CS4386 AI Game Programming

Lecture 03
Action Prediction:
N-Grams and Naïve Bayes Classifier

Semester B, 2020-2021
Department of Computer Science
City University of Hong Kong

Action Prediction

- It is often useful to guess what players will do next
  - E.g., which passage the player will take, which weapon the player will select, which route player will attack from, etc.
  - A game that can predict a player's actions can mount a more challenging opposition
- Humans are notoriously bad at behaving randomly
  - Psychological research has shown that we cannot accurately randomize our responses, even if we specifically try
  - Often we have shared characteristics so that learning to anticipate one player's actions can often lead to better play against a completely different player

Raw Probability

- The simplest way to predict the choice of a player is to keep a record of the number of times he/she chooses each option
  - This will then form a raw probability of that player choosing that action again
  - E.g., after observing the player over 10 rock-paper-scissor (RPS) games, R: 5; P: 3; S: 2. So P(R)=5/10=0.5, P(P)=3/10=0.3, P(S)=2/10=0.2. Thus the AI can predict that the player will choose R as the next move
- This kind of raw probability prediction is very easy to implement, but it gives a lot of feedback to the player, who can use the feedback to make their decisions more random
  - E.g., the player may choose S instead knowing how the AI predicts his move
- When the choice is made only once, then this kind of prediction may be all that is possible
  - If the probabilities are gained from many different players, then it can be a good indicator of which way a new player will go

String Matching

- When a choice is repeated several times, a simple string matching algorithm can provide good prediction
- The sequence of choices made is stored as a string
- E.g., in a game where the possible moves are either left (L) or right (R), the sequence of moves may look like "L$\boxed{\text{RR}}$LRLLL$\boxed{\text{RR}}$LRL$\boxed{\text{RR}}$"
  - To predict the next choice, the last few choices are searched for in the string, and the choice that normally follows is used as the prediction
  - The last two moves were "RR." Looking back over the sequence, the choice sequence RR is always followed by L, so we predict that the player will go for L next time
  - In this case we have looked up the last two moves, which is called the "window size": we are using a window size of two

N-Grams

- The string matching technique is rarely implemented by matching against a string
- It is more common to use a set of probabilities known as an N-Gram predictor (where N is one greater than the window size parameter, so 3-Gram would be a predictor with a window size of two)
- In an N-Gram we keep a record of the probabilities of making each move given all combinations of choices for the previous N moves
- For the example "LRRLRLLLRRLRLRR" with N=3:
  - we keep track of probability for L and R after the 4 sequences "LL" "LR" "RL" "RR"
  - 8 probabilities in total, but each pair must add up to one

	..R	..L
LL	$\frac{1}{2}$	$\frac{1}{2}$
LR	$\frac{3}{5}$	$\frac{2}{5}$
RL	$\frac{3}{4}$	$\frac{1}{4}$
RR	$\frac{0}{2}$	$\frac{2}{2}$

Pseudo Code – N-Grams

Each time an action occurs, the game registers the last n actions

When the game needs to predict what will happen next, it feeds only the window actions

```
class NGramPredictor:

  # Holds the frequency data
  data

  # Holds the size of the window + 1
  nValue

  # Registers a set of actions with predictor, updating
  # its data. We assume actions has exactly nValue
  # elements in it.
  def registerSequence(actions):

    # Split the sequence into a key and value
    key = actions[0:nValue]
    value = actions[nValue]

    # Make sure we've got storage
    if not key in data:
      data[key] = new KeyDataRecord()

    # Get the correct data structure
    keyData = data[key]

    # Make sure we have a record for the follow on value
    if not value in keyData.counts:
      keyData.counts[value] = 0

    # Add to the total, and to the count for the value
    keyData.counts[value] += 1
    keyData.total += 1
```

```
  # Gets the next action most likely from the given one.
  # We assume actions has nValue - 1 elements in it (i.e.
  # the size of the window).
  def getMostLikely(actions):

    # Get the key data
    keyData = data[actions]

    # Find the highest probability
    highestValue = 0
    bestAction = None

    # Get the list of actions in the store
    actions = keyData.counts.getKeys()

    # Go through each
    for action in actions:

      # Check for the highest value
      if keyData.counts[action] > highestValue:

        # Store the action
        highestValue = keyData.counts[action]
        bestAction = action

    # We've looked through all actions, if best action
    # is still None, then its because we have no data
    # on the given window. Otherwise we have the best
    # action to take
    return bestAction
```

Frequencies instead of probabilities are stored (easier to update)

