summary

inverted index

inverted index

- to improve search performance
- stores statistics
- bag of words: order is not important
- document data:
  - ids, metadata
  - doc length
  - avg doc length
- term data:
  - term frequency
  - document frequency
- dictionary:
  - posting list:
    - consists of `[doc-id:freq]` pairs for each term
    - doesn't store values with 0 freq

generating inverted index

- i) register metadata, assign doc id
- ii) tokenization
  - split on whitespace, punctuation character
  - keep abbreviations, names, numbers together
- iii) case folding
  - lowercase everything
- iv) stemming
  - reduce terms to 'root' form
  - ie. {changing, changed, change} $\mapsto$ chang
  - sometimes also lemmatization: {am, are, is} $\mapsto$ be
- v) filter stop words
  - ie. 'a', 'the', 'is', 'are'
- vi) add to dictionary, update posting list

querying inverted index

- find most relevant doc ids based on a scoring model (sum of scores for each query term)
- query types: exact matching, boolean queries, expanded queries (use all synonyms), wildcard queries, …
- spell checking: can suggest different queries
- lookup data-structure: hash table, prefix tree, b tree, …

scoring models

$\text{score}(q, d)$ = relevance score of query term q and document d

tf-idf

$$TF\text{-}IDF(q,d) = w_{t,d} = tf_{t,d} \cdot \underbrace{\log\left(\frac{|D|}{df_t}\right)}_{idf_t}$$

- term frequency within this doc
  - increases with the number of occurrences within a document, but logistically
  - $tf_{t,d} = \log(1 + f_{t,d})$
- inverse document frequency
  - the more docs contain the term, the less significant it is
  - $|D|$ = total num docs
  - df_t = num of docs with this term.
- generates weights used by other models (ie. vector space model vsm)

bm25

$$BM25(q,d) = \sum_{t \in T_d \cap T_q} \frac{tf_{t,d}}{k_1 \cdot ((1-b) + b \cdot \frac{dl_d}{avgdl}) + tf_{t,d}} \cdot \log\left(\frac{|D| - df_t + 0.5}{df_t + 0.5}\right)$$

- improvement over tf-idf: more saturated than logarithm as the term frequency increases
- variables:
  - $tf_{t,d}$ = term frequency
  - $|D|$ = num of all documents
  - df_t = document frequency of term = number of documents containing term
  - dl_d = doc length
  - $avgdl_d$ = average doc length
- hyperparameters:
  - k_1 = term frequency scaling
  - b = document length normalization

bm25f

$$BM25F(q,d) = \sum_{t \in T_d \cap T_q} \frac{\widetilde{tf}_{t,d}}{k_1 + \widetilde{tf}_{t,d}} \cdot \log\left(\frac{|D| - df_t + 0.5}{df_t + 0.5}\right)$$

$$\widetilde{tf}_{t,d} = \sum_{s=1}^{s_d} w_s \cdot \frac{tf_{t,s}}{(1-b_s) + b_s \cdot \frac{sl_s}{avgsl}}$$

- improvement over bm25: can weigh document segments ie. title, abstract, body
- each segment is called a 'stream'
- new variables:
  - sl_s = stream length
  - w_s = stream weight
  - $avgsl$ = average stream length (for that doc index)

evaluation

we want IR models to be effective, efficient (fast, scalable), interpretable.

online vs. offline eval

- online:
  - observing user behavior in production system
  - ie. user study through A/B testing
- offline:
  - prepared dataset
  - documents, queries, judgements (= test data, expected query results)

test collections

- documents
- queries
- relevance-judgements:
  - binary labels (relevant vs. not relevant) / graded labels (score usually between 0;3)
  - sparse (~1/query) / dense (>100/query)
  - implicit feedback / explicit feedback

metrics

- **precision P:
  - intuition: correctness
  - $P = TP/(TP + FP)$
- recall R:
  - intuition: completeness
  - $R = TP/(TP + FN)$

ranking metrics

binary labels:

