CS635_A3_Report

210050152, 210050068

Part1:- Link Prediction and Retrieval on Graphs

Datasets

CoraFull

- Data(x=[19793, 8710], edge_index=[2, 126842], y=[19793])
- The dataset consists of 19,793 nodes, 8,710 features per node and are classified into 70 classes. The graph contains 126,842 directed edges, these edges are stored in the matrix `edge_index`.
- Specifically, for each edge, the first entry in `edge_index[0]` represents the source node, and the first entry in `edge_index[1]` represents the target node.

Data Splitting

- The graph's edges were split into training and testing sets with a 60:40 ratio.
- The graph contains a total of 126,842 edges, which were split into training and testing.
- The training set consists of 76,105 edges (60% of the total edges), while the testing set contains 50,737 edges (40% of the total edges).
- The unique nodes involved in the training set are 17,426, while the testing set involves 16,178 unique nodes.
- The model is trained on the training subset of the graph and evaluated on the separate testing subset to evaluate its performance.

DeezerEurope

- Data(x=[28281, 128], edge_index=[2, 185504], y=[28281])
- The dataset consists of 28281 nodes, 128 features per node and are classified into 2 classes. The graph contains 185504 directed edges, these edges are stored in the matrix `edge_index`.
- Specifically, for each edge, the first entry in `edge_index[0]` represents the source node, and the first entry in `edge_index[1]` represents the target node.

Data Splitting

- The graph's edges were split into training and testing sets with a 60:40 ratio.
- The graph contains a total of 185504 edges, which were split into training and testing.
- The training set consists of 111302 edges (60% of the total edges), while the testing set contains 74202 edges (40% of the total edges).
- The unique nodes involved in the training set are 27040, while the testing set involves 24526 unique nodes.
- The model is trained on the training subset of the graph and evaluated on the separate testing subset to evaluate its performance.

Q Set Construction

In the test set, we identified a set Q of 1000 nodes, each of which is part of at least one triangle in the induced subgraph.

- To find these nodes, we iterated over the test edges and used the function `get_connected_nodes` to retrieve the neighbors of each node.
- For each edge, we checked if the two sets of neighbors intersected, indicating the presence of a triangle.
- Nodes involved in triangles were added to the set, and the first 1000 nodes were selected as the Q set.
- In total, we identified 4,835 nodes in CoraFull dataset and 4926 nodes in DeezerEurope dataset that were part of at least one triangle, with the process taking approximately 24.62028909 seconds for CoraFull and 43.53522778 seconds fro DeezerEurope dataset.

Training

During training, the subgraph induced by all edges and nodes in the train set was passed as input to the model. Each embedding in the loss equation was replaced by node embeddings generated by the chosen Graph Neural Network (GNN) model, and the scoring function was cosine similarity between embeddings.

To train the model effectively, we used the **AUC-inspired loss function** as defined in Equation (9) of the reference (paper. This loss aims to provide high-quality ranking by leveraging a margin-based ranking approach. For each epoch:

- Positive Scores: Computed as cosine similarity between embeddings of nodes connected by positive edges.

- Negative Scores: Computed similarly for node pairs without edges, sampled using negative sampling.
  - **negative Sampling**
    - Negative edges were sampled randomly from the graph.
    - Each sampled edge was checked to ensure it did not exist in the positive edge set.
    - If necessary, a function `negative_sampling_fix` was invoked to replace invalid samples with valid negative edges.
    - The final negative edge set was verified to ensure no overlap with positive edges.
- Loss Calculation: The margin-based ranking loss, defined as $\text{ReLU}(1.0 + \text{negative\_score} - \text{positive\_score})$, is minimized during training.

Models

Experimented with three GNNs:

1. **Graph Convolutional Network (GCN)**

GCN Model Architecture

- First Layer: Maps input features (`in_channels`) to hidden features (`hidden_channels`) using the `GCNConv` layer.
- Second Layer: Maps hidden features to output embeddings (`out_channels`).
- Non-linearity is introduced between the layers using the ReLU activation function.

The GCN model generated embeddings for all nodes in the graph. Query embeddings were extracted for nodes in Q. Cosine similarity scores were computed between query embeddings and all test embeddings. Self-matching scores were set to $-\infty$. Precision@K and MRR were calculated based on ranked retrievals.

