# Lab 2: How Multilingual is Multilingual BERT?
# Multilingual NLP

Manon Fleury
22410952

## 2. Fine-Tuning mBERT

```
!curl -O https://raw.githubusercontent.com/UniversalDependencies/UD_French-Sequoia/master/fr_sequoia-ud-test.conllu

import conllu
def load_conllu(filename):
  for sentence in conllu.parse(open(filename, "rt", encoding="utf-8").read()):
    tokenized_words = [token["form"] for token in sentence]
    gold_tags = [token["upos"] for token in sentence]
    yield tokenized_words, gold_tags
corpus = list(load_conllu("fr_sequoia-ud-test.conllu")) # list of sets of two lists sentence and glod_tags

len(corpus) # 456 sentences
```

## 1 The Universal Dependencies Project

### 1. Why do we need "consistent annotation" for our experiment?

We need consistent annotation because it allows a fair comparison across languages. Indeed, with multilingual models, standardized grammatical categories ensure that the annotations (PoS, morphological features, syntactic dependencies) are labeled the same way, regardless of the language. This consistency is key for evaluating how well the model transfers knowledge from one language to another.

### 2. What do we use the yield keyword rather than a simple return (line 7) ? Why do we have to call the list constructor (line 9)?

The yielf keyword produces a series of values over time, rather than computing them all at once and returning them in a list. This is particularly useful for processing large datasets, as it allows for more efficient memory usage since it doesn't load everything into memory at once, it generates data on demand. And if we only need a few sentences at a time for processing, yield allows to avoid loading the entire file.

The list constructor is called to convert the generator returned by load_conllu into a list, to store all the parsed sentences and their PoS tags in memory for further analysis.

### 3. What is the distribution of labels in the test set of the Sequoia treebank?

```
from collections import Counter
all\_tags = [tag for \_, tags in corpus for tag in tags] # list of all the
    tags (one after the other, so with repetition)
label\_distribution = Counter(all\_tags)
for label, count in label\_distribution.items():
    print(f"{label}: {count}")
```

Output:

PRON: 410
VERB: 781
SCONJ: 106
DET: 1492
NOUN: 2161
ADJ: 638
PUNCT: 1084
ADV: 411
NUM: 243
ADP: 1634
PROPN: 478
CCONJ: 221
AUX: 345
_: 310
X: 36
SYM: 4

## 4. What are the multiword tokens in the test set of Sequoia? How are they annotated?

When we look at the Sequoia conllu, we see that the multiword token have a range of numbers (like 15-16) to indicate it covers multiple tokens and the next line explicits the multiple token with the first range number corresponding to the first token extracted from the multiple token, and so on.
For example:

| | | | |
|---|---|---|---|
| 15-16 | au | _ | _ |
| 15 | à | ADP | _ |
| 16 | le | DET | _ |

```
def extract_multiword_tokens(filename):
  multiword_tokens = []
  for sentence in conllu.parse(open(filename, "rt", encoding="utf-8").read
      ()):
    for token in sentence:
      if isinstance(token["id"], tuple): # (like '15-16')
        token = token["form"].lower()
        if token not in multiword_tokens:
          multiword_tokens.append(token)
  return multiword_tokens
print(extract_multiword_tokens("fr_sequoia-ud-test.conllu"))
```

Output:

```
['des', 'du', 'aux', 'au', 'desdites']
```

This is the list of the multiple tokens occurring in the conllu file, with no repetition.

### 5. Why did the UD project made the decision of splitting multiword tokens into several (grammatical) words? What is your opinion on this decision?

The Universal Dependencies (UD) project decided to split multiword tokens into separate grammatical words to maintain consistency across different languages and enhance the syntactic representation of each word's role. This approach simplifies tokenization, allowing for clearer syntactic dependencies and better handling of grammatical phenomena like agreement and case marking. By treating each component as an individual token, models can learn relationships more effectively, potentially improving performance in multilingual contexts. However, this decision may complicate the treatment of idiomatic expressions that might be better understood as single units.