- mean average precision MAP:
  - = sum of relative rel-doc positions / total num of rel-docs (but also measures area under precision-recall-curve, hard to interpret)
  - $MAP(Q) = \frac{1}{|Q|} * \sum_{q \in Q} \frac{\sum_{i=1}^{k} P(q)_{@i} \cdot rel(q)_i}{|rel(q)|}$
  - $P(q)_{@i}$ = precision metric at i – can be interpreted as position of rel-docs among other rel-docs
  - $rel(q)_i$ = relevance (binary)
  - $|rel(q)_i|$ = number of rel-docs in result
- mean reciprocal rank MRR:
  - = 1 / absolute position of first rel-doc
  - $MRR(Q) = \frac{1}{|Q|} * \sum_{q \in Q} \frac{1}{FirstRank(q)}$
  - $FirstRank$ = position of first rel-doc

graded labels:

- discounted cumulative gain DCG:
  - = relevance score / logarithm of absolute position for each doc (discounted means we're normallizing scores based on ranks)
  - $DCG(D) = \sum_{d \in D,\, i=1} \frac{rel(d)}{\log(i+1)}$
  - $rel(d)$ = relevance of doc for given query
  - i = absolute position in ranking
- normalized discounted cumulative gain nDCG:
  - = normalizes by best possible ranking per query, where docs are sorted based on their relevance
  - $nDCG(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{DCG(q)}{DCG(\text{sorted}(rel(q)))}$

statistical significance

- goal: proving that difference in two systems isn't by chance
- set a significance level / p-value first
- ie. 5% p-level means there is a 5% chance that the result is just by chance
- types:
  - paired: paired student's t-test, wilcoxson signed-rank test, …
  - non-paired: student's t-test, mann-whitney-u-test, …

creating test collections

see: https://ir-datasets.com/

we're measuring relevance to the information-need rather than the query. this is subjective.

- sampled (ie. ms-marco) → real usage query logs only accessible to search companies
- handcrafted (ie. trec) → expensive to hire people

steps:

- i. create k-cutoff-set of docs (= pooling-process)
  - use many diverse existing models to reduce work / number of initial documents to annotate
  - called candidate-docs
  - problem: lowers recall
- ii. let people annotate doc-query pairs
  - usually crowdsourced
  - majority voting for quality assurance = inter-annotator agreement (iaa) measured with kappa κ value
- iii. create a model
  - test model on full dataset and if it retrieves docs outside the k-cutoff-set, then assume that they're false positives
  - be aware of bias:
    - biased dataset → biased results, overfit models
    - word embeddings are trained on biased data from wikipedia
    - gender, racial, lingual bias
    - term-position bias: snippets in the beginning of passage are always more relevant
    - we're not sure if you can really de-bias data

representation learning

dense vector representations / embeddings = tensors that can capture a large subset of all possibilities in a latent/hidden space for chars/words/sentences

n-gram

- char-n-gram = capturing at n characters at a time
- word-n-gram = capturing at n words at a time
  - ie. each float in tensor is a weight describing relationships between words

word embeddings

- language modelling = unsupervised learning of vector representation for words
- word2vec:
  - 1-word-1-vector
  - input: 1-hot encoding of entire vocabulary, for each input word, chosen with a sliding window
    - cbow = predict word from context
    - skip-gram = predict context from word
  - output: 1-hot encoding of entire vocabulary, that's usually ignored → softmax probability of each word being in the context of the input $p(w_{t+j} \mid w_t)$
  - hidden: embeddings with trained weights
- fastText:
  - 1-word-1-vector
  - input doesn't take one-hot-encoded vocuabulary but char-n-grams / sub-words of vocabulary
  - can handle out-of-vocabulary words as a combinations of known words
  - very effective because low-frequency terms are very significant
- transformers based:
  - BERT, ELMo, …
  - can also capture word ordering

query expansion

- add related words to query based on nearest embeddings
- topic-shifting = neighbours of embedding might not make sense because they're from a different topic
- retrofitting = use more data to tune embeddings with unsupervised learning (not common anymore)

- lsi = find latent (hidden) semantic structure in text

sequence modelling

models for sequential data that can capture the input-order

cnn

- = convolutional neural networks
- on sequential data: 1d-cnns with a sliding window
- use cases:
  - n-gram representation learning = generating word embeddings as char-n-grams
  - dimensionality reduction = capturing an embedding-n-gram
- pooling:
  - usually applied after cnn
  - max-pooling = keep strongest feature per region
  - avg-pooling = keep average over each feature per region
  - adaptive/dynamic-pooling = change window size of pool (so output size is the same independent of input size)