Evaluating the GCN model's ability to retrieve top-K relevant node pairs for a given query node q from the test set Q, based on Precision@K (K = 1, 5, 10) and Mean Reciprocal Rank (MRR)

GCN inference - CoraFull dataset

- Precision@1: 0.5704295704295704
- Precision@5: 0.42017982017982014
- Precision@10: 0.3158841158841166
- Mean Reciprocal Rank (MRR): 0.6966583019843102
- Average Retrieval Time per Query: 0.15776622796058654 seconds

Faster Inference with LSH- CoraFull dataset

Random LSH

- **n_hyperplanes = 2:**
  P@1: 0.557, P@5: 0.4026, P@10: 0.2986,
  MRR: 0.6821, Average Retrieval Time: 0.06426516771316529 seconds
- **n_hyperplanes = 4:**
  P@1: 0.545, P@5: 0.3854, P@10: 0.2821,
  MRR: 0.6656, Average Retrieval Time: 0.01882152009010315 seconds
- **n_hyperplanes = 6:**
  P@1: 0.523, P@5: 0.3650, P@10: 0.2580,
  MRR: 0.6408, Average Retrieval Time: 0.008074583530426026 seconds

Neural LSH

- Precision@1: 0.57
- Precision@5: 0.4201999999999999
- Precision@10: 0.31580000000000064
- Mean Reciprocal Rank (MRR): 0.697051367161806
- Average Retrieval Time per Query: 0.006876643180847168 seconds

GCN inference - DeezerEurope dataset

- Precision@1: 0.2827172827172827
- Precision@5: 0.1994005994005988
- Precision@10: 0.14855144855144772
- Mean Reciprocal Rank (MRR): 0.42901625152713285
- Average Retrieval Time per Query: 0.21283300232887267 seconds

Faster Inference with LSH- DeezerEurope dataset

Random LSH

- **n_hyperplanes = 2:**
  P@1: 0.282, P@5: 0.1888, P@10: 0.1391,
  MRR: 0.4217, Average Retrieval Time: 0.10030516362190246 seconds
- **n_hyperplanes = 4:**
  P@1: 0.255, P@5: 0.1736, P@10: 0.1255,
  MRR: 0.3932, Average Retrieval Time: 0.0364566650390625 seconds
- **n_hyperplanes = 6:**
  P@1: 0.241, P@5: 0.1554, P@10: 0.1104,
  MRR: 0.3632, Average Retrieval Time: 0.010139692306518554 seconds

Neural LSH

- Precision@1: 0.282
- Precision@5: 0.19679999999999942
- Precision@10: 0.14759999999999923
- Mean Reciprocal Rank (MRR): 0.4272218802272243
- Average Retrieval Time per Query: 0.011473605632781983 seconds

2. **Graph Attention Network (GAT)**

   GAT Model Architecture

- First Layer: Maps input features (*in channels*) to hidden features (*hidden channels*) using the `GATConv` layer with multi-head attention. Multiple attention heads capture diverse node relationships, and the outputs are concatenated to form richer node representations.
- Second Layer: Maps concatenated hidden features to output embeddings (*out channels*) using the `GATConv` layer with a single attention head for dimensionality reduction and final embedding computation.
- Non-linearity: Introduced between the layers using the ELU activation function to enhance expressiveness and stabilize training.

The GAT model generated embeddings for all nodes in the graph. Query embeddings were extracted for nodes in Q. Cosine similarity scores were computed between query embeddings and all test embeddings. Self-matching scores were set to $-\infty$. Precision@K and MRR were calculated based on ranked retrievals.

Evaluating the GAT model's ability to retrieve top-K relevant node pairs for a given query node q from the test set Q, based on Precision@K (K = 1, 5, 10) and Mean Reciprocal Rank (MRR)

GAT inference - CoraFull dataset

- Precision@1: 0.5274725274725275
- Precision@5: 0.3832167832167827
- Precision@10: 0.28831168831168835
- Mean Reciprocal Rank (MRR): 0.6569225002421069
- Average Retrieval Time per Query: 0.13898867177963256 seconds

