| كلية الحوسبة والمعلوماتية
College of Computing and Informatics

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

Second Semester – 2021/2022

Course Code	DS660
Course Name	Deep Learning Techniques
Assignment type	Critical Thinking
Module	07
Total Points	105 Points

Student ID	G200007615
Student Name	Abdulaziz Aqlumayzi
CRN	21604

Solutions:

Critical Thinking Assignment 2

Preprocessing Steps for Text Manipulation in Machine Learning

Introduction

In this activity, we will Define and illustrate typical preprocessing procedures in Machine Learning for text manipulation. Following that is a Python program example that loads a text into memory as strings, splits strings into tokens, and builds a vocabulary table to map the divided tokens to number indices.

Definition

Text is one of the most common types of sequence data. An article, for example, might be thought of as a series of words or even a series of characters. To make future experimentation with sequence data easier, we'll concentrate this part to explaining common text preparation techniques. Typically, these stages are:

1- Load text as strings into memory.

2- Separate strings into tokens (words and characters).

3- Construct a vocabulary table that maps the divided tokens to number indexes.

4- Convert text into numerical indices that can be easily modified by models.

Load text as strings into memory

Text from H. G. Wells' “The Time Machine” is loaded. This is a somewhat tiny corpus of little over 30000 words, but it suffices for the purposes of what we want to demonstrate.

Python Programming Code:

```
#@save
d2l.DATA_HUB['time_machine'] = (d2l.DATA_URL + 'timemachine.txt',
                                '090b5e7e70c295757f55df93cb0a180b9691891a')

def read_time_machine():  #@save
```

```
    """Load the time machine dataset into a list of text lines."""
    with open(d2l.download('time_machine'), 'r') as f:
        lines = f.readlines()
    return [re.sub('[^A-Za-z]+', ' ', line).strip().lower() for line in
lines]

lines = read_time_machine()
print(f'# text lines: {len(lines)}')
print(lines[0])
print(lines[10])
```

Python Programming Result:

```
Downloading ..\data\timemachine.txt from http://d2l-data.s3-accelerate.
amazonaws.com/timemachine.txt...
# text lines: 3221
the time machine by h g wells
twinkled and his usually pale face was flushed and animated the
```

Separate strings into tokens

The tokenize function below accepts a list (lines) as input, with each element being a text sequence. Each text sequence is divided into tokens. A token is the fundamental unit of text. Finally, a list of token lists is returned, with each token being a string.

Python Programming Code:

```
def tokenize(lines, token='word'):  #@save
    """Split text lines into word or character tokens."""
    if token == 'word':
        return [line.split() for line in lines]
    elif token == 'char':
        return [list(line) for line in lines]
    else:
        print('ERROR: unknown token type: ' + token)

tokens = tokenize(lines)
for i in range(11):
    print(tokens[i])
```

Python Programming Result:

```
['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
[]
[]
[]
[]
['i']
```

```
[]
[]
['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be', 'convenie
nt', 'to', 'speak', 'of', 'him']
['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his', 'g
rey', 'eyes', 'shone', 'and']
['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed',
'and', 'animated', 'the']
```

Construct a vocabulary table that maps the divided tokens to number indexes

Here we construct a dictionary, also known as a vocabulary, to map string tokens into number indices beginning with 0. To do this, we first count the unique tokens in all of the documents from the training set, referred to as a corpus, and then give a numerical index to each unique token based on its frequency. To decrease complexity, tokens that appear only seldom are frequently eliminated. Any token that does not exist in the corpus or has been deleted is mapped to a unique unknown token "< unk >". We optionally provide a list of reserved tokens, such as "< pad >" for padding, "< bos >" for presenting the start of a series, and "< eos >" for presenting the end of a sequence.

Python Programming Code:

```
class Vocab:  #@save
    """Vocabulary for text."""
    def __init__(self, tokens=None, min_freq=0, reserved_tokens=None):
        if tokens is None:
            tokens = []
        if reserved_tokens is None:
            reserved_tokens = []
        # Sort according to frequencies
        counter = count_corpus(tokens)
        self._token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                   reverse=True)
        # The index for the unknown token is 0
        self.idx_to_token = ['<unk>'] + reserved_tokens
        self.token_to_idx = {token: idx
                             for idx, token in
enumerate(self.idx_to_token)}
        for token, freq in self._token_freqs:
            if freq < min_freq:
                break
            if token not in self.token_to_idx:
                self.idx_to_token.append(token)
                self.token_to_idx[token] = len(self.idx_to_token) - 1

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

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):
        if not isinstance(indices, (list, tuple)):
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]

    @property
    def unk(self):  # Index for the unknown token
        return 0

    @property
    def token_freqs(self):  # Index for the unknown token
        return self._token_freqs

def count_corpus(tokens):  #@save
    """Count token frequencies."""
    # Here `tokens` is a 1D list or 2D list
    if len(tokens) == 0 or isinstance(tokens[0], list):
        # Flatten a list of token lists into a list of tokens
        tokens = [token for line in tokens for token in line]
    return collections.Counter(tokens)
```

Convert text into numerical indices that can be easily modified by models

As the corpus, we build a vocabulary using the time machine dataset. Then we publish the first five frequently occurring tokens together with their indexes.

