Cats and Dogs

Multi-Armed Bandits - Assignment 1

PREPARED FOR

Assignment 1
CS6046: Multi-Armed Bandits

PREPARED BY

Pragalbh Vashishtha

MM19B012

SUMMARY

The problem has been modelled as follows:

This problem is similar to the Online Learning:- Realisable Adversary case. Each word can be treated as an ‘expert’ or hypothesis. There is only one correct hypothesis or word, and the number of hypotheses are finite. However there are a few differences; the adversary waits for the algorithm’s prediction and gives a score to it. Each hypothesis also gives a prediction and is divided into sets on that basis. Based on the score, wrong hypotheses sets are eliminated. The score is the “cats and dogs” which are entailed in the problem statement.

As the algorithm is required to move first, the majority doesn’t come into the picture. Each round, a word from the correct class from the previous round is chosen. Initially, a random word was chosen, however later the algorithm was modified to choose a word with the most ‘branching’, i.e, the other words will produce more combinations of ‘nC mD’ (where n,m are numbers). If branching is increased, the algorithm can discard more hypotheses each round and reach the adversary’s chosen hypothesis.

*This problem is similar to the ‘true experts case’, however it is inverted. The algorithm also uses a similar approach to the majority algorithm and the Standard optimization algorithm, however has no solid Littlestone dimension and hence employs branching heuristic

Pseudocode:

```
current list of hypotheses=total list

while the Current list of hypotheses is not empty do:

      choose hypothesis (A) that produces most branching

      for each hypothesis in current list do:

            separate hypothesis into sets based on predictions
            generated by comparing each hypothesis to selected
            hypothesis A. (Cats and Dogs)

            if an unseen prediction is encountered:
                  create a new set and add hypothesis

      print the hypothesis A as output and receive prediction from
      adversary

      if prediction is "0C 4D", all letters are in correct
      positions: return hypothesis A

      else: current list of hypotheses= set of hypotheses with
      predictions that match the adversary
```

The various Modules of the Algorithm:-

1. Basic functions:-

   a. To get the prediction values from the hypothesis, a cats and dogs function is used:. It is called `cd(s1,s2)` and returns cats and dogs in the form of 'nC mD', where n,m are integers
   b. The function `return_dataset()` returns the hypothesi from a csv containing all the words in the website. Another function `finalist(datalist)` takes the list of csv extracted and removes words which have repeating letters. After this preprocessing, we have got our hypothesis set
2. Auxiliary functions for the algorithm:-
   a. The `ldim(word,aset)` function is named after the littlestone dimension, as it is inspired from the same, however due to the differences between the Cats and Dogs problem and the Online learning with experts case, it only does branching. It counts the branches that are formed, i.e, new predictions. By maximising the branches using the conjunct function `best_choice(aset)`, it is argued that more pruning of incorrect hypotheses (sets) can be done each round.