Faster Inference with LSH- CoraFull dataset

Random LSH

- **n_hyperplanes = 2:**
  P@1: 0.523, P@5: 0.3736, P@10: 0.2771,
  MRR: 0.6502, Average Retrieval Time: 0.05581317496299744 seconds
- **n_hyperplanes = 4:**
  P@1: 0.496, P@5: 0.3486, P@10: 0.2539,
  MRR: 0.6164, Average Retrieval Time: 0.015758037328720092 seconds
- **n_hyperplanes = 6:**
  P@1: 0.497, P@5: 0.3380, P@10: 0.2452,
  MRR: 0.6181, Average Retrieval Time: 0.007600168228149414 seconds

Neural LSH

- Precision@1: 0.528
- Precision@5: 0.38299999999999956
- Precision@10: 0.2882
- Mean Reciprocal Rank (MRR): 0.6576668099465003
- Average Retrieval Time per Query: 0.006227587938308716 seconds

GAT inference - DeezerEurope dataset

- Precision@1: 0.365634365634365563
- Precision@5: 0.2607392607392599
- Precision@10: 0.19600399600399487
- Mean Reciprocal Rank (MRR): 0.5210989576937342
- Average Retrieval Time per Query: 0.20916166114807128 seconds

Faster Inference with LSH- DeezerEurope dataset

Random LSH

- **n_hyperplanes = 2:**
  P@1: 0.345, P@5: 0.2384, P@10: 0.1747,
  MRR: 0.4938, Average Retrieval Time: 0.09204198455810547 seconds
- **n_hyperplanes = 4:**
  P@1: 0.335, P@5: 0.2242, P@10: 0.1658,
  MRR: 0.4795, Average Retrieval Time: 0.03946417713165283 seconds
- **n_hyperplanes = 6:**
  P@1: 0.339, P@5: 0.2132, P@10: 0.1497,
  MRR: 0.4737, Average Retrieval Time: 0.017916815757751465 seconds

Neural LSH

- Precision@1: 0.364
- Precision@5: 0.2601999999999992
- Precision@10: 0.19559999999999889
- Mean Reciprocal Rank (MRR): 0.5194996007360587
- Average Retrieval Time per Query: 0.010334732532501221 seconds

3. **Graph Isomorphism Network (GIN)**

   GIN Model Architecture

   - First Layer: Maps input features (*in channels*) to hidden features (*hidden channels*) using the `GINConv` layer. This layer employs a Multi-Layer Perceptron (MLP) to transform node features, with a `ReLU` activation function to introduce non-linearity.
   - Second Layer: Maps hidden features from the first layer to output embeddings (*out channels*) using another `GINConv` layer. The second layer further reduces dimensionality and computes the final node embeddings, using an MLP followed by a `ReLU` activation.
   - Non-linearity: Introduced between the layers using the `ReLU` activation function. This enhances the model's expressiveness and stabilizes training by helping to mitigate the vanishing gradient problem.

The GIN model generated embeddings for all nodes in the graph. Query embeddings were extracted for nodes in Q. Cosine similarity scores were computed between query embeddings and all test embeddings. Self-matching scores were set to $-\infty$. Precision@K and MRR were calculated based on ranked retrievals.

Evaluating the GIN model's ability to retrieve top-K relevant node pairs for a given query node q from the test set Q, based on Precision@K (K = 1, 5, 10) and Mean Reciprocal Rank (MRR)

GIN inference - CoraFull dataset

- Precision@1: 0.21078921078921078
- Precision@5: 0.15304695304695323
- Precision@10: 0.3386168448210884
- Mean Reciprocal Rank (MRR):
- Average Retrieval Time per Query: 0.13791158246994017 seconds

Faster Inference with LSH- CoraFull dataset

Random LSH

- **n_hyperplanes = 2:**
  P@1: 0.21, P@5: 0.1478, P@10: 0.1164,
  MRR: 0.3349, Average Retrieval Time: 0.05862618112564087 seconds
- **n_hyperplanes = 4:**
  P@1: 0.204, P@5: 0.1476, P@10: 0.1155,
  MRR: 0.3309, Average Retrieval Time: 0.016658509492874146 seconds
- **n_hyperplanes = 6:**
  P@1: 0.202, P@5: 0.1492, P@10: 0.1147,
  MRR: 0.3274, Average Retrieval Time: 0.009355814933776855 seconds

Neural LSH

- Precision@1: 0.211
- Precision@5:0.1530000000000002
- Precision@10: 0.12009999999999978
- Mean Reciprocal Rank (MRR): 0.33864532608515313
- Average Retrieval Time per Query: 0.0069580554962158205 seconds