Implementation

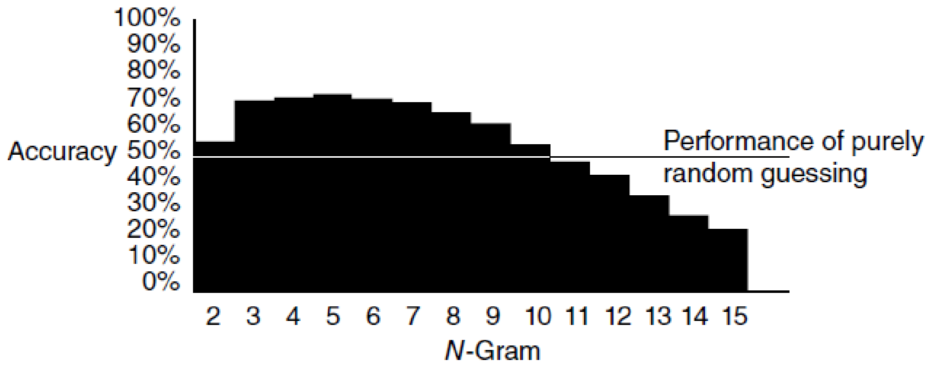
- The implementation in previous slide uses hash tables to avoid growing too large when most combinations of actions are never seen
- If there are only a small number of actions, and all possible sequences can be visited, then it will be more efficient to replace the nested hash tables with a single array
- Performance
  - $O(1)$ in time for function `registerSequence`
  - $O(m)$ in time for function `getMostLikely` where m: the number of possible actions

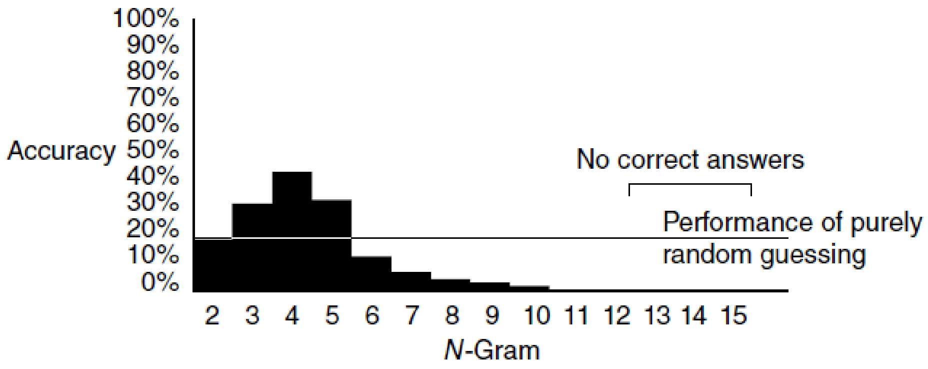
  We can keep the hash tables sorted by value, then
  - $O(m)$ in time for function `registerSequence`
  - $O(1)$ in time for function `getMostLikely`

  - $O(n^m)$ in memory where n: the N value, i.e., window size + 1

Window Size

- Increasing the window size initially increases the performance of the prediction algorithm
  - For each additional action in the window, the improvement reduces until there is no benefit to having a larger window, and eventually the prediction gets worse with a larger window, which may even be worse than guessing at random
- If there is a certain degree of randomness in our actions, then a very long sequence will likely have a fair degree of randomness in it
  - The very large window size is likely to include more randomness and, therefore, be a poor predictor