Timeline and developments

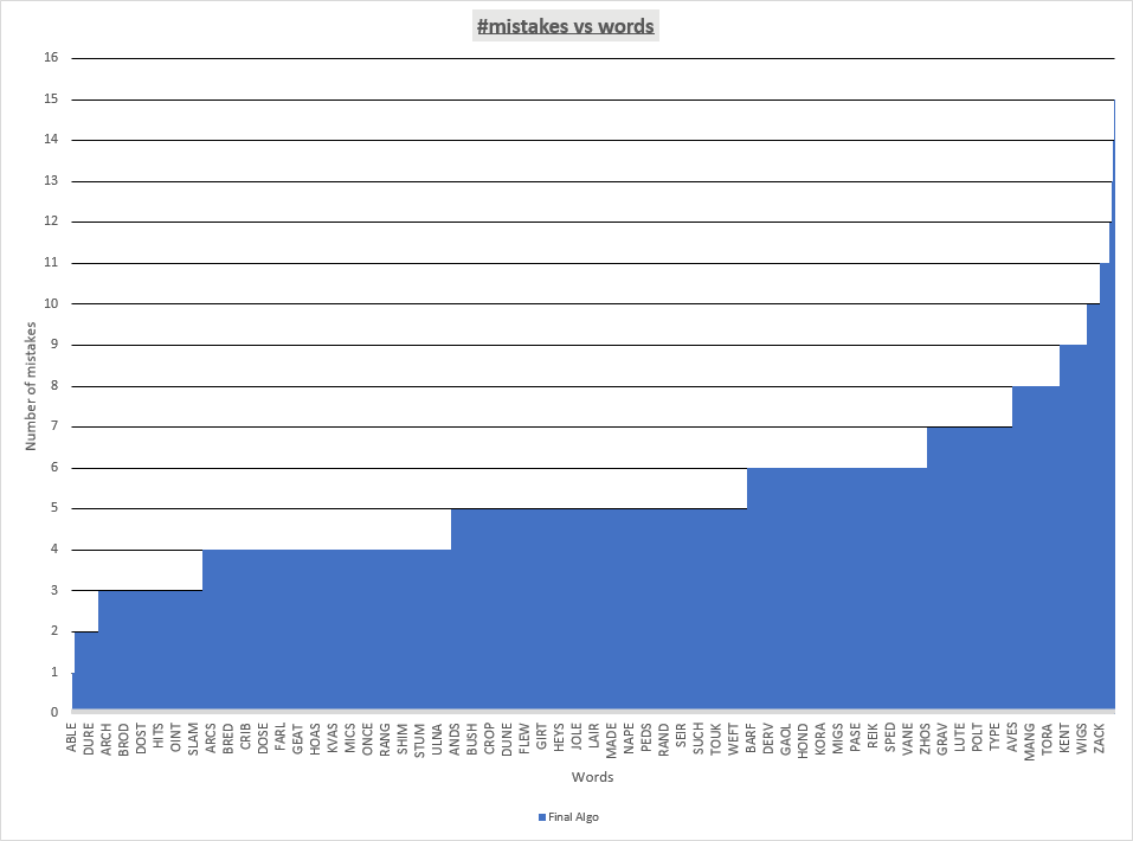
- The cd function for returning cats and dogs was developed. Along with this, the algorithm was developed at a basic level.
- The initial version of the algorithm used the first element of the round's set as the base hypothesis for comparison. This was suboptimal as it did not take advantage of the pruning that can be caused by higher branching
- Soon after this, branching heuristic based ldim was introduced. This reduced the average mistakes made by the algorithm.
- To increase computational speed to compare all the words as 'test', memoization was done. This is not reflected in the submission as it is not required there

Analysing the results of the algorithm

- In the initial case the algorithm had no branching these were the observed statistics:-
  - ABED is the word with 0 mistakes (chosen to be the first hypothesis)
  - The average number of mistakes are 5.733348804827106
  - The word with maximum number of mistakes is ZEST with 15 mistakes
- In the new algorithm branching heuristic is utilized. These were the statistics observed
  - ABLE is the word with 0 mistakes (chosen to be the first hypothesis)

- The average number of mistakes are 5.210721745184498. This is a clear upgrade of more than 0.5 mistakes over the set
- The word with the maximum number of mistakes is ZINS with 15 mistakes. The mistake bound remains the same

- Below is a graph of the mistakes



Suboptimality and scope of optimization

The algorithm uses the branching heuristic and hence only looks at the number of 'children' subsets each round. It is like counting the branches in the tree for the littlestone dimension. As the level of analysis is only one round deep, it is not optimal. To make it optimal, a tree similar to that in the littlestone dimension needs to be constructed and then used. If that is done the algorithm will be optimal. However the time complexity of that algorithm is large and even after memoization it did not complete as of time of writing this report. The optimal algorithm will make the least amount of mistakes and lower average as well.

Code attached for reference:

```
def finalgo():
      ################### Base defs#################

      def predict(word,aset):
      a=[]
      for i in aset:
            a.append([i,cd(word,i)])
      return a
      ##############################################
      #we choose the word that produces "greatest branching".
      totset=datset
      count=0
      dictset={}
      while totset:
#           word=totset[0]
      word=best_choice(totset)
      #now we find yt predicts
      ytot=predict(word,totset)
      for i in ytot:#getting yts
            y=i[1]
            w=i[0]
            if y in dictset:
                  dictset[y].append(w)
            else:
                  dictset[y]=[w]
      print(word)
      yt=input()
      if yt=="0C 4D":
            return word
#           return count
      else:
            totset=dictset[yt]
            count+=1
#           tree.append([count,totset])
            dictset={}
      return ""
```

Thank you