GIN inference - DeezerEurope dataset

- Precision@1: 0.14385614385614387
- Precision@5: 0.10849150849150891
- Precision@10: 0.08941058941058914
- Mean Reciprocal Rank (MRR): 0.2592916988985791
- Average Retrieval Time per Query: 0.21550560760498047 seconds

Faster Inference with LSH- DeezerEurope dataset

Random LSH

- **n_hyperplanes = 2:**
  P@1: 0.144, P@5: 0.1048, P@10: 0.0864,
  MRR: 0.2557, Average Retrieval Time: 0.08700841450691223 seconds
- **n_hyperplanes = 4:**
  P@1: 0.141, P@5: 0.1054, P@10: 0.0854,
  MRR: 0.2526, Average Retrieval Time: 0.028602280616760255 seconds

- **n_hyperplanes = 6:**
  P@1: 0.141, P@5: 0.1048, P@10: 0.0835,
  MRR: 0.2512, Average Retrieval Time: 0.012951849937438965 seconds

(DeezerEurope dataset, GIN model): FASTER INFERENCE - NEURAL LSH

Neural LSH

- Precision@1: 0.143
- Precision@5: 0.1086000000000006
- Precision@10: 0.08969999999999972
- Mean Reciprocal Rank (MRR): 0.25916327981406156
- Average Retrieval Time per Query: 0.011039289474487304 seconds

4. **Adamic Adar Score**

   Adamic Adar Score Calculation

   The Adamic Adar score between two nodes u and v is based on the common neighbors of these nodes. For each common neighbor w, the score is calculated as:

   $$\text{Score}(u, v) = \sum_{w \in \text{common neighbors}(u,v)} \frac{1}{\log(\deg(w))}$$

   Where $\deg(w)$ represents the degree of the neighbor node w.

   The Adamic Adar score was computed for all pairs of nodes in the query set Q and test nodes. The results were evaluated using Precision@K, where $K \in \{1, 5, 10\}$, and the Average Retrieval Time was recorded.

   CoraFull dataset

   - Precision@1: 0.6510
   - Precision@5: 0.5296
   - Precision@10: 0.4318
   - Mean Reciprocal Rank (MRR): 0.7741454204567364
   - Average Retrieval Time per Query: 8.21797911 seconds

   DeezerEurope dataset

   - Precision@1: 0.6490
   - Precision@5: 0.5654
   - Precision@10: 0.4836

- Mean Reciprocal Rank (MRR): 0.7807400436236114
- Average Retrieval Time per Query: 16.59718637 seconds

5. **Common Neighbours Score**

   Common Neighbours Score Calculation

   The Common Neighbors (CN) score between two nodes u and v measures the number of shared neighbors between them. Specifically, for each node u and v, the Common Neighbors score is calculated as:

   $$\text{Score}(u, v) = \sum_{w \in \text{common neighbors}(u,v)} 1$$

   where the sum is taken over all neighbors w that are common to both nodes u and v. In this case, a higher CN score indicates a greater number of shared neighbors and, consequently, a higher similarity between nodes u and v.

   Common Neighbours Score was computed for all pairs of nodes in the query set Q and test nodes. The results were evaluated using Precision@K, where $K \in \{1, 5, 10\}$, and the Average Retrieval Time was recorded.

   CoraFull dataset

   - Precision@1: 0.5820
   - Precision@5: 0.5002
   - Precision@10: 0.4107
   - Mean Reciprocal Rank (MRR): 0.7194956217656225
   - Average Retrieval Time per Query: 8.30217773 seconds

   DeezerEurope dataset

   - Precision@1: 0.6010
   - Precision@5: 0.5456
   - Precision@10: 0.4732
   - Mean Reciprocal Rank (MRR): 0.7462036814492697
   - Average Retrieval Time per Query: 15.84732808 seconds

Part2:- Zero-Shot Retrieval Report

Zero-Shot Retrieval Implementation

Feed-Forward Layer (FFL)

The FFL projects CLS token embeddings to scalar scores for retrieval tasks. It consists of two linear layers with ReLU activation. Code: `FeedForwardLayer(nn.Module)` defined with 256 hidden units and an output scalar.

