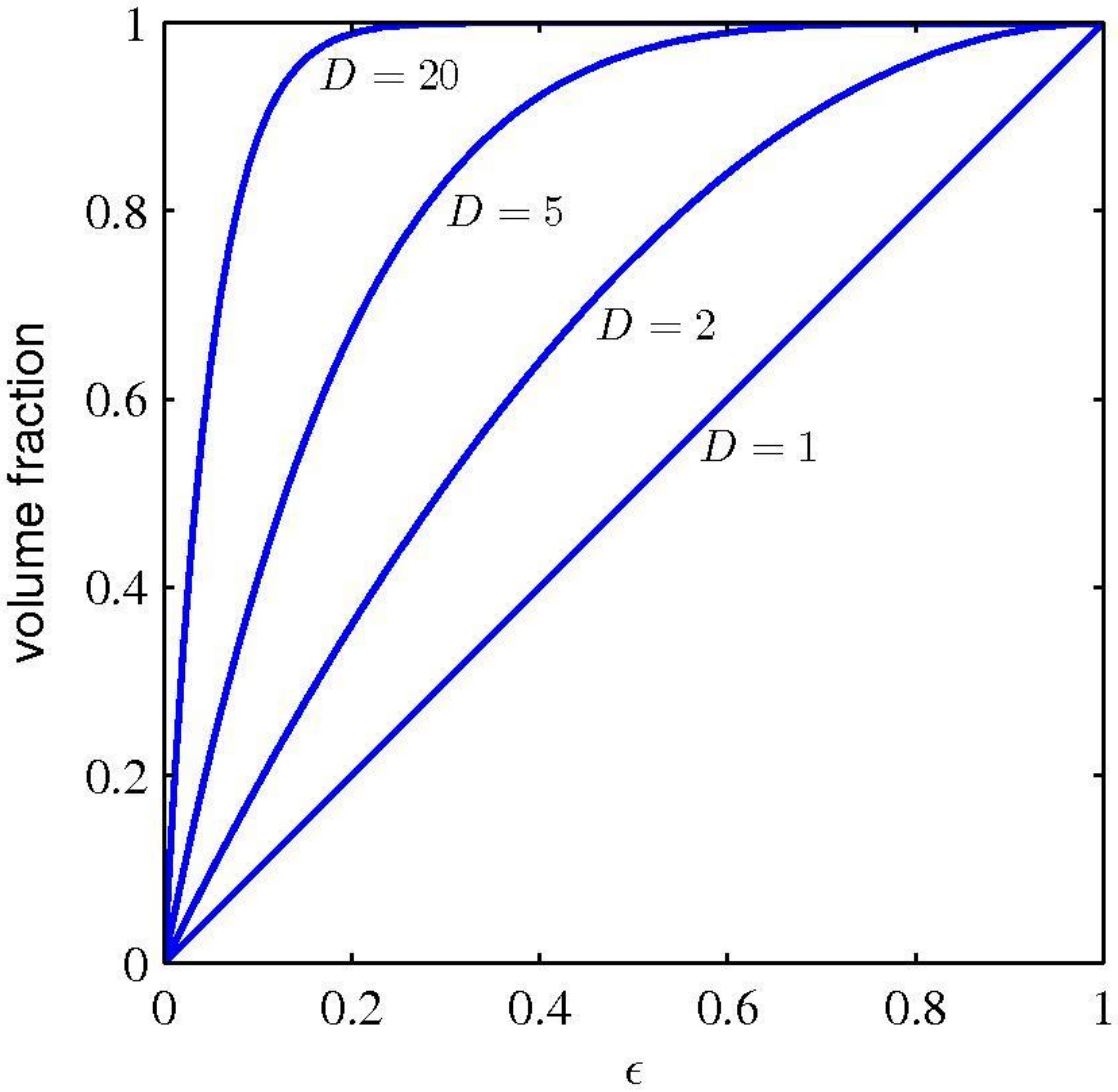


Figure 1.22 from Bishop

Curse of Dimensionality

- Problem: In very high dimensions, distances become less meaningful
- This problem applies to all types of classifiers, not just K-NN

Problems with Instance-Based Learning

- Too many features?
  - Doesn't work well if large number of irrelevant features, distances overwhelmed by noisy features
  - Distances become meaningless in high dimensions (the **curse of dimensionality**)

- What is the impact of the **value of K**?

- Expensive
  - No learning: most real work done during testing
  - For every test sample, must search through all dataset – very slow!
  - **Must use tricks like approximate nearest neighbor search**
  - Need to store all training data

Adapted from Dhruv Batra

kNN Decision Boundary

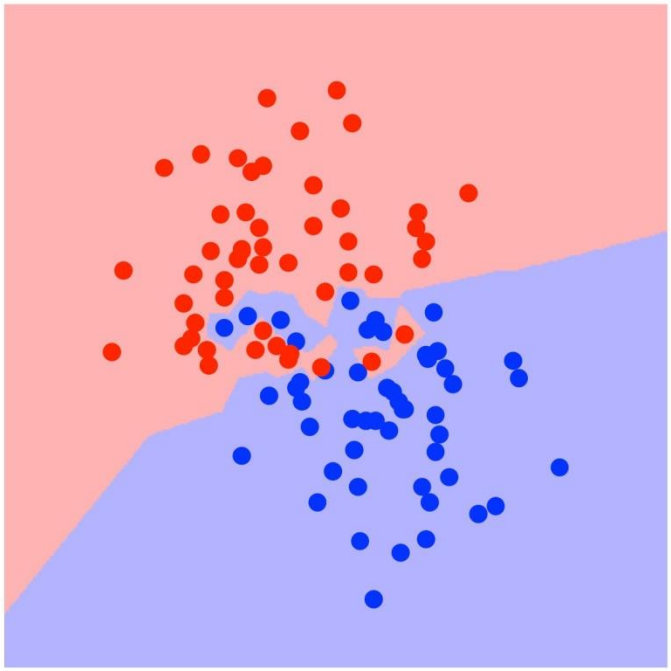
- Increasing k simplifies / complicates decision boundary