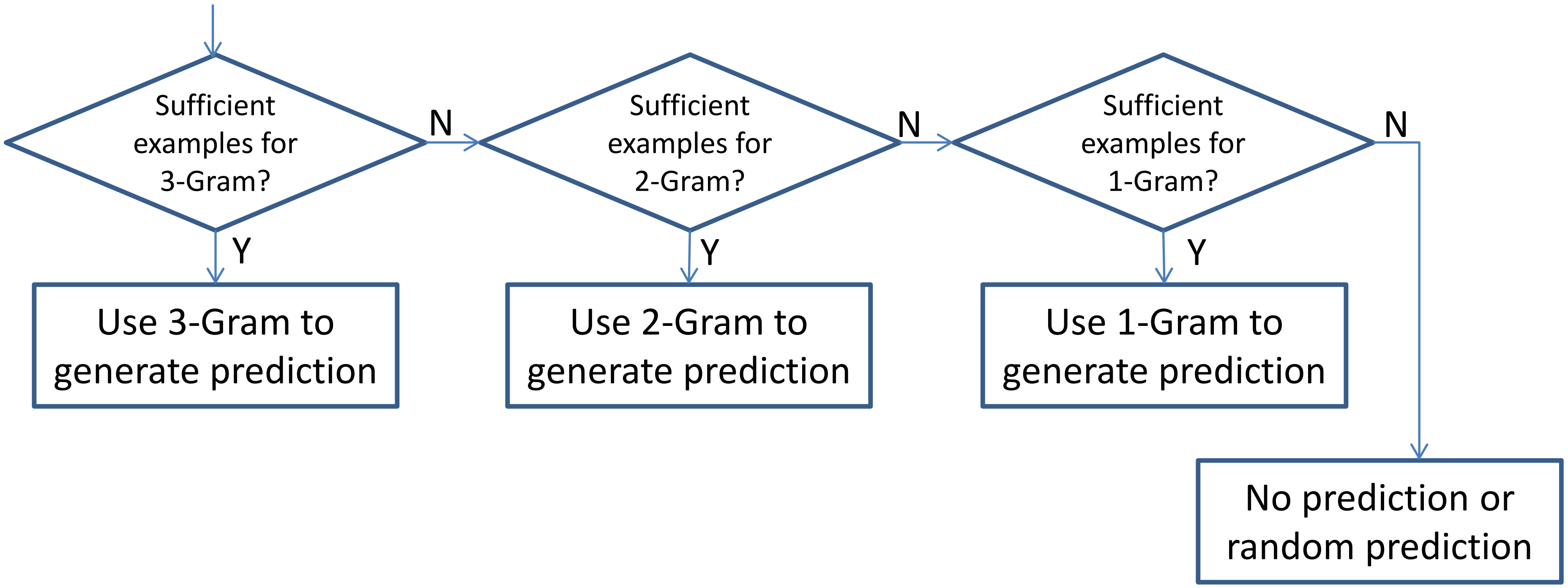


Predictions with 2 choices

Predictions with 5 choices

Hierarchical N-Grams

- The hierarchical N-Grams algorithm has several N-Gram algorithms working in parallel, each with increasingly large window sizes
  - A hierarchical 3-Gram will have regular 1-Gram (i.e., the raw probability approach), 2-Gram, and 3-Gram algorithms working on the same data

