# CPSC 340:
# Machine Learning and Data Mining

Fun Examples

(Bonus Lecture)

Summer 2021

# In This Bonus Lecture

- Regression-versions of classifiers (10 minutes)
- Recommender Systems (20 minutes)
- Games (20 minutes)

Coming Up Next

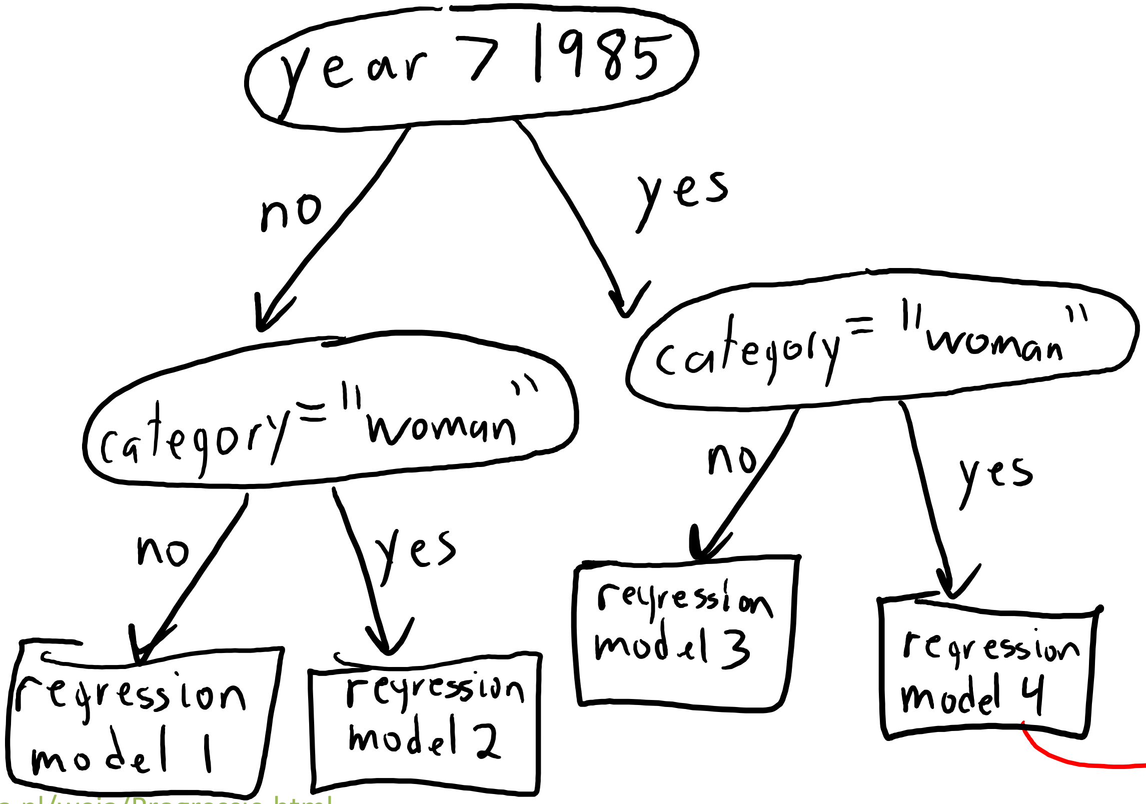
# REGRESSION-VERSIONS OF CLASSIFIERS WE'VE COVERED

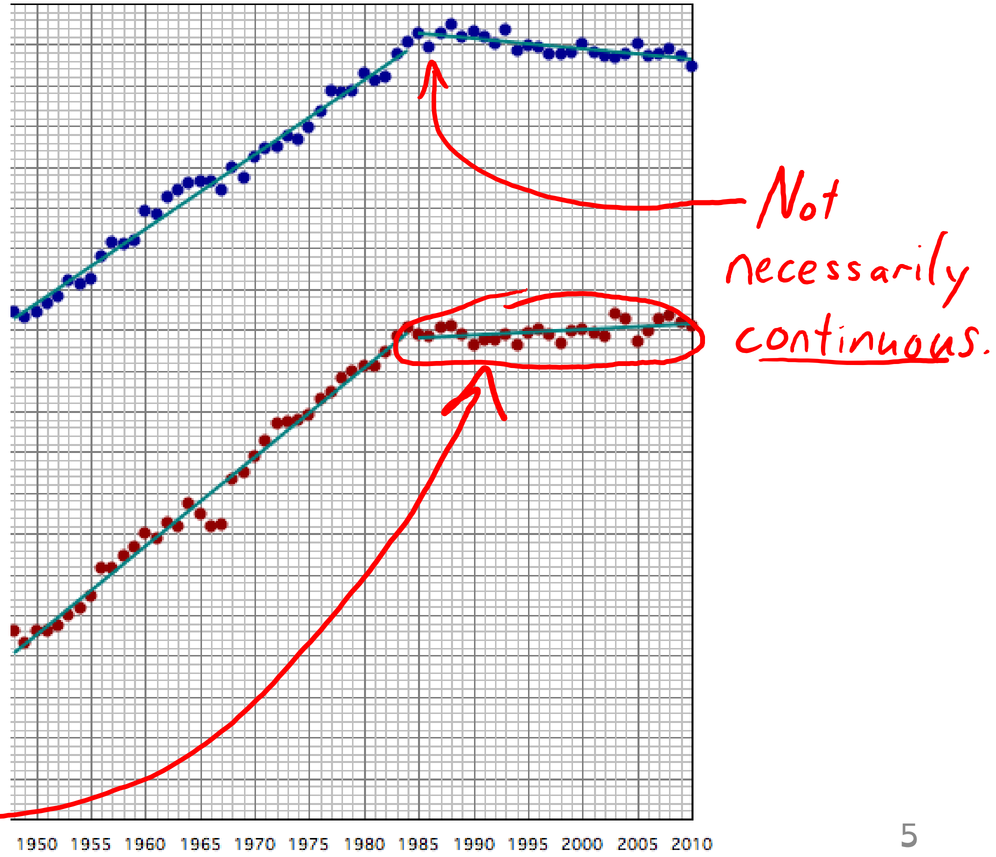
# Adapting Counting/Distance-Based Methods

- We can adapt our classification methods to perform regression:

# Adapting Counting/Distance-Based Methods

- We can adapt our classification methods to perform regression:
  - Regression tree: tree with mean value or linear regression at leaves.

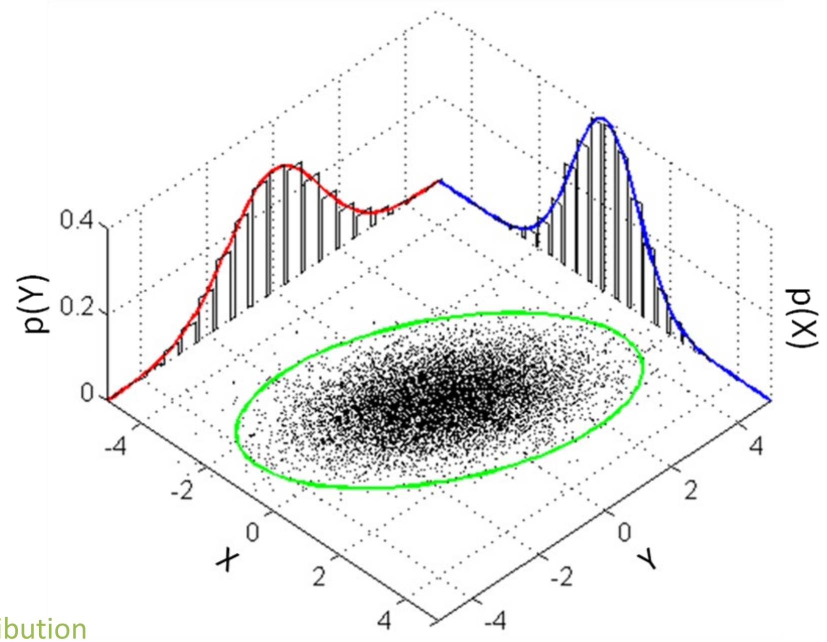



http://www.at-a-lanta.nl/weia/Progressie.html

# Adapting Counting/Distance-Based Methods

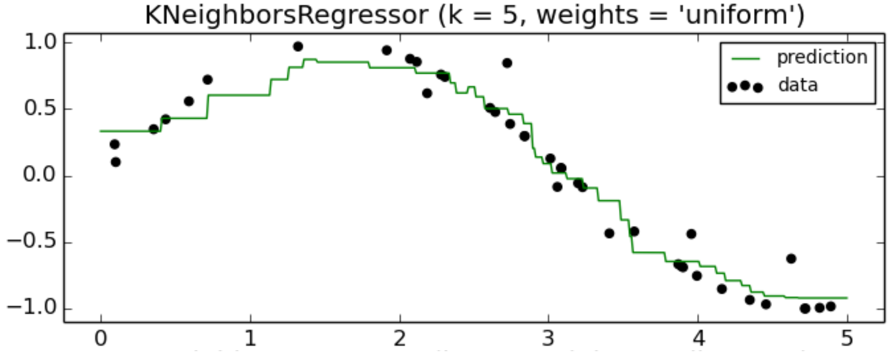
- We can adapt our classification methods to perform regression:
  - Regression tree: tree with mean value or linear regression at leaves.
  - Probabilistic models: fit $p(x_i \mid y_i)$ and $p(y_i)$ with Gaussian or other model.
    - Take CPSC 440/540.




https://en.wikipedia.org/wiki/Multivariate_normal_distribution

# Adapting Counting/Distance-Based Methods

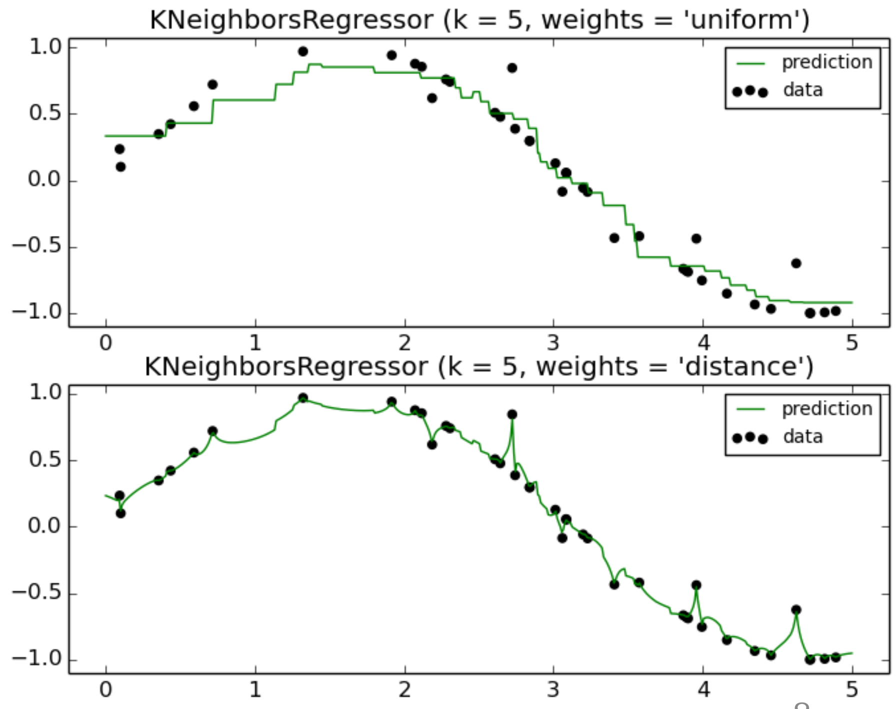
- We can adapt our classification methods to perform regression:
  - Regression tree: tree with mean value or linear regression at leaves.
  - Probabilistic models: fit $p(x_i \mid y_i)$ and $p(y_i)$ with Gaussian or other model.
  - Non-parametric models:
    - KNN regression:
      - Find 'k' nearest neighbours of $\tilde{x}_i$.
      - Return the mean of the corresponding $y_i$.




http://scikit-learn.org/stable/modules/neighbors.html

# Adapting Counting/Distance-Based Methods

- We can adapt our classification methods to perform regression:
  - Regression tree: tree with mean value or linear regression at leaves.
  - Probabilistic models: fit $p(x_i \mid y_i)$ and $p(y_i)$ with Gaussian or other model.
  - Non-parametric models:
    - KNN regression.
    - Could be weighted by distance.
      - Close points 'j' get more "weight" $w_{ij}$.

