BK-Trees: Efficient Retrieval of Similar Strings

Natural Language Processing

Daniel Ortiz Martínez, David Buchaca Prats

Building a Basic Spellchecker

- Based on a notion that we have already seen, edit distance, we can build a spellchecker
- Let us consider that we want to correct misspelled words (i.e. words that are not in the vocabulary)
- Base algorithm: Let x be a sentence and V the vocabulary

```
for w in x
    if w not in V
        Find the closest words to w # (candidate search)
        Evaluate each of the candidate words # (candidate evaluation)
        Return the most probable candidate
```

Finding the Closest Items to a Query

- This is an extremely relevant problem for many data science and machine learning problems
- Scikit-learn implements the kdtree for dense vectors
- What if we have strings?

Finding the Closest Items to a Query

- Efficient search of similar values in a dataset is a very challenging problem. In particular, computing distances between a word and a huge vocabulary can be computationally expensive

- Let w be a string that is out of the vocabulary

- Let us consider $W_k(w; X) = \{w | w \in V, d(w, w_j) < k\}$

- Finding $W_k(w; X)$ can be done in two different ways:
    1. compute $d(w, w_j)$ for all w_j keeping elements at distance at most k
    2. Use a data structure to avoid computing $d(w, w_j)$ for all w_j in X

Tree Intuition

- We can build a tree to do efficient search of similar words. This will allow us to prune a lot of the search space, with the objective of avoiding many distance computations on a big part of the vocabulary

- Example: consider $w =$ pleistation
  ana $\rightarrow d$(pleistation, ana)
  playstation $\rightarrow d$(pleistation, playstation)
  house $\rightarrow d$(pleistation, house)

- If "pleistation" has 11 characters and we want candidates at most at distance $k = 3$, is there any need to compute d(pleistation, ana) ? Ana has 3 characters!

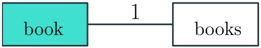
BK-Tree: Construction Algorithm

1. Select any word from the vocabulary and use it as the root node
2. Keep adding words until all vocabulary is in the tree
    2.1 Each time we add a word the distance between the word and the root node is computed, let us assume this distance is d
    2.2 If no node from the root node is at distance d we add a new leave as a descendant of the root node with edge value equal to d
    2.3 If there exist another node at distance d then we repeat this process redefining the root node as the node that produced the collision

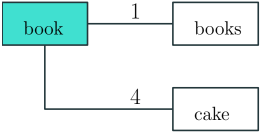
BK-Tree: Construction Example

- Let us consider the data [book, books, cake, boo, cape, cart, boon, cook]
- Insert **book** (which becomes root node)

- Insert **books**: compute d(book, books)=1

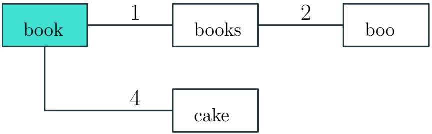


- Insert **cake**: compute d(book, cake)=4



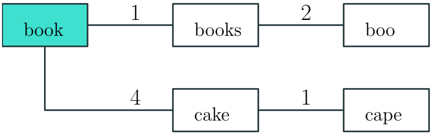
BK-Tree: Construction Example

- Insert **boo**: compute d(book, boo)=1
  - The BK-tree has to respect that every node have all children with different distances, since there is already a word at the same edit distance 1 we go to the branch of words at distance 1
  - If there is a collision (like we have now) the new word must become a children of the collisioned word. In this case, a children of "book"
  - The new distance from "books" to "boo" is 2



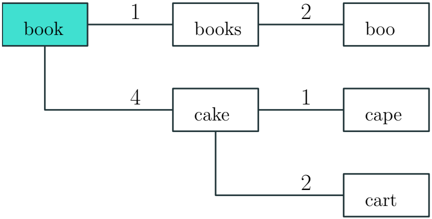
BK-Tree: Construction Example

- Insert **cape**: compute d(book, cape)=4
  - Collision! There is already cake at distance 4 from “book”
  - Root node is now “cake”
  - Root=cake: compute d(cake, cape)=1
  - There is no descendant from “cake” at distance 1 $\rightarrow$ we can add it