Slide credit: Alexander Ihler

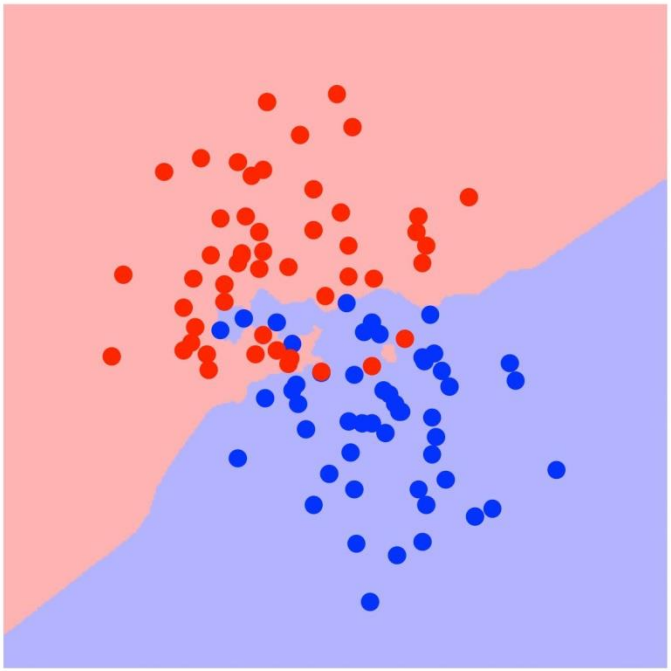
kNN Decision Boundary

- Increasing k “simplifies” decision boundary
  - Majority voting means less emphasis on individual points

K = 1



K = 3

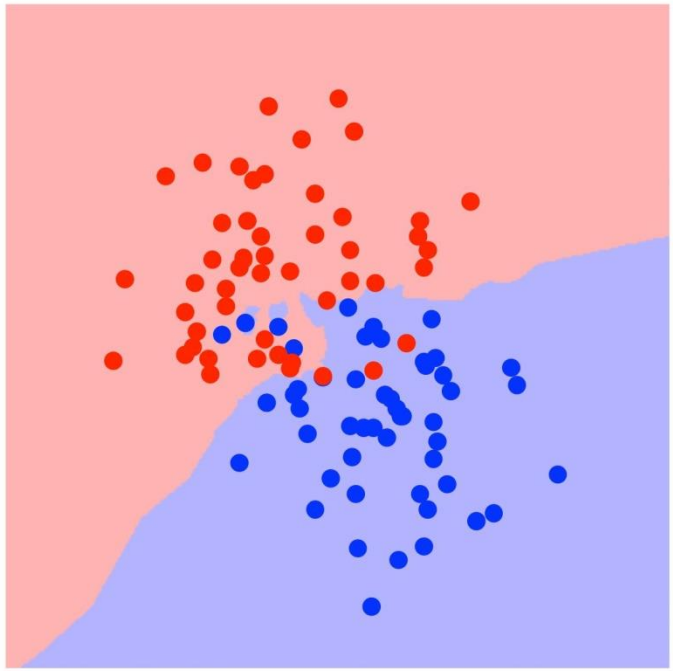


Slide credit: Alexander Ihler

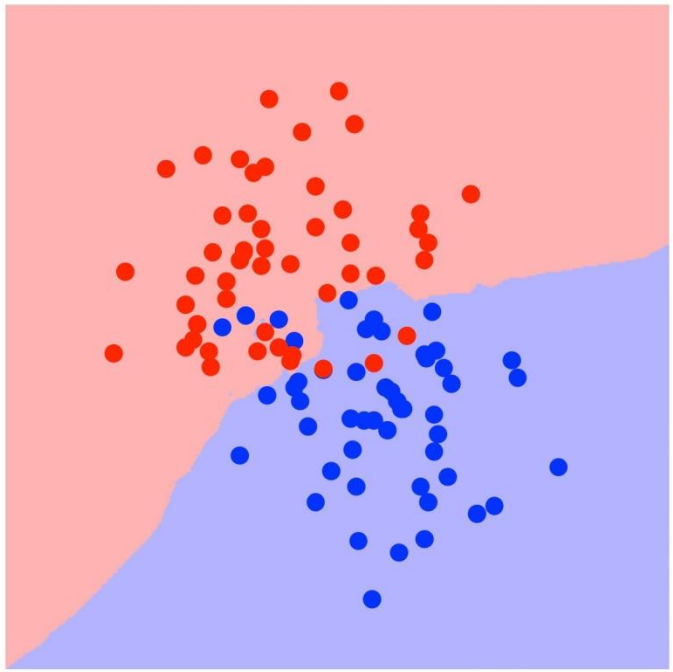
kNN Decision Boundary

- Increasing k “simplifies” decision boundary
  - Majority voting means less emphasis on individual points