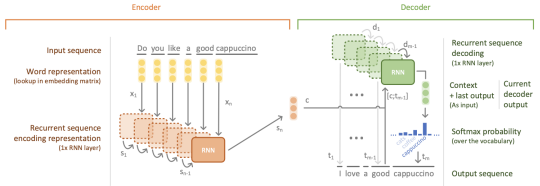
rnn

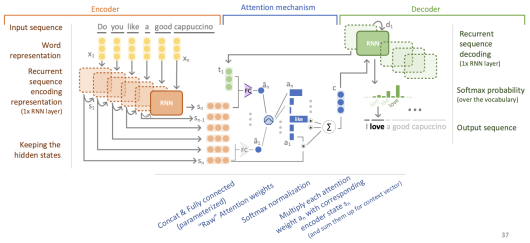
- = recurrent neural network
- vanishing gradient problem: unrolling means adding exponent to weight, leading to information loss
- can be used bidirectionally or hierarchically (multi-layer/stacked)
- $s_i = R_{SRNN}(x_i, s_{i-1}) = \quad g(s_{i-1} \cdot W^s + x_i \cdot W^x + b)$
  - s_i = state at position i (depends on position s_{i-1}, recursive definition)
  - g = activation function
  - b = bias vector (trainable)
  - W^s, W^x = weight matrices (trainable)

lstm

- = long short term memory
- helps with vanishing gradient problem
- binary-gate is not differentiable, we have to use sigmoid function $\sigma(g')$ to map it to range [0;1]
- $s_j = R_{LSTM}(x_j, s_{j-1}) = \quad [c_j; h_j]$
  - $c_j = f \odot c_{j-1} + i \odot z \longrightarrow$ gated memory
  - $h_j = o \odot \tanh(c_j) \longrightarrow$ hidden state
    - $f = \sigma(x_j W^{xf} + h_{j-1} W^{hf}) \longrightarrow$ forget
    - $i = \sigma(x_j W^{xi} + h_{j-1} W^{hi}) \longrightarrow$ input
    - $z = \tanh(x_j W^{xz} + h_{j-1} W^{hz}) \longrightarrow$ update candidate
    - $o = \sigma(x_j W^{xo} + h_{j-1} W^{hz}) \longrightarrow$ output
  - $\odot$ = hedamard-product, element-wise multiplication
  - $[a; b]$ = `concat(a,b)`

seq2seq (+ attention)



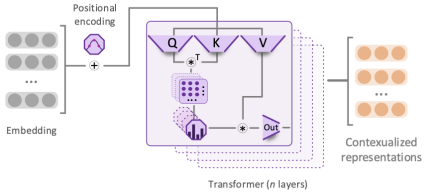


- encoder/decoder architecture, used for translation, question answering, chat bots, …
- i. encoder:
  - $c = RNN_{Enc}(x_{1:n})$
  - get encoding / representation of entire input sequence $x_{1:n}$ as a single embedding
  - for unknown reasons, reversing input works best
- ii. attention mechanism:
  - improve encoder output c by generating weights (context vector) $a^j_{[i]}$ based on encoder state $s_{1:n}$, and decoder state t_j
  - $attend(s_{1:n}, t_j) = c^j = \sum_{i=1}^{n} a^j_{[i]} \cdot s_i$
    - $a^j = \text{softmax}(\bar{a}^j_{[1]},\ \ldots,\ \bar{a}^j_{[n]})$ ⟶ attention weights sum up to 1
    - $\bar{a}^j_{[i]} = \nu \cdot \tanh([t_j; s_i] \cdot U + b)$ ⟶ implemented as neural network
    - j = current rnn iteration number
    - $s_{1:n}$ = all encoder states so far
    - t_j = decoder state at iteration j
    - c^j = context vector at position j
    - $a_{[i]}$ = `a[i]`
    - $[a; b]$ = `concat(a,b)`
    - ν, U, b = learnable params
- iii. decoder:
  - $p(t_v \mid t_{1:v-1}) = \text{softmax}(RNN_{Dec}([t_{1:v-1};\ c]))$
  - $p(t_v \mid t_{1:v-1}) = \text{softmax}(RNN_{Dec}(attend(t_{v-1}, s_{1:n})))$ ⟶ alternative with attention mechanism
  - uses embedding from previous stage, returns most likely output sequence T / $t_{1:m}$ as: $\arg_T \max(p(T|x_{1:n}))$
  - beam-search: heuristic graph-search-algorithm to pick highest softmax output, based on softmax distribution (works better than greedy-search)