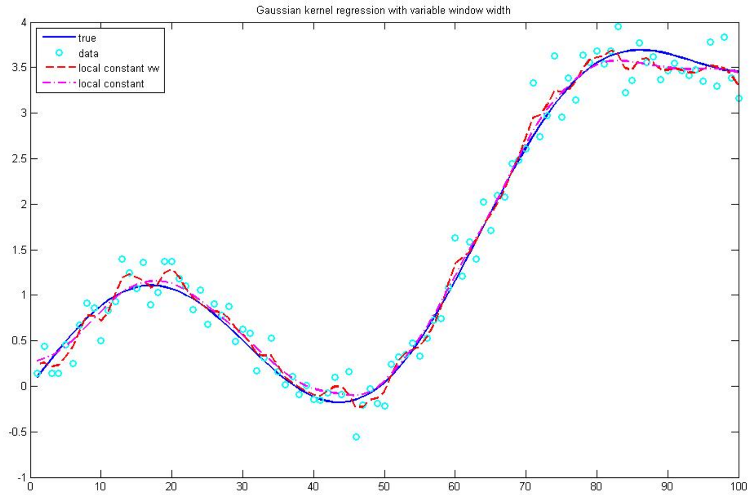



http://scikit-learn.org/stable/modules/neighbors.html

# Adapting Counting/Distance-Based Methods

- We can adapt our classification methods to perform regression:
  - Regression tree: tree with mean value or linear regression at leaves.
  - Probabilistic models: fit $p(x_i \mid y_i)$ and $p(y_i)$ with Gaussian or other model.
  - Non-parametric models:
    - KNN regression.
    - Could be weighted by distance.
    - 'Nadaraya-Waston': weight *all* $y_i$ by distance to $x_i$.

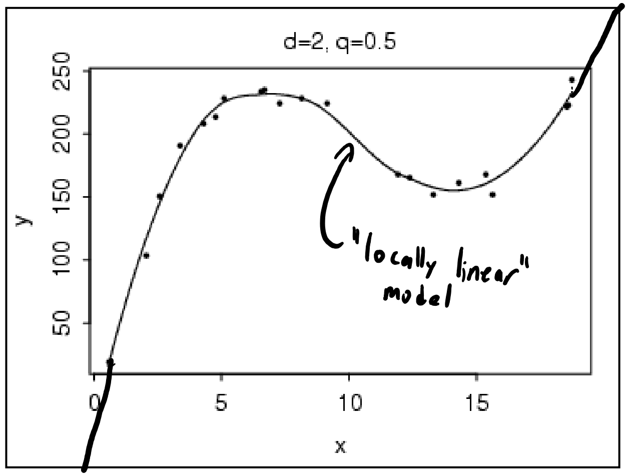
$$\hat{y}_i = \frac{\sum_{j=1}^{n} v_{ij} y_j}{\sum_{j=1}^{n} v_{ij}}$$




http://www.mathworks.com/matlabcentral/fileexchange/35316-kernel-regression-with-variable-window-width/content/ksr_vw.m

# Adapting Counting/Distance-Based Methods

- We can adapt our classification methods to perform regression:
  - Regression tree: tree with mean value or linear regression at leaves.
  - Probabilistic models: fit $p(x_i \mid y_i)$ and $p(y_i)$ with Gaussian or other model.
  - Non-parametric models:
    - KNN regression.
    - Could be weighted by distance.
    - 'Nadaraya-Waston': weight *all* $y_i$ by distance to $x_i$.
    - 'Locally linear regression': for each $x_i$, fit a linear model weighted by distance.
      (Better than KNN and NW at boundaries.)




http://www.itl.nist.gov/div898/handbook/pmd/section4/pmd423.htm

# Adapting Counting/Distance-Based Methods

- We can adapt our classification methods to perform regression:
  - Regression tree: tree with mean value or linear regression at leaves.
  - Probabilistic models: fit $p(x_i \mid y_i)$ and $p(y_i)$ with Gaussian or other model.
  - Non-parametric models:
    - KNN regression.
    - Could be weighted by distance.
    - 'Nadaraya-Waston': weight *all* $y_i$ by distance to $x_i$.
    - 'Locally linear regression': for each $x_i$, fit a linear model weighted by distance. (Better than KNN and NW at boundaries.)
  - Ensemble methods:
    - Can improve performance by averaging predictions across regression models.

# Adapting Counting/Distance-Based Methods

- We can adapt our classification methods to perform regression.

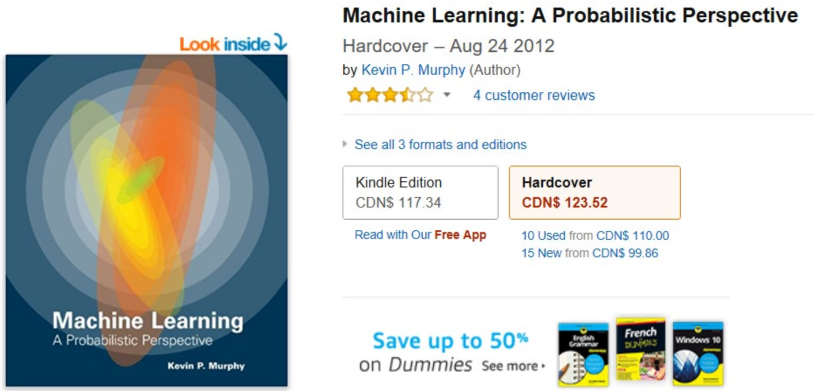
- Applications:
  - Regression forests for fluid simulation:
    - https://www.youtube.com/watch?v=kGB7Wd9CudA
  - KNN for image completion:
    - http://graphics.cs.cmu.edu/projects/scene-completion
    - Combined with "graph cuts" and "Poisson blending".
    - See also "PatchMatch": https://vimeo.com/5024379
  - KNN regression for "voice photoshop":
    - https://www.youtube.com/watch?v=I3l4XLZ59iw
    - Combined with "dynamic time warping" and "Poisson blending".

Coming Up Next

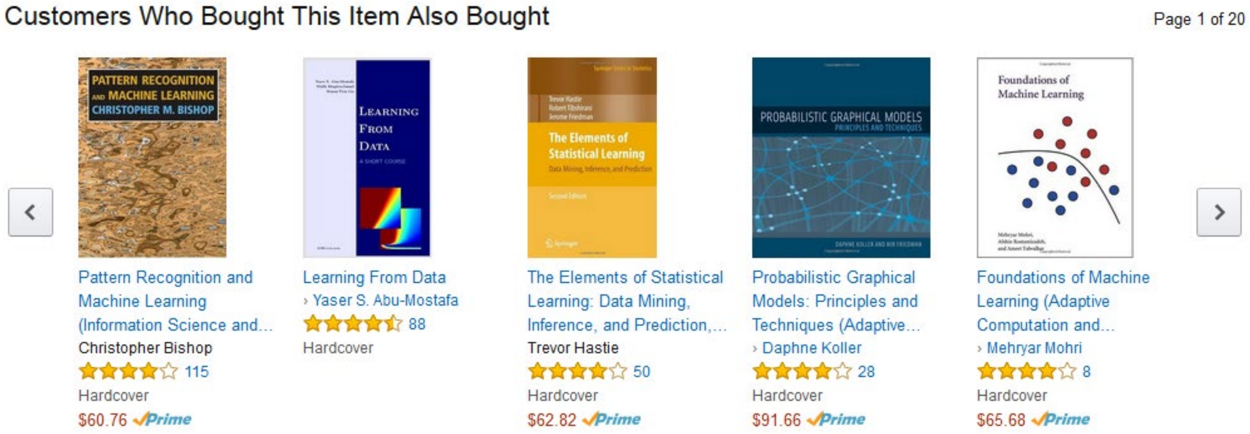
# RECOMMENDER SYSTEMS

# Motivation: Product Recommendation

- A customer comes to your website looking to buy at item:



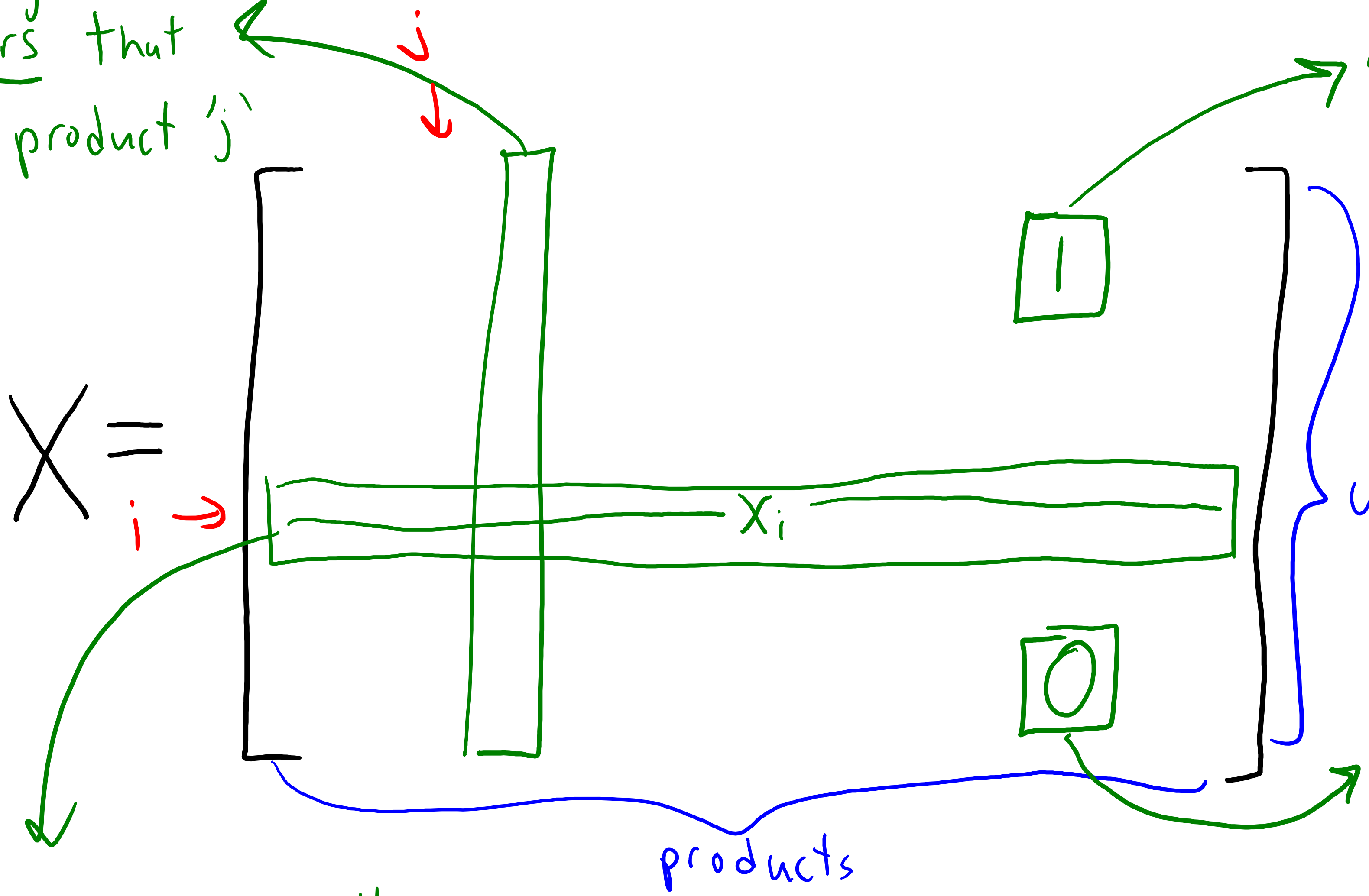

- You want to find similar items that they might also buy:



# User-Product Matrix

Column $x^j$ gives all users that bought product 'j'

$X_{ij} = 1$ means user 'i' bought item 'j'!

