Embedded Graph Representation

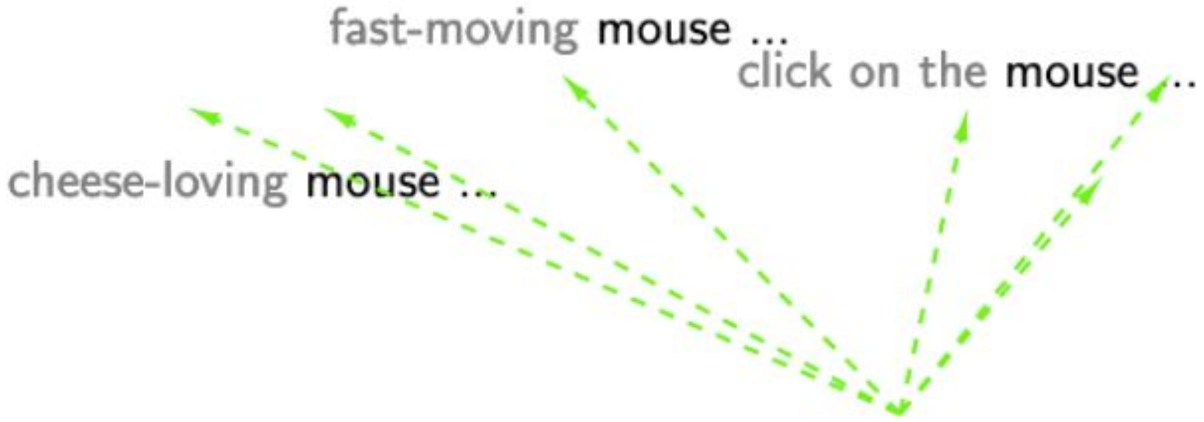
Problems with current approach

- Goal: Update graph with new documents and topics in streaming fashion
- Few existing online approaches and difficult to train
  - Online Hierarchical Dirichlet Process gives poor results
  - Topic distribution change too much over time

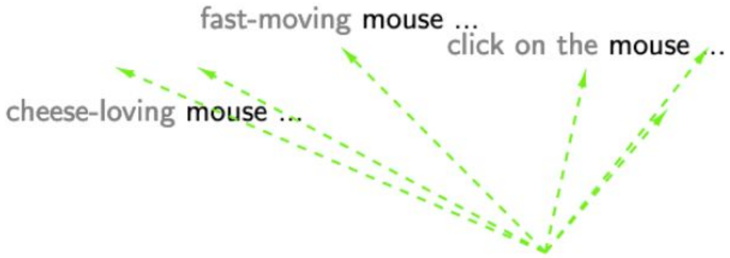
- Batch training necessary
- Need to connect topics trained from different batches
  - Highly heuristical with classical approaches like LDA
  - Leads to blow-up in graph size (edges between topics)

Why move to an embedding space?

- Contextualised vectorisation
- Allows streaming - all documents added to same space as vectors
- Clustering - Vectors can be compared in terms of similarity



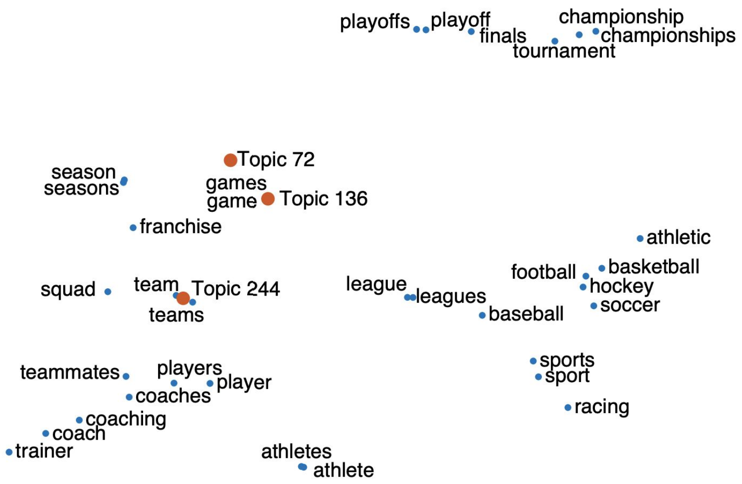
From unstructured text to contextual embedding



Storage: 4kb/word vector in binary format
English vocabulary: ~170,000 words in Oxford dictionary
Leads to upper bound of 680Mb

Idea

- Embedding-based topic modeling (Model: ETM)
- Words and topics represented as vectors
- Topics clustered together in vector space
  - From different batches
  - Topic hierarchy



Enriching embedding space

- Bringing the graph in the embedding space
  - More accurate
  - Faster querying

Embed the query!

- First stage: representing documents
  - Vector representation by
    - Convex combination of topic weights/probabilities
    - Learning the embedding (unsupervised representation learning)
  - Query ex: “Quantum Computing” → Embed & find neighbouring documents
- Second stage: authors, departments and organisations
  - Query ex: “Quantum Computing” → Embed & find neighbouring authors

Upper bound on storage requirement

```python
def compute_storage_upper_bound(nrpubs, nrdeps, nrorgs, nrpeople):
    """Gives a rough upper bound of the storage required for a graph (GB) with the given input parameter values"""
    import numpy as np

    # Record size per node: 15B
    # Record size per edge: 34B
    # Record size per attribute: 41B
    # Record size per string or array attribute: 128B
    # https://neo4j.com/developer/kb/understanding-data-on-disk/

    nrtopics = np.log(nrpubs) # assume that the number of topics grows logarithmically with the number of publications

    # for each type on node, multiply the number of nodes with the storage required for the node annd its attributes
    node_storage = (nrpubs*(15+2*41+4*128) + nrdeps*(15+41+128) + nrorgs*(15+41+128) + nrpeople*(15+41+128) +
            nrtopics*(15+41+128))

    dep_people_edges = nrdeps*40 # assume max 40 professors per department on average
    org_people_edges = nrorgs*5 # assume max 5 professors per organisation on average
    pub_people_edges = nrpubs*10 # assume max 10 authors per publication on average
    pub_topic_edges = nrpubs*20 # assume max 10 topics per publication on average

    # for each type on edge, multiply the number of nodes with the storage required for the node annd its attributes
    edge_storage = dep_people_edges*34 + org_people_edges*34 + pub_people_edges*34 + pub_topic_edges*(34+128)

    # storage required for indices
    # following neo4j heuristics: average property value size * (1/3)
    # we have four indices, one for each node
    avg_prop_size = (6*41+9*128)/15
    index_storage = avg_prop_size*(nrpubs + nrdeps + nrorgs + nrpeople)*(1/3)

    # add and return in GB
    return (node_storage + edge_storage + index_storage)/10**9
```

```python
compute_storage_upper_bound(170000, 16, 400, 10000)
```

```
0.7197434437472171
```