K = 5



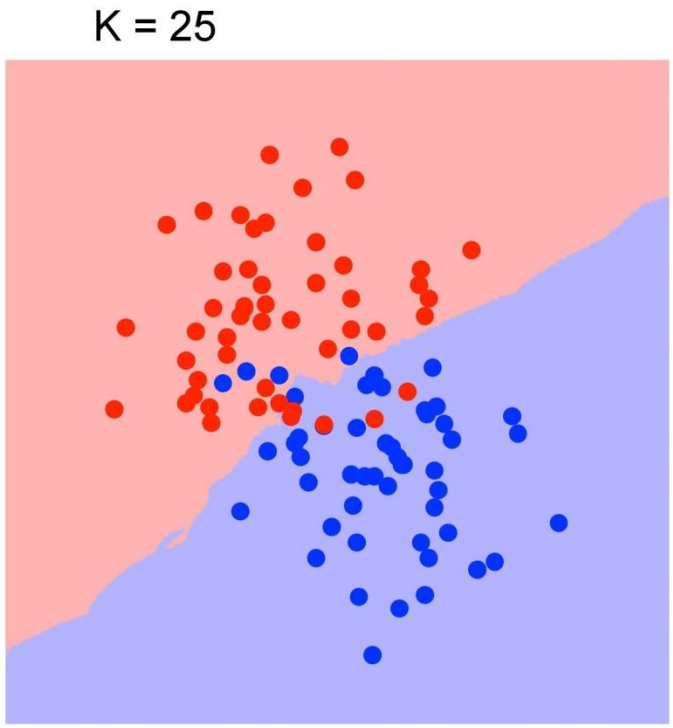
K = 7



Slide credit: Alexander Ihler

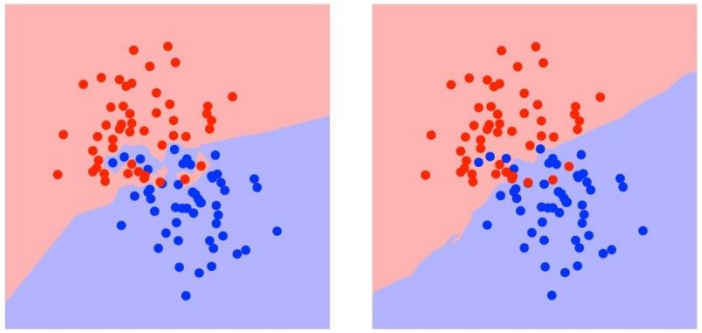
kNN Decision Boundary

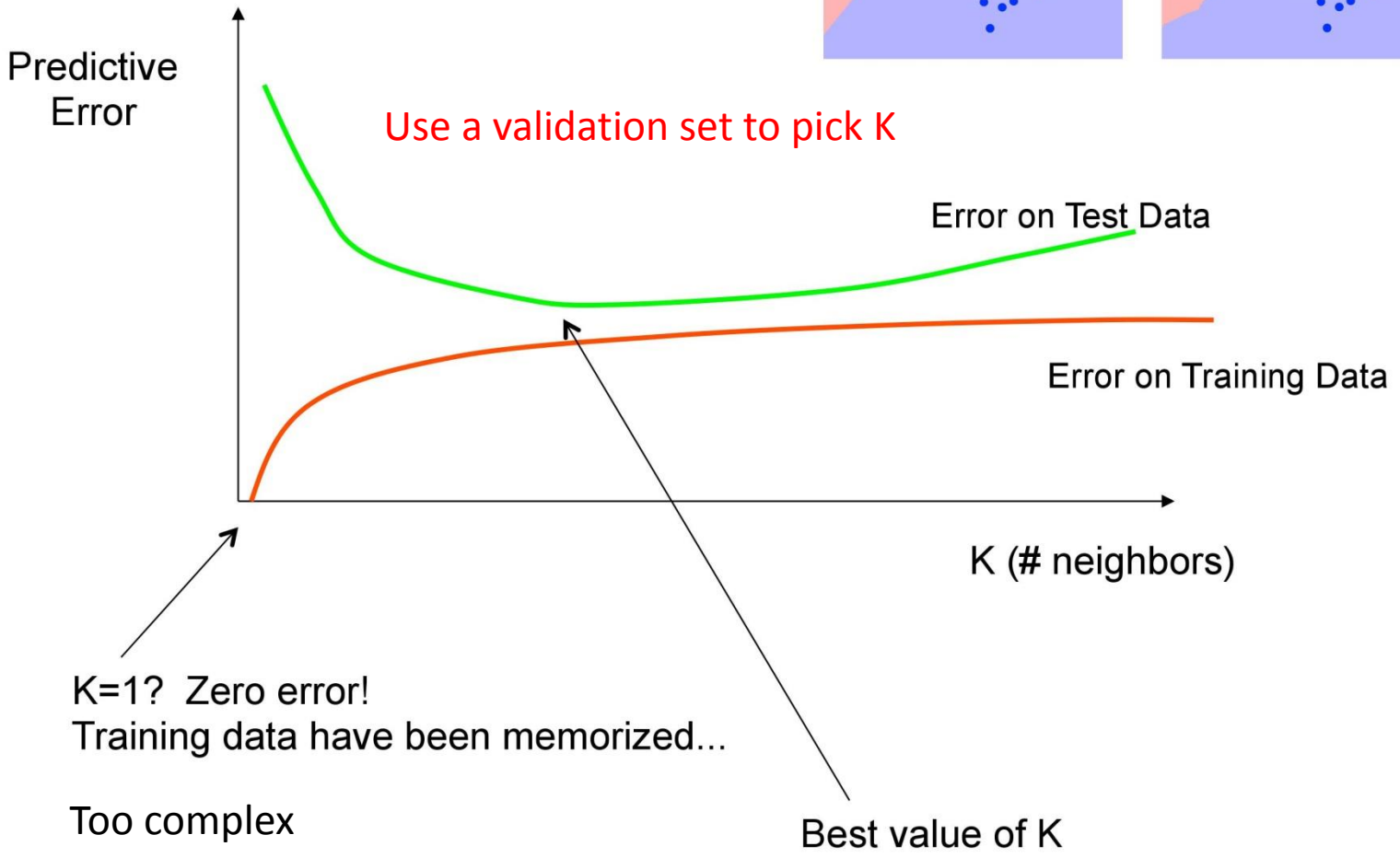
- Increasing k “simplifies” decision boundary
  - Majority voting means less emphasis on individual points



Slide credit: Alexander Ihler

Error rates and K





Slide credit: Alexander Ihler

Problems with Instance-Based Learning

- Too many features?
  - Doesn't work well if large number of irrelevant features, distances overwhelmed by noisy features
  - Distances become meaningless in high dimensions (the **curse of dimensionality**)

- What is the impact of the **value of K**?

- Expensive
  - No learning: most real work done during testing
  - For every test sample, must search through all dataset – very slow!
  - **Must use tricks like approximate nearest neighbor search**
  - Need to store all training data

