Experiment 9 - Introduction to Natural Language Processing

1.1 Theory and Introduction

Machine Learning has different applications in different spaces. In previous experiments, we have applied machine learning algorithms on tabular data, like IRIS dataset for example. Later on, we saw how those algorithms can be applied in different spaces like the vision space and we've solved image processing tasks using machine learning/deep learning techniques, like image segmentation. In this experiment, we're going to explore a different space in which machine learning/deep learning algorithms are used. That is the natural language space, where the input is a text (sentence, paragraph, document, etc.) and we're trying to obtain a desired output from that input. The output could be a class (e.g., classification tasks), a sentence (e.g., summarization or translation tasks), or a word (e.g., next-word prediction tasks). Those tasks are usually referred to as Natural Language Processing (NLP) tasks.

Tweet	**Sentiment**
I broke my leg	Negative
I need to find a way to be happy	Negative
so excited for the new JB album	Positive

Table 1.1: Sample data from twitter's sentiment140 dataset for Sentiment Analysis task.

1.1.1 Representation

Similar to visual data in computer vision tasks, the first question in NLP tasks is how can we represent the text data in a way that machine learning algorithms can handle. In computer vision, we had both classical and deep learning methods to handle visual data. NLP isn't any different.

Before jumping to the actual methods, let's try to think about how should the representation be. Let's examine the input characteristics. We're expecting to get a text, a sequence of words. For example, "I need to find a way to be happy" is a

text, made up of 9 words. Not all texts are made of 9 words so we need to be able to handle that undetermined length.

So that leaves us with two different questions:

1. How can we represent a *word*?
2. How can we represent a *sequence* with *undetermined length*?

Word Representation

For this section, we should think about only one aspect of the task. If we have only one word, how can we feed that word to the model to predict the sentiment?

Word	Sentiment
broke	Negative
happy	Positive

Table 1.2: Sample data for single word sentiment.

We can treat the *word* as a categorical feature where its values are all the distinct words in the dataset. So if we have 20000 words in our dataset, that leaves us with a categorical feature with 20000 possible values. What if we have a model that requires numerical input? the encoding of the word would be a vector of size 20000, assuming we're using one-hot encoding.

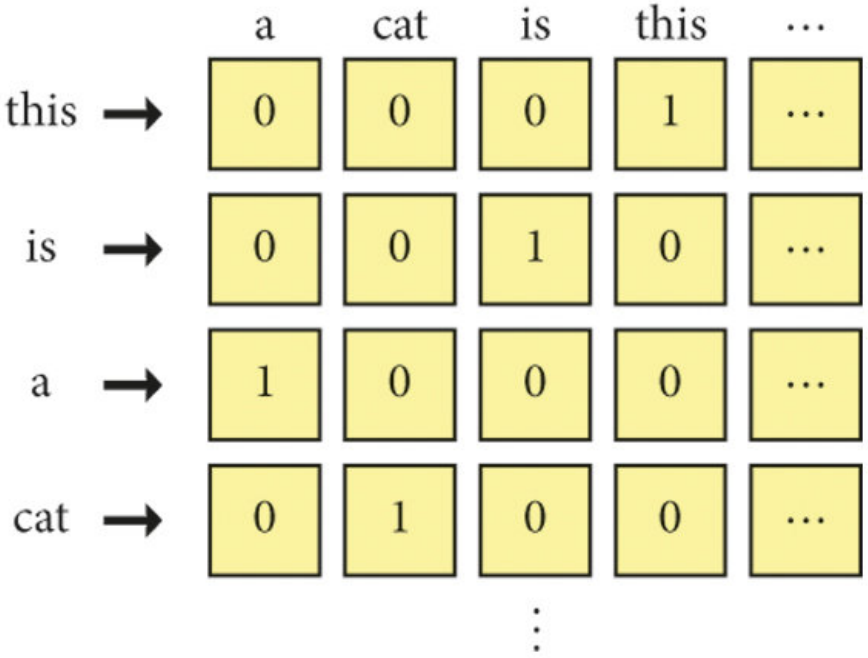

Figure 1.1: One-hot encoding to represent words as vectors.