Python Programming Code:

```
vocab = Vocab(tokens)
print(list(vocab.token_to_idx.items())[:10])
```

Python Programming Result:

```
[('<unk>', 0), ('the', 1), ('i', 2), ('and', 3), ('of', 4), ('a', 5), (
'to', 6), ('was', 7), ('in', 8), ('that', 9)]
```

Each text line may now be converted into a list of number indexes.

Python Programming Code:

```
for i in [0, 10]:
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])
```

Python Programming Result:

```
words: ['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
indices: [1, 19, 50, 40, 2183, 2184, 400]
words: ['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'fl
ushed', 'and', 'animated', 'the']
indices: [2186, 3, 25, 1044, 362, 113, 7, 1421, 3, 1045, 1]
```

Using the preceding functions, we wrap everything into the load_corpus_time_machine function, which returns corpus, a list of token indices, and vocab, the vocabulary of the time machine corpus.

Python Programming Code:

```
def load_corpus_time_machine(max_tokens=-1):  #@save
    """Return token indices and the vocabulary of the time machine 
dataset."""
    lines = read_time_machine()
    tokens = tokenize(lines, 'char')
    vocab = Vocab(tokens)
    # Since each text line in the time machine dataset is not necessarily a
    # sentence or a paragraph, flatten all the text lines into a single 
list
    corpus = [vocab[token] for line in tokens for token in line]
    if max_tokens > 0:
        corpus = corpus[:max_tokens]
    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)
```

Python Programming Result:

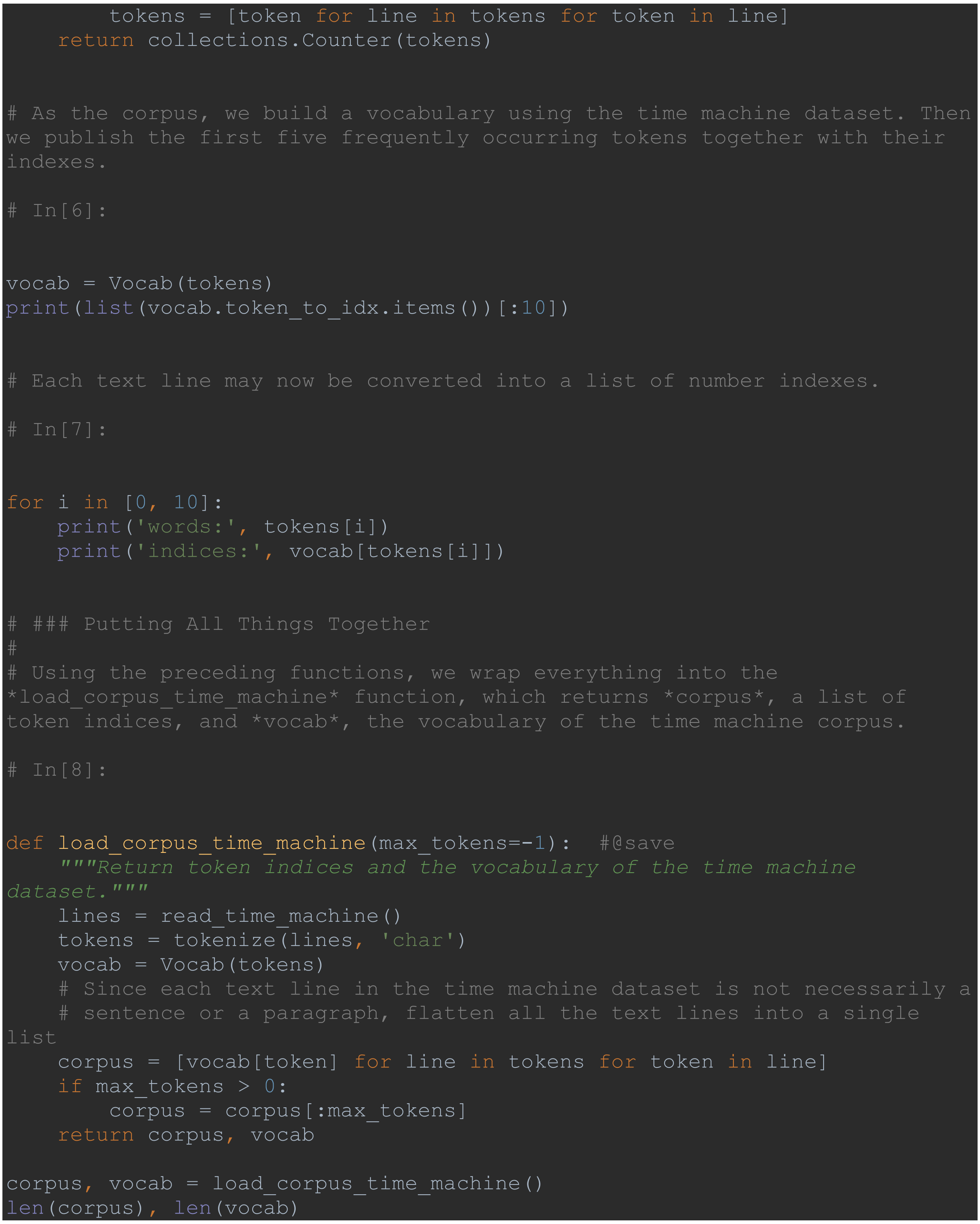
```
(170580, 28)
```

Full Python Programming Code

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


pip install keras
```

```
# In[2]:


import collections
import re
from d2l import tensorflow as d2l


# ### Reading the Dataset
#
# Text from H. G. Wells' **The Time Machine** is loaded. This is a somewhat
tiny corpus of little over 30000 words, but it suffices for the purposes of
what we want to demonstrate.

