Playing with the Melbot

Binchilling
CS771 - Introduction to Machine Learning
Indian Institute of Technology, Kanpur
2022-23 Semester II

1 Notation

We define the notations which will be consistently used throughout this report.

$Tree \rightarrow$ The trained decision tree object

$Node \rightarrow$ An internal node of the tree storing its parent and children along with query history and word history

$Leaf \rightarrow$ The last node in a particular branch and query history which returns the final predicted word

$Entropy \rightarrow$ The measure of uncertainty or randomness of a distribution. We use the discrete set notion of entropy

$D_i \rightarrow$ Dictionary of words that reach a node 'i'

$\#(X) \rightarrow$ Number of elements of a set X

The dictionary of words given to us contains 5167 words. Each word is written using the Latin lowercase letter **a - z**.

2 Training a Decision Tree

The decision tree consists of three types of nodes:

1. Root
2. Internal Node
   (a) Parent
   (b) Child
3. Leaf

Essentially, we apply the ID3 algorithm with a slight change. An internal node splits into children nodes based on an optimal query word $w*$ which tries to minimize the entropy post split. For a node n, the optimal query word is chosen **from** D_n. This is guaranteed to always split an internal node, so we may not end up in a node which cannot be split.

Initially, implementation of the actual ID3 algorithm, where the optimal query word was chosen from the entire dictionary took almost 3 minutes to train and gave higher average query count than the current algorithm.

The pseudo-code of the algorithm is given below:

For a set X_0 splitting into K sets $X_i \, \forall \, i = \{1, 2, \cdots, K\}$, with a query w, the entropy for this split is defined in the following manner:

$$H_w(X_0) = \sum_{i=1}^{K} \log_2(\#(X_i)) \times \frac{\#(X_i)}{\#(X_0)}$$

Since $\#(X_0)$ is a normalization constant, to speed up computation, we ignore this term while choosing the maximum entropy decrease.

For a node N_0 we obtain node split, $N_j \, \forall \, j = \{1, 2, \cdots, M\}$. We decide the optimal word w^* based on the following minimization problem:

$$w^* = \underset{w \in D_{N_0}}{\arg\min} \; H_w(D_{N_0}) \tag{1}$$

Expanding the definition of $H(D_{N_i})$ we get:

$$w^* = \underset{w \in D_{N_0}}{\arg\min} \; \sum_{i=1}^{M} \log_2(\#(D_{N_i})) \times \#(D_{N_i})) \tag{2}$$

Thus Melbo needs to ask w^* at each node to optimally reach solution. The node N_0 is the parent node while its children are $N_j, j \in \{1, 2, \cdots, M\}$

Next we need to decide to assign a leaf node, which records Melbo's final response. A node i is declared as a leaf node when $\#(D_i) = 1$ or depth of a node is more then the parameter `max_depth`. In case we reach the maximum depth before ending up with a single word at a leaf, the first index of D_i is returned.

We train the model with `max_depth` = 15.

3 The Code

```python
import numpy as np

# You are not allowed to import any libraries other than numpy

# SUBMIT YOUR CODE AS A SINGLE PYTHON (.PY) FILE INSIDE A ZIP ARCHIVE
# THE NAME OF THE PYTHON FILE MUST BE submit.py
# DO NOT INCLUDE OTHER PACKAGES LIKE SKLEARN, SCIPY, KERAS ETC IN YOUR CODE
# THE USE OF PROHIBITED LIBRARIES WILL RESULT IN PENALTIES

# DO NOT CHANGE THE NAME OF THE METHOD my_fit BELOW
# IT WILL BE INVOKED BY THE EVALUATION SCRIPT
# CHANGING THE NAME WILL CAUSE EVALUATION FAILURE

# You may define any new functions, variables, classes here
# For example, classes to create the Tree, Nodes etc

class Tree:
	def __init__( self, min_leaf_size, max_depth ):
		self.root = None
		self.words = None
		self.min_leaf_size = min_leaf_size
		self.max_depth = max_depth

	def fit( self, words, verbose = False):
		self.words = words
		self.root = Node( depth = 0, parent = None )
		if verbose:
```

```
			print( "root" )
			print( "", end = '' )
		# The root is trained with all the words
		self.root.fit( all_words = self.words, my_words_idx = np.arange( len( self.words ) ), min_leaf_size = self.min_leaf_size, max_depth = self.max_depth, verbose = verbose )

class Node:
	# A node stores its own depth (root = depth 0), a link to its parent
	# A link to all the words as well as the words that reached that node
	# A dictionary is used to store the children of a non-leaf node.
	# Each child is paired with the response that selects that child.
	# A node also stores the query-response history that led to that node
	# Note: my_words_idx only stores indices and not the words themselves
	def __init__( self, depth, parent ):
		self.depth = depth
		self.parent = parent
		self.all_words = None
		self.my_words_idx = None
		self.children = {}
		self.is_leaf = True
		self.query_idx = None
		self.history = False

	# Each node must implement a get_query method that generates the
	# query that gets asked when we reach that node. Note that leaf nodes
	# also generate a query which is usually the final answer
	def get_query( self ):
		return self.query_idx

	# Each non-leaf node must implement a get_child method that takes a
	# response and selects one of the children based on that response
	def get_child( self, response ):
		# This case should not arise if things are working properly
		# Cannot return a child if I am a leaf so return myself as a default action
		if self.is_leaf:
			print( "Why is a leaf node being asked to produce a child? Melbot should look into this!!" )
			child = self
		else:
			# This should ideally not happen. The node should ensure that all possibilities
			# are covered, e.g. by having a catch-all response. Fix the model if this happens
			# For now, hack things by modifying the response to one that exists in the dictionary
			if response not in self.children:
				print( f"Unknown response {response} -- need to fix the model" )
				response = list(self.children.keys())[0]

			child = self.children[ response ]

		return child

	# Dummy leaf action -- just return the first word
	def process_leaf( self, my_words_idx, history ):
		return my_words_idx[0]

	def reveal( self, word, query ):
		# Find out the intersections between the query and the word
		mask = [ *( '_' * len( word ) ) ]