This task is referred to as *Word Embedding*. In this task, we're trying to represent the words in a way that can be understood by machine learning models. However, the representation should preserve the word characteristics, that is, similar words should have similar representations. If one-hot encoding is used, then every pair of words would have orthogonal vectors, which means that the similarity is always 0. Such type of embeddings is referred to as Localist embeddings because every word is localized in its dimension.

Another type of embeddings is the Distributed embeddings. In this type, each word is represented by a vector of a defined length, that doesn't have to be equal to the number of distinct words. The embeddings of *semantically similar* words are similar as well. Distributed embeddings are trained on a large text (the corpus), and they use statistical analysis techniques to identify the similarity between words based on the context in which they occur. For example, if it's frequent to see sentences like "Let's meet on *Monday*" and "Let's meet on *Tuesday*" the model would conclude that *Monday* and *Tuesday* are similar words. Word2Vec and GloVe are examples of distributed embeddings.

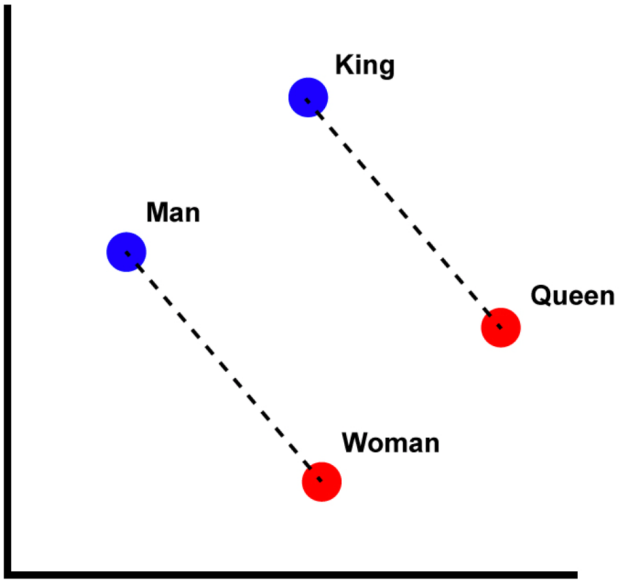

Figure 1.2: Representing words with a vector distributed in space.

Both of the previous types are called to be Static embeddings. Once they're trained, they'd produce the same embedding for each word no matter what context the word came in. For example, the word *can* would have the same embedding in the following sentences even though it has a different meaning in each of them: "A *can* of cola." and "You *can* do it.". Another type of embedding that can solve this issue is the Contextualized embeddings. Instead of feeding a single word to the embeddings model, we provide the context in which this word occurred. This allows the model to return different embeddings for the same word depending on the context. ELMo and BERT are examples of contextualized embeddings.

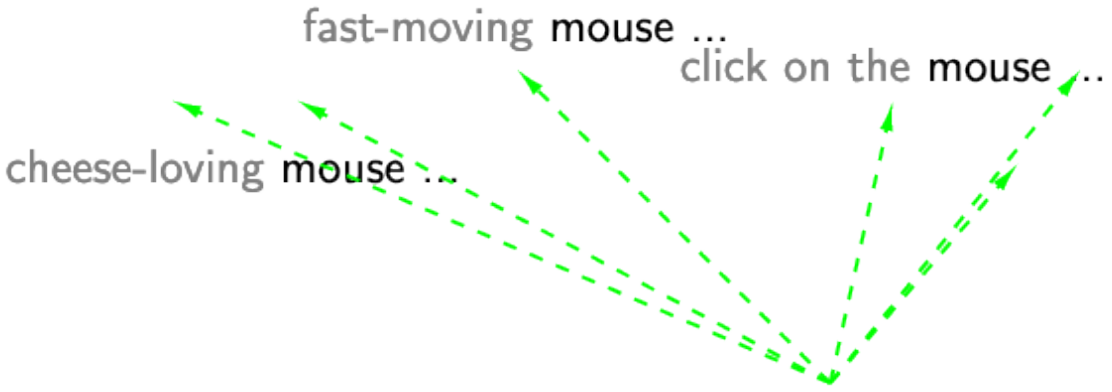

Figure 1.3: Representing words based on the context in which they appear.

