CS6046 Multi-Armed Bandits: Assignment 1 Report

-Akash Reddy A, EE17B001

1 The Algorithm

The game of Cats and Dogs prescribed is one where the adversary gives feedback based on each guess the algorithm makes. The information is not obtained in batches or as an entire dataset, therefore the learning has to be online. Of course, we can accumulate the information over multiple guesses, prepare a batch, and learn on it. However, since we wish to minimise the number of guesses, it would make more sense to learn from each adversary response that we get at the end of each round, and try to reduce the eventual number of guesses with each round's information, instead of waiting to gather more.

Therefore, an online learning algorithm is used where we learn from each example/guess.

Overview: After filtering the given list of words to consider only the words with no repeating letters (initially 5526 words, 4309 words left after removal of words with repeated letters), the algorithm makes a guess and obtains an ordered pair (cats, dogs) from the adversary, based on which it can refine its guess. This particular algorithm uses the (cats, dogs) information of the current guess with the adversary word, to reduce the set of candidate words to those that share the same (cats, dogs) values with the guess as the adversary word does. The adversary word also necessarily belongs to this truncated set which is the new set of candidate words, and we repeat until the required word is found.

At the very first round when there is no information whatsoever, any word is equally likely to be the adversary word (let us denote it by W). However, we can make low number of mistakes/guesses if we make a pick at each round such that the cardinality of the set of candidate words remaining at the end of the round is smallest. This is reminiscent of the Halving Algorithm discussed in class, where we try to ensure that the cardinality of the set that the adversary points us towards is the smallest.

The steps of the algorithm are described below, with further explanation on individual steps:

1. We use the metric of "average length of next list across all possible adversary words" (let us denote it by L_n) to measure the above-mentioned cardinality.

   Consider that the (cats, dogs) pair between the ith and the jth word is denoted by cd_{ij}. **We calculate the metric L_n for the ith word** as follows:

   $$(L_n)_i = \frac{1}{|P|}\sum_{j=0}^{|P|}|P'_{cd_{ij}}|$$

   where $|P'_{cd_{ij}}|$ is the size of the set of all words k such that $cd_{ik} = cd_{ij}$. Thus, we get the average length of the next list over all adversary words (summation over j), for the ith word.

   Then, **we pick the guess G from the list of possible words P which has the lowest value of L_n**, such that:

   $$G = argmin_i(L_n)_i$$

   and pass it to the adversary, who returns a (x, y) cats-dogs ordered pair.

2. Now, the number of cats = x and number of dogs = y is all the information we have. We know neither the positions of these cats and dogs, nor which letters they are. Moreover, the (cats, dogs) pair is a commutative relationship that holds both ways - if a word A shares (x, y) cats and dogs with B, then B also shares (x, y) cats and dogs with A.

   Therefore, since G shares (x, y) with W, W must also share (x, y) with G. **We use this information by filtering the current list P for the words that share (x, y) with G, and make them the new list of possible words P'_{xy} that can be our guess for W.** W cannot lie outside this new list P'_{xy}, since it definitely shares (x, y) with G. Since we picked the best G such that the average length of next set L_n is smallest, we know that P'_{xy} is the smallest set we can reduce P to, on average.

   Moreover, this is the best way to use this information to truncate the list of possible words - we know that W shares **exactly** (x, y) cats and dogs with our current guess G. Therefore, we take only the words which share **exactly** (x, y) cats and dogs with G. There is no use considering, for instance, words that share **atleast** (x, y) cats and dogs instead. This new list still contains W, but is simply larger and leads to more guesses prior to the answer. Similarly, there is no use considering all words with whom all $x + y$ shared letters are cats, for the same reason (unnecessarily larger list).

3. Set $P = P'_{xy}$ and go back to step 1 and repeat, until we either guess the adversary word correctly, or only one word remains in the list of possible words. It turns out that the word that remains is always the adversary word that we are looking for.

