Contents

1 Information Retrieval

IR is a sub-branch of Computer Science that investigates systems that enable searching on unstructured data.

In this domain we typically model process of search by assuming that user gives **queries** which are answered by **relevant documents**.

To formalize this, we define **gold standard** results as queries $(q)_{q \in Queryset}$ with their respective relevant document lists $(Rel_q)_{q \in Q}$.

In text retrieval setting for example queries are typically short sequences of words, and the documents come from agreed-upon corpus.

Our problem belongs to **multimodal** search, because we use different modalities for queries (natural language) and we do not treat code repositories as natural language texts (or even sequences of programming language tokens).

1.1 Evaluating Information Retrieval

2 [TODO] Neural Networks

(basic definitions)

3 Neural Networks for unsupervised text feature extraction

Although some neural network approaches have long history, with Recurrent Neural Networks being proposed in 80s, it is only the 2010s that witnessed widespread adoption of practical methods.

3.1 Classical approaches for text data

Pre-neural machine learning methods were mostly based either on linguistical information (part of speech tags, grammatical information) for sequences, or use token or ngram count data.

One standard method of obtaining text features is the so-called Bag of Words model where documents are treated as sets of words or ngrams.

The drawback of manual labeling is that it can't be easily scaled to leverage massive unlabeled text datasets obtained from book corpuses or crawling the web.

Bag-of-words model on the other hand runs into problems with polysemy (the same word might mean different concepts) and synonymy (different tokens define features which are completely unrelated). This approach also poses problems for machine learning methods that suffer from high dimensionality, because for representing reasonably sized corpus one typically

needs to use thousands of features. In general optimization problems become harder, and some algorithms, notably tree-based models, fare poorly in such regimes.

3.2 Language modeling

LM is a problem of predicting words from their contexts (in the simplest version the problem is to predict next word given preceding text).

Formally the task is to model probability $p(w_t|(w'_t)_{t'<t})$

Language Modeling is an old subdomain of artificial intelligence, because for a given corpus one can easily define this probability using word cooccurrences. While this approach was shown to be useful for some tasks like machine translation, it suffers from the same problems as Bag of Words approach. Since it is not possible to estimate probabilities for every accurately using only n-gram counts, generalization capabilities of such models are limited.

3.3 Why are neural networks useful for NLP?

Neural models in NLP have several advantages compared to older feature extraction methods. These methods typically contain a part that passess tokens (or their parts) through lower-dimensional representation (this is called **an embedding**). This embedding phase distills linguistic information so that network might generate similar outputs for similar inputs. The exact similarity depends on the method and might encode different aspects - for example Word Embedding methods capture *distributional similarity* as in the phrase *"linguistic items with similar distributions have similar meanings"* One might compare intermediate representation obtained in this way to compression, because it works by encoding information about tokens (which might come from a very big vocabulary, typically thousands of tokens) using fewer parameters (typicaly several hundred features).

Another advantage compared to older methods is that neural networks are highly composable, which is important for *transfer learning* - for example given a pretrained neural LM one might put classification layer on top, which would turn it into a sequence labeling model.

3.3.1 Neural vs count-based Language Models

To compare neural and count-based language models consider the task of estimating probability of a word given previous word, formally $p(w|v)$. In count-based LM we would use $p(w|v) = count(w, v)/count(v)$ (in general

left size might be proportional to right size to use smoothing, for example to account for rare words) Note this approach needs to store $|V|^2$ parameters, where V is the vocabulary.

In contrast neural-based method might use $p(w|v) = f_{\theta_0}(g_{\theta_1}(w), h_{\theta_2}(v))$ Where f, g, h are some neural network parametrized by $\theta_0, \theta_1, \theta_2$. Because in each of these networks only input or output depends on $|V|$, but not both, and the number of parameters is sum of θ sizes, number of parameters is $O(|V|)$.

3.4 Recurrent Neural Networks

RNN is a neural sequence model that in addition to using current input also contains connections with previous input. This architecture makes it possible to propagate information along the sequence, which in theory can be arbitrarily long.

Formally one layer of RNN takes two inputs x_t, h_t (called *hidden state*) and provides two outputs y_t, h_{t+1} (we omit bias for notational convenience)

$y_t = \sigma(W_y h_t)$

$h_t = \sigma(W_h h_{t-1} + W_x x_t)$

Where σ is some kind of nonlinearity.

Because h_t depends on h_{t-1}, output at time t is influenced by hidden states for $t' < t$. Note that number of parameters for such network does not depend at sequence length at all.

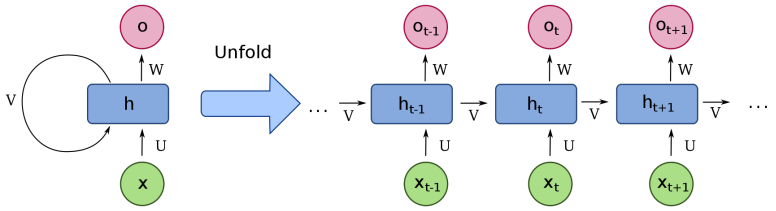

Figure 1: folded (left) and unfolded (right) RNN diagram

Procedure for training RNNs is called Backpropagation Through Time and is illustrated in the image above. This works almost like training regular feedforward network, but the parameters of connections for each step are tied.