Adapted from Dhruv Batra

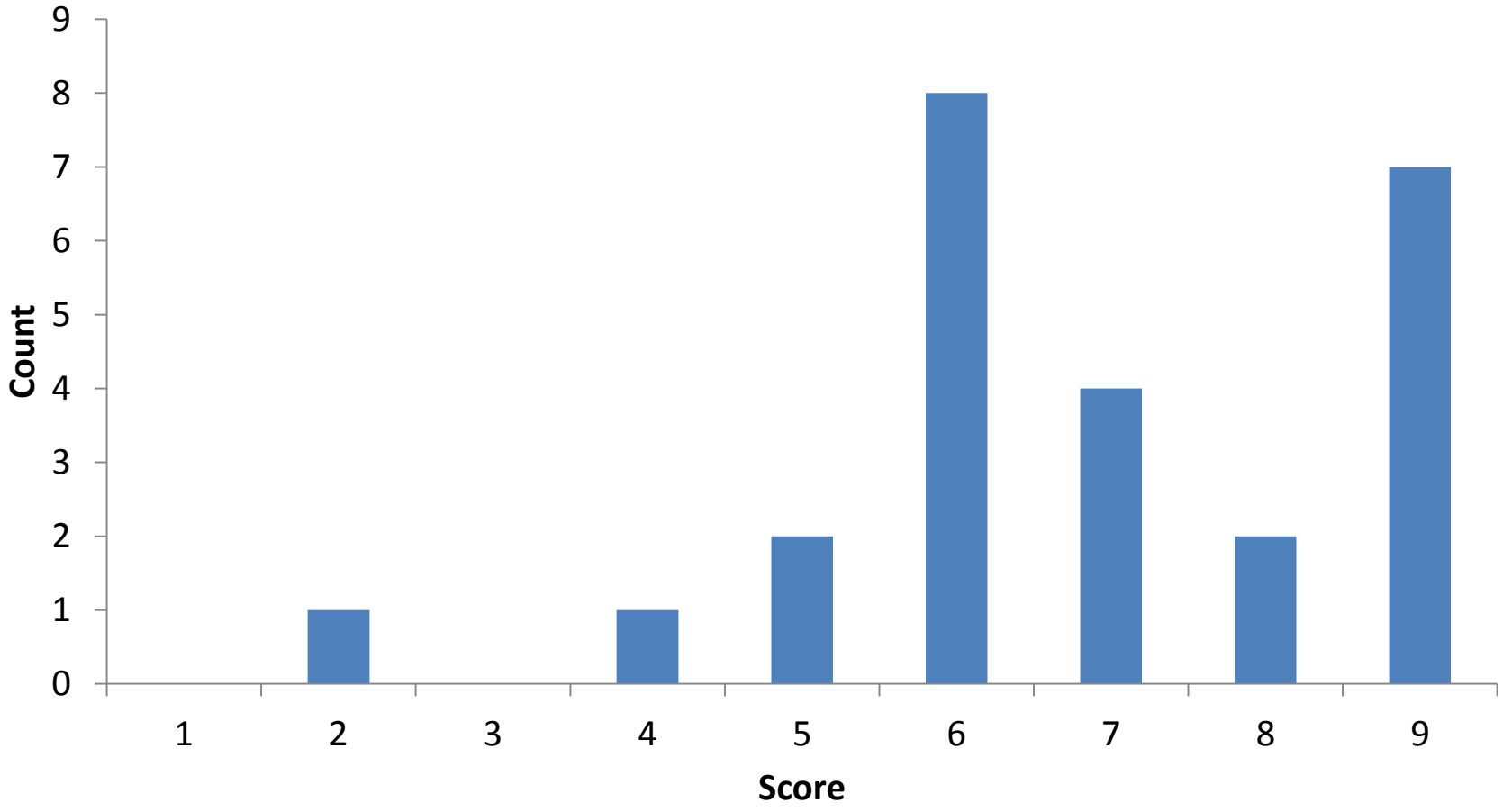
Approximate distance methods

- Build a balanced tree of data points (*kd*-trees), splitting data points along different dimensions
- Using tree, find “current best” guess of nearest neighbor
- Intelligently eliminate parts of the search space if they cannot contain a better “current best”
- Only search for neighbors up until some budget exhausted

Summary

- K-Nearest Neighbor is the most basic and simplest to implement classifier
- Cheap at training time, expensive at test time
- Unlike other methods we'll see later, naturally works for any number of classes
- Pick K through a validation set, use approximate methods for finding neighbors
- Success of classification depends on the amount of data and the meaningfulness of the distance function

Quiz 1



Review

- Today:
  - Linear algebra
  - PCA / dimensionality reduction
- Next time:
  - Mean shift vs k-means
  - Regularization and bias/variance

Other topics

- Hierarchical agglomerative clustering
  - Read my slides, can explain in office hours
- Graph cuts
  - Won’t be on any test, I can explain in office hours
- Lagrange multipliers
  - Will cover in more depth later if needed
- HW1 Part I
  - We’ll release solutions

Terminology

- Representation: The vector x_i for your data point i
- Dimensionality: the value of d in your 1xd vector representation x_i
- Let x_i^j be the j-th dimension of x_i , for $j = 1, \ldots, d$
- Usually the different x_i^j are independent, can think of them as responses to different "tests" applied to x_i

Terminology

- For example: In HW1 Part V, I was asking you to try different combinations of RGB and gradients
- This means you can have a *d*-dimensional representation where *d* can be anything from 1 to 5
- There are 5 “tests” total (R, G, B, G_x, G_y)

Linear Algebra Primer

Professor Fei-Fei Li

Stanford

Another, very in-depth linear algebra review from CS229 is available here:
http://cs229.stanford.edu/section/cs229-linalg.pdf
And a video discussion of linear algebra from EE263 is here (lectures 3 and 4):
http://see.stanford.edu/see/lecturelist.aspx?coll=17005383-19c6-49ed-9497-2ba8bfcfe5f6

Vectors and Matrices

- Vectors and matrices are just collections of ordered numbers that represent something: movements in space, scaling factors, word counts, movie ratings, pixel brightnesses, etc. We'll define some common uses and standard operations on them.

Vector

- A column vector $\mathbf{v} \in \mathbb{R}^{n \times 1}$ where

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

- A row vector $\mathbf{v}^T \in \mathbb{R}^{1 \times n}$ where

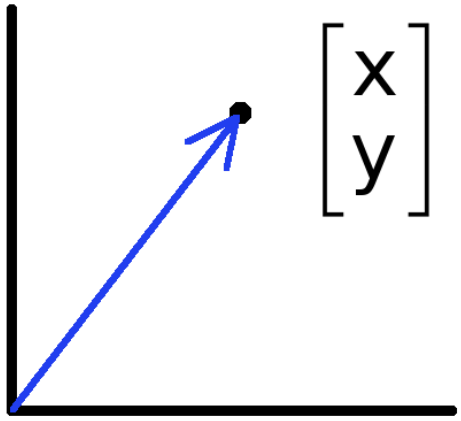
$$\mathbf{v}^T = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}$$

T denotes the transpose operation

Vector

- You'll want to keep track of the orientation of your vectors when programming in MATLAB.
- You can transpose a vector *V* in MATLAB by writing **V'**.

Vectors have two main uses



- Vectors can represent an offset in 2D or 3D space
- Points are just vectors from the origin

- Data can also be treated as a vector
- Such vectors don't have a geometric interpretation, but calculations like "distance" still have value

Matrix

- A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is an array of numbers with size $m \downarrow$ by $n \rightarrow$, i.e. m rows and n columns.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & & & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{bmatrix}$$

- If $m = n$, we say that $\mathbf{A}$ is square.