pointer generator model

- out-of-vocabulary words are replaced with `UNK` in seq2seq
- this model can copy them back or regenerate them from the vocabulary

transformers



- = attention with matrix-multiplications instead of rnn
- types: encoder-only, encoder-decoder, stacked
- contextualized representations through self-attention:
  - unsupervised learning
  - masked language modelling = find relevance of words in context, by trying to reconstruct words with parts of the context being masked, then updating weights
  - self-attention = capture most relevant neighboring words in embedding
  - multi-head = without a fixed window size - is expensive: $O(n^2)$
- positional encoding = turning each token (subword) embedding into Q query, K key, V value
  - the key has d_k dimensions
  - $\text{SelfAttention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}}) \cdot V$

bert

- = bidirectional encoder representations from transformers
- limited to <512 tokens
- pre-training:
  - huge model, multiple layers of stacked transformers
  - fine-tuned for different use-cases
  - you can reduce vocabulary size by splitting words up into tokens with the wordPiece or bytePair algorithm
- special tokens:
  - `CLS` = end of 1-2 sentences, used for parallelism, prediction of sentences
  - `MASK` = masked word to predict
  - `SEP` = end of segment in sentence

huggingface:

- initially started as tensorflow to pytorch port of BERT
- share code, models, datasets via git-lfs, …

extractive q&a

- ≠ generative question answering (chat bots)
- find sequences in text that answer question
- can be done with segment start/end token predictions of BERT
- open domain qa = first retrieve the relevant documents, then find segments that answer question

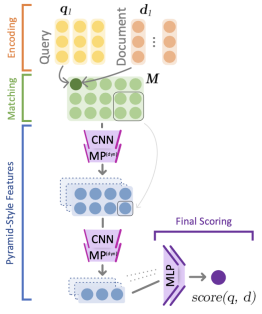
neural re-ranking

content-based ad-hoc retrieval = we only have access to query and doc text

architecture:

- first-stage ranker:
  - traditional, ie. bm25
  - most re-rankers rely on bm25 for the first stage, but it's by far not the best
- neural re-ranker:
  - input triples = `(query, rel-doc, non-rel-doc)`
    - hard to find truly non-rel-docs, and false-negatives confuse the model
  - loss = maximize margin between rel/non-rel-passages for given query
    - here we have a plain margin-loss
    - $L(Q, P^+, P^-) = MSE(Ms(Q, P^+) - Ms(Q, P^-), M_t(Q, P^+) - Mt(Q, P^-))$
  - online learning = use user activity-history-logs to improve model

matchPyramid

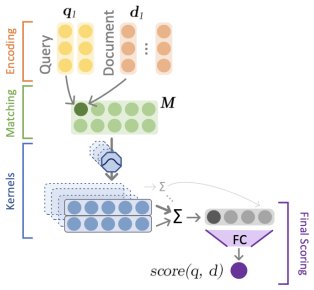


applying image processing techniques on match-matrix

- i. compute match matrix:
  - all query-doc cosine-similarities
  - measures direction of vectors, but not the magnitude
  - $M_{ij} = \cos(q_i, d_j) = \frac{d_j \cdot q_i}{|d_j||q_i|}$