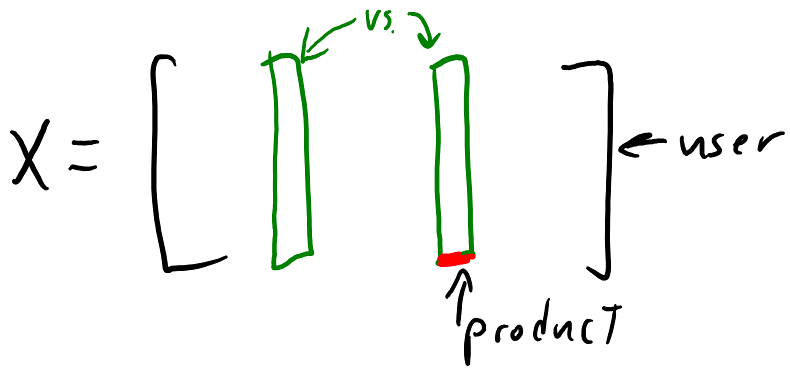



$x_{ij} = 0$ means user 'i' has not buy item 'j'

Row $x_i$ gives all items bought by user 'i'.

# Amazon Product Recommendation

- Amazon product recommendation method:




- Return the KNNs across columns.
  - Find 'j' values minimizing $\|x^i - x^j\|$.
  - Products that were bought by similar sets of users.

- But first divide each column by its norm, $x^i/\|x^i\|$.
  - This is called normalization.

Q: Why is normalization helpful here?

# Amazon Product Recommendation

- Consider this user-item matrix:

$$
X = \begin{array}{c} \\ \text{John} \\ \text{Paul} \\ \text{George} \\ \text{Ringo} \\ \text{Yoko} \end{array}
\begin{array}{c} \begin{array}{cccccc} \text{Product 1} & \text{Product 2} & \text{Product 3} & \text{Product 4} & \text{Product 5} & \text{Product 6} \end{array} \\
\left[\begin{array}{cccccc}
1 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 0
\end{array}\right] \end{array}
$$

- Product 1 is most similar to Product 3 (bought by lots of people).
- Product 2 is most similar to Product 4 (also bought by John and Yoko).
- Product 3 is equally similar to Products 1, 5, and 6.
  - Does not take into account that Product 1 is more popular than 5 and 6.

# Amazon Product Recommendation

- Consider this user-item matrix (normalized):

$$X = \begin{array}{c} \\ \text{John} \\ \text{Paul} \\ \text{George} \\ \text{Ringo} \\ \text{Yoko} \end{array} \begin{array}{c} \begin{array}{cccccc} \text{Product 1} & \text{Product 2} & \text{Product 3} & \text{Product 4} & \text{Product 5} & \text{Product 6} \end{array} \\ \left[\begin{array}{cccccc} 1/\sqrt{5} & 1/\sqrt{2} & 1/\sqrt{4} & 1/\sqrt{2} & 0 & 1/\sqrt{3} \\ 1/\sqrt{5} & 0 & 1/\sqrt{4} & 0 & 1/\sqrt{3} & 0 \\ 1/\sqrt{5} & 0 & 1/\sqrt{4} & 0 & 1/\sqrt{3} & 1/\sqrt{3} \\ 1/\sqrt{5} & 0 & 1/\sqrt{4} & 0 & 1/\sqrt{3} & 1/\sqrt{3} \\ 1/\sqrt{5} & 1/\sqrt{2} & 0 & 1/\sqrt{2} & 0 & 0 \end{array}\right] \end{array}$$

- Product 1 is most similar to Product 3 (bought by lots of people).
- Product 2 is most similar to Product 4 (also bought by John and Yoko).
- Product 3 is most similar to Product 1.
  - Normalization means it prefers the popular items.

# Cost of Finding Nearest Neighbours

- With 'n' users and 'd' products, finding KNNs for one item costs O(nd).
  - Not feasible if 'n' and 'd' are in the millions+.

- It's faster if the user-product matrix is sparse: O(z) for z non-zeroes.
  - But 'z' is still enormous in the Amazon example.

# Closest-Point Problems

- We've seen a lot of "closest point" problems:
  - K-nearest neighbours classification.
  - K-means clustering.
  - Density-based clustering.
  - Hierarchical clustering.
  - KNN-based outlier detection.
  - Outlierness ratio.
  - Amazon product recommendation.
- How can we possibly apply these to Amazon-sized datasets?

# But first the easy case: "Memorize the Answers"

- Easy case: you have a limited number of possible test examples.
  - E.g., you will always choose an existing product (not arbitrary features).

- In this case, just memorize the answers:
  - For each test example, compute all KNNs and store pointers to answers.
  - At test time, just return a set of pointers to the answers.

for each value of k, massive pre-computes, fast runtime

- The answers are called an inverted index, queries now cost O(k).
  - Needs an extra O(nk) storage, which is fine for small 'k'.

cost of popularity k nearest examples as output.

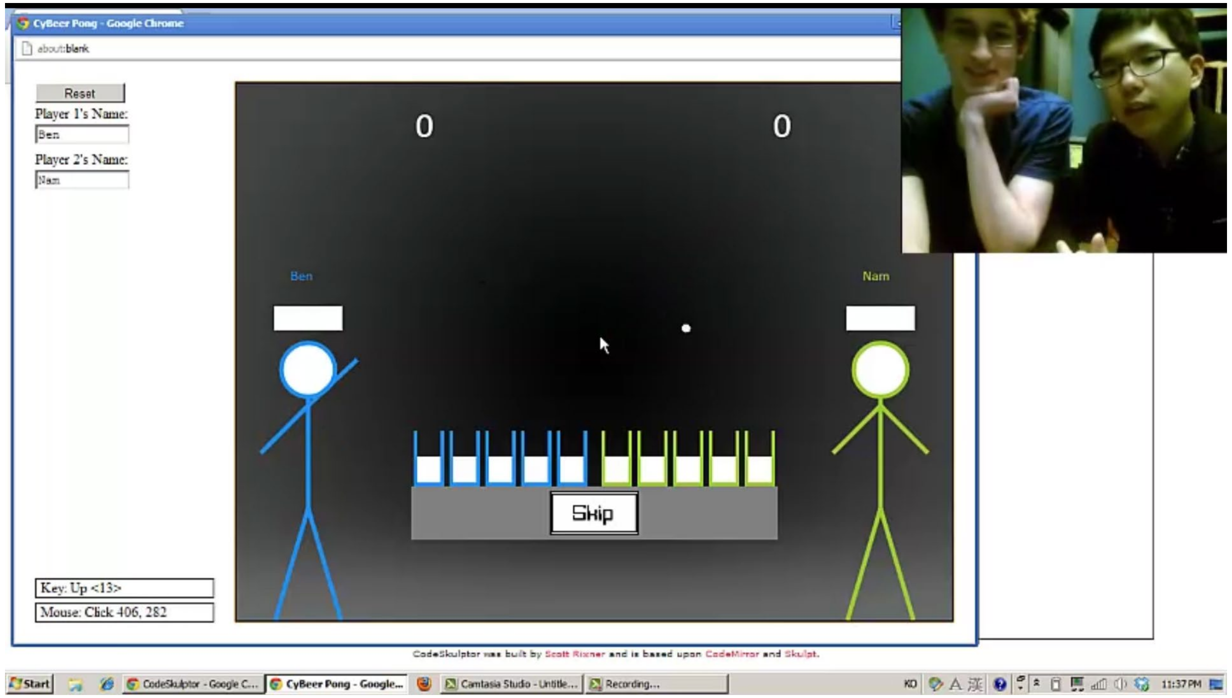
Q: What if we had continuous features?

Coming Up Next

# GRID-BASED PRUNING

# "Grid-Based Pruning"

- A classic method for fast collision detection in physics simulation
- I have 1 million objects. Are objects 1 and 2 running into each other?




Frosh Nam Hee

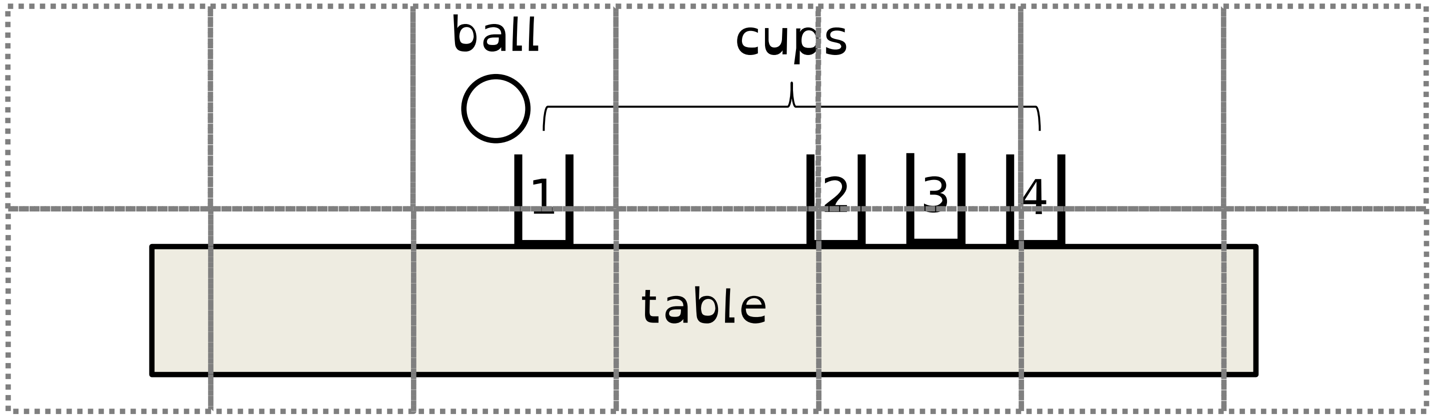
"CyBeer Pong"

- Expensive: check all pairs in 'n' objects ($O(n^2)$) and check their positions.

Q: Can we avoid unnecessary checks?

# Grid-Base Pruning for Collisions

- Smarter collision detection: check for "rough" distances first

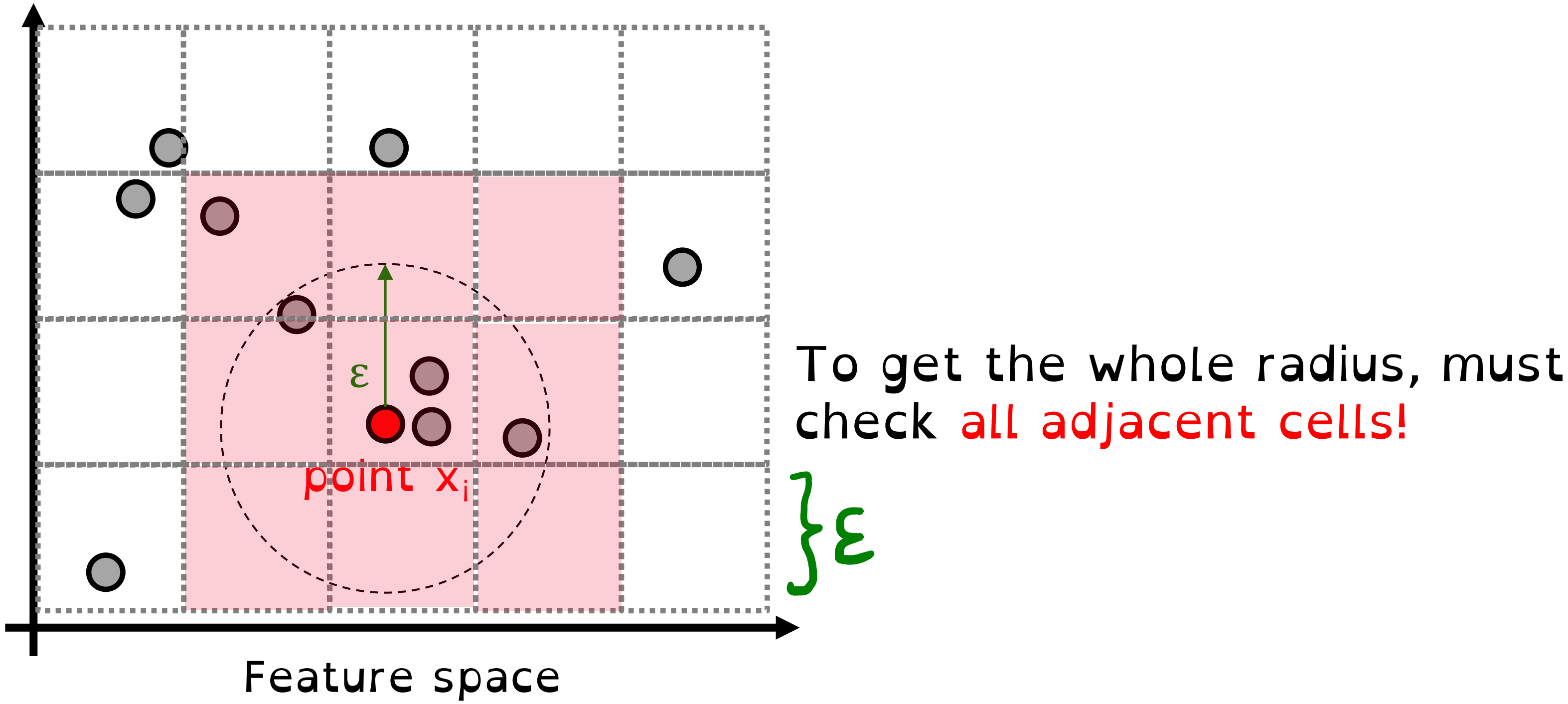



Q: Do we need to check ball vs. every cup?