Text Representation

Now we have a way to represent the words, how can we represent a sequence of words given that we can't determine the length of the sequence?

The naive approach would be to assume a constant length for the sequences. If a sequence exceeds that length, we simply truncate it. If the sequence is less than the predefined length, then we pad it with a special embedding vector, for example, the zero vector.

#1	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**Sentiment**
I	broke	my	leg	$\langle pad \rangle$	$\langle pad \rangle$	$\langle pad \rangle$	Negative
I	need	to	find	a	way	to	Negative
so	excited	for	the	new	JB	album	Positive

Table 1.3: Converting all sequences to 7-words sequence using padding and truncation.

The main issue with such representation is that it requires setting a constant for the sequence length that might require some knowledge of the anticipated data. Furthermore, it might lead to severe information loss. For example, in the second sentence, we've lost the words "be happy" which are crucial to determine the sentiment of that sentence.

Another approach is to *aggregate* the sequence in a way that reduces any arbitrary length sequence to a single element of specific length. The easiest way might be to average the vectors of all words. For a sequence of length $|S|$, and an embedding function f, the sequence representation would be:

$$x = \frac{1}{|S|} \sum_{i=0}^{|S|} f\left(w_i\right) \tag{1.1}$$

Where w_i is the word at position i. The problem with such an approach is that it doesn't preserve the positional information of the words. For example, both "I am happy because I did *not* fail the exam" and "I am *not* happy because I did fail the exam" have the same representation, while they've opposite meanings and sentiments.

Another option would be Sequence Models. Which are special architectures of models that support sequence data (i.e., ordered varying length data). Examples of sequence models are: Recurrent Neural Networks (RNNs), Long- Short-Term Memory (LSTM), and Transformers.

1.1.2 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are networks with loops. A neural network $\mathbf{A}$ can transform the input $\vec{x}$ to the output $\mathbf{A}(\vec{x})$. While an RNN would have a loop/feedback that allows it to handle sequences. The output of the RNN is in the form of $\mathbf{A}\left(\vec{x}, \vec{h}\right)$ where $\vec{h}$ is the hidden state passed through the loop connection.

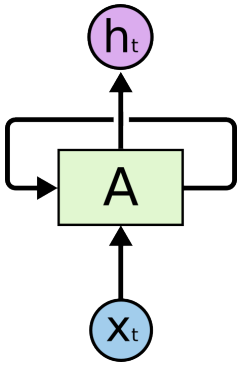

Figure 1.4: Recurrent Neural Network (RNN)

If we passed a sequence of inputs $\mathbf{X}$, each input $\vec{x_t}$ would be mapped to the corresponding output $\vec{h_t}$. This output $\vec{h_t}$ is going to be passed through the loop connection when the next input $\vec{x_{t+1}}$ comes in. Thus, the output of the RNN is described by the following equation:

$$\vec{h_t} = \mathbf{A}\left(\vec{x_t}, \vec{h_{t-1}}\right) \tag{1.2}$$

Figure 1.5 shows how an RNN unit can be viewed as a sequence of units with arbitrary length, each unit takes the previous state as input along with the current element.

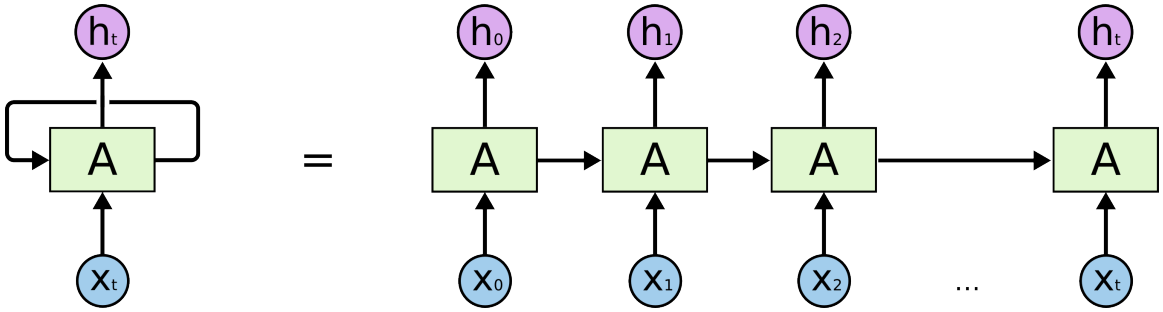

Figure 1.5: A simple RNN unit on the LHS, and its equivalent on the RHS.