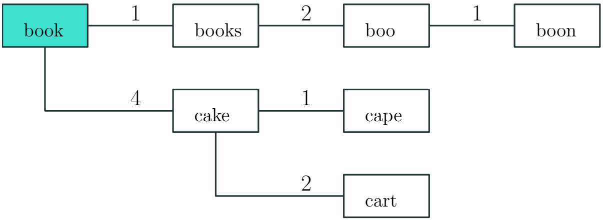
BK-Tree: Construction Example

- Insert **cart**: compute d(book, cart)=4
  - Collision! There is already cake at distance 4 from "book"
  - Root node is now "cake"
  - Root=cake: compute d(cake, cart)=2
  - There is no descendant from "cake" at distance 2 $\rightarrow$ we can add it



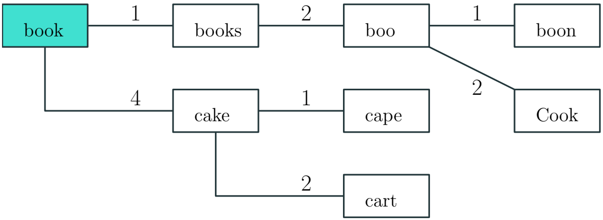
BK-Tree: Construction Example

- Insert **boon**: compute d(book, cart)=4
  - Collision! There is already cake at distance 4 from "book"
  - Root node is now "books"
  - Root=books: compute d(books, boon)=2
  - Collision! There is already "boo" at distance 2 from "boon"
  - Root=books: compute d(boo, boon)=1
  - There is no descendant from "boo" at distance 1 → we can add it



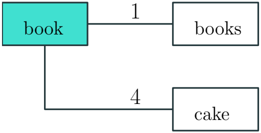
BK-Tree: Construction Example

- Insert **cook**: compute d(book, cook)=1
  - Collision! There is already cake at distance 1 from "book"
  - Root node is now "books"
  - Root=books: compute d(books, cook)=2
  - Collision! There is already "boo" at distance 2 from "cook"
  - Root=books: compute d(boo, cook)=2
  - There is no descendant from "boo" at distance 2 → we can add it



BK-Tree: Storage in Memory

- Let us consider the following tree:

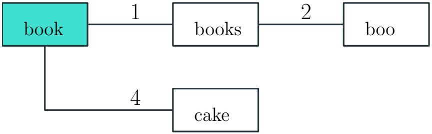


- We can use tuples to represent the tree in memory:
  - The first element is the word assigned to the node
  - The second element is the subtree that spawns from that node
  - A subtree can be represented as a Dict[Int, Tuple]
  - Keys are the distances to the root node
  - Values are tuples which represent subtrees

- For the previous example:

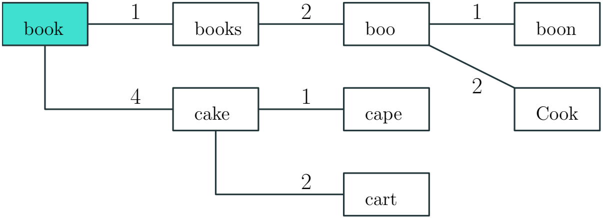
```
('book', {1: ('books', {}), 4: ('cake', {})})
```

BK-Tree: Storage in Memory




```
('book',
{1: ('books', {2: ('boo', {})}),
4: ('cake', {})})
```

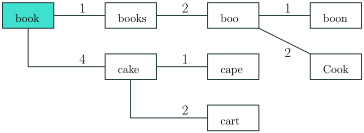
BK-Tree: Storage in Memory




```
('book',
{1: ('books', {2: ('boo', {1: ('boon', {}), 2: ('cook', {})})}),
4: ('cake', {1: ('cape', {}), 2: ('cart', {})})})
```

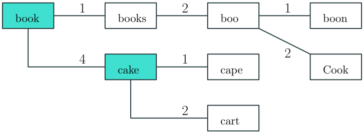
Searching in a BK-Tree

- Problem: search all words that appear at distance less or equal than a tolerance T form a query word q
- Bad solution: compute all edit distances between q and w for w in the vocabulary
- Key idea: visit all words w that are at distance $[d(w,q) - T, d(w,q) + T]$