- Idea: organize space into a coarse "grid" and check only cups within same cell
  - Instance of spatial discretization
- Still $O(n^2)$ checks in worst case, but works well in practice

# Grid-Based Pruning

- Instead of collision detection, let's find examples within L2-distance of 'ε' of point $x_i$.

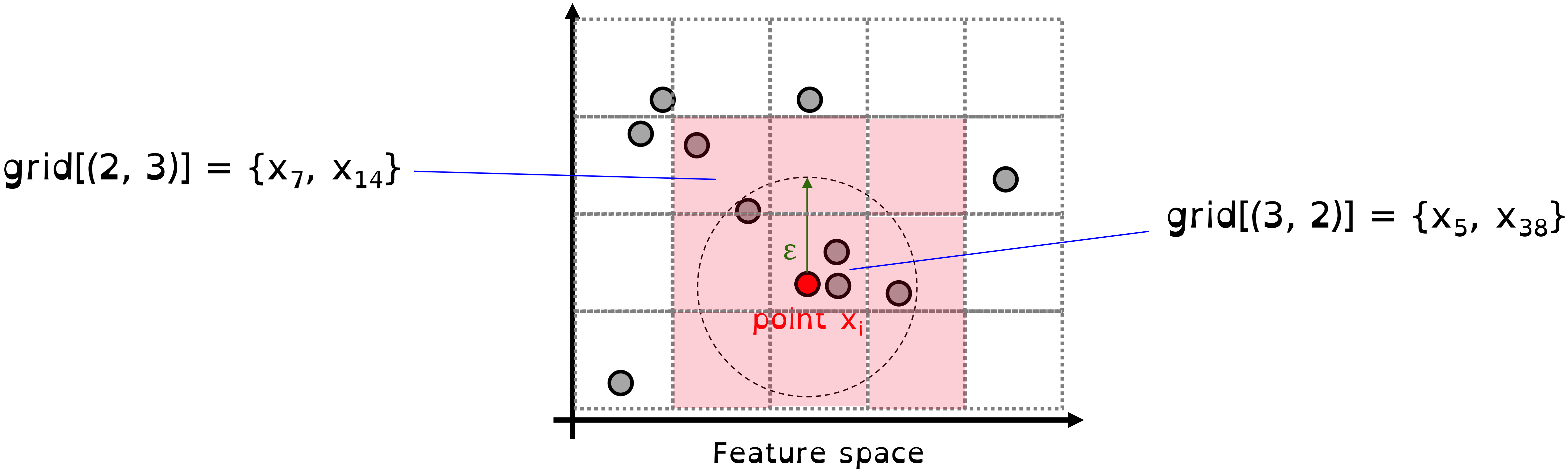



**Q: Do we need to check $x_i$ vs. every other point?**

- Idea: organize feature space into a coarse grid and check only points in same cell (?)

# Implementing Grid-Based Pruning

We need to pre-compute the grid for each value of $\epsilon$ beforehand.

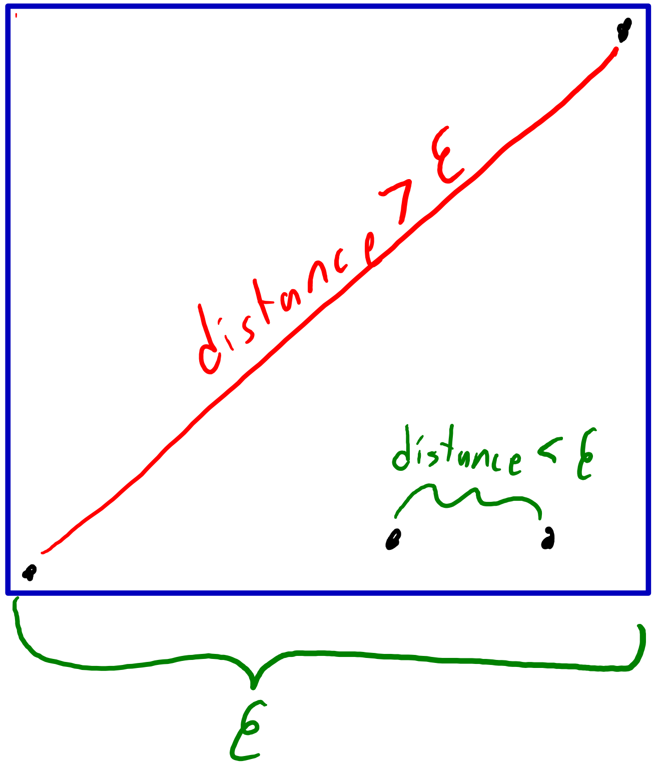



Q: Which data structure can represent these grids efficiently?

grid = dict()

# Grid-Based Pruning

- Which squares do we need to check?

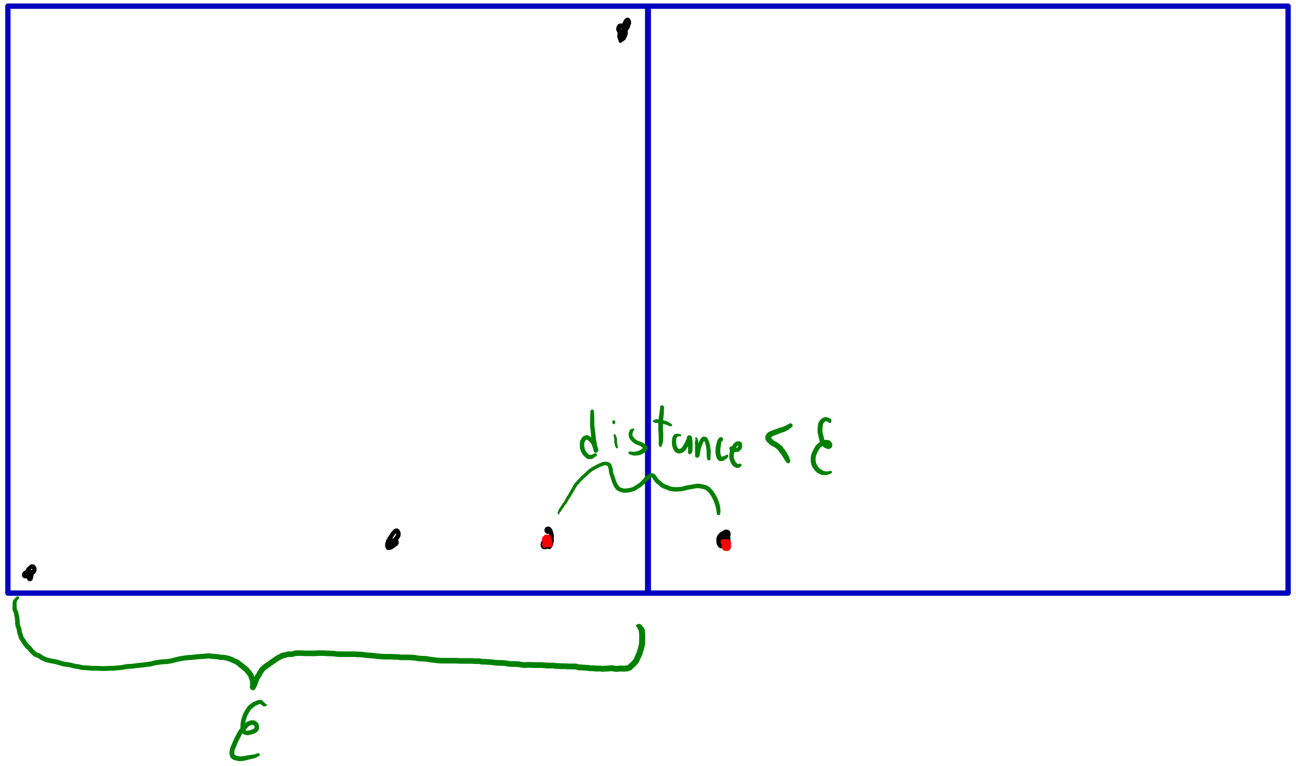



Points in same square can have distance less than 'ε'.

# Grid-Based Pruning

- Which squares do we need to check?

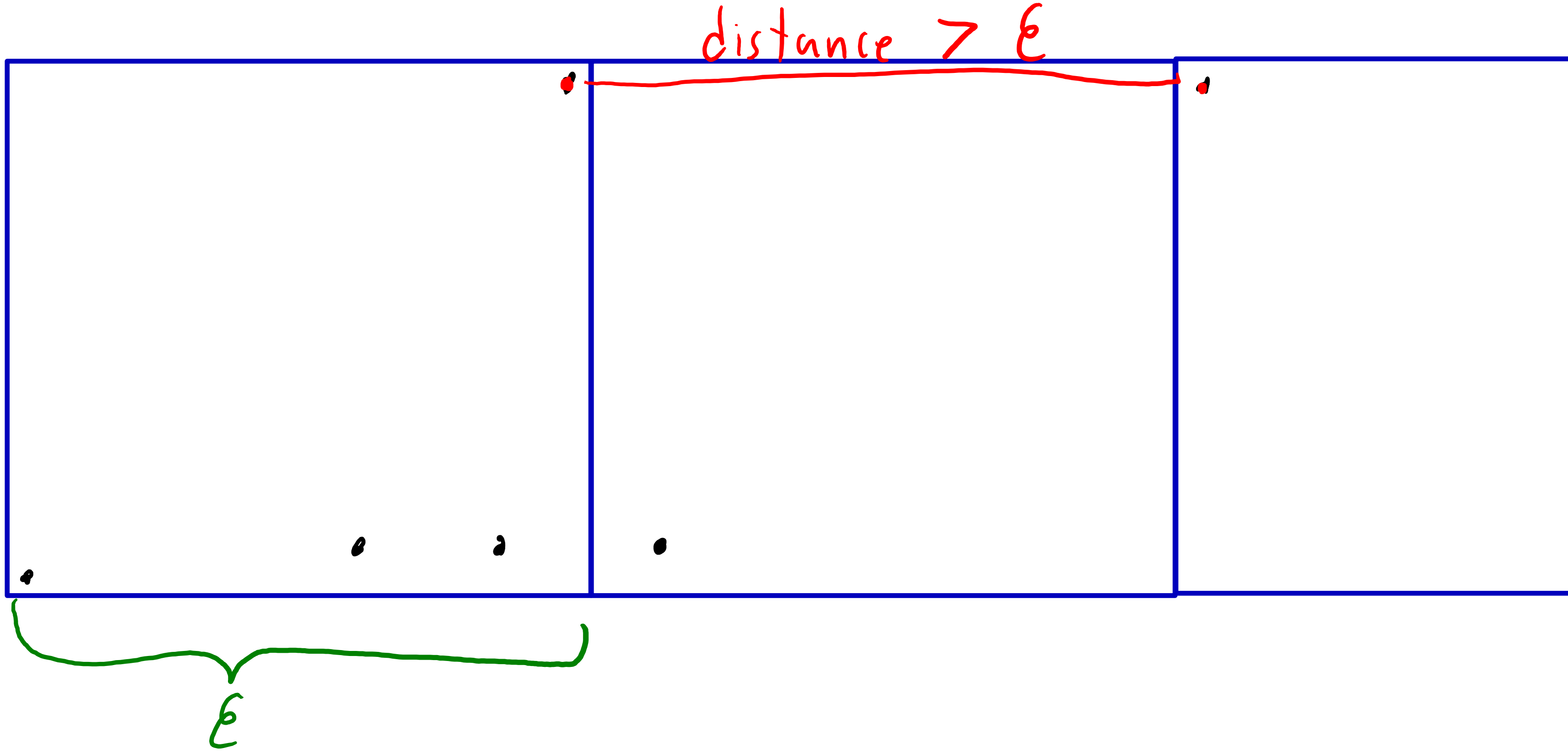
Points in adjacent squares can have distance less than distance ‘ε’.