Matrix Operations

- Addition

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} a+1 & b+2 \\ c+3 & d+4 \end{bmatrix}$$

  - Can only add a matrix with matching dimensions, or a scalar.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + 7 = \begin{bmatrix} a+7 & b+7 \\ c+7 & d+7 \end{bmatrix}$$

- Scaling

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times 3 = \begin{bmatrix} 3a & 3b \\ 3c & 3d \end{bmatrix}$$

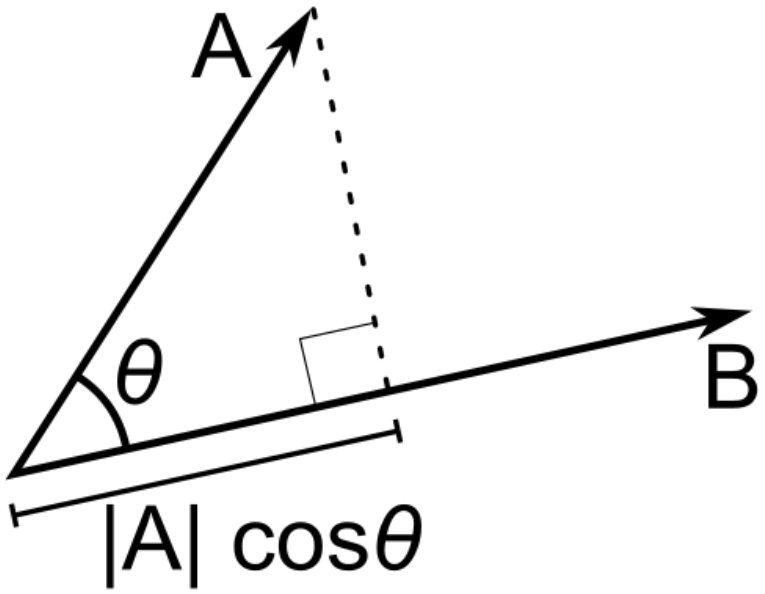
Matrix Operations

- Inner product (dot product) of vectors
  - Multiply corresponding entries of two vectors and add up the result
  - x·y is also |x||y|Cos(*the angle between x and y*)

$$\mathbf{x}^T\mathbf{y} = \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} x_i y_i \qquad \text{(scalar)}$$

Matrix Operations

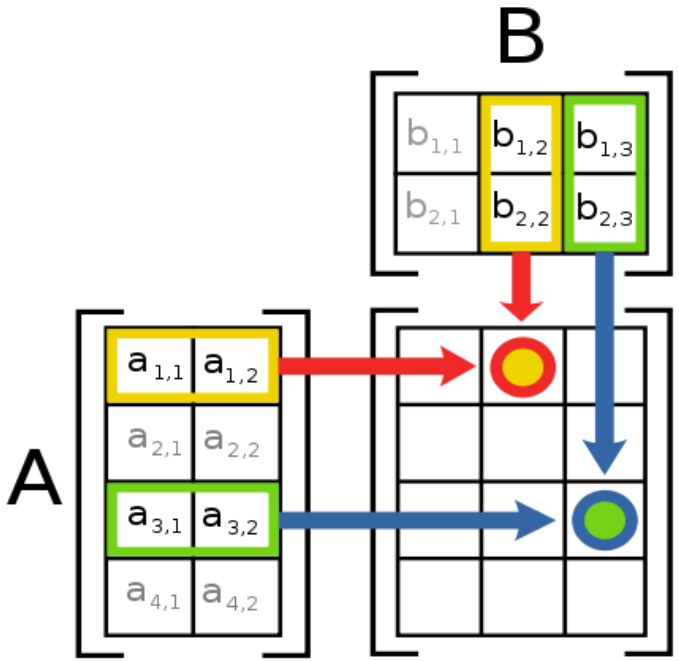
- Inner product (dot product) of vectors
  - If B is a unit vector, then A·B gives the length of A which lies in the direction of B (projection)



Matrix Operations

- Multiplication

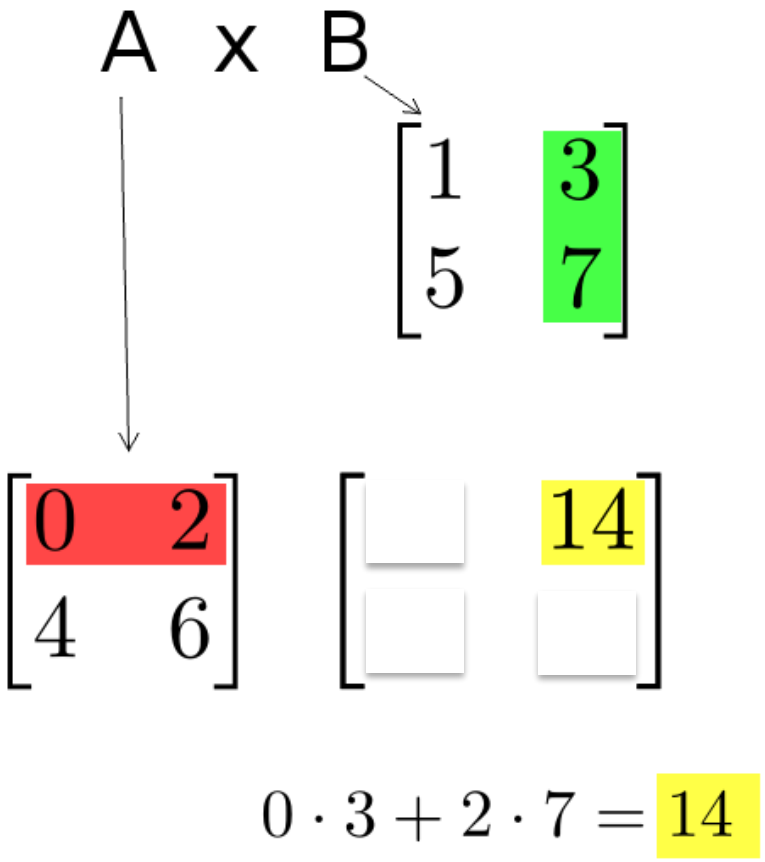
- The product AB is:



- Each entry in the result is (that row of A) dot product with (that column of B)

Matrix Operations

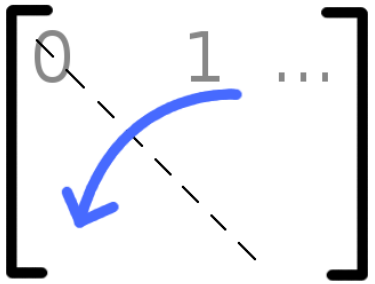
- Multiplication example:



– Each entry of the matrix product is made by taking the dot product of the corresponding row in the left matrix, with the corresponding column in the right one.