### 6. According to UD tokenization guidelines, a token can contain spaces. Are there any token in the Sequoia corpora that contain spaces ? If so these spaces should be removed.

```
def tokens_space(filename):
    tokens_with_space = []
    for sentence in conllu.parse(open(filename, "rt", encoding="utf-8").
        read()):
      for token in sentence:
        if " " in token["form"]:
          tokens_with_space.append(token["form"])
    return tokens_with_space

tokens_with_space = tokens_space("fr_sequoia-ud-test.conllu")

if tokens_with_space:
  for token in tokens_with_space:
    print(token)

# Function to remove the spaces
def load_and_clean_conllu(filename):
  for sentence in conllu.parse(open(filename, "rt", encoding="utf-8").read
      ()):
    tokenized_words = [token["form"].replace(" ", "") for token in
        sentence]
    gold_tags = [token["upos"] for token in sentence]
    yield tokenized_words, gold_tags

cleaned_corpus = list(load_and_clean_conllu("fr_sequoia-ud-test.conllu"))
```

"print(token)" displayed numbers like "500 000", "3 862", that represent the tokens containing a space. So with this function, these numbers become "500000".

## 2 mBERT tokenization

### 7. Why does mBERT use a subword tokenization?

mBERT uses subword tokenization to effectively handle out-of-vocabulary words by breaking them down into smaller, meaningful components, allowing the model to process even rare or infrequent terms. This approach also reduces vocabulary size while capturing the rich morphological structures found in various languages, enhancing generalization across tasks and languages.

## 8. How is the sentence "Pouvez-vous donner les mêmes garanties au sein de l'Union Européene" 4 tokenized according to the UD convention? How is it tokenized by mBERT tokenizer?

According to the UD convention, this sentence is tokenized as:

```
["Pouvez",
 "-vous",
 "donner",
 "les",
 "mêmes",
 "garanties",
 "au",
 "sein",
 "de",
 "l",
 "'",
 "Union",
 "Européenne".]
```

While mBERT tokenizer:

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-multilingual-cased")
tokenizer.tokenize("Pouvez-vous donner les m mes garanties au sein de
    l Union  Europ enne")
```

Output:

```
['Po',
 '##uve',
 '##z',
 '-',
 'vous',
 'donner',
 'les',
 'mêmes',
 'gara',
 '##nties',
 'au',
 'sein',
 'de',
 'l',
 '[UNK]',
 'Union',
 'Euro',
 '##pée',
 '##nne']
```

We see that mBERT tokenizer splits infrequent words in smaller factors and the subword tokens are prefixed with ##.

## 9. Why is this difference in tokenization a problem for training a mBERT-based PoS tagger?

This mismatch means that the model may not directly learn the PoS tags that correspond to the tokens used in the training data, leading to inconsistencies in predictions.

Additionally, when the mBERT tokenizer generates [UNK] (unknown) tokens, it creates a gap in the training data for PoS tagging, that can degrade the model's ability to make correct predictions for those words.

## 3 Reconciling the Two Tokenizations

**10. Write a function that takes as parameter a sentence tokenized according to UD rules (i.e. a variable of type List[str] and its gold labels (also a variable of type List[str]) and implements the first principle explained above.**

```
    def update_ud_tokenization(tokens, labels):
  '''tokens = List[str], sentence tokenized according to the UD rules
  labels = List[str] of the labels corresponding to the tokens
  Returns UD updated tokenzation and UD updated labels'''
  new_tokens = []
  new_labels = []
  i = 0
  while i < len(tokens):
    token = tokens[i]
    label = labels[i]
    if label == '_' and i+2 < len(tokens): # i+2: we must have 2 more
        tokens after
      multiword_token = token
      concatene_label = labels[i+1] + '+' + labels[i+2]
      new_tokens.append(token)
      new_labels.append(concatene_label)
      i += 2
    else: # if the token is not a multiword token
      new_tokens.append(token)
      new_labels.append(label)
    i += 1
  return new_tokens, new_labels

#test
print(update_ud_tokenization(['aller','au','a','le','marche'], ['VERB', '_
    ', 'ADP','DET','NOUN']))
```

(['aller', 'au', 'marché'], ['VERB', 'ADP+DET', 'NOUN'])

**11. Write a function that takes as input the sentences and labels created by "normalizing" UD sentences, apply the mBERT tokenizer and compute the corresponding label. You must ensure each subtoken (including padding symbols) has a label.**

