WEAKLY SUPERVISED LABEL SMOOTHING

Presented By
CHELOUCHE Keryan
SADYKOVA Karima

MAI 2023

Implementation of the Research Proposal
by Gustavo Penha and Claudia Hauff
TU Delft

SOMMAIRE

L'idée générale

Weakly Supervised Label Smoothing

RQ1 : Le lissage des étiquettes est-il un régulariseur efficace pour les modèles neuronaux de classement de rang (L2R), et si oui, dans quelles conditions ?

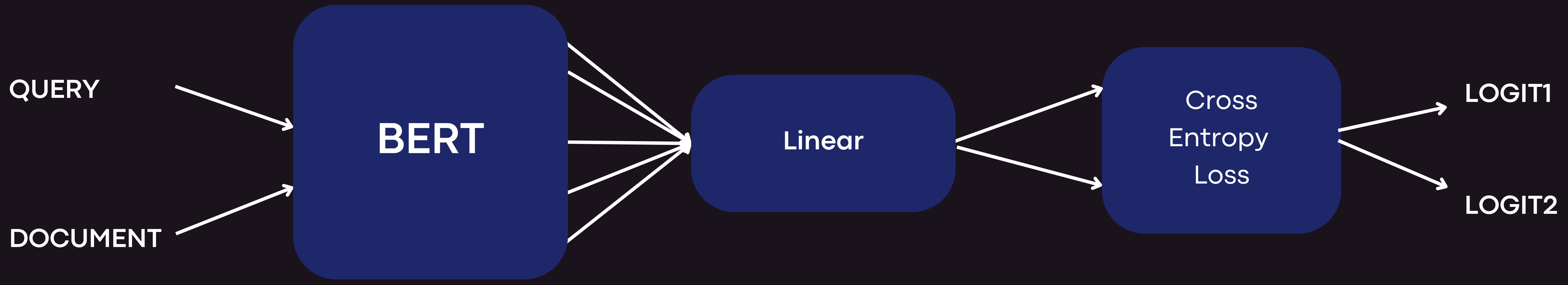
RQ2 : Le WSLS est-il plus efficace que le LS pour l'apprentissage des modèles neuronaux de classement de rang (L2R) ?

Modèle L2R

Classement basé sur BERT comme une référence solide pour l'apprentissage neural L2R

```
class BertLTRModel(nn.Module):
    def __init__(self, bert_model):
        super(BertLTRModel, self).__init__()
        self.bert = bert_model
        self.linear = nn.Linear(self.bert.config.hidden_size, 2)

    def forward(self, input_ids, attention_mask):
        output = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = output.pooler_output
        relevance_score = self.linear(pooled_output)
        return relevance_score
```



Negative Sampling

Comment choisir quel document non pertinent associé à chaque requête?

NS Random

Au hasard parmi tout les documents non pertinents de la base

NS BM25

On sélectionne les documents non pertinents au BM25 le plus élevé

LS

$$q'(k \mid x) = (1-\epsilon)\delta_{k,y} + \epsilon u(k)$$

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 1-\varepsilon \\ \varepsilon \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \longrightarrow \begin{bmatrix} \varepsilon \\ 1-\varepsilon \end{bmatrix}$$

WSLS

$$q'(k \mid x) = (1-\epsilon)\delta_{k,y} + \epsilon BM25(x)$$

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 1-\varepsilon \\ \varepsilon \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \longrightarrow \begin{bmatrix} \varepsilon \cdot BM25 \\ 1-\varepsilon \cdot BM25 \end{bmatrix}$$

CROSS-ENTROPIE

$$\ell = -\sum_{k=0}^{K} \log(p(k))q(k)$$

T-LS / T-WSLS

Après X instances d'entraînement on utilise eps = 0

Protocole d'évaluation

1. Collection des données libre accès -> **MS-MARCO-Hard-Negatives.jsonl**
2. Transformation des données en triplets de **(qid, pid, relevance) en fonction du Negative Sampling**
3. Pour chaque requête on considère 1 triplet pertinent et 9 non-pertinents
4. Apprentissage du modèle L2R et Lissage d'étiquettes selon le choix : LS, T-LS, WSLS, T-WSLS
5. Evaluation des performances