# Grid-Based Pruning

- Which squares do we need to check?

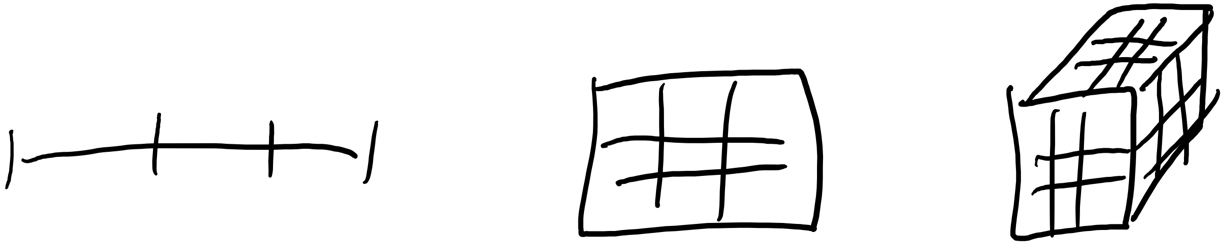
Points in non-adjacent squares must have distance more than ‘ε’.



# Grid-Based Pruning Discussion

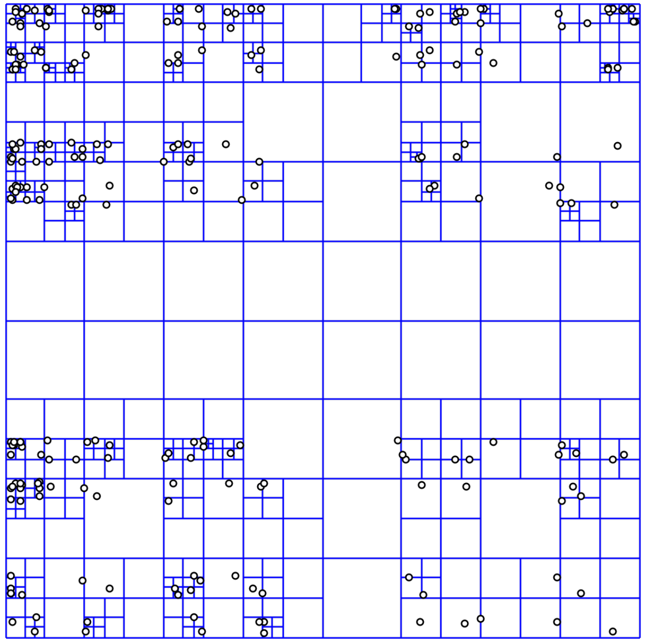
- Similar ideas can be used for other "closest point" calculations.
  - Can be used with any norm.
  - If you want KNN, can use grids of multiple sizes.

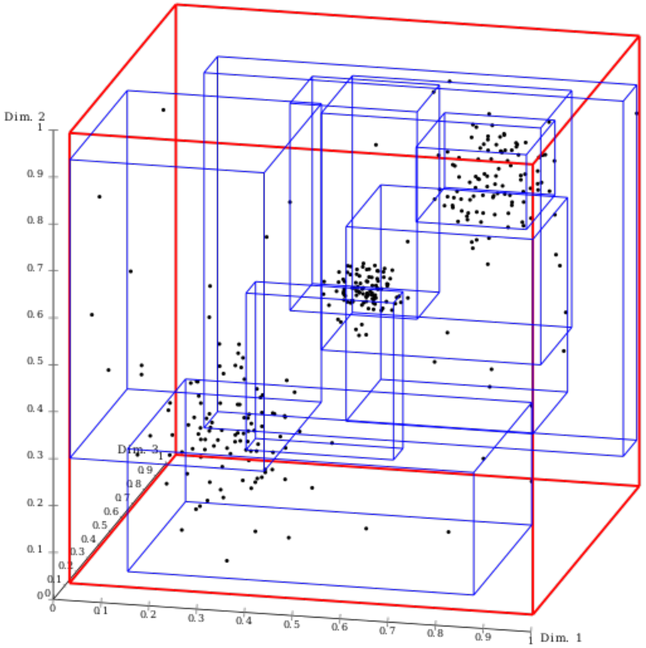
- But we have the "curse of dimensionality":
  - Number of adjacent regions increases exponentially. ______________:
    - 2 with d=1, 8 with d=2, 26 with d=3, 80 with d=4, 252 with d=5, $3^d$-1 in d-dimension.

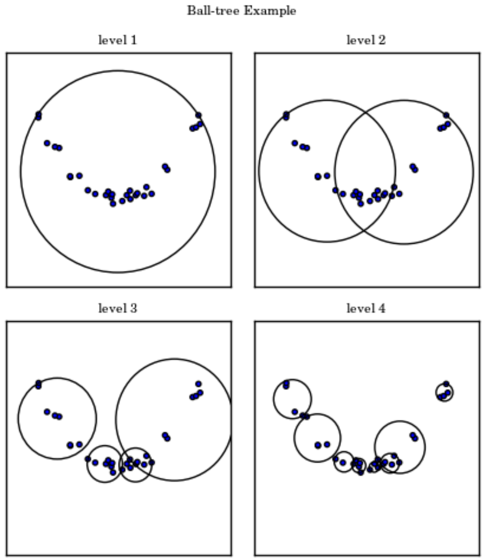


# Grid-Based Pruning Discussion

- Better choices of regions:
  - Quad-trees.
  - Kd-trees.
  - R-trees.
  - Ball-trees.






- Work better than squares, but worst case is still exponential.

https://en.wikipedia.org/wiki/Quadtree
https://en.wikipedia.org/wiki/R-tree
http://www.astroml.org/book_figures/chapter2/fig_balltree_example.html

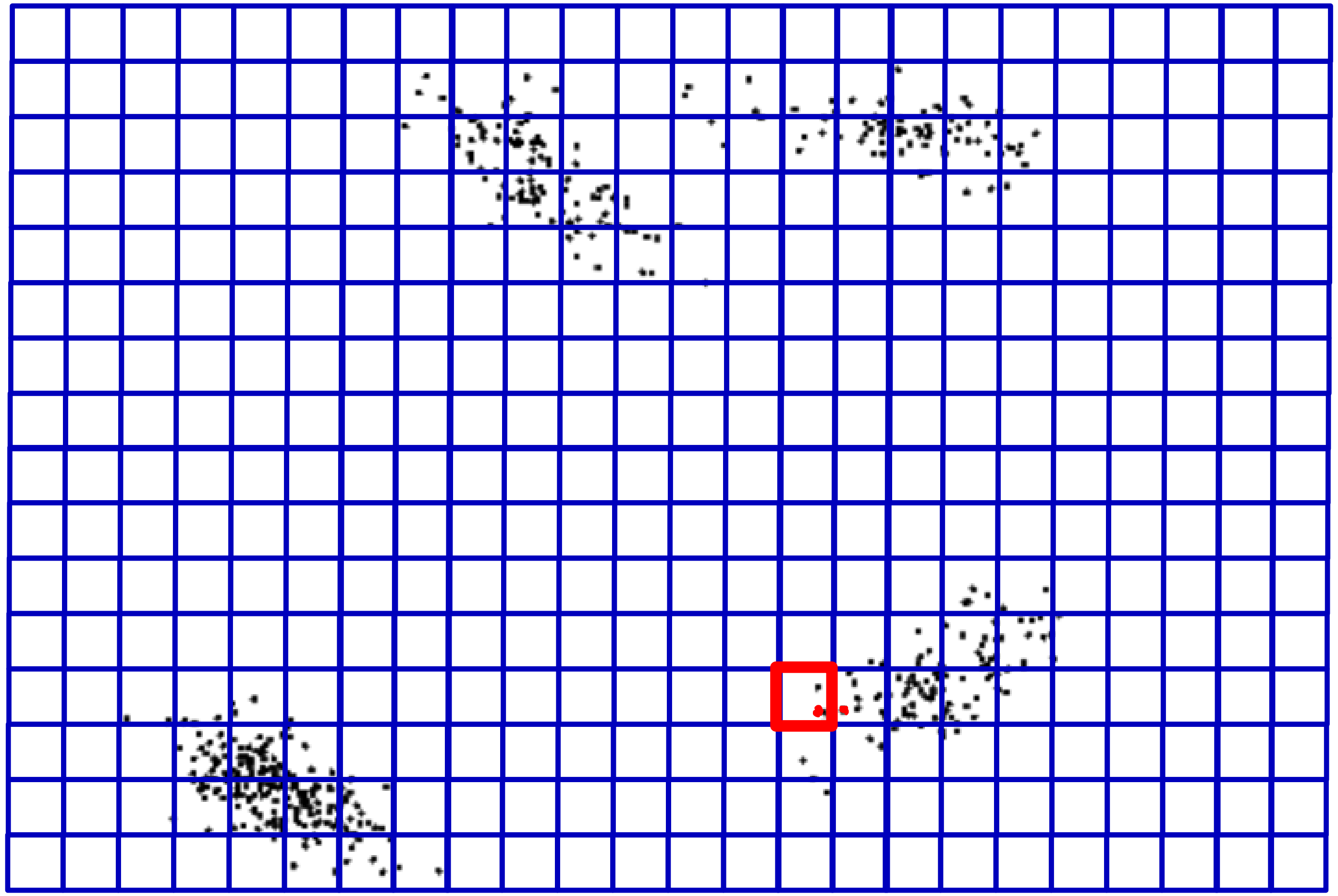
# Approximate Nearest Neighbours

- Only check points within the same square.
  - Works if neighbours are in the same square.
  - But misses neighbours in adjacent squares.

- A simple trick to improve the approximation quality:
  - Use more than one grid.
  - So “close” points have more “chances” to be in the same square.

# Approximate Nearest Neighbours

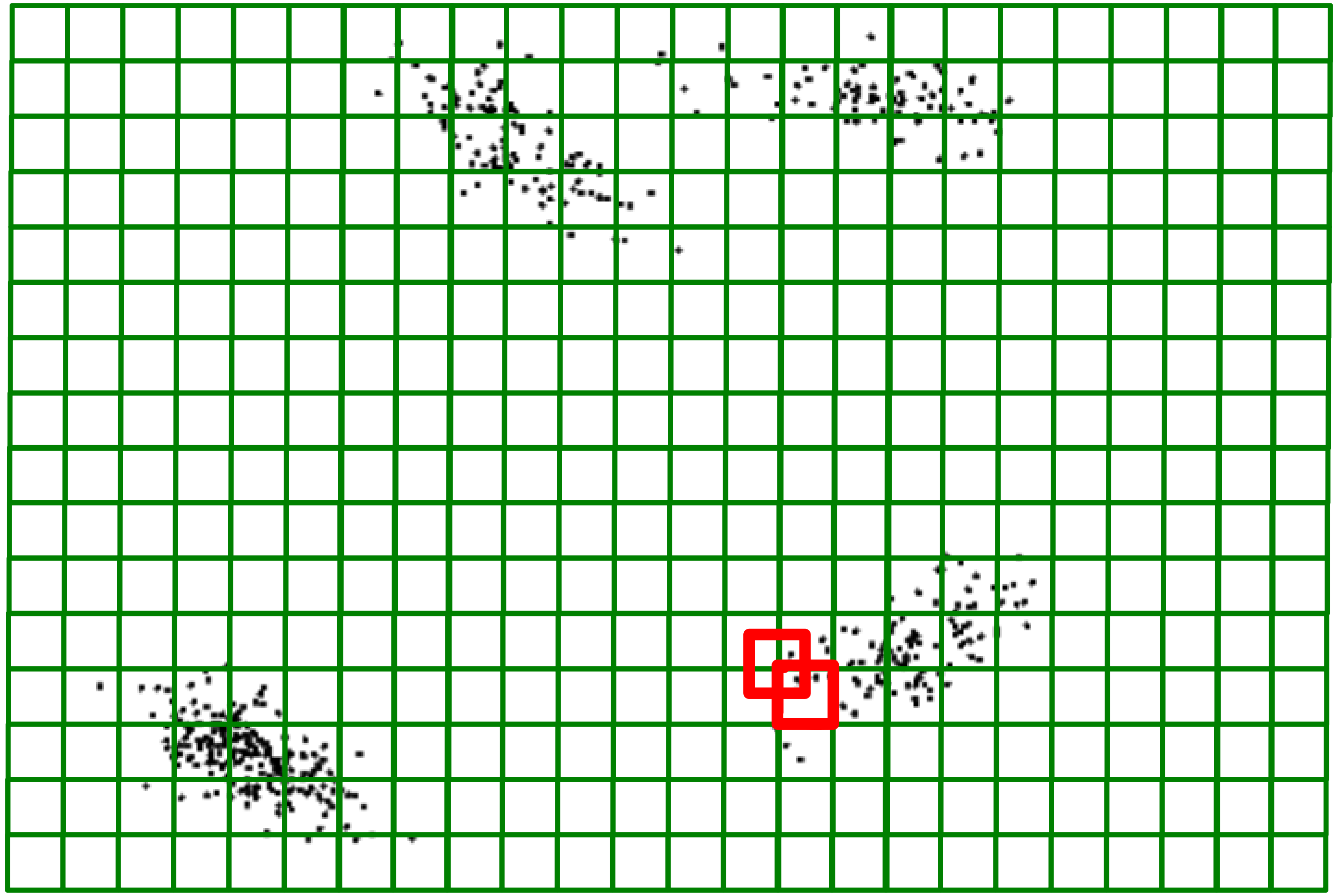
Grid 1:



# Approximate Nearest Neighbours

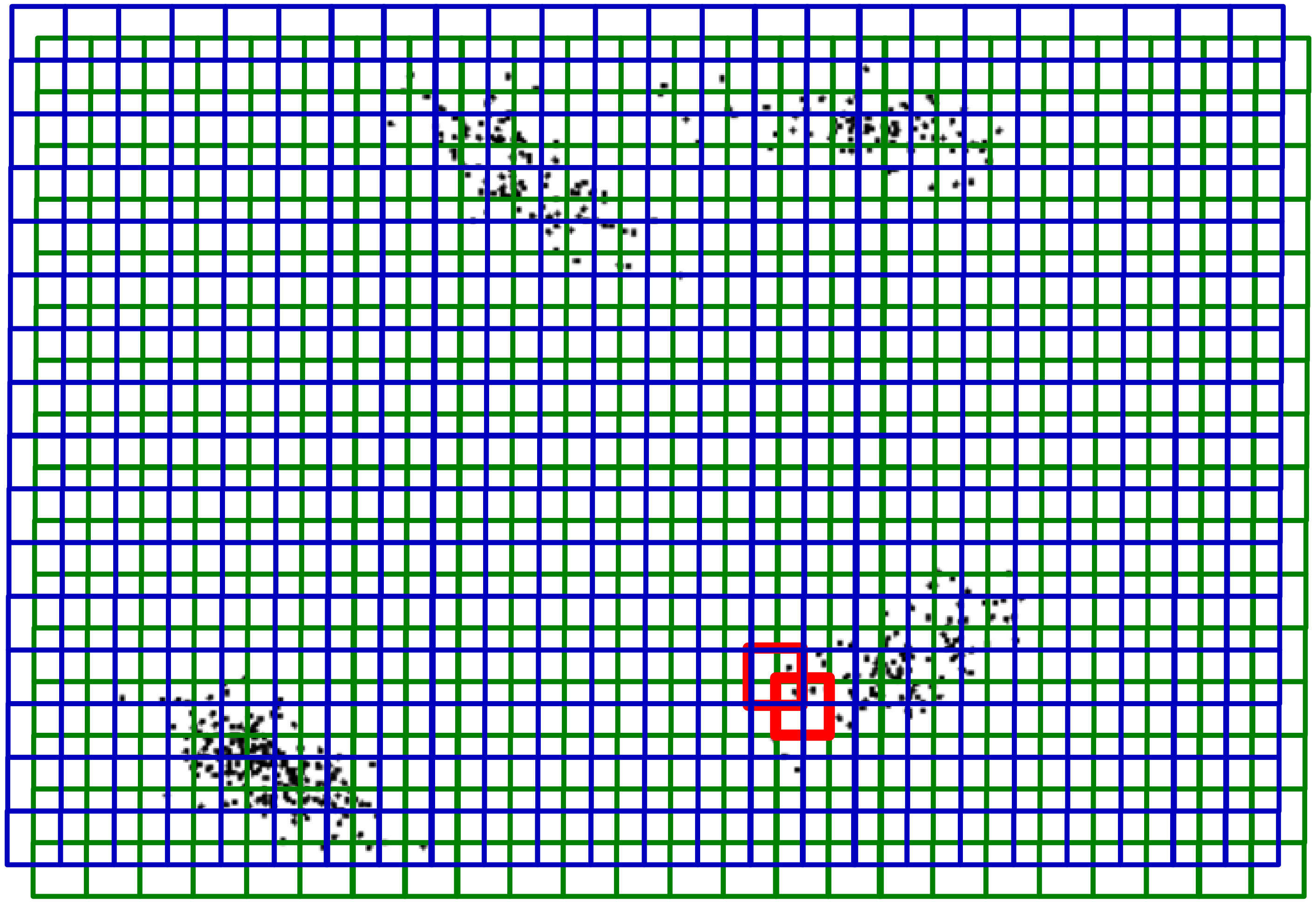
- Using multiple sets of regions improves accuracy.

Grid 2:



# Approximate Nearest Neighbours

- Using multiple sets of regions improves accuracy.

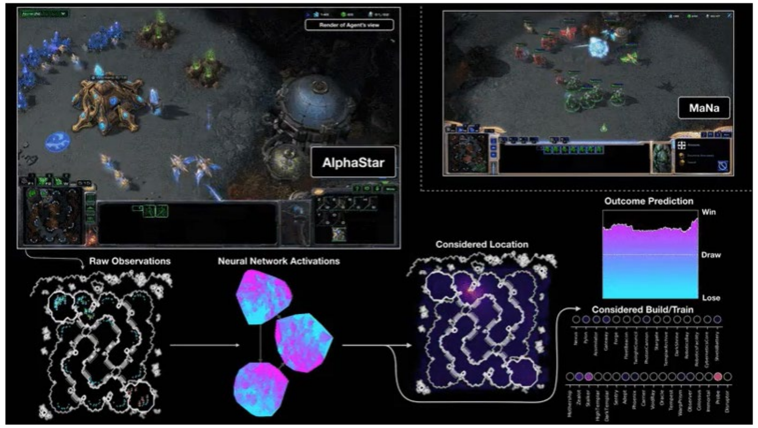


Coming Up Next

# MACHINE LEARNING FOR GAMES

# Motivation: “AI” in Games


Playing Go




Playing StarCraft II

Playing Dota 2

- An AI must judge the situation (“state” of the game)
  - **Go**: the board looks like this, and the opponent has captured 5 stones…
  - **Dota 2:** opponent team’s hero A is level 6 with items 1, 2, 3, my team’s heroes have…
  - **StarCraft**: opponent has unit A, building B, and a group of units are moving…
- …and make a good decision (“action” of the agent)
  - **Go**: place stone in position (x,y)
  - **Dota 2:** cast my hero B’s ability Q on opponent hero A
  - **StarCraft**: build unit C, move my units to location (x,y)

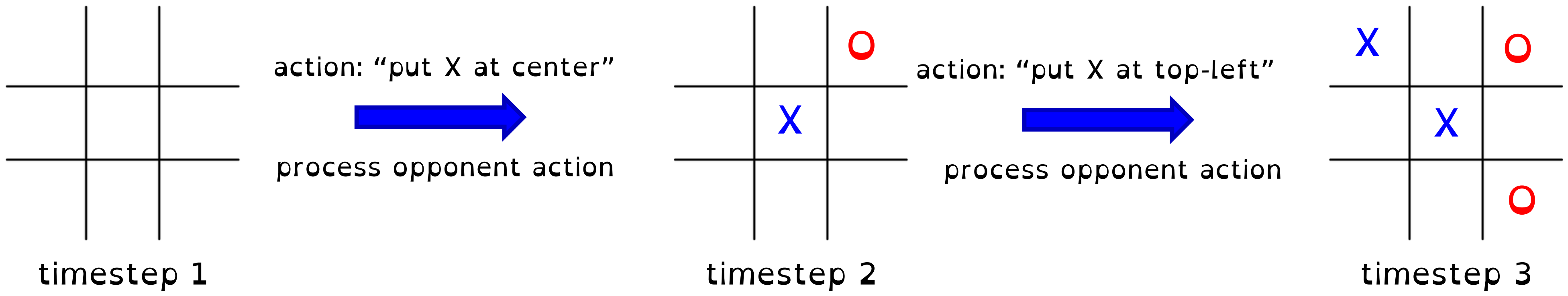
Q: Can we make this a supervised learning problem?

# "Optimal Control"

- Optimal control: a popular mathematical framework for computer games

- Assumption: for every situation ("state"), there is a correct move ("action")
  - A "controller" (or a "policy") is a mapping of state → action
  - Our goal is to use machine learning to produce an automated controller

- Let's assume that games follow a Markov Decision Process (MDP)
  - At each "timestep" in the game, you are given the current game state
  - You decide on the best action for that timestep
  - The game incorporates your action and runs its engine (aka "taking a step")
  - Then you move onto the next timestep in the game.



# Classic Approaches to Gameplay

- Hard-coded policies (fast but labour-intensive)
  - Game developer sits down to make a complicated, hard-coded decision tree.
  - e.g. "World of Warcraft" raid boss
    if 'my_hp' < 20%: use_special_ability()

- Simulation-based control (expensive)
  - At each timestep, play the game multiple times with different strategies, then choose the best one
  - e.g. chess, go, card games, board games
  - Requires knowledge of what the opponent might do

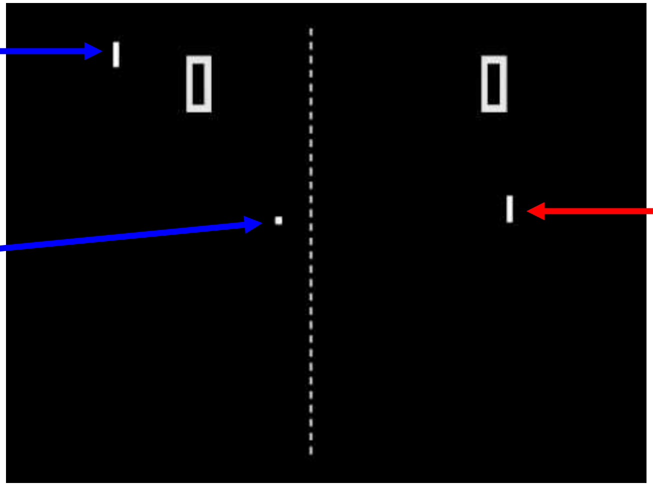
Coming Up Next

# CONTROLLER LEARNING

# Toy Example: “Pong”

my paddle

ball



opponent’s paddle

(assume we’re up against the computer)

- Goal: beat the opponent!
- The situation (“state” of the game) is captured by:
  - position of my paddle (scalar)
  - position of opponent’s paddle (scalar)
  - position of ball (2d vector)
  - velocity of ball (2d vector)

  continuous features
- The decision (“action” of the agent) is:
  - {UP, DOWN, STAY} ← categorical label

# Imitation Learning for Pong

- Idea: gather play data from human players (experts),
  - Look at winners' play data
  - Learn "winners' action" at each state

| MyPos | YourPos | BallXPos | BallYPos | BallXVel | BallYVel |
|---|---|---|---|---|---|
| 0 | 16 | 25 | 30 | 2 | 0 |
| 125 | 126 | 50 | 192 | 1 | -2 |
| 137 | 10 | 10 | 21 | 2 | 1 |
| … | … | … | … | … | … |

"state features"

| Action |
|---|
| STAY |
| DOWN |
| UP |
| … |

"action labels"

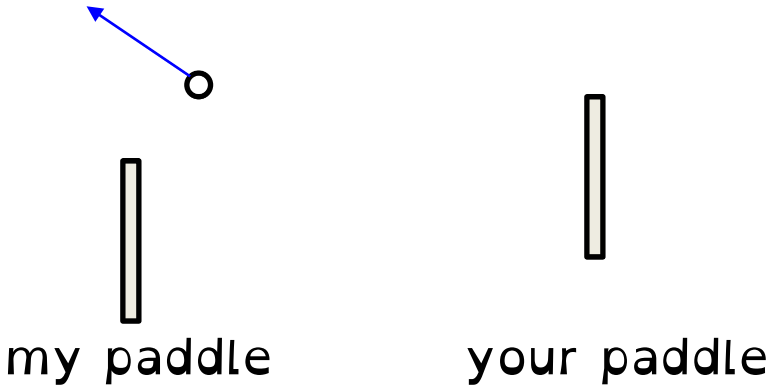
- Also called "imitation learning" or "policy cloning"
  - Assumes that both human experts and automated agents are policies

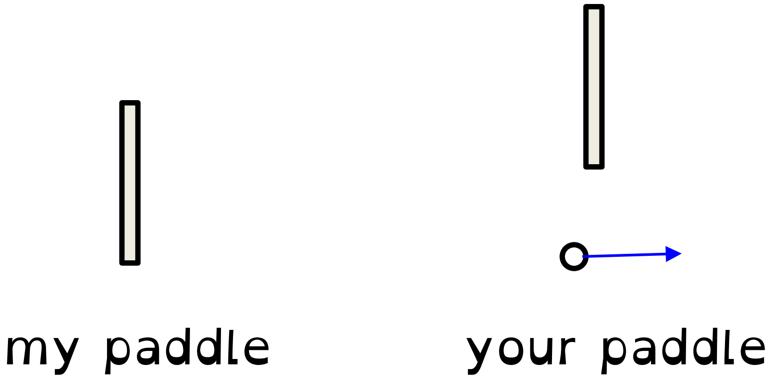
Q: What kind of models can we train on this data?