1.1 Some Implementation Details

- Our step 1 requires the calculation of the (cats, dogs) metric for each pair of (guess word, adversary word) when we calculate the average length metric $(L_n)_i$ for each i, and minimise over i to pick the guess. In the first round, our set of possible words includes all 4309 words. This means $4000 \times 4000 \approx 1.6 \times 10^7$ iterations of calculating the (cats, dogs) metric, once for each time we play the game. This is highly computationally expensive.

  One workaround we have is to create an initial cats-dogs matrix CD such that:

  $$CD[i, j] = cd_{ij}$$

  This matrix can be used to look up the (cats, dogs) values of any pair of words $(i, j) \epsilon P$ whenever needed without the overhead of calculation for every round in a single game, and even each time the game is played. For each round, we obtain the new CD matrix by taking the rows and columns of the original CD matrix corresponding to the words remaining in the candidate list.

- $|P'_{cd_{ij}}|$ can now be calculated using the matrix CD. The ith row of CD contains the cd (cats, dogs) values for i with each adversary word j. Given a certain word i, $|P'_{cd_{ij}}|$ is the length of the list of words k such that $cd_{ik} = cd_{ij}$. We obtain this number by

  - Fixing an adversary word j.
  - Counting the number $m_{cd_{ij}}$ of words k in the ith row of CD such that $cd_{ik} = cd_{ij}$. Since the cd metrics that we need to count are all equal to cd_{ij}, we simply take the frequency of cd_{ij} itself in the ith row to obtain the value of $m_{cd_{ij}}$. Hence, the value of $|P'_{cd_{ij}}| = freq(cd_{ij}) = m_{cd_{ij}}$.

  Also, in each row, all these $m_{cd_{ij}}$ words have the same cd_{ij}, therefore $|P'_{cd_{ij}}| = m_{cd_{ij}}$ for all these $m_{cd_{ij}}$ words. We can take advantage of this fact during computation like so - from each unique cd pair, we get $m_{cd_{ij}} \times m_{cd_{ij}} = m^2_{cd_{ij}}$ contribution towards the summation $\sum_{j=0}^{|P|} |P'_{cd_{ij}}|$. We have:

  $$\frac{1}{|P|} \sum_{j=0}^{|P|} |P'_{cd_{ij}}| = \frac{1}{|P|} \sum_{cd_{ij}=(0,0)}^{(4,4)} m^2_{cd_{ij}} = (L_n)_i$$

  Therefore, all we need to do to calculate our average length of next set metric $(L_n)_i$ for the ith word is to take the frequency of each unique cd value in the ith row and add up their square values, and divide

it by the number of words in the set P! We then minimise $(L_n)_i$ over all i to obtain the best guess for the algorithm in the current round.

We also pre-calculate the best guess of word in the very first round, since it is calculated over all the possible words. There is no need to recalculate it for different adversary words.

Pseudocode of Core Algorithm

```
 W = adversary_word
P = list_of_all_possible_words
G = bestChoice(P)

while (len(P) > 1) and (G != W) {
    (x,y) = catsDogs(W, G)
    P' = []
    for word in P
        if (catsDogs(word, G) == (x,y))
            add word to P'
    P = P'
    G = bestChoice(P)
}

final_guess = G

**********
function catsDogs(word1, word2):
    y = 0, x = 0
    common_letters = len(intersection(set(letters in word 1), set(letters in word 2)))

    for (i = 0, i < len(word1), i++)
        if (word1[i] == word2[i])
            y++

    # (Dogs are calculated now)

    x = common_letters - y
    # (Remaining common letters must be cats)

    return (x,y)

**********
function bestChoice(P'):
    avg_lengths = []
    if (len(P')==len(P))
        matrix = initial_CD_matrix
    else
        matrix = initial_CD_matrix[rows and columns corresponding to words in P']

    for row in matrix
        append sum of squares of all unique cd values in row, to avg_lengths
    best_choice = word in P' corresponding to argmin(avg_lengths)

    return best_choice
```

2 Results

Upon running the algorithm over each adversary word and taking the average of all these numbers of guesses, we get the overall average number of guesses over all possible words.

Average number of guesses over all possible words ≈ 6.09

Toughest word to figure out: WATS

Corresponding highest number of guesses ≈ 16

In order to get a feel for the number of guesses made, we plot the histogram of the number of words that took each unique number of guesses.