```

```
        for i in range( min( len( word ), len( query ) ) ):
            if word[i] == query[i]:
                mask[i] = word[i]

        return ' '.join( mask )

    # Dummy node splitting action -- use a random word as query
    # Note that any word in the dictionary can be the query
    def process_node( self, all_words, my_words_idx, history, verbose ):
        # For the root we do not ask any query -- Melbot simply gives us the length of the secret word
        best_split_dict = {}
        best_query = ""
        best_query_idx = 0
        best_entropy = len(all_words) * 100000          # very high default entropy value
        if history == False:
            best_query_idx = -1                         # this ensures that the root node is split into children containing words of different lengths
        else:
            for try_idx in my_words_idx:                # try every word in the dictionary as a query
                split_dict = {}
                cnt_entropy = 0
                for idx in my_words_idx:
                    mask = self.reveal(all_words[ idx ], all_words[try_idx] )
                    if mask not in split_dict:
                        split_dict[mask] = []
                    split_dict[mask].append(idx)


                for split in split_dict.values():
                    cnt_entropy += len(split) * np.log2(len(split))

                if cnt_entropy < best_entropy:
                    best_entropy = cnt_entropy
                    best_query = all_words[try_idx]
                    best_query_idx = try_idx

        for idx in my_words_idx:
            mask = self.reveal( all_words[ idx ], best_query )
            if mask not in best_split_dict:
                best_split_dict[ mask ] = []

            best_split_dict[ mask ].append( idx )

        if len( best_split_dict.items() ) < 2 and verbose:
            print( "Warning: did not make any meaningful split with this query!" )

        return ( best_query_idx, best_split_dict )

    def fit( self, all_words, my_words_idx, min_leaf_size, max_depth, fmt_str = "    ", verbose = False ):
        self.all_words = all_words
        self.my_words_idx = my_words_idx

        # If the node is too small or too deep, make it a leaf
        # In general, can also include purity considerations into account
        if len( my_words_idx ) <= min_leaf_size or self.depth >= max_depth:
```

```
            self.is_leaf = True
            self.query_idx = self.process_leaf( self.my_words_idx, self.history )
            if verbose:
                print( '' )
        else:
            self.is_leaf = False
            ( self.query_idx, split_dict ) = self.process_node( self.all_words, self.my_words_idx, self.history, verbose )

            if verbose:
                print( all_words[ self.query_idx ] )

            for ( i, ( response, split ) ) in enumerate( split_dict.items() ):
                if verbose:
                    if i == len( split_dict ) - 1:
                        print( fmt_str + "", end = '' )
                        fmt_str += "     "
                    else:
                        print( fmt_str + "", end = '' )
                        fmt_str += "    "

                # Create a new child for every split
                self.children[ response ] = Node( depth = self.depth + 1, parent = self )
                history = self.history
                history = True
                self.children[ response ].history = history

                # Recursively train this child node
                self.children[ response ].fit( self.all_words, split, min_leaf_size, max_depth, fmt_str, verbose )

################################
# Non Editable Region Starting #
################################
def my_fit( words ):
################################
#  Non Editable Region Ending  #
################################

    # Use this method to train your decision tree model using the word list provided
    # Return the trained model as is -- do not compress it using pickle etc
    # Model packing or compression will cause evaluation failure
    model = Tree(min_leaf_size=1, max_depth=15)
    model.fit(words)

    return model                                    # Return the trained model
```