BÁO CÁO TUẦN 2

Người làm: Trần Đức Thọ

**Bài 1**

def sparseSearch(arr , key , low , high):

    left = 0; right = 0

    while low <= high:

            mid = low + (high - low) // 2

            if arr[mid] == '':

                left = mid - 1

                right = mid + 1

                while True:

                    if left < low and right > high:

                        return -1

                    elif left >= low and arr[left] != '':

                        mid = left

                        break

                    elif right <= high and arr[right] != '':

                        mid = right

                        break

                    left -= 1

                    right += 1

            if arr[mid] == key:

                print(mid)

                return mid

            elif arr[mid] > key:

                high = mid - 1

            elif arr[mid] < key:

                low = mid + 1

    return -1

if \_\_name\_\_ == '\_\_main\_\_':

    arr = ["for", "", "geeks", "ball", "", "", "ide",

           "practice", "", "", "", "quiz"]

    key = 'geeks'

    low = 0

    high = len(arr) - 1

    sparseSearch(arr , key , low , high)

ý tưởng thuật toán:

tìm điểm nằm giữa của mảng

sử dụng thuật toán tìm kiếm để giải quyết bài toán.

**Bài 2:** Trình bày paper “LightGBM: A Highly Efficient Gradient Boosting Decision Tree”

Tổng quan: Gradient Boosting Decision Tree là thuật toán học máy được sử dụng rộng rãi, do hiệu quả độ chính xác và khả năng diễn giải của nó, nó thường được nhắc nhiều trong các tác vụ học máy như phân loại nhiều lớp , xếp hạng, dự đoán.

Với sự phát triển của dữ liệu lớn, chúng ta luôn phải cân nhắc giữa hiệu quả và chi phí với từng loại mô hình.

Gradient Boosting Decision Tree luôn phải quét toàn bộ các trường hợp nên khi số lượng instances và số features tăng, độ phức tạp tính toán tăng.

Để giải quyết bài toán này, ý tưởng là giảm số lượng features và số instances, nhưng để thực hiện việc này thì rất khó.

Bài báo đề xuất ra 2 ý tưởng đó là Gradient-based One-Side Sampling (GOSS) và Exclusive Feature Bundling (EFB)

* Gradient-based One-Side Sampling (GOSS): Nên giữ tốc độ học lớn với 1 số cá thể dữ liệu.
* Exclusive Feature Bundling (EFB): với các features độc quyền, tức là hiếm khi chúng hiếm khi nhận các giá trị đồng thời, chúng ta đóng gói đồng thời tập trung

Phân tích thuật toán Gradient Boosting Decision Tree: thuật toán GBDT là thuật toán phân tích từ các điểm dữ liệu từ ra thành các nút lá, từ độ chênh lệch của đầu ra thực tế và trên lí thuyết, chúng ta sẽ xây dựng tốc độ học cho bài toán, cứ tiếp tục các vòng lặp như vậy cho đến khi độ chênh lệch nhỏ hơn 1 ngưỡng nào đó thì có thể dừng.

Tức là ở đây chúng ta cần tốn O(#data × #feature) cho việc xây dựng biểu đồ và O(#bin × #feature) cho việc tìm điểm phân chia. Như vậy, nếu có thể giảm số data và số feature xuống chúng ta có thể rất có lợi và đẩy nhanh tiến độ thuật toán GBDT.

Có một cách để có thể giảm kích thước đào tạo đó là phương pháp lấy mẫu với tỉ lệ nhất định, tỉ lệ sẽ liên tục được chuyển đổi trong quá trình traning mô hình để đưa ra tỉ lệ lấy mẫu một cách hợp lý nhất. Tuy nhiên thuật toán GBDT không thể áp dụng phương pháp lấy mẫu vì nó không có trọng số gốc cho dữ liệu, không thế đánh giá để lất mẫu được.

Tiếp theo là vấn đề giảm số lượng các tính năng, chúng ta nên lọc các tính năng yếu. Các tính ăng thường được phân tích thành phần nguyên tắc hoặc đuổi theo phép chiếu. Tuy nhiên nhưng tính năng này dựa trên giả định các tính năng chứa những dư thừa đáng kể, tuy nhiên điều này hầu như khó xảy ra trong thực tế.

(2 thuật toán GOSS và EFB đều có khá nhiều về công thức toán nên em chưa thể hiểu hết, em sẽ cố gắng để trình bày trong tuần tới)

**Bài 2 (tuần 1)**

class Solution(object):

   def findWords(self, board, words):

      self.result = []

      t = {}

      for word in words:

         self.insert(word,t)

      for i in range(len(board)):

         for j in range(len(board[0])):

            self.solve(board,t,i,j)

      return self.result

   def solve(self,board,d,i,j,s=""):

      if i<0 or j<0 or i>=len(board) or j>=(len(board[0])):

         return

      l = board[i][j]

      if l in d:

         d = d[l]

         s+=l

         if "#" in d and d['#']:

            self.result.append(s)

            d['#'] = 0

         board[i][j] = '\*'

         if i+1<len(board) and board[i+1][j] in d :

            self.solve(board,d,i+1,j,s)

         if j+1 < len(board[0]) and board[i][j+1] in d:

            self.solve(board,d,i,j+1,s)

         if i-1>=0 and board[i-1][j] in d :

            self.solve(board,d,i-1,j,s)

         if j-1>=0 and board[i][j-1] in d :

            self.solve(board,d,i,j-1,s)

         board[i][j] = l

   def insert(self, word,t):

      current = t

      for i in word:

         if i not in current:

            current[i] = {}

         current =current[i]

      current['#']=1

ob = Solution()

print(ob.findWords([["o","a","a","n"],["e","t","e","a"],["i","h","k", "r"],["i","f","l","v"]],["oath","pea","tea","rain"]))

Sau khi tìm hiểu mô hình bài toán em có hiểu vấn đề như sau

- Tạo 1 bảng 2 chiều chứa các kí tự, các từ trong từ điển sẽ được tìm kiếm sẽ tuần tự theo chiều ngang hoặc chiều dọc và là các ô lân cận.

-Không lặp lại từ đã đi qua

Ý tưởng bài toán:

Tạo 1 kết quả mảng

Hàm solve với các đối số board, d, i, j ,s(s rỗng)

I,j không còn nằm trong bảng thì trả về false

L = board[i, j]

Nếu l có trong d thì:

* d = d[l] cho l vào s
* Nếu # nằm trong d và d [#] không rỗng, thì
  + Chèn s vào kết quả:
  + Đặt d[#]=0
* Board[I,j]=0;
* Cho I tiến lùi 1 giữ nguyên j hoặc j tiến lùi 1 gọi lại hàm đệ quy với hàm solve() (đối số I,j thay đổi thì thay đổi các đối số còn lại giữ nguyên)
* Tiếp theo chúng ta triển khai hàm insert() lấy từ và từ điển t
* Current=t
* Cho I chạy trong word
* Nếu I không có trong curren trả về rỗng
* current =current[i]
* current['#']=1

-đến method chính

-khởi tạo map t

-insert(word,t)

Solver(board,t,I,j)

Trả về result

(thuật toán trên tham khảo từ word search II em tìm kiếm được)