Matrix Operations

- Transpose – flip matrix, so row 1 becomes column 1



$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}^T = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix}$$

- A useful identity:

$$(ABC)^T = C^T B^T A^T$$

Special Matrices

- Identity matrix **I**
  - Square matrix, 1's along diagonal, 0's elsewhere
  - **I** · [another matrix] = [that matrix]

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Diagonal matrix
  - Square matrix with numbers along diagonal, 0's elsewhere
  - A diagonal · [another matrix] scales the rows of that matrix

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & 7 & 0 \\ 0 & 0 & 2.5 \end{bmatrix}$$

Special Matrices

- Symmetric matrix

$$\mathbf{A}^T = \mathbf{A}$$

$$\begin{bmatrix} 1 & 2 & 5 \\ 2 & 1 & 7 \\ 5 & 7 & 1 \end{bmatrix}$$

Inverse

- Given a matrix **A**, its inverse $\mathbf{A}^{-1}$ is a matrix such that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$
- E.g. $\begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}^{-1} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{3} \end{bmatrix}$
- Inverse does not always exist. If $\mathbf{A}^{-1}$ exists, **A** is *invertible* or *non-singular*. Otherwise, it's *singular*.

Pseudo-inverse

- Say you have the matrix equation AX=B, where A and B are known, and you want to solve for X
- You could use MATLAB to calculate the inverse and premultiply by it: $A^{-1}AX=A^{-1}B \rightarrow X=A^{-1}B$
- MATLAB command would be **inv(A)*B**
- But calculating the inverse for large matrices often brings problems with computer floating-point resolution, or your matrix might not even have an inverse. Fortunately, there are workarounds.
- Instead of taking an inverse, directly ask MATLAB to solve for X in AX=B, by typing **A\B**
- MATLAB will try several appropriate numerical methods (including the pseudoinverse if the inverse doesn't exist)
- MATLAB will return the value of X which solves the equation
  - If there is no exact solution, it will return the closest one
  - If there are many solutions, it will return the smallest one

Matrix Operations

- MATLAB example:

$$AX = B$$

$$A = \begin{bmatrix} 2 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

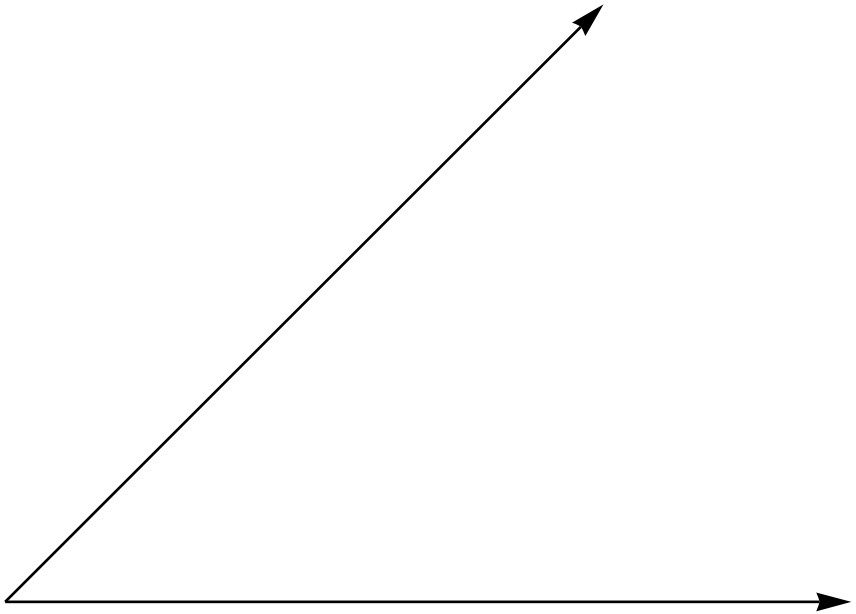
```
>> x = A\B
x =
    1.0000
   -0.5000
```

Linear independence

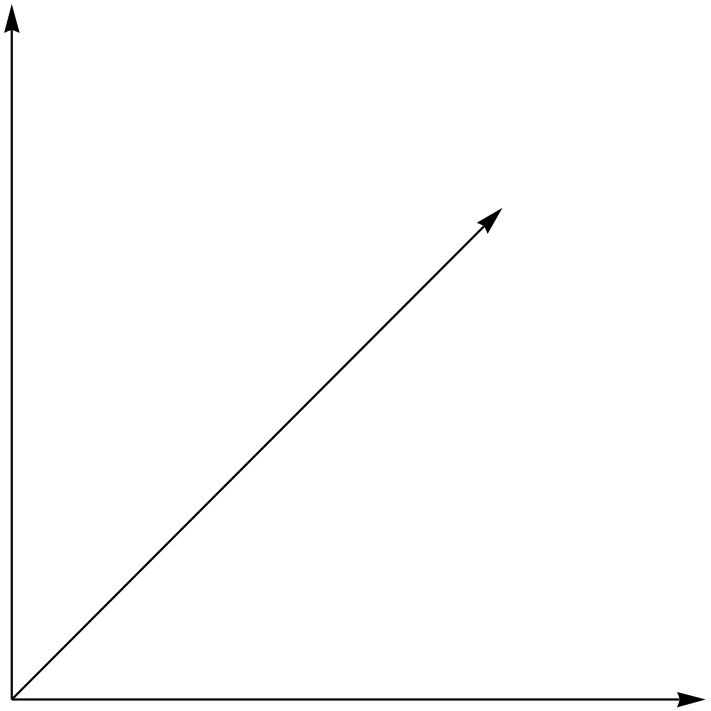
- Suppose we have a set of vectors $\mathbf{v_1}, \ldots, \mathbf{v_n}$
- If we can express $\mathbf{v}_1$ as a linear combination of the other vectors $\mathbf{v}_2 \ldots \mathbf{v}_n$, then $\mathbf{v}_1$ is linearly *dependent* on the other vectors.
  - The direction $\mathbf{v}_1$ can be expressed as a combination of the directions $\mathbf{v}_2 \ldots \mathbf{v}_n$. (E.g. $\mathbf{v}_1 = .7\ \mathbf{v}_2 - .7\ \mathbf{v}_4$)
- If no vector is linearly dependent on the rest of the set, the set is linearly *independent*.
  - Common case: a set of vectors $\mathbf{v_1}, \ldots, \mathbf{v_n}$ is always linearly independent if each vector is perpendicular to every other vector (and non-zero)