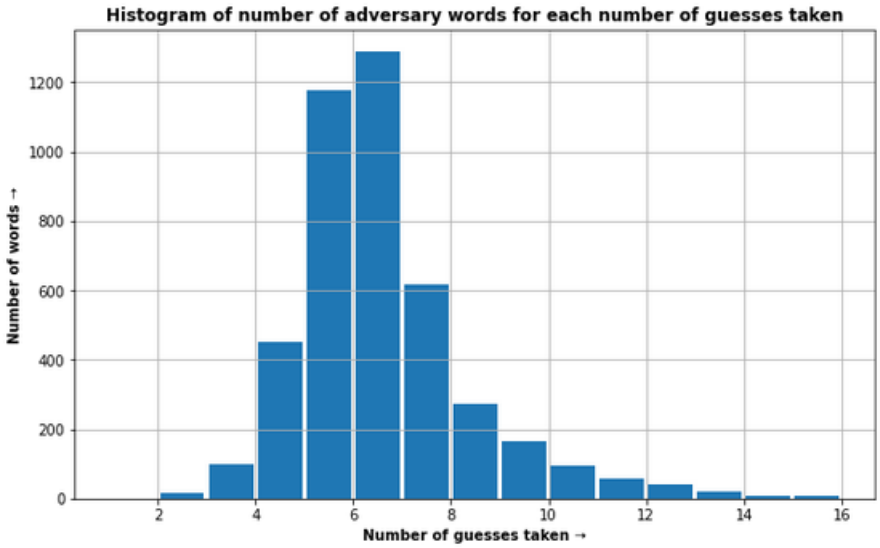

Figure 1: Histogram of number of words for each unique number of guesses

The curve is bell-shaped due to the large number of words in the dataset - most words tend to take an average number of guesses with only a few words on either extreme. However, it is a right-skewed distribution, which means there are enough words that take fewer guesses to balance out the words that take larger number of guesses.

In order to try to understand why WATS is the hardest word to guess, we track the history of guesses until we arrive at WATS while following the algorithms.

```
['TAES', 'TASK', 'BATS', 'CATS', 'FATS', 'GATS', 'HATS', 'LATS', 'MATS', 'NATS', 'OATS', 'PATS', 'QATS', 'RATS', 'VATS', 'WATS']
```

Figure 2: History of guesses until WATS

We observe that most of the guesses end in ATS. When the remaining list of words is all words that end in ATS, any guess apart from the right one will give (0 cats, 3 dogs). Therefore, we cannot eliminate any words apart from the guess once we reach this list, and the detection of the right word is completely dependent on randomness. In the worst case, if there are K words ending in ATS, it can take K guesses to land on the right word while eliminating all the wrong words one-by-one.

In fact, whenever we reach a point where the list has (0 cats, 3 dogs) left, the algorithm slows down drastically in detecting the right word. All words with a large number of (0 cats, 3 dogs) co-words will take long to guess. It turns out that our manner of identifying best guesses based on least average future cardinality takes the longest time for the large set of "ATS" words. Within these words, the random seeding of Python 3 selects the guesses in such a way that WATS takes the longest.

3 Optimality Analysis

This algorithm is near-optimal, because at each round of the game, the only information we get is the exact number of cats and dogs shared with the adversary word. The algorithm makes best use of this information to shorten the list of candidate words. Once whatever information we have obtained from the adversary responses has been exhausted, we then select the word that has the lowest measured cardinality for the next round's list of words. This is in line with the Halving algorithm, where we try to make the adversary push the algorithm in the direction of least cardinality of remaining hypotheses, at each round.

However, this algorithm may be sub-optimal owing to the following potential factors:

- The derived metric of average next list length over all adversary words $(L_n)_i$ need not be the best way of picking the least cardinality. The cardinality differs based on the adversary word. However, incorporating this improvement is computationally very expensive, so we settle for this average metric of next list cardinality instead.
- The usage of Littlestone dimension (depth of mistake trees) instead of cardinality to make a guess could be even more optimal, similar to the Standard Optimality Algorithm (SOA).

****