Pre-trained BERT and Tokenizer

Loaded `bert-base-uncased` from Hugging Face. The model was used for masked language modeling (MLM) reconstruction.

HotpotQA Dataset

Subsets of the HotpotQA corpus, queries, and qrels were extracted:

- **Corpus:** 5,233,329 documents
- **Queries:** 3,000 queries
- **Qrels:** 54,000 query-document pairs (each query- 2 positive docs and 16 negative docs sampling)

Dataset structures:

- `Corpus`: _id, title, text
- `Queries`: _id, title, text
- `Qrels`: query-id, corpus-id, score

Subsets were created to ensure efficient training and evaluation.

Data Preprocessing and Pair Generation

- **Dictionary Construction for Efficient Lookups**: This step involves creating dictionaries for both the query and document texts, indexed by their respective IDs, to enable efficient O(1) lookups during the subsequent processing steps.
- **Identification of Positive Query-Document Pairs**: Positive query-document pairs, where the relevance score is 1, are identified and stored in a set for fast access. This ensures only relevant pairs are included in the final output.
- **Negative Sampling Strategy**: For each positive pair, negative samples are generated by randomly selecting documents that are not relevant to the query. The selection ensures that no document is repeated or used as a positive pair for the same query.

- **Output of Query-Document Pairs**: Both positive and negative query-document pairs are added to a final dictionary, labeled accordingly with 1 for relevant pairs and 0 for non-relevant ones. A total of 54,000 query-document pairs are generated for further analysis.

- $$\text{query\_doc\_pairs} \rightarrow \text{List of dictionaries} \rightarrow \text{Dataset.from\_list(query\_doc\_pairs\_list)}$$

  converted a dictionary of query-document pairs into a Hugging Face Dataset, `query_doc_dataset`, with 54,000 rows.

- **T**he preprocessing function tokenizes the `query_text` and `doc_text` together using the tokenizer, producing:

  $$\text{input\_ids, attention\_mask, label}$$

  The function also ensures that inputs are truncated or padded to a maximum length of 512 tokens.

- The preprocessing function is applied to each example in the dataset:

  $$\text{hotpotqa} = \text{query\_doc\_dataset.map(preprocess\_function)}$$

  The resulting dataset includes the tokenized inputs (`input_ids`, `attention_mask`) alongside the original `label`.

Training the Feed-Forward Layer (FFL) on Top of BERT

In this section, we detail the process of fine-tuning a Feed-Forward Layer (FFL) on top of a pre-trained BERT model using the training data.

Model Setup

The BERT model and the FFL are set up on the appropriate device (either CPU or GPU). During training, BERT is frozen (set to `eval()` mode), meaning that its weights are not updated, and only the FFL's weights are trained. The FFL is set to `train()` mode, which enables the computation of gradients for its parameters.

Training Process

The training loop runs for a maximum of 5 epochs, and the following steps are performed during each epoch:

- **Data Loading:** The training data is loaded in batches using the `DataLoader` with a batch size of 16.

- **Forward Pass:**

  - The input sequence is passed to the frozen BERT model. We extract the CLS embeddings from the last hidden state of BERT using `torch.no_grad()` to avoid updating BERT's weights during training.
  - These CLS embeddings are then passed through the FFL to produce the output scores.

- **Loss Computation:** The loss is computed using Mean Squared Error (MSE) between the predicted scores and the actual labels. The MSE loss function is used as the training objective.
- **Backward Pass:** The gradients of the FFL weights are computed, and the optimizer (`AdamW`) is used to update the FFL parameters.
- **Early Stopping:**
  - The model monitors the loss during each epoch, and if the loss improvement is less than a predefined threshold (`min_delta`), the training is stopped early to prevent overfitting. The best model (with the lowest loss) is saved during training.

Training Results

The training proceeds as follows for each epoch:

- **Epoch 1:** Average loss = 0.0648. Model improves, and weights are saved as `ffl_weights.pth`.
- **Epoch 2:** Average loss = 0.0500. Model improves, and weights are saved.
- **Epoch 3:** Average loss = 0.0440. Model continues to improve, and weights are saved.
- **Epoch 4:** Average loss = 0.0405. Model improves, and weights are saved.
- **Epoch 5:** Average loss = 0.0376. Best performance, weights saved.