LSTM is a special type of RNNs that evaluates more state variables that are different from the output variable. It has been introduced to save the long-term context and allow distant words to affect each other if they're within the same context.

1.1.3 Text Pre-processing

Although representing words and sequences is the heart of NLP. Text pre-processing plays a crucial role in the performance of an NLP model. Text pre-processing may contain the following stages:

1. **Tokenization**: the process of converting a text to a list of tokens (words). Tokenization may include converting all words to lowercase to avoid words being unmatched due to the casing. It may also include removing punctuation as well.

2. **Stop-Words Removal**: English text may contain stop-words, such as "the", "is", or "are", which are very frequent functional words that are in some NLP applications removed from the text.

3. **Stemming**: the process of replacing a word with its stem, which is the main part of the word in a sense, and it is obtained by removing a word suffix. For example, the stem of the word *waiting* is *wait*.

1.2 Procedure

In this experiment, HuggingFace is used for the datasets and the pre-trained models. Thus, start by installing huggingface packages:

```
$ pip install -U datasets transformers[torch] evaluate
```

The other dependencies required for this experiment are already installed on Google Colab.

The objective of this experiment is to solve the Sentiment Analysis task, in which you're given a text that need to be classified either to negative or positive sentiment. The `'sentiment140'` dataset is to be used in this experiment. However, due to the size of that dataset, use the `'MrbBakh/Sentiment140'` dataset, which is a subset of the `'sentiment140'` dataset. To load the data set:

```
from datasets import load_dataset

dataset = load_dataset('MrbBakh/Sentiment140')
```

1.2.1 Text Pre-processing

Using the NLTK package for text pre-processing, start by tokenizing the text:

```
import nltk
from nltk.tokenize import word_tokenize

nltk.download('punkt')

def tokenize(row):
    tokens = word_tokenize(row['text'])

    # to lowercase and remove punctuation
    tokens = [token.lower() for token in tokens if token.isalpha()]

    return {
        'tokens': tokens
    }

dataset = dataset.map(tokenize)
```

*Note: **`map`** and **`filter`** are common operators in functional APIs. The **`map`** operator maps every sample in the dataset based on the provided function. While the **`filter`** operator filters the dataset to keep only the samples that satisfy the given predicate.*

Then, remove stop words:

```
from nltk.corpus import stopwords
```

```
nltk.download('stopwords')

def remove_stopwords(row):
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in row['tokens'] if token not in stop_words]

    return {
        'tokens': tokens
    }

dataset = dataset.map(remove_stopwords)
```

Task 1: Use `PorterStemmer` from NLTK to stem the tokens.

1.2.2 Word Embedding

Now that you have pre-processed tokens, use those to train the embedding model.`Word2Vec` model from `gensim` is to be used in this experiment:

```
from gensim.models import Word2Vec

word_embedding = Word2Vec(dataset['train']['tokens'], vector_size=100,
    min_count=1, window=5, sg=1, hs=0, negative=10)
```

The model is trained on the training set's tokens, and the size of the embedding vector is 100. The other parameters are related to the method used in `Word2Vec`, which are not covered in this introductory experiment.

After training the model, you can save it and load it again if you wish to:

```
word_embedding.save('w2v.model')

word_embedding = Word2Vec.load('w2v.model')
```

1.2.3 Average Vector

The first model would be an Average Vector model. A Naive Bayes classifier is going to accept the average vector as input in order to classify samples into positive or negative sentiments.

```
def filter_tokens(example):
    return {
        'tokens': [token for token in example['tokens'] if token in
            word_embedding.wv]
    }

def mean_vector(example):
```

```
    return {
        'mean': word_embedding.wv[example['tokens']].mean(axis=0)
    }

dataset = dataset.map(filter_tokens) \
                 .filter(lambda e: len(e['tokens']) > 0) \
                 .map(mean_vector)
```

The `filter_tokens` function would return only the tokens that do have embeddings in the trained `Word2Vec` model. After that, samples that do not contain any valid token are filtered out. Then the `mean_vector` function would return the average of tokens' vectors for each sample.