Q: Are these examples IID? What can go wrong?

out-of-dist errors

# "State Value Function"




Q: Is this a good state?

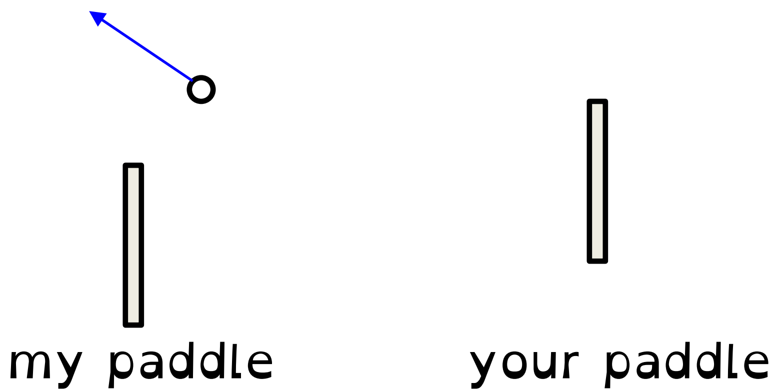
1.2

Q: Is this a good state?

9.9

- Some states are inherently "better" than others
- State value function measures which states are better
- The "true values" can be computed with dynamic programming
  - Expensive but accurate

# "Action Value Function"




| Q: Is "DOWN" a good action here? | 0.2 |
|---|---|
| Q: Is "UP" a good action here? | 9.9 |

- Some actions are inherently "better" than others
- Action value function measures which actions are better
- However, actions are __State-specific__ Value(state, action)
  - We need to compute the value of action in a specific state
- The "true values" can be computed with dynamic programming
  - Expensive but accurate

# Action Value Learning for Pong

- Idea: gather play data from human players (experts),
  – Compute action value by using expensive solution
  – Learn the mapping of state-action → value

| MyPos | YourPos | BallXPos | BallYPos | BallXVel | BallYVel | Action | | Value |
|---|---|---|---|---|---|---|---|---|
| 0 | 16 | 25 | 30 | 2 | 0 | STAY | ⇒ | 10.5 |
| 125 | 126 | 50 | 192 | 1 | -2 | DOWN | ⇒ | 2.3 |
| 137 | 10 | 10 | 21 | 2 | 1 | UP | ⇒ | 30.1 |
| … | … | … | … | … | … | … | | -5.0 |

"state features" "action labels" "action values"

- Also called "Q-Learning" if done without an expert

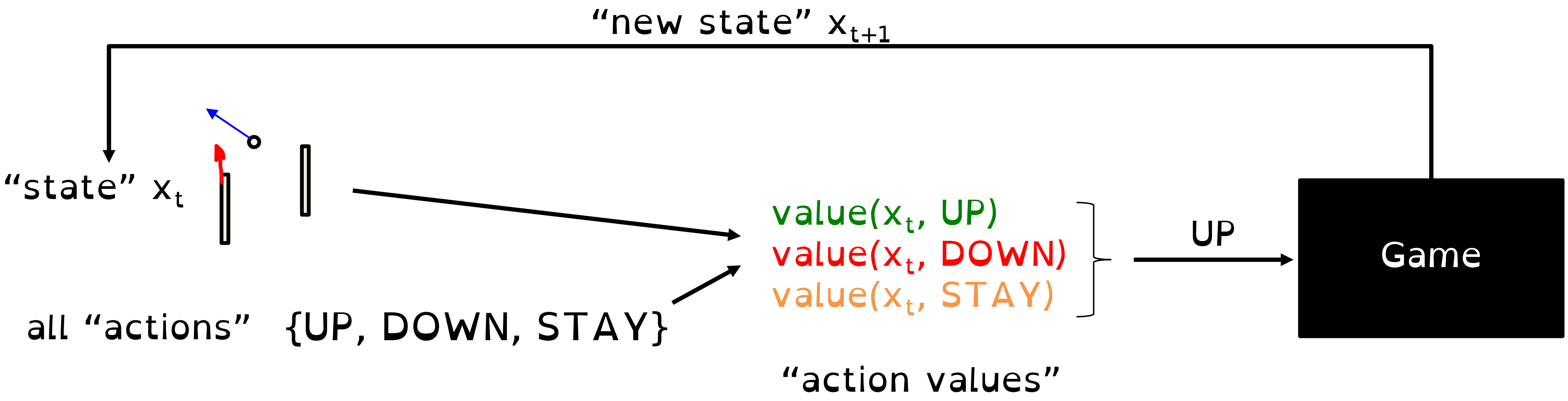
Q: What kind of models can we train on this data?

KNN regression random forest. linear regression

# What If We Don't Have Experts?

- Vanilla imitation learning: impossible without an expert.
  - Also requires lots of gameplay when state space is large
- Idea: instead of a human expert, let's use a game-playing bot
  - Make LOTS of random actions and record their values
  - Do it over MANY rounds of Pong
- Learn the action values. Then we have a controller! (WHAT?!)

# Action Value for Optimal Control




- Taking "argmax" of action value gives you the best action for current timestep.
- Next timestep, you receive a __new state__________.
- With the new state, take "argmax" of action value again, and repeat.
- If getting action values is fast, then the controller will be fast!

Q: Will this controller be perfect?

# “Reinforcement Learning”

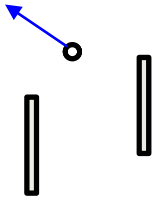
- Earlier: instead of a human expert, let’s use a game-playing bot
  - Make LOTS of random actions and record their values
  - Do it over MANY rounds of Pong

Q: Are random actions that useful?

- Instead of random actions, lets use the “argmax” of action value idea

# “Reinforcement Learning”

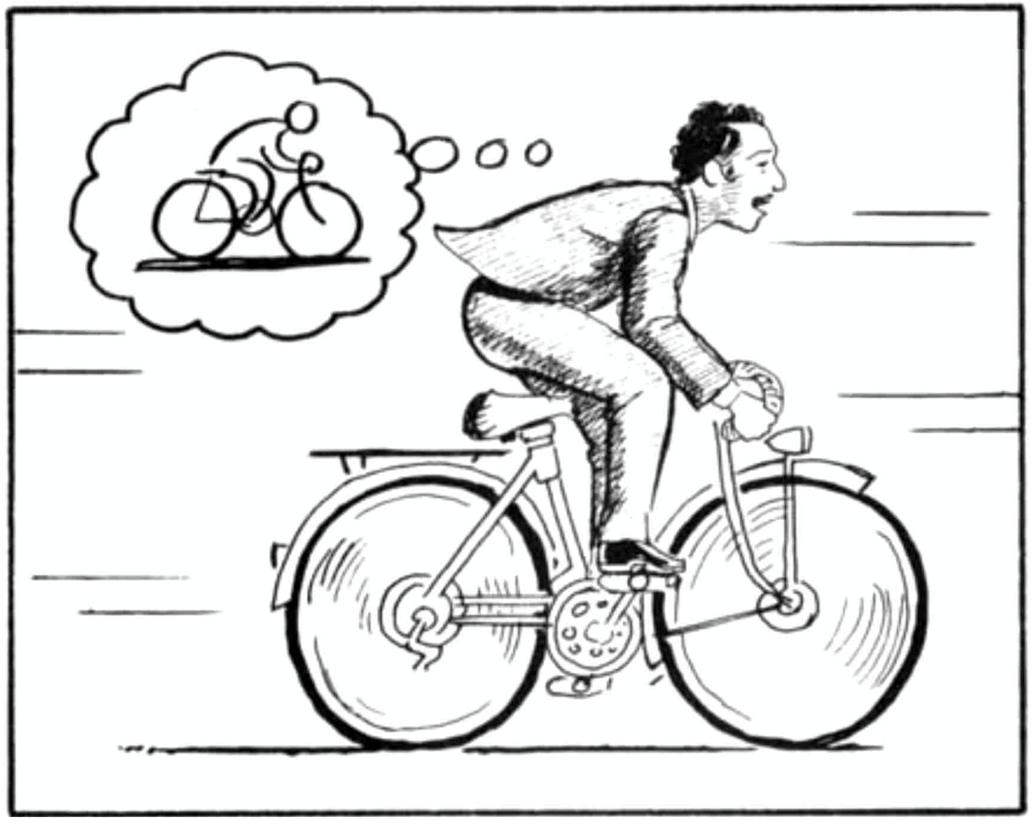
- We can iteratively improve the learned action values like this:

“state” $x_t$ 

  - When in this state, do “UP” sometimes and make random actions sometimes
  - Do it over MANY rounds of Pong
  - Learn action values with new data, and repeat

Q: How is this better than using random actions?

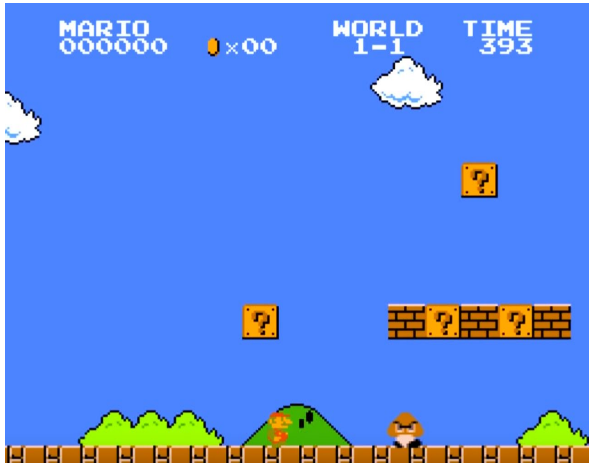
- Coverage of state space is crucial:
  - Using “good actions” will lead to Progress in game (exploitation)
  - Using random actions will lead to diversity of states (exploration)
- This is an (watered-down) instance of “reinforcement learning” (RL)
- Core ideas of RL:
  - iteratively improve a controller
  - let it play the game better every time



Coming Up Next

# DYNAMICS LEARNING

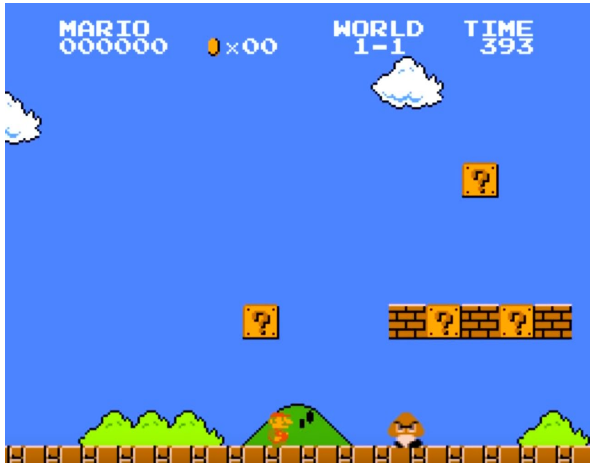
# Another Example: “Super Mario Brothers”




- The decision (“action” of the agent) is:
  - {LEFT, RIGHT, UP, DOWN, SPECIAL, JUMP, NONE}

Q: How should we represent the game state?