- ii. apply 2D convolution layers on matrix:
  - layers:
    - 1st — $z_{ij}^{(1,c)} = \text{2D\_Conv}(M_{ij}) = ReLU\left(\sum_{s=0}^{r_c-1}\sum_{t=0}^{r_c-1} w_{s,t}^{(1,c)} \cdot M_{i+s,j+t} + b^{(1,c)}\right)$
    - 2nd — $z_{ij}^{(2,c)} = \text{dyn\_max\_pool}\left(z_{ij}^{(1,c)}\right) = \max_{0\le s<d_c} \ \max_{0\le t<d_c} z_{i\cdot d_c+s,j\cdot d_c+t}^{(1,c)}$ → makes output size static
    - ... other kernels, each learning a different feature
    - L'th — $z_{ij}^{(l,c)} = \text{max\_pool}\left(\text{2D\_Conv}(z_{ij}^{(l-1)})\right)$
  - where:
    - $z^{(n,-)}$ = sequential variable
    - c = channels
    - d_c = dynamic pool kernel size
    - r_c = channel size
    - W_*, b_* = weights, biases
- iii. get final score:
  - multi-layer-perceptron that returns float
  - $\text{score(q,d)} = MLP(z^l) = W_2 \cdot ReLU(W_1 \cdot z^l + b_1) + b_2$

(conv-) knrm



- knrm = kernel based neural ranking model
- roughly as effective as matchPyramid but a lot faster and simpler
- counts similarities in match-matrix
- i. encode docs and queries (if conv-knrm):
  - use cnn to generate word-n-gram embedding
  - $q_{1\ldots n}^h = 1D\_CNN(q_{1\ldots n})$
  - $d_{1\ldots m}^h = 1D\_CNN(d_{1\ldots m})$
  - where:
    - h = n-gram size
- ii. compute match matrix:
  - $M_{ij} = \cos(q_i, d_j) = \frac{d_j \cdot q_i}{|d_j||q_i|}$
- iii. apply radial-basis-function kernel:
  - rbf kernel for a single match, summed along document dimension j
  - $K_k(M) = \sum_{i=1}^{n} \log\left(\ \sum_j \exp\left(-\frac{(M_{ij}-\mu_k)^2}{2\sigma_k^2}\right)\right)$
  - where:
    - μ_k = similarity level
    - σ_k = kernel width / range
- iv. get final score:
  - $s = FC(K) = W \cdot K + b$
  - where:
    - FC = fully connected neural network
    - K = all kernels
    - W_*, b_* = weights, biases

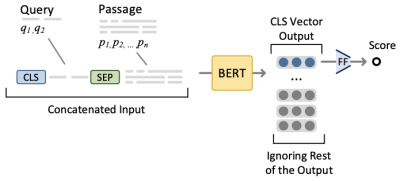
transformer contextualized re-ranking

these perform a lot better

transformers generate contextualized representations

bert-cat

effect: 1 – latency: 950ms – memory: 10.4GB



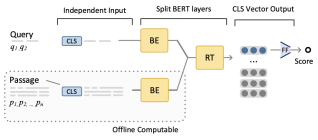
- most effective, slowest
- aka vanilla-bert, bert-cat, mono-bert
- steps:
  - concatenate query and passage, get representation, repeat for all passages
  - get final score with linear layer
  - $r = \text{BERT}([\texttt{CLS}]; q_{1..n}; [\texttt{SEP}]; p_{1..m})_{\texttt{CLS}}$
  - $s = r \cdot W$

improvements:

- input size: bert's can only read 512 tokens
  - a) limit input
  - b) sliding window with max-pooling
- accuracy: mono-duo pattern
  - use the larger T5 model instead of BERT
  - mono-phase: compute $\text{score}(q, p)$@1000
  - duo-phase: compute $\text{score}(q, p1, p2)$@50 but 50^2 = 2500 times ⟶ improves results from previous stage
- efficiency:
  - a) using a simpler model
  - b) precomputing as in preTTR

preTTR

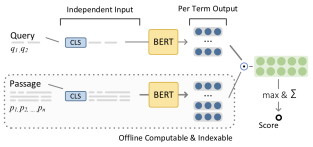
effect: 0.97 – latency: 455ms – memory: 10.9GB



- first n layers of BERT are precomputed seperately, maybe also compressed, then merged
- n is a hyperparameter
- steps:
  - $\hat{q}_{1..n} = \text{BERT}_{1..b}([\texttt{CLS}]; q_{1..n})$
  - $\hat{p}_{1..m} = \text{BERT}_{1..b}([\texttt{CLS}]; p_{1..m})$
  - $s = W \cdot \text{BERT}_{b..l}(\hat{q}_{1..n}[\texttt{SEP}]; \hat{p}_{1..m})$

colBERT

effect: 0.97 – latency: 28ms – memory: 3.4GB



- very fast, very memory intensive
- uses entire BERT representations (not layers, as in preTTR) to get a match-matrix, that then gets max-pooled
- steps:
  - . . . same
  - $s = \sum_{i=1..n} \max_{t=1...m} \hat{q}_i \cdot \hat{p}_t$