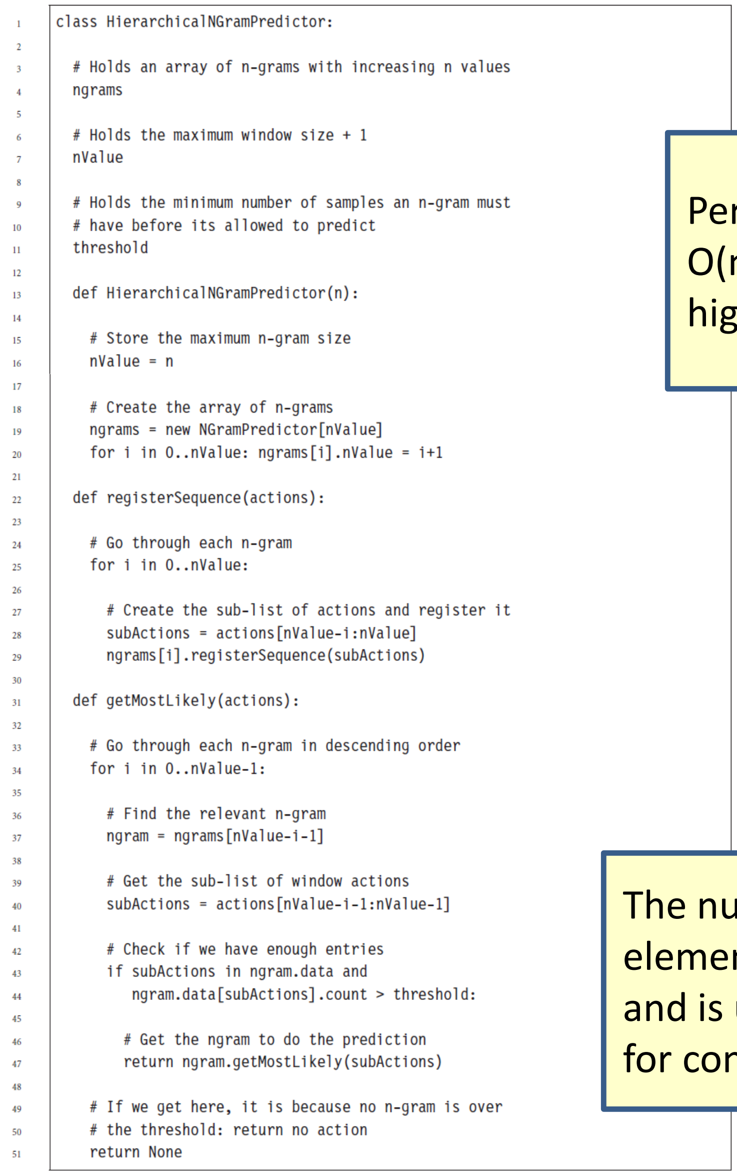


Sufficient examples for 3-Gram?
N
Sufficient examples for 2-Gram?
N
Sufficient examples for 1-Gram?
N
Y
Use 3-Gram to generate prediction
Y
Use 2-Gram to generate prediction
Y
Use 1-Gram to generate prediction
No prediction or random prediction

Hierarchical N-Gram (2)

- When an N-Gram algorithm is used for online learning, there is a balance between the maximum predictive power and the performance of the algorithm during the initial stages of learning
  - A larger window size may improve the potential performance, but will mean that the algorithm takes longer to get to a reasonable performance level
- There is no single correct threshold value for the number of entries required for confidence
  - To some extent it needs to be found by trial and error
  - In online learning, however, it is common for the AI to make decisions based on very sketchy information, so the confidence threshold can be small (say 3 or 4)

Pseudo Code – Hierarchical N-grams



```
class HierarchicalNGramPredictor:

  # Holds an array of n-grams with increasing n values
  ngrams

  # Holds the maximum window size + 1
  nValue

  # Holds the minimum number of samples an n-gram must
  # have before its allowed to predict
  threshold

  def HierarchicalNGramPredictor(n):

    # Store the maximum n-gram size
    nValue = n

    # Create the array of n-grams
    ngrams = new NGramPredictor[nValue]
    for i in 0..nValue: ngrams[i].nValue = i+1

  def registerSequence(actions):

    # Go through each n-gram
    for i in 0..nValue:

      # Create the sub-list of actions and register it
      subActions = actions[nValue-i:nValue]
      ngrams[i].registerSequence(subActions)

  def getMostLikely(actions):

    # Go through each n-gram in descending order
    for i in 0..nValue-1:

      # Find the relevant n-gram
      ngram = ngrams[nValue-i-1]

      # Get the sub-list of window actions
      subActions = actions[nValue-i-1:nValue-1]

      # Check if we have enough entries
      if subActions in ngram.data and
         ngram.data[subActions].count > threshold:

        # Get the ngram to do the prediction
        return ngram.getMostLikely(subActions)

    # If we get here, it is because no n-gram is over
    # the threshold: return no action
    return None
```

Performance:
O(n) in time, where n is the highest numbered N-Gram used

The number of samples is just one element that affects confidence, and is used as an approximation for confidence here

Application in Combat

- N-Gram prediction can be applied in combat games involving timed sequences of moves
- Using an N-Gram predictor allows the AI to predict what the player is trying to do as they start their sequence of moves and then select an appropriate rebuttal
- This approach is so powerful, however, that it can provide unbeatable AI
  - A common requirement in this kind of game is to remove competency from the AI so that the player has a sporting chance

Bayes' Rule

- Definition of conditional probability:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \qquad P(B|A) = \frac{P(A \cap B)}{P(A)}$$

$$P(A \cap B) = P(A|B)P(B) \qquad P(A \cap B) = P(B|A)P(A)$$

- Bayes' Rule:

$$\underbrace{P(B|A)}_{\boxed{\text{posterior}}} = \frac{\overbrace{P(A|B)}^{\boxed{\text{likelihood}}}\overbrace{P(B)}^{\boxed{\text{prior}}}}{P(A)}$$

Bayes' Rule Example

- A soldier is either trained or untrained and an AI is trying to decide whether it will win a fight against a trained soldier
- Given that from past combats, the AI has won 80% of the time when it fights against a soldier
- Among all the soldiers that the AI has fought (AI may win or lose), 25% of the soldiers are trained (and thus 75% untrained)
- Among the soldiers against whom the AI has won the fight, 12.5% of them are trained (and thus 87.5% untrained)
- What is the probability that the AI will win a fight against a trained solider?