CHOIX DE LA MÉTRIQUE D'EVALUATION

1. Pour chaque requête on classe les documents selon leur score de pertinence
2. On calcule la position moyenne du document pertinent dans l'ensemble des documents
3. Le score correspond à 10 - Avg_pos

Requete_i →

Document non pertinent	6.55	1
Document non pertinent	5.05	2
Document pertinent	5.25	3
....	....	....
Document non pertinent	4.50	7
Document non pertinent	3.25	8
Document non pertinent	2.50	9
Document non pertinent	1.20	10

RESULTATS OBTENUS

Lissage/Choix triplets	NS_BM25	NS_Random
BERT	3.75	5.46
LS	4.07	3.84
T-LS	4.69	4.71

Lissage	NS_BM25
T-LS	4.69
WSLS	4.23
T-WSLS	4.90

Resultats sont obtenus pour l'apprentissage de chaque modèle avec:

- nb_epochs = 20
- learning rate = 1e-5
- smoothing = 0.2
- 50 requêtes (1 document pertinent + 9 non-pertinents par requête)

CONCLUSION

RQ1 : Le lissage des étiquettes est-il un régulariseur efficace pour les modèles neuronaux de classement de rang (L2R), et si oui, dans quelles conditions ?

RQ2 : Le WSLS est-il plus efficace que le LS pour l'apprentissage des modèles neuronaux de classement de rang (L2R) ?

	NS_{BM25}			NS_{random}		
	TREC-DL	QQP	MANtIS	TREC-DL	QQP	MANtIS
BERT	$0.568{\pm}.00$	$0.581{\pm}.03$	$0.612{\pm}.01$	$\mathbf{0.385}{\pm}.01$	$0.444{\pm}.01$	$\mathbf{0.350}{\pm}.01$
w. LS	$0.564{\pm}.01^{\blacktriangledown}$	$0.593{\pm}.01^{\blacktriangle}$	$0.612{\pm}.01$	$0.304{\pm}.05^{\blacktriangledown}$	$0.440{\pm}.03^{\blacktriangledown}$	$0.348{\pm}.01^{\blacktriangledown}$
w. T-LS	$\mathbf{0.570}{\pm}.01^{\blacktriangle}$	$\mathbf{0.598}{\pm}.01^{\blacktriangle}$	$0.612{\pm}.01$	$0.382{\pm}.02^{\blacktriangledown}$	$0.444{\pm}.01$	$0.345{\pm}.01^{\blacktriangledown}$

	TREC-DL	QQP	MANtIS
BERT	$0.599{\pm}.00$	$0.595{\pm}.01$	$0.609{\pm}.01$
w. T-LS	$0.601{\pm}.00^{\blacktriangle}$	$0.596{\pm}.01$	$0.607{\pm}.01$
w. T-WSLS	$\mathbf{0.604}{\pm}.00^{\blacktriangle\vartriangle}$	$\mathbf{0.598}{\pm}.01^{\blacktriangle\vartriangle}$	$0.609{\pm}.01^{\vartriangle}$

Resultats sont obtenus pour l'apprentissage de chaque modèle avec:

- nb_epochs = **50 000**
- learning rate = 5^(-6)
- smoothing = 0.2

l2r_class.py

```python
class BertLTRModel(nn.Module):
    def __init__(self, bert_model):
        super(BertLTRModel, self).__init__()
        self.bert = bert_model
        self.linear = nn.Linear(self.bert.config.hidden_size, 2)

    def forward(self, input_ids, attention_mask):
        output = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = output.pooler_output
        relevance_score = self.linear(pooled_output)
        return relevance_score
```

```python
class LTRDataset(Dataset):
    def __init__(self, data, tokenizer, max_len):
        self.data = data
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        query, document, relevance = self.data[idx]
        encoding = self.tokenizer.encode_plus(
            query,
            document,
            add_special_tokens=True,
            max_length=self.max_len,
            return_token_type_ids=False,
            truncation=True,
            padding='max_length',
            return_attention_mask=True,
            return_tensors='pt',
        )
        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'relevance': torch.tensor(relevance, dtype=torch.float)
        }
```

```python
def create_one_hot(target):
    # Ensure the target is a 2D tensor of shape (batch_size, 1)
    if len(target.shape) == 1:
        target = target.view(-1, 1)
    # Create a one-hot tensor of shape (batch_size, 2)
    one_hot = torch.zeros(target.size(0), 2).to(target.device)
    # Fill the one-hot tensor with (score, 1-score) values
    one_hot[:, 0] = target.squeeze()
    one_hot[:, 1] = 1 - target.squeeze()

    return one_hot

class WSLSCrossEntropyLoss(nn.Module):
    def __init__(self, smoothing=0.2):
        super(WSLSCrossEntropyLoss, self).__init__()
        self.smoothing = smoothing

    def forward(self, input, target):
        with torch.no_grad():
            # We first create a tensor with same shape as input filled with smoothing value
            true_dist = torch.ones_like(target)
            # This is to ensure that the non-relevant documents (target != 1) have the (1-target*smoothing, target*smoothing) condition
            non_relevant_indices = (target != 1).squeeze()
            relevant_indices = (target == 1).squeeze()


            # target = target.view(-1, 1)
            true_dist[non_relevant_indices] = target[non_relevant_indices].squeeze() * self.smoothing

            # Next, scatter the remaining confidence to the correct class index
            true_dist[relevant_indices] = 1-self.smoothing
            true_dist = create_one_hot(true_dist)
        # We then calculate the log probability of the inputs (this is done internally in CrossEntropyLoss)
        log_prob = F.log_softmax(input, dim=-1)

        # Our final loss is then the mean negative log likelihood
        return torch.mean(torch.sum(-true_dist * log_prob, dim=-1))
```

Utils

```python
def create_tripletWSLS(data_list):
    # Now data_list contains multiple JSON objects, one per line from the file
    data_id = []
    bm25_max = 0
    pos_examples = []
    neg_examples = []

    for data in data_list:
        # Extract positive examples
        pos_examples.extend([(data['qid'], pos['pid'], 1) for i, pos in enumerate(data['pos']) if i < 1])

        # Extract negative examples with a bm25_score
        for i, neg in enumerate(data['neg']['bm25']):
            if i < 9:
                if neg['bm25-score'] > bm25_max:
                    bm25_max = neg['bm25-score'] + 1
                neg_examples.append((data['qid'], neg['pid'], neg['bm25-score']))

    neg_examples = [(qid, pid, float(score) / float(bm25_max)) for qid, pid, score in neg_examples]
    max_len = max(len(pos_examples), len(neg_examples))

    for i in range(max_len):
        if i < len(pos_examples):
            data_id.append(pos_examples[i])
        for j in range(9*i, 9*(i+1)):
            if j < len(neg_examples):
                data_id.append(neg_examples[j])

    # Print the final_data
    print("---------")
    print("data size", len(data_id))
    print("bm25_max : ", bm25_max)
    print(data_id[:20])
    print("---------")
    return data_id
```

```python
def create_triplet(data_list):
    # Now data_list contains multiple JSON objects, one per line from the file
    data_id = []

    for data in data_list:
        # Extract positive examples
        pos_examples = [(data['qid'], pos['pid'], 1) for i, pos in enumerate(data['pos']) if i < 1]

        # Extract negative examples with a bm25_score
        neg_examples = [(data['qid'], neg['pid'], 0) for i, neg in enumerate(data['neg']['bm25']) if i < 9]

        # Combine positive and negative examples
        data_id.extend(pos_examples + neg_examples)

    # Print the final_data
    print("data size", len(data_id))
    print(data_id[:20])
    return data_id
```

```python
import numpy as np

def create_nsbm25(data_id, wsls=False):
    # Create an empty list to store the modified data
    data_bm25 = []

    for data in data_id:
        # Extract the values from the original data
        dict_value = dict_query[data[0]]
        store_value = store.get(data[1])[1]
        class_value = data[2]


        if not wsls:
            # Perform one-hot encoding
            class_vector = np.zeros(2)
            if class_value == 1:
                class_vector[0] = 0
                class_vector[1] = 1
            elif class_value == 0:
                class_vector[0] = 1
                class_vector[1] = 0
        else:
            class_vector = class_value

        # Create a modified tuple with the encoded class vector
        modified_tuple = (dict_value, store_value, class_vector)

        # Append the modified tuple to the list
        data_bm25.append(modified_tuple)
    return data_bm25

data_bm25 = create_nsbm25(data_id)
data_bm25_wsls = create_nsbm25(data_id_wsls, wsls=True)
```

```python
def learnModelLS(data, num_epochs=1, tls=-1, eps=0.2):
    dataset = LTRDataset(data, tokenizer, max_len=512)

    # Create data loaders
    loader = DataLoader(dataset, batch_size=16, num_workers=4)

    # Instantiate the model
    model = BertLTRModel(bert_model)

    # Move model to GPU if available
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = model.to(device)

    # Define loss function and optimizer
    criterion = nn.CrossEntropyLoss(label_smoothing=eps)
    optimizer = torch.optim.AdamW(model.parameters(), lr=1e-5)

    list_loss = []
    # Training loop
    for epoch in range(num_epochs):
        model.train()
        total_loss = 0

        for batch in loader:
            # Move batch tensors to the same device as the model
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            relevance = batch['relevance'].to(device)

            # Forward pass
            outputs = model(input_ids=input_ids, attention_mask=attention_mask)

            # Compute loss
            if tls == num_epochs:
                criterion = nn.CrossEntropyLoss()
            loss = criterion(outputs, relevance)

            # Backward pass and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
        loss = total_loss/len(loader)
        list_loss.append(loss)
        print(f'Epoch {epoch}, Loss: {loss}')
    return model, list_loss
```