3.4.1 [TODO???] Problems with RNNs LSTM and Gated Units

- ELMO, AWD-LSTM?

3.5 Neural Language Models

First neural LMs were using RNNs where predicted sequence was shifted to the right (at time t, given x_t try to predict x_{t+1})

These models were shown to be useful for various NLP tasks in *Natural Language Processing (Almost) from Scratch* (Collobert et al, 2011) . Authors used the method to work only using text. The obtained results were close to previous state-of-the-art approaches that used hand-crafted features, while using significantly simpler models.

3.6 Word embeddings

Training LMs using RNNs requires unrolling which is hard to parallelize. This makes it time-consuming, and sometimes useful word embeddings might be obtained not using information about whole sequences.

Simpler methods for word embeddings, only making use of fixed-size contexts, were proposed based on *distributional hypothesis*. They obtain word vectors such that $vector(w)$ is similar to $vector(w')$ if w and w' occur in similar contexts, for example "Italy" and "France".

These methods given a sequence $(w_0, w_{t-1}, w_t, w_{t+1}, ..., w_T)$ try to predict w_t by the rest of the context.

3.6.1 Word2Vec

This method, proposed in Mikolov et al, 2013 is posed as a standard classification problem and solved using a shallow neural network. The weights between input and hidden layer can be then used as word vectors. Empirically they have been shown that they capture the similarity of words that have similar contexts. In addition to that it also has been shown that geometry of these word embeddings encodes semantic relations, for example $vector(\text{"}king\text{"}) - vector(\text{"}man\text{"}) \approx vector(\text{"}queen\text{"}) - vector(\text{"}woman\text{"})$.

3.6.2 FastText

One problem of using words as tokens is that they treat them as atomic units, so either tokens need to be stemmed or lemmatized, or they will get treated as unrelated.

A series of papers from starting from Bojanowski et al, 2017 proposed to refine Word2Vec model with subword information. These methods split words into tokens, for example "technical" might be split into "techn*" "*ical", so it will have similar parts with "technician".

In our work FastText is useful for embedding Python function names, as it might figure out that for example "*get_http_request*" is not an atomic entity.

3.7 [TODO???] Attention and Transformer-based models

3.8 NLP for Programming Languages

3.8.1 CodeSearchNet

3.8.2 CodeBERT

4 Zero-shot Learning

Because usable repository search systems can't know all possible queries in advance we use Zero-shot Learning as a more realistic proxy task than Supervised Learning.

Zero-shot Learning (ZSL) is a branch of Machine Learning where classes from test set possibly do not occur in training set.

Because it is impossible to transfer between seen to unseen classes using label encoding or one-hot encoding, to circumvent this ZSL assumes that classes are represented by their feature vectors.

These features might be manually constructed as shown in the following image, or they might consist of NLP-extracted features of class names.

Figure 2: samples from AWA2 dataset showing images and class features

For our problem ZSL is a useful framework because methods used to solve it typically match different modalities (such as image features vs text features).

It also enables some flexibility in the choice of both input and class features.

4.1 Proxy ZSL problem

Because ZSL needs class features, we evaluate several methods for extracting representations from PapersWithCode task names:

- pretrained word embeddings available in Gensim
- word embeddings trained with Word2Vec on Python file corpus
- FastText trained on Word2Vec Python file corpus

5 Graphs in Machine Learning

GRL chap 1

Most older methods for node embeddings are *transductive*, so they assume the same graph structure at training time and test time.

- PageRank
- Manifold Learning - neighborhood graph (Isomap, Laplacian Eigenmaps)
- Graph embeddings (Node2Vec)
- Graph Neural Networks (GCN, GraphSAGE, GAT)

5.1 Node Embeddings

Node2Vec - Word2Vec with contexts obtained from random walks

5.2 Graph Neural Networks

GRL Chap 5

Using neural networks for graph-structured data is problematic, because

- node neighborhoods $\mathcal{N}(u)$ may vary in size, and NN layers usually assume fixed-size input
- there is no natural ordering on neighbors

First issue is less severe because we might just sample neighbors so that their number becomes fixed in each iteration. The second one is usually circumvented by explicitly making output layer *permutation invariant* or *equivariant*.

Formally, let us assume that our NN layer f takes A, graph adjacency matrix as input.

Then

$$f(PAP^T) = f(A)$$

means that the layers is *permutation invariant*, whereas

$$f(PAP^T) = Pf(A)$$

means *permutation equivariance*.

5.2.1 General framework, message passing

The general approach for defining GNNs is called *message passing* and can be interpreted as generalization of convolution to graph data or differentiable graph isomorphism test. The intuition is to incorporate information from neighborhoods so that after each epoch it propagates through graph.