Early Stopping and Final Model

The training stops after 5 epochs, as no further significant improvement is observed in the loss. The final best loss achieved is 0.0376, and the model's weights are saved as `ffl_weights.pth`. The fine-tuning process successfully trains the Feed-Forward Layer on top of BERT, achieving a final average loss of 0.0376 after 5 epochs. The model was saved when improvements were made, and early stopping was employed to prevent overfitting.

SCIDOCS dataset Loading and Preprocessing

- Load the SCIDOCS corpus, queries, and qrels datasets using the `load_dataset()` function from the BeIR collection.
- Extract the relevant subsets of the corpus, queries, and qrels data.

- Initialize dictionaries `queries_dict` and `docs_dict` to map query and document IDs to their respective texts.
- Iterate through the qrels subset to retrieve the corresponding query and document for each entry, checking that both exist.
- For valid query-document pairs, extract the texts and relevance score, storing them in the `scidocs_query_doc_pairs` dictionary.

The number of query-document pairs extracted is printed as confirmation.

- The dictionary `scidocs_query_doc_pairs` is converted into a list and used to create a Hugging Face dataset `scidocs_query_doc_dataset` containing 29,928 rows.

  $$\texttt{scidocs\_query\_doc\_pairs} \rightarrow \text{List} \rightarrow \texttt{Dataset.from\_list(scidocs\_query\_doc\_pairs\_list)}$$

- A function tokenizes the `query_text` and `doc_text` using a tokenizer, producing `input_ids`, `attention_mask`, and `label` as output.

  $$\texttt{preprocess\_scidocs}(\textit{example}) \rightarrow \text{tokenized inputs with labels}$$

- The preprocessing function is applied to the dataset:

  $$\texttt{scidocs} = \texttt{scidocs\_query\_doc\_dataset.map(preprocess\_scidocs)}$$

  This results in a dataset with tokenized query-document pairs and their labels.

Masked Language Modeling Step

- **Data Collator:** A data collator is defined using `DataCollatorForLanguageModeling` with a 15% probability for masking tokens.
- **Masked Language Modeling Step Function:** The `masked_language_modeling_step` function processes a batch, prepares inputs and attention masks, and applies the data collator to create features for MLM.
- **Model Processing:** The model is called with the inputs and labels, and the loss is computed based on the model's output.

We evaluate the model using Mean Reciprocal Rank (MRR), Precision at 10 (P@10), and Recall at 10 (R@10) for various granularities during test-time inference.

Without Masked Language Modeling (MLM Reconstruction)

When no masked language modeling (MLM) reconstruction is performed, the following results were obtained for nearly 1000 scidocs queries pairs tested:

$$\text{MRR} = 0.2537, \quad \text{P@10} = 0.1343, \quad \text{R@10} = 0.2776$$

With Masked Language Modeling (MLM Reconstruction)

When MLM reconstruction is enabled, we observed the following metrics for different granularities K on nearly 1000 scidocs queries tested::

- For $K = 1$: MRR = 0.2666, P@10 = 0.1433, R@10 = 0.2970
- For $K = 8$: MRR = 0.2424, P@10 = 0.1343, R@10 = 0.2791
- For $K = 16$: MRR = 0.2120, P@10 = 0.1254, R@10 = 0.2612
- For $K = 64$: MRR = 0.2216, P@10 = 0.1269, R@10 = 0.2634

Analysis

- **Without MLM Reconstruction:** MRR is 0.2537, P@10 is 0.1343, and R@10 is 0.2776. These metrics are relatively lower compared to the masked inference case, which suggests that masking helps improve performance.
- **With MLM Reconstruction:**
  - When $K = 1$, the metrics improve, with MRR rising to 0.2666, indicating better ranking quality.
  - As K increases, there is a decrease in all metrics, especially in MRR and P@10. For $K = 8$, MRR drops to 0.2424, and for larger granularities like $K = 16$ and $K = 64$, the performance continues to degrade.

Smaller granularities (like $K = 1$) yield better performance in ranking tasks, especially with masked language modeling, which helps improve retrieval accuracy. Larger granularities lead to worse performance, especially for MRR and precision metrics.

Reconstructing on the query + document combination is expected to perform worse compared to reconstructing on just the document. The query adds additional complexity and noise to the reconstruction, making the task harder. Masking is not required on query part so instead of sending token(query+doc) for masking we will get better results if only token(doc) is sent for masking