Linear independence

Linearly independent set



Not linearly independent



Matrix rank

- Column/row rank

$$\text{col-rank}(\mathbf{A}) = \text{ the maximum number of linearly independent column vectors of } \mathbf{A}$$
$$\text{row-rank}(\mathbf{A}) = \text{ the maximum number of linearly independent row vectors of } \mathbf{A}$$

- Column rank always equals row rank
- Matrix rank $\text{rank}(\mathbf{A}) \triangleq \text{col-rank}(\mathbf{A}) = \text{row-rank}(\mathbf{A})$
- If a matrix is not full rank, inverse doesn't exist
  - Inverse also doesn't exist for non-square matrices

Singular Value Decomposition (SVD)

- There are several computer algorithms that can "factor" a matrix, representing it as the product of some other matrices
- The most useful of these is the Singular Value Decomposition
- Represents any matrix **A** as a product of three matrices: $\mathbf{U\Sigma V^T}$
- MATLAB command: [U,S,V] = svd(A);

Singular Value Decomposition (SVD)

$$\mathbf{U\Sigma V^T = A}$$

- Where **U** and **V** are rotation matrices, and **Σ** is a scaling matrix. For example:

$$\underset{U}{\begin{bmatrix} -.40 & .916 \\ .916 & .40 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 5.39 & 0 \\ 0 & 3.154 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.05 & .999 \\ .999 & .05 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 3 & -2 \\ 1 & 5 \end{bmatrix}}$$

Singular Value Decomposition (SVD)

- In general, if **A** is *m* x *n*, then **U** will be *m* x *m*, **Σ** will be *m* x *n*, and $\mathbf{V}^T$ will be *n* x *n*.

$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

Singular Value Decomposition (SVD)

- **U** and **V** are always rotation matrices.
  - Geometric rotation may not be an applicable concept, depending on the matrix. So we call them “unitary” matrices – each column is a unit vector.
- **Σ** is a diagonal matrix
  - The number of nonzero entries = rank of **A**
  - The algorithm always sorts the entries high to low

$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

Singular Value Decomposition (SVD)

$$\mathbf{M} = \mathbf{U}\Sigma\mathbf{V}^T$$

Illustration from Wikipedia

SVD Applications

- We've discussed SVD in terms of geometric transformation matrices
- But SVD of a data matrix can also be very useful
- To understand this, we'll look at a less geometric interpretation of what SVD is doing

SVD Applications

$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

- Look at how the multiplication works out, left to right:
- Column 1 of **U** gets scaled by the first value from

$$\underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}}$$

- The resulting vector gets scaled by row 1 of $\mathbf{V^T}$ to produce a contribution to the columns of **A**

SVD Applications

$$\underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \rightarrow \underset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

$$+ \underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \rightarrow \underset{A_{partial}}{\begin{bmatrix} -.6 & -.1 & .4 \\ .2 & 0 & -.2 \end{bmatrix}}$$

$$= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- Each product of (*column i of* ***U***)·(*value i from* ***Σ***)·(*row i of* $\boldsymbol{V^T}$) produces a component of the final **A**.

SVD Applications

$$U\Sigma \begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix} \times V^T \begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix} \quad A_{partial} \begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}$$

$$A \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$U\Sigma \begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix} \times V^T \begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix} \quad A_{partial} \begin{bmatrix} -.6 & -.1 & .4 \\ .2 & 0 & -.2 \end{bmatrix}$$

- We're building **A** as a linear combination of the columns of ***U***
- Using all columns of ***U***, we'll rebuild the original matrix perfectly
- But, in real-world data, often we can just use the first few columns of ***U*** and we'll get something close (e.g. the first $\boldsymbol{A_{partial}}$, above)

SVD Applications

$$U\Sigma \begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix} \times V^T \begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix} \quad A_{partial} \begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}$$

$$A \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$U\Sigma \begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix} \times V^T \begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix} \quad A_{partial} \begin{bmatrix} -.6 & -.1 & .4 \\ .2 & 0 & -.2 \end{bmatrix}$$

- We can call those first few columns of ***U*** the *Principal Components* of the data
- They show the major patterns that can be added to produce the columns of the original matrix
- The rows of $\mathbf{V^T}$ show how the *principal components* are mixed to produce the columns of the matrix

SVD Applications

$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

We can look at **Σ** to see that the first column has a large effect

while the second column has a much smaller effect in this example

Principal Component Analysis

$$\underbrace{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}}_{U\Sigma} \times \underbrace{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}}_{V^T} \quad \underbrace{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}_{A_{partial}}$$

- Remember, columns of ***U*** are the *Principal Components* of the data: the major patterns that can be added to produce the columns of the original matrix
- One use of this is to construct a matrix where each column is a separate data sample
- Run SVD on that matrix, and look at the first few columns of ***U*** to see patterns that are common among the columns
- This is called *Principal Component Analysis* (or PCA) of the data samples

Principal Component Analysis

$$\underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \underset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

- Often, raw data samples have a lot of redundancy and patterns
- PCA can allow you to represent data samples as weights on the principal components, rather than using the original raw form of the data
- By representing each sample as just those weights, you can represent just the "meat" of what's different between samples
- This minimal representation makes machine learning and other algorithms much more efficient

Addendum: How is SVD computed?

- For this class: tell MATLAB to do it. Use the result.
- But, if you're interested, one computer algorithm to do it makes use of Eigenvectors
  - The following material is presented to make SVD less of a "magical black box." But you will do fine in this class if you treat SVD as a magical black box, as long as you remember its properties from the previous slides.