```
    def labels_mBERT(sentences, labels):
  '''sentences : list of list of words by UD tokenization
  labels : list of list of labels by UD original labels'''
  L_mBERT_labels = [] # to return
  for sentence, label in zip(sentences,labels):
    # HERE we process ONE sentence and the labels corresponding (the '
        label' list)

    # 1. Apply the mBERT tokenizer
    info_sentence = tokenizer(sentence, is_split_into_words=True,
```

```
            return_offsets_mapping=True,
            padding=True,
            truncation=True)
    L_offset_mapping = info_sentence['offset_mapping'] # list of tuples of offmap for this sentence, with two more tuple (0,0) one at the beginning and one at the end
    # We get rid of the first element and the last element of the list ((0,0)), unrelevant for our processing
    L_offset_mapping.pop(0)
    L_offset_mapping.pop(-1)

    # 2. Compute the corresponding label
    for i,couple in enumerate(L_offset_mapping):
      if couple[0] != 0: # <pad>
        label.insert(i, '<pad>')
    L_mBERT_labels.append(label) # we add the list of labels for one sentence in the broader list labels containing all the labels
  return L_mBERT_labels

# test
print(labels_mBERT([['Revenons', '-en', 'aux', 'choses', 'essentielles', '.'], ['Parcourir',"l'",'itineraire','!']], [['VERB', 'PRON', 'ADP+DET', 'NOUN', 'ADJ', 'PUNCT'],['VERB','DET','NOUN','PUNCT']]))
```

```
[['VERB', '<pad>', '<pad>', 'PRON', '<pad>', 'ADP+DET', 'NOUN', 'ADJ', '<pad>', 'PUNCT'], ['VERB', '<pad>', '<pad>', 'DET', '<pad>', 'NOUN', '<pad>', '<pad>', 'PUNCT']]
```

## 12. Write a function that encodes the labels into integers.

```
def encode_labels(labels):
  '''labels: [list of [list of labels for one sentence]]'''

  # GENERATE DICT OF UNIQUE LABEL TO UNIQUE INTEGER
  unique_labels = set(label for sublist in labels for label in sublist)
  label_dict = {label: (-100 if label == "<pad>" else idx + 1) for idx, label in enumerate(unique_labels)}

  # ENCORE LABELS INTO INTEGERS
  list_integers_of_all_label_lists = []
  for L_label in labels:
    list_integers_of_one_label_list = []
    for label in L_label:
      list_integers_of_one_label_list.append(label_dict[label])
    list_integers_of_all_label_lists.append(
      list_integers_of_one_label_list)
  return list_integers_of_all_label_lists, label_dict

# test
print(encode_labels([['VERB', '<pad>', '<pad>', 'PRON', '<pad>', 'ADP+DET', 'NOUN', 'ADJ', '<pad>', 'PUNCT'],
 ['VERB', '<pad>', '<pad>', 'DET', '<pad>', 'NOUN', '<pad>', '<pad>', 'PUNCT']]))
```

```
([[6, -100, -100, 3, -100, 4, 2, 5, -100, 1], [6, -100, -100, 8, -100, 2, -100, -100, 1]], 'PUNCT': 1, 'NOUN': 2, 'PRON': 3, 'ADP+DET': 4, 'ADJ': 5, 'VERB': 6, '<pad>':  -100, 'DET':
```

8)

## 4 Creating a Dataset

### 13. Using the Dataset.from_list method, write a method that creates a Dataset that encapsulates a corpus in the conllu.

```
from datasets import Dataset

def create_dataset(sentences, labels):
    '''sentences : list of list of words by UD tokenization
    labels : list of list of labels by encoded_labels'''
    tokenized_sentences = tokenizer(sentences,
                                    is_split_into_words=True,
                                    return_offsets_mapping=False,
                                    padding=True, truncation=True)
    max_length = max(len(ids) for ids in tokenized_sentences['input_ids'])

    padded_labels = []
    for label in labels:
        padded_label = label + [-100] * (max_length - len(label))
        padded_labels.append(padded_label)

    dataset_entries = []
    for i in range(len(sentences)):
        dataset_entries.append({
            'input_ids': tokenized_sentences['input_ids'][i],
            'attention_mask': tokenized_sentences['attention_mask'][i],
            'labels': padded_labels[i]
        })

    dataset = Dataset.from_list(dataset_entries)
    return dataset
```

### 14. Create three instances of Dataset : one for the train set, one for the dev set and the last one for the test set.