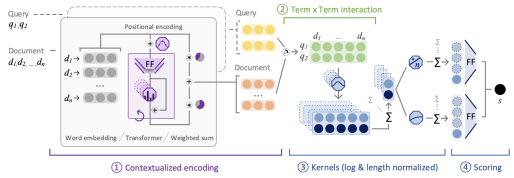
bert-dot

effect: 0.87 – latency: 23ms – memory: 3.6GB

- uses nearest-neighbor-index, computes cosine similarity
- steps:
  - . . . same
  - $s = \hat{q} \cdot \hat{p}$

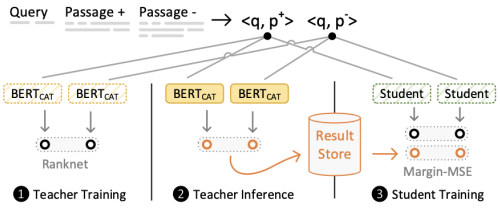
tk

effect: 0.89 – latency: 14ms – memory: 1.8GB (also interpretable)



- limits number of transformer layers and context, then applies kernel pooling
- types:
  - tk = transformer kernel ranking
  - tkl = tk for long documents → topography saturation & scoring module applied on match-matrix
  - tk-sparse = faster → removing stopwords after contextualization with sparsity-module
- i. precompute embedding-match-matrix:
  - apply transformer, get match-matrix of contextualized query and passage representations
  - $\hat{q}_{1..n} = TF(q_{1..n})$
  - $\hat{d}_{1..m} = TF(d_{1..m})$
  - $M_{ij} = \mathbf{cos}(\hat{q}_i, \hat{d}_j)$
- ii. apply radial-basis-function kernel:
  - rbf kernel for a single match, summed along passage dimension j
  - $K_k(M) = \sum_{i=1}^{n} \log \left(\sum_j \exp \left(-\frac{(M_{ij} - \mu_k)^2}{2\sigma_k^2} \right) \right)$
- iii. get final score:
  - $s = FC(K) = W \cdot K + b$

idcm



- = intra document cascade
- knowledge distillation / distilled training
  - = small efficient model learning from large accurate model
  - reduces noise in data

    - usually ~4x speedup with similar accuracy
- components:
    - etm = effective teacher model (vanilla-bert)
    - estm = effective student model (ck)
    - ps = passage score aggregator
- training steps:
    - i. teacher training
        - train teacher on binary loss
        - ensemble of teachers is better
    - ii. teacher inference (just once)
        - generate some initial labels and scores (the more the better)
    - iii. student training
        - uses labels (and other supervision signals) from teacher to improve margin-mse score (between rel-passages and non-rel-passages)

dense retrieval

dense retrieval lifecycle

dense retrieval DR = neural first-stage retrieval

- i. training stage: train bert-dot, encode all passages
    - train by maximizing distance between rel vs. non-rel-passages
    - sampling for training:
        - false-negatives confuse the model
        - find good non-rel-passages to reduce noise, either with bm25 or generative model
        - ANCE = approximate nearest neighbor negative contrastive learning
- ii. indexing stage: storing all embeddings in nearest-neighbor-index
    - index is also used during training step
    - cosine similarity is cheap
- iii. searching stage: use bert-dot to encode query, look up closest passage-neighbors

TAS-balancing

- TAS = topic aware sampling
- i. cluster queries and passages by topic
    - relatively cheap in practice
    - much more effective than comparing with non-rels in the same training batch
- ii. bin pairs by margins
    - normalizes results
- iii. train

zero-shot benchmarking

- BEIR benchmark
- dense-retrival models don’t generalize well to other query distributions
- neural models can have false-positives that don't make sense (bm25 must have lexical overlap as a constraint) but mostly only on sparse-judgements