Eigenvector definition

- Suppose we have a square matrix **A**. We can solve for vector x and scalar λ such that Ax= λx
- In other words, find vectors where, if we transform them with **A**, the only effect is to scale them with no change in direction.
- These vectors are called eigenvectors (German for "self vector" of the matrix), and the scaling factors λ are called eigenvalues
- An *m* x *m* matrix will have ≤ *m* eigenvectors where λ is nonzero

Finding eigenvectors

- Computers can find an x such that $Ax = \lambda x$ using this iterative algorithm:
  - x=random unit vector
  - while(x hasn't converged)
    - x=Ax
    - normalize x
- x will quickly converge to an eigenvector
- Some simple modifications will let this algorithm find all eigenvectors

Finding SVD

- Eigenvectors are for square matrices, but SVD is for all matrices
- To do svd(A), computers can do this:
  - Take eigenvectors of AA^T (matrix is always square).
    - These eigenvectors are the columns of **U**.
    - Square root of eigen*values* are the singular values (the entries of **Σ**).
  - Take eigenvectors of A^TA (matrix is always square).
    - These eigenvectors are columns of **V** (or rows of $\mathbf{V}^T$)

Finding SVD

- Moral of the story: SVD is fast, even for large matrices
- It's useful for a lot of stuff
- There are also other algorithms to compute SVD or part of the SVD
  - MATLAB's svd() command has options to efficiently compute only what you need, if performance becomes an issue

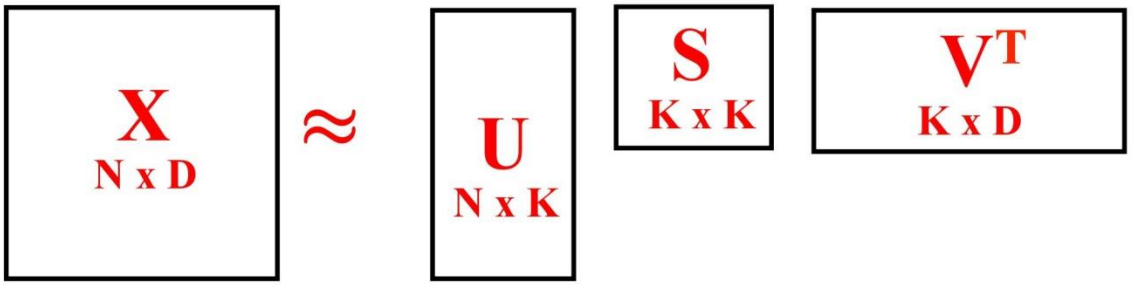
A detailed geometric explanation of SVD is here:
http://www.ams.org/samplings/feature-column/fcarc-svd

Singular Value Decomposition

- Alternative method to calculate (still subtract mean 1st)
- Decompose $X = U\ S\ V^T$
  - Orthogonal: $X^T X = V\ S\ S\ V^T = V\ D\ V^T$
  - $X\ X^T = U\ S\ S\ U^T = U\ D\ U^T$

- U*S matrix provides coefficients
  - Example $x_i = (U_{i,1}\ S_{11})v_1 + (U_{i,2}\ S_{22})v_2 + \ldots$

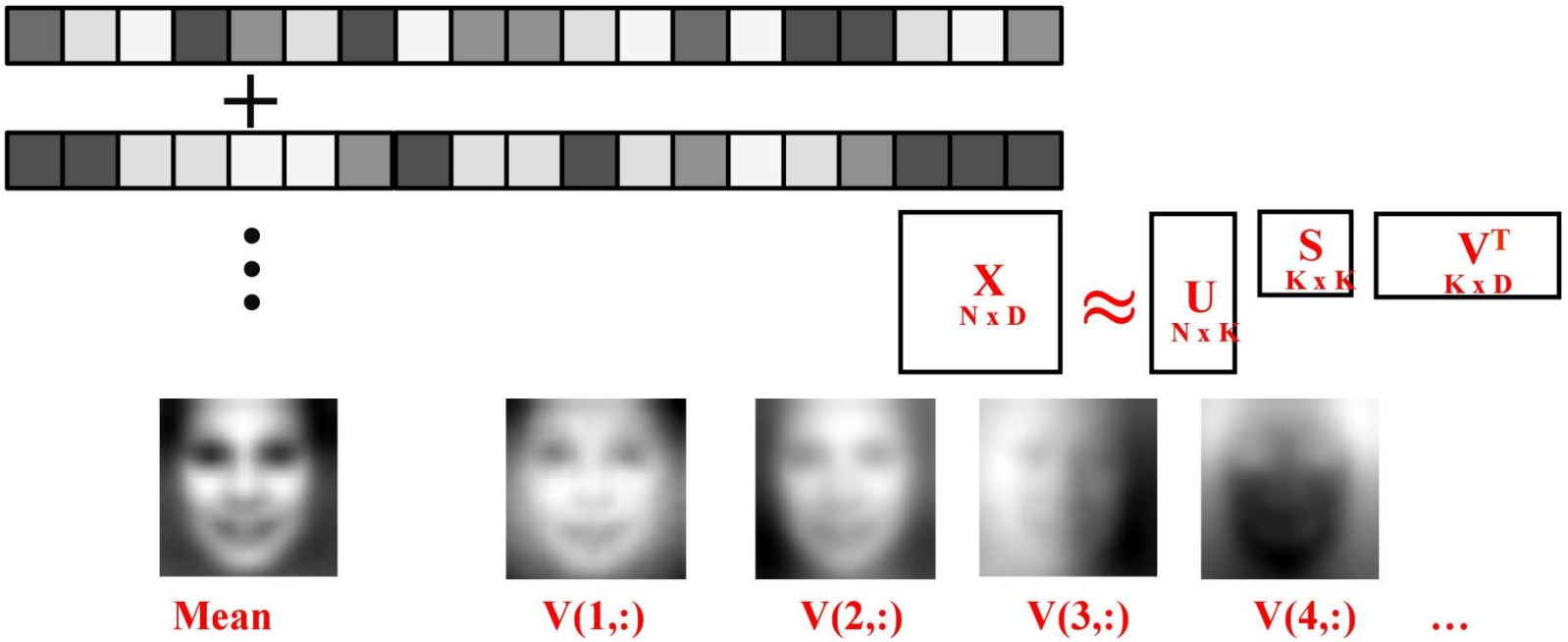
- Gives the least-squares approximation to X of this form



Slide credit: Alexander Ihler

“Eigen-faces”

- “Eigen-X” = represent X using PCA
- Ex: Viola Jones data set
  - 24x24 images of faces = 576 dimensional measurements
  - Take first K PCA components



Slide credit: Alexander Ihler