```
from datasets import Dataset
import random
import numpy as np

def split_dataset(dataset, train_ratio=0.8, dev_ratio=0.1, test_ratio=0.1)
    :
    total_samples = len(dataset)
    train_size = int(train_ratio * total_samples)
    dev_size = int(dev_ratio * total_samples)
    test_size = total_samples - train_size - dev_size

    indices = list(range(total_samples))
    random.shuffle(indices)

    train_indices = indices[:train_size]
```

```
    dev_indices = indices[train_size:train_size + dev_size]
    test_indices = indices[train_size + dev_size:]

    train_set = dataset.select(train_indices)
    dev_set = dataset.select(dev_indices)
    test_set = dataset.select(test_indices)

    return train_set, dev_set, test_set
```

**Main program**:

```
unique_labels_list = []
L_tokens = []
L_labels = []
for one_set in corpus: # corpus = list of sets of 2 lists (sentence, labels)
  # one_set = ([list tokens], [list labels])
  tokens = one_set[0]
  labels = one_set[1]
  updated_tokens, updated_labels = update_ud_tokenization(tokens,labels)
  L_tokens.append(updated_tokens) # list of lists of sentence tokens # 456 lists
  L_labels.append(updated_labels) # list of lists of sentence labels # 456 lists

aligned_labels = labels_mBERT(L_tokens, L_labels) # <pad>
encoded_labels, dico_label = encode_labels(aligned_labels) # integers
print('dico_label:',dico_label)

dataset = create_dataset(L_tokens,encoded_labels)

train_set, dev_set, test_set = split_dataset(dataset)
print(train_set)
print(dev_set)
print(test_set)

unique_labels = set(label for sublist in aligned_labels for label in sublist)
len(unique_labels) # 17 unique labels, useful later for the model
```

Output:

```
dico_label: {'ADV': 1, 'ADP': 2, 'PUNCT': 3, 'CCONJ': 4, 'X': 5, 'NUM': 6, 'NOUN': 7,
'PRON': 8, 'ADP+DET': 9, 'SYM': 10, 'SCONJ': 11, 'PROPN': 12, 'ADJ': 13, 'VERB': 14,
'<pad>': -100, 'DET': 16, 'AUX': 17}

Dataset({
    features: ['input_ids', 'attention_mask', 'labels'],
    num_rows: 364
})
Dataset({
    features: ['input_ids', 'attention_mask', 'labels'],
    num_rows: 45
})
Dataset({
```

```
    features: ['input_ids', 'attention_mask', 'labels'],
    num_rows: 47
})
```

# 5 Fine-Tuning mBERT

**15. How can you modify the code of Figure 4 to report the PoS tagging accuracy during optimization. Why is this information important?**

```python
from evaluate import load
import numpy as np
from transformers import AutoModelForTokenClassification, TrainingArguments, Trainer

base_model_checkpoint = "bert-base-multilingual-cased"
model = AutoModelForTokenClassification.from_pretrained(base_model_checkpoint, num_labels=len(list(unique_labels))+1)

metric = load("accuracy")

def compute_metrics(pred):
    logits, labels = pred.predictions, pred.label_ids
    predictions = np.argmax(logits, axis=-1)
    # Mask of the -100 labels to ignore the padding labels
    true_labels = labels[labels != -100]
    true_predictions = predictions[labels != -100]
    accuracy = metric.compute(predictions=true_predictions, references=true_labels)
    return {"accuracy": accuracy["accuracy"]}


training_args = TrainingArguments(
    output_dir="./mBERT_fine_tuning",
    eval_strategy="epoch",  # evaluate at each epoch
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    logging_dir='./logs',
    logging_steps=10,
    report_to="none"  #Deactivate the integration with 'wandb'
    )

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset= train_set,
    eval_dataset= dev_set,
    compute_metrics=compute_metrics
)

trainer.train()
```

| Epoch | Training Loss | Validation Loss | Accuracy |
|---|---|---|---|
| 1 | 2.204400 | 1.821981 | 0.506024 |
| 2 | 1.598800 | 1.264914 | 0.653614 |
| 3 | 1.254000 | 1.050898 | 0.707831 |

Figure 1: Fine-tuning results for French

# 6 Evaluating the multilingual capacity of mBERT

**16. Choose 5 languages from the UD project. For each language train a PoS tagger using the code of the previous section and test it on the 5 languages you have chosen.**