Bayes' Rule Example (2)

- T = the solider fought against AI is trained
- W = AI wins the fight against a soldier
- T | W = the soldier is a trained one given that the AI wins the fight
- W | T = AI wins the fight given that the soldier is trained
- Applying Bayes' Rule:

$$P(W|T) = \frac{P(T|W)P(W)}{P(T)} = \frac{(0.125)(0.8)}{0.25} = 0.4$$

- Note that in this example, we estimate the probability from the relative frequency, i.e., deriving from the event occurrences in the past experience

Naïve Bayes Classifier

- The Naïve Bayes Classifier applies Bayes Rule to make a decision
- For example, in a racing game, the AI would like to learn a player's style of going around corners. In particular, the AI would like to determine whether the player will apply brake with a given speed at a given distance to the corner. Suppose that the following gameplay data has been recorded from the player:

brake?	distance	speed
Y	2.4	11.3
Y	3.2	70.2
N	75.7	72.7
Y	80.6	89.4
N	2.8	15.2
Y	82.1	8.6
Y	3.8	69.4

Naïve Bayes Classifier Example (1)

- To make the problem easier to solve, the distance and speed features can be converted into discrete states:
  - If distance < 10, then label it as *near*. Otherwise, label it as *far*
  - If speed > 20, then label it as *fast*. Otherwise, label it as *slow*

brake?	distance	speed
Y	near	slow
Y	near	fast
N	far	fast
Y	far	fast
N	near	slow
Y	far	slow
Y	near	fast

Naïve Bayes Classifier Example (2)

- From Bayes' Rule

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

$$= \alpha P(A|B)P(B)$$

Indicating that posterior is linearly proportional to the product of likelihood and prior

- So the problem becomes solving the following equation:

$$P(brake|distance, speed) = \alpha P(distance, speed|brake)P(brake)$$

- Assuming conditional independence,

$$P(distance, speed|brake) = P(distance|brake)P(speed|brake)$$

- The equation to solve thus becomes:

$$P(brake|distance, speed) = \alpha P(distance|brake)P(speed|brake)P(brake)$$

Naïve Bayes Classifier Example (3)

$$P(brake|distance, speed) = \alpha P(distance|brake)P(speed|brake)P(brake)$$

- Estimated from the observed relative frequency of the gameplay data:

$P(near|brake = Y) = \frac{3}{5}$ $\quad$ $P(far|brake = Y) = \frac{2}{5}$

$P(slow|brake = Y) = \frac{2}{5}$ $\quad$ $P(fast|brake = Y) = \frac{3}{5}$

$P(near|brake = N) = \frac{1}{2}$ $\quad$ $P(far|brake = N) = \frac{1}{2}$

$P(slow|brake = N) = \frac{1}{2}$ $\quad$ $P(fast|brake = N) = \frac{1}{2}$

$P(brake = Y) = \frac{5}{7}$ $\quad$ $P(brake = N) = \frac{2}{7}$

brake?	distance	speed
Y	near	slow
Y	near	fast
N	far	fast
Y	far	fast
N	near	slow
Y	far	slow
Y	near	fast

Naïve Bayes Classifier Example (4)

$$P(brake|distance, speed) = \alpha P(distance|brake)P(speed|brake)P(brake)$$

$$P(brake = Y|near, slow) = \alpha P(near|brake = Y)P(slow|brake = Y)P(brake = Y)$$
$$= \alpha\left(\frac{3}{5}\right)\left(\frac{2}{5}\right)\left(\frac{5}{7}\right) = \frac{6}{35}\alpha$$
$$P(brake = N|near, slow) = \alpha P(near|brake = N)P(slow|brake = N)P(brake = N)$$
$$= \alpha\left(\frac{1}{2}\right)\left(\frac{1}{2}\right)\left(\frac{2}{7}\right) = \frac{1}{14}\alpha$$

Since $\frac{6}{35}\alpha > \frac{1}{14}\alpha$ (regardless of the value of α as long as it is positive),

$P(brake = Y|near, slow)$>$P(brake = N|near, slow)$, thus the AI predicts that the player will apply brake when distance is near with slow speed

You should carry out similar calculations to determine whether the AI predicts that the player will apply brake under the conditions (near, fast), (far, slow), (far, fast)

Implementation

- One of the problems with multiplying small numbers together (like probabilities) is that, with the finite precision of floating point, they very quickly lose precision and eventually become zero
- The usual way to solve this problem is to represent all probabilities as logarithms and then, instead of multiplying, they are added. It is denoted by the term “log-likelihood”

Reference

- Artificial Intelligence for Games
  - Chapter 7.3, 7.5