Evaluation

```python
print(f'LS-NS-Random : {get_avg_pos(outputs_random_ls)}')
print(f'TLS-NS-Random : {get_avg_pos(outputs_random_tls)}')
     get_avg_pos

print(f'NS-BM25 : {get_avg_pos(outputs_bm25)}')
print(f'NS-Random : {get_avg_pos(outputs_random)}')

print(f'LS : {get_avg_pos(outputs_bm25_ls)}')
print(f'T-LS : {get_avg_pos(outputs_bm25_tls)}')

print(f'WSLS : {get_avg_pos(outputs_wsls)}')
print(f'T-WSLS : {get_avg_pos(outputs_wsls_tls)}')
```

```python
def get_avg_pos(all_outputs, wsls=False):
    ranked_docs = []
    #Rank the documents by their scores
    for i in range(10,len(all_outputs)+1, 10):
        query_docs = all_outputs[i-10:i, 1-int(wsls)]
        ranked_docs_by_query = np.argsort(query_docs)
        ranked_docs.append(ranked_docs_by_query)
    # Store the positions
    positions = []

    for arr in ranked_docs:
        position = np.where(arr == 0)[0]
        if position.size > 0:
            positions.append(position[0])

    # Calculate the average position of 0
    average_position = sum(positions) / len(positions)
    return average_position
```

```python
 Create data loaders
ef eval_model(dataset, model):
    loader = DataLoader(dataset, batch_size=16, num_workers=4)

    # Create an empty list to store the outputs
    all_outputs = []

    # Move model to GPU if available
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = model.to(device)

    # Evaluate model
    model.eval()
    for batch in loader:
        # Move batch tensors to the same device as the model
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        relevance = batch['relevance'].to(device)

        # Forward pass
        with torch.no_grad():  # Ensure no gradients are calculated
            outputs = model(input_ids=input_ids, attention_mask=attention_mask)

        # Move outputs to CPU and convert to numpy array
        outputs = outputs.detach().cpu().numpy()

        # Append the outputs to the list
        all_outputs.append(outputs)

    # Concatenate all outputs
    all_outputs = np.concatenate(all_outputs)
    return all_outputs
```