```
corpus_breton = list(load_conllu("br_keb-ud-test.conllu"))
corpus_chinese = list(load_conllu("zh_hk-ud-test.conllu"))
corpus_russian = list(load_conllu("ru_pud-ud-test.conllu"))
corpus_french = corpus

def preprocess_corpus(corpus):
  unique_labels_list = []
  L_tokens = []
  L_labels = []
  for one_set in corpus:
    tokens = one_set[0]
    labels = one_set[1]
    updated_tokens, updated_labels = update_ud_tokenization(tokens,labels)
    L_tokens.append(updated_tokens)
    L_labels.append(updated_labels)

  aligned_labels = labels_mBERT(L_tokens, L_labels)
  encoded_labels, dico_label = encode_labels(aligned_labels)

  dataset = create_dataset(L_tokens,encoded_labels)
  train_set, dev_set, test_set = split_dataset(dataset)

  unique_LABELS = set(label for sublist in aligned_labels for label in
      sublist)

  return train_set, dev_set, test_set,unique_LABELS


from evaluate import load
import numpy as np
from transformers import AutoModelForTokenClassification,
    TrainingArguments, Trainer

metric = load("accuracy")

def compute_metrics(pred):
    logits, labels = pred.predictions, pred.label_ids
```

```
    predictions = np.argmax(logits, axis=-1)
    true_labels = labels[labels != -100]
    true_predictions = predictions[labels != -100]
    accuracy = metric.compute(predictions=true_predictions, references=
        true_labels)
    return {"accuracy": accuracy["accuracy"]}

def evaluate_on_all_languages(trainer, test_datasets, languages):
    """Evaluate the fine-tuned model on all chosen languages.
    test_sets (dict): Dictionary of test datasets keyed by language code.
    languages (list): List of languages.
    Returns:dictionary of evaluation accuracies keyed by language code."""

    scores = {}
    for lang in languages:
        print(f"Evaluating on language: {lang}")
        metrics = trainer.evaluate(test_datasets[lang])
        scores[lang] = metrics.get("eval_accuracy", 0.0)
        print(f"{lang}: {scores[lang]}")
    return scores


def finetune(train_set,dev_set,unique_LABELS,language,test_datasets):
  base_model_checkpoint = "bert-base-multilingual-cased"
  model = AutoModelForTokenClassification.from_pretrained(
      base_model_checkpoint, num_labels=len(list(unique_LABELS))+1)

  training_args = TrainingArguments(
      output_dir=f"./mBERT_fine_tuning_{language}",
      eval_strategy="epoch",
      learning_rate=2e-5,
      per_device_train_batch_size=16,
      per_device_eval_batch_size=16,
      num_train_epochs=3,
      weight_decay=0.01,
      logging_dir=f"./logs_{language}",
      logging_steps=10,
      report_to="none"
      )

  trainer = Trainer(
      model=model,
      args=training_args,
      train_dataset= train_set,
      eval_dataset= dev_set,
      compute_metrics=compute_metrics
  )

  trainer.train()

  # Evaluate the model on all languages
  scores = evaluate_on_all_languages(trainer, test_datasets, list(corpora.
      keys()))
  return scores
```

```python


# Main program

corpora = {'breton':corpus_breton,
           'chinese':corpus_chinese,
           'russian':corpus_russian,
           'french':corpus_french}
languages = list(corpora.keys())
evaluation_results = {}

for language, corpus in corpora.items():
    train_set, dev_set, test_set, unique_LABELS = preprocess_corpus(corpus)
    test_datasets = {lang: preprocess_corpus(corpus)[2] for lang, corpus in corpora.items()}