Formally specifying one layer consists of defining node's u embedding $h_u^{(k+1)}$ (superscripts denote layer number).

$h_u^{(0)}$ are either initialized with some other methods (for example using word embeddings of node names) or by using graph features (node degree et c).

The embeddings of next layer are then

$$m_{neigh}(u) = \textsc{aggregate}(\{h_v^{(k)}\}_{v \in \mathcal{N}(u)})$$

$$h_u^{(k+1)} = \textsc{update}(h_u^{(k)}, m_{neigh})$$

(general GNN equation)

where UPDATE and AGGREGATE are some differentiable functions.

Note that second argument of UPDATE is a set, so this part needs to be permutation invariant. This can be achieved using functions that don't depend on ordering of the input. It is usually achieved using pooling functions like averaging or taking maximum, or by averaging method that is order-sensitive (like LSTM encoder) over sample of permutations.

5.2.2 GNN

$$h_u^{(k+1)} = \sigma(W_{self}^{(k)} h_u^{(k)} + W_{neigh}^{(k)} \sum_{v \in \mathcal{N}(u)} h_v^{(k)}) \qquad \text{(basic GNN equation)}$$

5.2.3 Scalability, GraphSAGE

It is easy to come up with vectorized form of GNN equation that takes the whole graph into account to enable whole batch gradient descent:

$$H^{(k+1)} = \sigma(W_{self}^{(k)} H^{(k)} + A W_{neigh}^{(k)} H^{(k)} + b_k) \qquad \text{(vectorized basic GNN)}$$

Where A is graph's adjacency matrix. The problem with this equation is that it is not easy to turn this into minibatch version. This is because the second appearance of $H^{(k)}$ cannot be just replaced by batched version - one would need to take neighbors into account.

Because of this, several approaches for scaling GNNs were proposed.

One such example is GraphSAGE proposed in Hamilton et al, 2017. It creates minibatches by sampling nodes, and creating fixed size contexts from their neighbors.

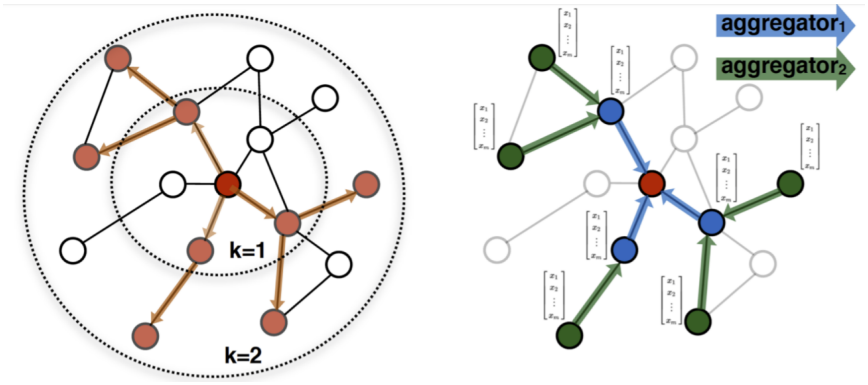

Figure 3: GraphSAGE: SAmple and AggregatE

Formally, for each minibatch B, K -layer GraphSAGE fetches embeddings of nodes sampled from K -hop neighborhoods of each $u \in B$.

This is achieved by the following algorithm:

Algorithm 1: GraphSAGE Batch embedding

function SAMPLE_EMBEDDED_VERTICES:
  Input:
    B - `batch of vertices`
    $(\text{SAMPLE\_NEIGHBORS}_k)_{k<K}$ – `sampling function`
  Output: $(\text{B}_k)_{k<K}$ –
    `neighbor samples required for embedding`
  $\text{B}_K := B$
  for *k in [K-1, ..., 1]* **do**
    $\text{B}_k := B_{k+1}$
    for *u in* B_k **do**
      $B_k := B_k \cup \text{SAMPLE\_NEIGHBORS}_k(u)$
    end
  end

function BATCH_EMBEDDING:
  Input:
    B - `batch of vertices`
    $(\text{SAMPLE\_NEIGHBORS}_k)_{k<K}$ – `sampling function`
  Output: $(\text{z}_u)_{u\in B}$ – `node embeddings`
  for $v \in B_0$ **do**
    $h_u^0 := x_u$ `// initialization`
  end
  $(B_k)_{k<K} = \text{SAMPLE\_EMBEDDED\_VERTICES}(B)$
  for *k in [1, ..., K]* **do**
    for *u in* B_k **do**
      $N_k(u) := \text{SAMPLE\_NEIGHBORS}_k(u)$
      $h_{neigh}^k := \text{AGGREGATE}_k(\{h_v^{k-1}\}_{v\in N_k(u)})$
      $h_u^k := \sigma(W^k \cdot [h_u^{k-1}; h_{neigh}^k])$
      $h_u^k := \text{NORMALIZE}(h_u^k)$
    end
  end
  for $v \in B$ **do**
    $z_u := h_u^K$ `// embeddings from last layer`
  end

5.3 [TODO???] Deep Graph Infomax

Method related to Masked Language Models