Finally, train the Naive Bayes classifier:

```
import numpy as np
from sklearn.naive_bayes import GaussianNB

X = np.array(dataset['train']['mean'])
y = np.array(dataset['train']['sentiment'])

clf = GaussianNB()
clf.fit(X, y)
```

Task 2: Compute the accuracy and the confusion matrix of the trained classifier on the test dataset.

1.2.4 LSTM

In this section, LSTM is to be used instead of feeding the average vector to a Naive Bayes classifier.

LSTM in PyTorch is a module that accepts a tensor of shape $L \times V$, where L is the sequence length and V is the token vector length. LSTM module would consume each sequence element x_t, along with the previous hidden state (h_{t-1}, c_{t-1}) and output o_{t-1}, to produce the hidden state (h_t, c_t) and output o_t. As a result, the output of the entire sequence would be of shape $L \times H$ where H is the hidden size.

Let's first convert the tokens to the corresponding vectors using the trained `Word2Vec` model.

```
def vectorize(example):
    return {
        'vectors': word_embedding.wv[example['tokens']]
    }

dataset = dataset.map(vectorize)
```

Try the following LSTM module and notice the output shape. Check the shape of the input as well.

```
import torch
import torch.nn as nn

lstm = nn.LSTM(100, 200)

sequence = torch.tensor(dataset['train'][0]['vectors'])

out, _ = lstm(sequence)
```

Note: for batched input with batch size being N, LSTM would expect input as $L \times N \times V$, and outputs the shape $L \times N \times H$. To avoid that, use ***`batch_first=True`*** *and then the input would be $N \times L \times V$ and the output would be $N \times L \times H$.*

To define and use a 2-layers LSTM that accepts a batched input:

```
lstm = nn.LSTM(100, 200, 2, batch_first=True)

batch = [torch.tensor(sequence) for sequence in
    dataset['train'][0:4]['vectors']]
padded_batch = nn.utils.rnn.pad_sequence(batch)

out, _ = lstm(padded_batch)
```

Task 3: slice the output of the LSTM to get the last token's output for every sample in the batch.

Another module from PyTorch that can be used is the `Embedding`. This module contains a learnable weights matrix, that maps every word to its embedding vector. However, the input isn't exactly the word, but its index in the weights matrix.

The trained `Word2Vec` is also a weights matrix (`word_embedding.wv.vectors`). To be able to fill the `Embedding` module with `Word2Vec` weights, map the words to their corresponding index.

```
def word_to_index(example):
    indices = [word_embedding.wv.key_to_index[token] for token in
        example['tokens']]

    return {
        'indices': indices
    }

dataset = dataset.map(word_to_index)
```

To accelerate the training, batches can be padded in order to be grouped in a single tensor and moved to GPU at once. Define a padding vector, for example, the zero vector, and give it the last index to avoid collision with other indices.

```
pad_vector = np.zeros(word_embedding.vector_size)
weights = np.vstack([word_embedding.wv.vectors, pad_vector])
```

```
vocab_size, embedding_size = weights.shape
pad_idx = vocab_size - 1
```

Now, pad the sequences. Arguments `batched=True` and `batch_size=None` are used to map all dataset samples at once, and `with_format('torch')` is used to ensure that the returning dataset is a PyTorch tensor.

```
def pad_sequences(batch):
    indices = [torch.tensor(sample, dtype=torch.long) for sample in
        batch['indices']]
    indices = nn.utils.rnn.pad_sequence(indices, batch_first=True,
        padding_value=pad_idx)

    return {
        'indices': indices
    }

dataset = dataset.map(pad_sequences, batched=True,
    batch_size=None).with_format('torch')
```