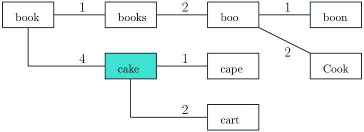
BK-Tree: Searching Example

- Let us consider q=caqe, T=1, `candidates`=[], `search`=[book]
- Select candidate "book" from `search`=[book]
  - d(book, caqe) = 4 → `candidates` is not updated
  - Crawl all children of "book" at distance `I`=[4-1,4+1]=[3,5]
  - Only node cake is connected to book and with distance in `I`=[3,5]
  - `search` = [book, cake]\book = [cake]



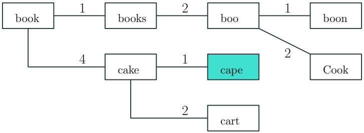
BK-Tree: Searching Example

- Let us consider q=caqe, T=1, `candidates`=[]
- Select candidate "cake" from `search`=[cake]
  - d(cake, caqe) = 1 → `candidates +=`[cake]
  - Crawl all children of "cake" at distance `I`=[1-1,1+1]=[0,2]
  - There are only 2 possible nodes, `search` = [cape, cart]



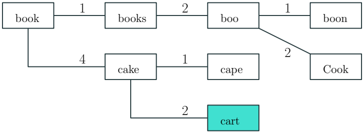
BK-Tree: Searching Example

- Let us consider q=caqe, T=1, `candidates`=[cake]
- Select candidate "cape" from `search`=[cape,cart]
  - d(cape, caqe) = 1 $\rightarrow$ `candidates +=`[cape]
  - Crawl all children of "cape" at distance `I`=[1-1,1+1]=[0,2]
  - "cape" has no children
  - `search` = [cape, cart]\cape=[cart]



BK-Tree: Searching Example

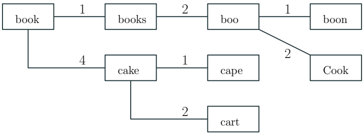
- Let us consider q=caqe, T=1, `candidates`=[cake,cape]
- Select candidate "cape" from `search`=[cart]
  - d(cart, caqe) = 2 $\rightarrow$ `candidates` is not updated
  - Crawl all children of "cart" at distance `I`=[2-1,2+1]=[1,3]
  - "cart" has no children
  - `search` = [cart]\cart=[] $\rightarrow$ Search space is empty, stop search



- The resulting set of possible candidates at distance 1 are: [cake,cape]

BK-Tree: Searching Example

- To sum up:
  - Start conditions: q=caqe, T=1, `candidates`=[], `search`=[book]
  - Result: the set of possible candidates at distance 1 are [cake, cape]
- Observation: we ended up computing 4 edit distances yet we have 8 nodes



BK-Tree Speedup

- In the case that the search space is drastically pruned, the speedup can be massive:

```
word = "anthropomorphologicaly"
max_dist = 2
sort_candidates=False
%timeit candidates_ext = get_candidates_exhaustive(word,max_dist,words)
```

245 ms ± 8.68 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

```
candidates_ext = get_candidates_exhaustive(word,max_dist,words)
candidates_ext
```

[(1, 'anthropomorphological'), (1, 'anthropomorphologically')]

```
word = "anthropomorphologicaly"
%timeit candidates_ext = t.query(word, 2)
```

123 µs ± 805 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

```
candidates_ext = t.query(word, 2)
candidates_ext
```

[(1, 'anthropomorphological'), (1, 'anthropomorphologically')]

BK-Tree Speedup

- If the pruned search space still contains a huge amount of words the speedup might note be that huge:

```
word = "astrologi"
max_dist = 2
sort_candidates=False
%timeit candidates_ext = get_candidates_exhaustive(word, max_dist, words)
```

221 ms ± 14.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

```
word = "astrologi"
%timeit t.query(word, 2)
```

99.6 ms ± 724 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)