# In[3]:


#@save
d2l.DATA_HUB['time_machine'] = (d2l.DATA_URL + 'timemachine.txt',
                                '090b5e7e70c295757f55df93cb0a180b9691891a')

def read_time_machine():  #@save
    """Load the time machine dataset into a list of text lines."""
    with open(d2l.download('time_machine'), 'r') as f:
        lines = f.readlines()
    return [re.sub('[^A-Za-z]+', ' ', line).strip().lower() for line in
lines]

lines = read_time_machine()
print(f'# text lines: {len(lines)}')
print(lines[0])
print(lines[10])


# ### Tokenization
# The *tokenize* function below accepts a list (*lines*) as input, with
each element being a text sequence. Each text sequence is divided into
tokens. A token is the fundamental unit of text. Finally, a list of token
lists is returned, with each token being a string.

# In[4]:


def tokenize(lines, token='word'):  #@save
    """Split text lines into word or character tokens."""
    if token == 'word':
        return [line.split() for line in lines]
    elif token == 'char':
        return [list(line) for line in lines]
    else:
        print('ERROR: unknown token type: ' + token)

tokens = tokenize(lines)
for i in range(11):
    print(tokens[i])


# ### Vocabulary
#
# Let us now construct a dictionary, also known as a vocabulary, to map
string tokens into number indices beginning with 0. To do this, we first
```

```
count the unique tokens in all of the documents from the training set,
referred to as a corpus, and then give a numerical index to each unique
token based on its frequency. To decrease complexity, tokens that appear
only seldom are frequently eliminated. Any token that does not exist in the
corpus or has been deleted is mapped to a unique unknown token "< unk >".
We optionally provide a list of reserved tokens, such as "< pad >" for
padding, "< bos >" for presenting the start of a series, and "< eos >" for
presenting the end of a sequence.

# In[5]:


class Vocab:  #@save
    """Vocabulary for text."""
    def __init__(self, tokens=None, min_freq=0, reserved_tokens=None):
        if tokens is None:
            tokens = []
        if reserved_tokens is None:
            reserved_tokens = []
        # Sort according to frequencies
        counter = count_corpus(tokens)
        self._token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                   reverse=True)
        # The index for the unknown token is 0
        self.idx_to_token = ['<unk>'] + reserved_tokens
        self.token_to_idx = {token: idx
                             for idx, token in
enumerate(self.idx_to_token)}
        for token, freq in self._token_freqs:
            if freq < min_freq:
                break
            if token not in self.token_to_idx:
                self.idx_to_token.append(token)
                self.token_to_idx[token] = len(self.idx_to_token) - 1

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

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):
        if not isinstance(indices, (list, tuple)):
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]

    @property
    def unk(self):  # Index for the unknown token
        return 0

    @property
    def token_freqs(self):  # Index for the unknown token
        return self._token_freqs

def count_corpus(tokens):  #@save
    """Count token frequencies."""
    # Here `tokens` is a 1D list or 2D list
    if len(tokens) == 0 or isinstance(tokens[0], list):
        # Flatten a list of token lists into a list of tokens
```



```
        tokens = [token for line in tokens for token in line]
    return collections.Counter(tokens)


# As the corpus, we build a vocabulary using the time machine dataset. Then
we publish the first five frequently occurring tokens together with their
indexes.

# In[6]:


vocab = Vocab(tokens)
print(list(vocab.token_to_idx.items())[:10])


# Each text line may now be converted into a list of number indexes.

# In[7]:


for i in [0, 10]:
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])


# ### Putting All Things Together
#
# Using the preceding functions, we wrap everything into the
*load_corpus_time_machine* function, which returns *corpus*, a list of
token indices, and *vocab*, the vocabulary of the time machine corpus.

# In[8]:


def load_corpus_time_machine(max_tokens=-1):  #@save
    """Return token indices and the vocabulary of the time machine
dataset."""
    lines = read_time_machine()
    tokens = tokenize(lines, 'char')
    vocab = Vocab(tokens)
    # Since each text line in the time machine dataset is not necessarily a
    # sentence or a paragraph, flatten all the text lines into a single
list
    corpus = [vocab[token] for line in tokens for token in line]
    if max_tokens > 0:
        corpus = corpus[:max_tokens]
    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)
```

References

Zhang, A., Lipton, Z., Li, M., Smola, A., Werness, B., Hu, R., Zhang, S., & Tay, Y. (2022). *Dive into Deep Learning*. https://d2l.ai/index.html.