    # Fine-tune the model and evaluate
    scores = finetune(train_set, dev_set, unique_LABELS, language, test_datasets)

    # Store the scores
    evaluation_results[language] = scores
```

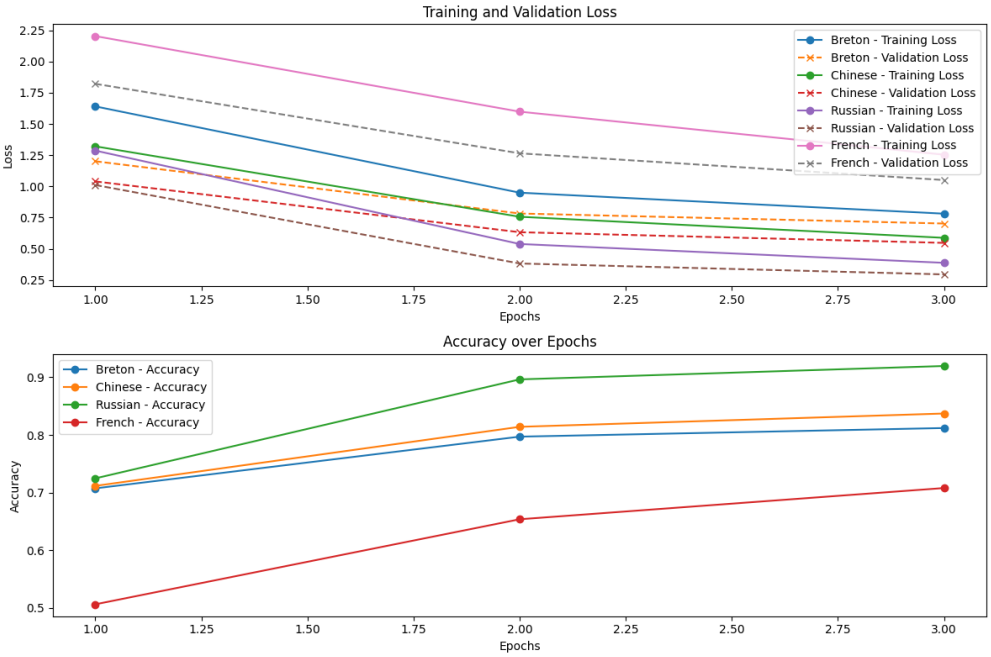



Figure 2: Finetuning results for multiple languages (Breton, Chinese, Russian, French)

These graphs show that the fine-tuning globally enhanced the performances: the training and validation loss decreases for all languages and the accuracy increases, with the Russian having better results than the others, suggesting potential challenges in fine-tuning.

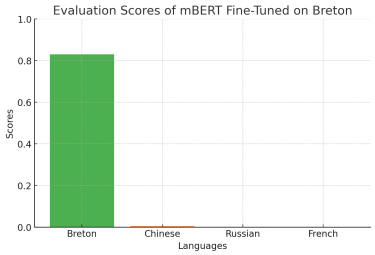
**Results for Breton:**

- Result of finetuning:

| Epoch | Training Loss | Validation Loss | Accuracy |
|---|---|---|---|
| 1 | 1.641100 | 1.201294 | 0.707265 |
| 2 | 0.949500 | 0.782655 | 0.797009 |
| 3 | 0.781400 | 0.702319 | 0.811966 |

- Result of evaluation on all languages (scores):

breton: 0.83
chinese: 0.0047
russian: 0.0010
french: 0.00085




The high score for Breton (0.83) indicates that the model fine-tuned on this language performs well for its own data. However, the dramatically lower scores for Chinese (0.0047), Russian (0.0010), and French (0.00085) reveal that this fine-tuning did not generalize well to other languages. This could indicate that mBERT struggles to transfer representations learned from a language like Breton.

## 17. Can you use the same hyperparameters for the different languages?

Using the same hyperparameters across all languages may not yield optimal results due to the variability observed in performance between languages. For instance, Breton achieved relatively high accuracy, while Chinese, Russian, and French had much lower scores when evaluated on the Breton-fine-tuned model. This suggests that adjusting hyperparameters, such as learning rate or training epochs, could improve results by better accommodating each language's unique characteristics.

## 18. What can you conclude?

The results indicate that mBERT's multilingual capabilities vary significantly across languages. While fine-tuning generally improves accuracy, some languages perform better than others, suggesting that mBERT's representations are more effective for certain languages. This highlights both the strengths and limitations of using a single multilingual model across diverse linguistic contexts, and suggests that language-specific adjustments may be necessary for optimal performance.