Now, define the `SentimentClassifierLSTM` module using the `Embedding` followed by the 2-layered `LSTM`. An extra fully connected layer (`Linear`) is used to project the last token to a single element, followed by a to produce output between $[0, 1]$.

```
import torch.nn.functional as F

class SentimentClassifierLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_size, hidden_size,
        num_layers):
        super().__init__()

        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.embedding = nn.Embedding(vocab_size, embedding_size)

        self.lstm = nn.LSTM(embedding_size, hidden_size, num_layers,
            batch_first=True)
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        embeddings = self.embedding(x)

        out, _ = self.lstm(embeddings)

        out = out[:, -1, :]
        out = self.fc(out)
```

```
        out = F.sigmoid(out)
        return out.squeeze(1)
```

Define the model and fill the embedding layer, it's important to set `requires_grad` to `False` for the embedding layer, as it's pre-trained and shouldn't be considered in backpropagation and weights update.

```
hidden_size = 128
num_layers = 2

model = SentimentClassifierLSTM(vocab_size=vocab_size,
    embedding_size=embedding_size, hidden_size=hidden_size,
    num_layers=num_layers)

model.embedding.weight = nn.Parameter(torch.FloatTensor(weights))
model.embedding.weight.requires_grad = False
```

As usual, define the optimizer and the loss function:

```
learning_rate = 0.001

criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Then move everything to GPU:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = model.to(device)
criterion = criterion.to(device)
```

Define the dataloaders:

```
from torch.utils.data import DataLoader, TensorDataset

batch_size = 2048

def to_dataloader(dataset, split, shuffle):
    dataset = TensorDataset(dataset[split]['indices'],
        dataset[split]['sentiment'])
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)

train_dataloader = to_dataloader(dataset, 'train', True)
test_dataloader = to_dataloader(dataset, 'test', False)
validation_dataloader = to_dataloader(dataset, 'validation', False)
```

Finally, define the training function:

```
def train_one_epoch(dataloader):
```

```
    for inputs, labels in dataloader:
        inputs = inputs.to(device)
        labels = labels.to(device).float()

        outputs = model(inputs)

        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Task 4: Use the `train_one_epoch` to train the model on 20 epochs. ***Bonus***: Evaluate the model on the validation set after each epoch and print the validation accuracy.
Task 5: Evaluate the model on the test set using the accuracy and confusion matrix.
Task 6: Compare the performance of the model with the performance of the Average Vector model.

1.2.5 Transformers

Transformer is a different architecture for sequence models. It's the state-of-the-art in NLP. Use the pre-trained BERT-mini model, which is available on HuggingFace under `'lyeonii/bert-mini'`.

Start by defining the tokenizer. The tokenizer is responsible for returning the indices of the tokens. Use HuggingFace's `AutoTokenizer` with the repository of the model, and it will be able to return the appropriate tokenizer that suits the model:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained('lyeonii/bert-mini')
```

Then tokenize the dataset, the tokenizer will handle padding as well. Make sure to set `return_tensors` to `'pt'` which stands for PyTorch:

```
tokenized_dataset = dataset.map(lambda x: tokenizer(
    x['text'],
    padding=True,
    return_tensors='pt'
), batched=True, batch_size=None).with_format('torch')
```

HuggingFace models expect to receive the labels under the key `'labels'`. Thus, we need to rename the column `'sentiment'` in our dataset to be `'labels'`.

```
tokenized_dataset = tokenized_dataset.rename_column('sentiment', 'labels')
```

Then load the model itself using `AutoModelForSequenceClassification`, similar to `AutoTokenizer`:

```
from transformers import AutoModelForSequenceClassification

model =
    AutoModelForSequenceClassification.from_pretrained('lyeonii/bert-mini',
    num_labels=2)
```

Finally, use HuggingFace's `Trainer` to train the model:

```
from transformers import TrainingArguments, Trainer

training_args = TrainingArguments(
    output_dir='sentiment-analysis',
    num_train_epochs=3,
    per_device_train_batch_size=512,
    per_device_eval_batch_size=512,
    weight_decay=0.01,
    evaluation_strategy='epoch',
    save_strategy='epoch',
    logging_strategy='epoch'
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset['train'],
    eval_dataset=tokenized_dataset['validation']
)

trainer.train()
```

Task 7: Use `compute_metrics` in `Trainer` constructor, with `evaluate` package, to compute validation accuracy.
Task 8: Evaluate the model on the test set using the accuracy and confusion matrix.
Task 9: Compare the model with the previously trained models.