# State Representation



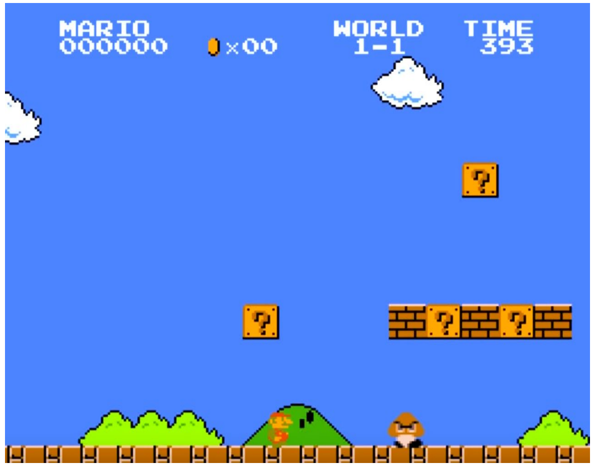

m-by-n image

grayscale intensity

| (1,1) | (2,1) | (3,1) | ... | (m,1) | ... | (m,n) |
|---|---|---|---|---|---|---|
| 45 | 44 | 43 | ... | 12 | ... | 35 |

mn x 1 vector

# “Dynamics”




“current state”

JUMP


“next state”

Q: What does the state look like after I use the “JUMP” action?

- A particular action at a particular state leads to a new state
  - Usually written as $x_{t+1} = f(x_t, u_t)$ or $s_{t+1} = f(s_t, a_t)$
  - called “dynamics” or “model” of the game

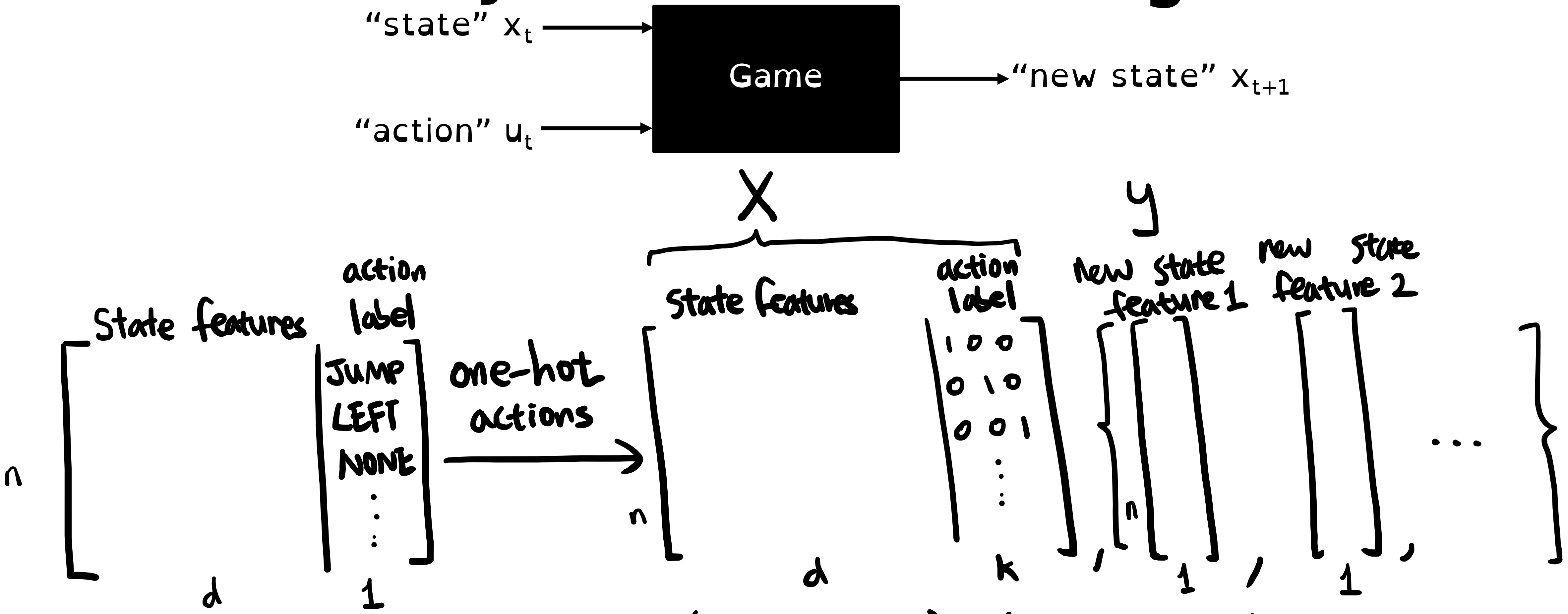
Q: Can we predict the consequence of an action without actually taking the step?

# Dynamics Learning


- Idea:
  - Look at gameplay data, including “state”, “action”, and “new state” at every timestep
  - Predict “new state” from “state” and “action”

| state features | | | action label | next state features | | |
|---|---|---|---|---|---|---|
| 25 | 13 | 42 | JUMP | 26 | 13 | 44 |
| 26 | 13 | 44 | NONE | 26 | 13 | 44 |
| 26 | 13 | 44 | LEFT | 24 | 13 | 46 |
| … | … | … | … | … | … | … |

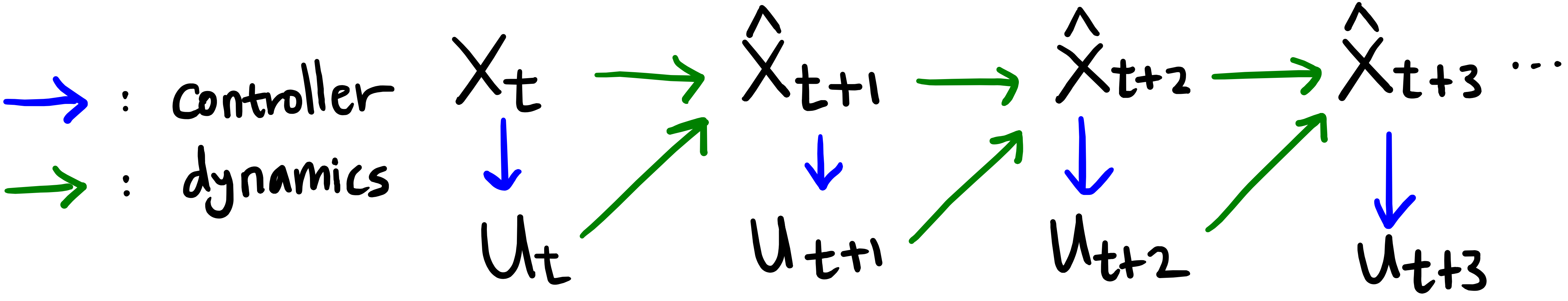
# Dynamics Learning




$$\Rightarrow (X, y_{\text{feature 1}}), (X, y_{\text{feature 2}}), \ldots, (X, y_{\text{feature } d})$$

Q: What kind of models can we train on this data?

# Learned Dynamics Can Be "Chained"!


- Some people call this "thinking"
- Some people call this "dreaming"

# Why Learn Dynamics?

Q: What does the state look like after I use the "JUMP" action?

- Using linear regression, I get $O(d^2+kd)$ time to predict a new state
  - (d + k) features means I have (d + k) weights
  - I predict d different state features
  - For complicated games, often faster than running the game
    - Rendering, physics handling, relocating objects, computing opponent action, etc.

- Simulation-based control methods can use learned dynamics to speed up computation
  - e.g. [model predictive control (MPC)]()
  - Learned dynamics abstracts away the opponent's strategy!

$$[u_1 \quad u_2 \quad u_3 \quad \cdots \quad u_T] \rightarrow \left.\begin{matrix} V(x_1,u_1) \\ V(x_2,u_2) \\ \vdots \\ V(x_T,u_T) \end{matrix}\right\} \text{fitness}$$

$$\text{argmax fitness} = [u_1^* \quad u_2^* \quad u_3^* \quad \cdots \quad u_T^*]$$

# Speeding Up Physics Simulations

**Subspace Neural Physics: Fast Data-Driven Interactive Simulation**

Daniel Holden
Ubisoft La Forge, Ubisoft
Montreal, QC, Canada
daniel.holden@ubisoft.com

Bang Chi Duong
Ubisoft La Forge, Ubisoft
Montreal, QC, Canada
bangchi.duong.20193@outlook.com

Sayantan Datta
McGill University
Montreal, QC, Canada
sayantan.datta@mail.mcgill.ca

Derek Nowrouzezahrai
McGill University
Montreal, QC, Canada
derek@cim.mcgill.ca

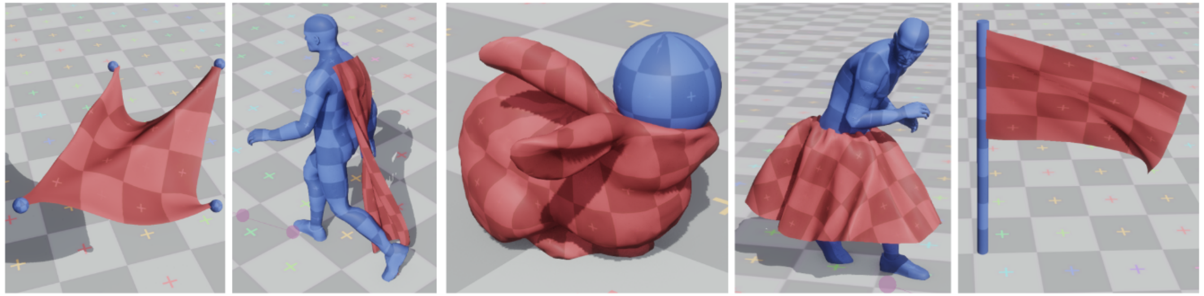


**Figure 1: Our method simulates deformation effects, including external forces and collisions, 300× to 5000× faster than standard offline simulation.**

- Cloth simulation: notoriously slow
  - due to complicated interactions and physical effects
- Learned dynamics: speeds up cloth simulation 5000 times
- Passive dynamics: action is not involved in these applications

# Speeding Up Physics Simulations

**Data-driven Fluid Simulations using Regression Forests**

Ľubor Ladický*† ETH Zurich
SoHyeon Jeong*† ETH Zurich
Barbara Solenthaler† ETH Zurich
Marc Pollefeys† ETH Zurich
Markus Gross† ETH Zurich, Disney Research Zurich

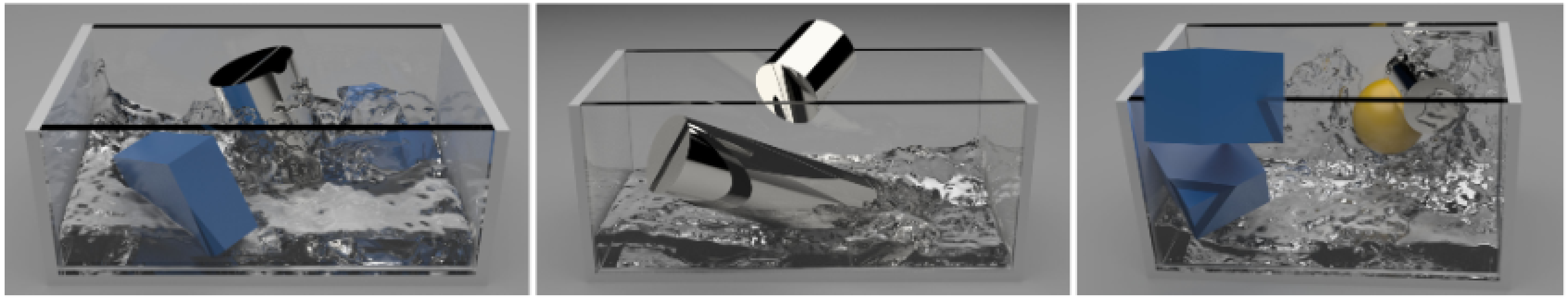


**Figure 1:** *The obtained results using our regression forest method, capable of simulating millions of particles in realtime. Our promising results suggest the applicability of machine learning techniques to physics-based simulations in time-critical settings, where running time matters more than the physical exactness.*

- Also applies to fluid simulation!
- Passive dynamics: action is not involved in these applications

# Summary

- Recommender systems: find similar items to recommend
- Closest-point problem: the bane of distance-based methods
  - Hard to do with lots of features!
- Grid-based pruning: use dictionary to speed up distances
- Controller learning: machine learning for game-playing agents
  - Reinforcement learning: iterative controller learning based on sample actions
- Dynamics learning: bypass real steps to get approximate steps
  - Useful for speeding up simulations

- Next time:
  - how to make least squares "smarter"