Table of Contents

In [ ]:

```
import os
import pandas as pd
import numpy as np

import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.models import Model
from keras.layers import Input, TextVectorization, Dense, Flatten, Embedding
```

NLP - Word representation

In [ ]:

```
texts = np.array(["I like chocolate",
            "I like tea",
            "You didn't know Paris",
            "You like chocolate",
            'You hate tea'])
labels = np.array([1,1,0,0,0])
```

BOW representation

We will use here the tools proposed by Keras in tensorflow.keras.preprocessing.text.

- The Tokenizer, separates a list of strings into tokens
- The `fit_on_sequences` function builds the vocabulary dictionary (i.e. assigns an index to each word).
- The `text_to_matrix` function builds the BOW representation

Binary BOW representation

In [ ]:

```
vocab_size = 10

tokenize = Tokenizer(num_words=vocab_size, char_level=False)
tokenize.fit_on_texts(texts) # Remember : you have to fit only on a train part
tokenize.texts_to_matrix(texts, mode='binary')
```

Out[ ]:

```
array([[0., 1., 0., 1., 1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 1., 0., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 1., 1., 1., 0.],
       [0., 1., 1., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 1., 0., 0., 0., 1.]])
```

Count BOW representation

In [ ]:

```
vocab_size = 10

tokenize = Tokenizer(num_words=vocab_size, char_level=False)
tokenize.fit_on_texts(texts) # Remember : you have to fit only on a train part
tokenize.texts_to_matrix(texts, mode='count')
```

Out[ ]:

```
array([[0., 1., 0., 1., 1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 1., 0., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 1., 1., 1., 0.],
       [0., 1., 1., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 1., 0., 0., 0., 1.]])
```

Frequence representation

In [ ]:
```
vocab_size = 10

tokenize = Tokenizer(num_words=vocab_size, char_level=False)
tokenize.fit_on_texts(texts) # Remember : you have to fit only on a train part
tokenize.texts_to_matrix(texts, mode='freq')
```

Out[ ]:
```
array([[0.        , 0.33333333, 0.        , 0.33333333, 0.33333333,
        0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.33333333, 0.        , 0.33333333, 0.        ,
        0.33333333, 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.25      , 0.        , 0.        ,
        0.        , 0.25      , 0.25      , 0.25      , 0.        ],
       [0.        , 0.33333333, 0.33333333, 0.        , 0.33333333,
        0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.33333333, 0.        , 0.        ,
        0.33333333, 0.        , 0.        , 0.        , 0.33333333]])
```

TfIDf representation

In [ ]:
```
vocab_size = 10

tokenize = Tokenizer(num_words=vocab_size, char_level=False)
tokenize.fit_on_texts(texts) # Remember : you have to fit only on a train part
tokenize.texts_to_matrix(texts, mode='tfidf')
```

Out[ ]:
```
array([[0.        , 0.81093022, 0.        , 0.98082925, 0.98082925,
        0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.81093022, 0.        , 0.98082925, 0.        ,
        0.98082925, 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.81093022, 0.        , 0.        ,
        0.        , 1.25276297, 1.25276297, 1.25276297, 0.        ],
       [0.        , 0.81093022, 0.81093022, 0.        , 0.98082925,
        0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.81093022, 0.        , 0.        ,
        0.98082925, 0.        , 0.        , 0.        , 1.25276297]])
```

MLP predictor

In [ ]:
```
# define the model
input_ = Input(shape=(vocab_size, ), name="input", dtype=tf.float32)
```

```
x = Dense(128, activation="relu", name="hidden")(input_)
output_ = Dense(1, activation='sigmoid', name="output")(x)
model = Model(input_, output_)
# summarize the model
model.summary()
```

```
Model: "model_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input (InputLayer)           [(None, 10)]              0         
_________________________________________________________________
hidden (Dense)               (None, 128)               1408      
_________________________________________________________________
output (Dense)               (None, 1)                 129       
=================================================================
Total params: 1,537
Trainable params: 1,537
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:

```
X = tokenize.texts_to_matrix(texts, mode='tfidf')
```

In [ ]:

```
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# fit the model
model.fit(X, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(X, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

```
Accuracy: 100.000000
```

Word embedding

Word Embedding is a technique for natural language processing. **The technique uses a neural network model** to learn word associations from a large corpus of text. Once trained, such a model can detect synonymous words or suggest additional words for a partial sentence. As the name implies, **Word Embedding represents each distinct word with a particular vector**. The vectors are chosen carefully such that a simple mathematical function (the cosine similarity between the vectors) indicates the level of semantic similarity between the words represented by those vectors.

The underlying idea here is that similar words will have a minimum distance between their vectors. For example fruits like apple, mango, banana should be placed close whereas books will be far away from these words. In a broader sense, word embedding will create the vector of fruits which will be placed far away from vector representation of books.

Somes techniques used to create embeddings:

- Word2vec (Google) - 2 techniques: Continuous Bag of Words (CBoW) and Skip-Gram;
- Global Vectors or GloVe (Stanford);
- fastText (Facebook) — interesting fact: accounts for out of vocabulary words.

It is also easy to build your own embedding:

- Using the Keras Embedding layer. The embedding will be adjusted to your dataset and does not require a specific corpus
- Using Gensim. The embedding will be adjusted to your corpus using one of the above techniques: word2vec or fastText

Keras embedding

In [ ]:

```
# Constants
vocab_size = 5000  # Maximum vocab size.
max_len = 10  # Sequence length to pad the outputs to.

# Create the layer.
vectorize_layer = TextVectorization(max_tokens=vocab_size,
                                    output_mode='int',
                                    output_sequence_length=max_len)
```

```
2022-01-21 15:34:07.497218: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized
with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operat
ions:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

In [ ]:

```
# Now that the vocab layer has been created, call `adapt` on the text-only
# dataset to create the vocabulary. You don't have to batch, but for large
# datasets this means we're not keeping spare copies of the dataset.
```

```
vectorize_layer.adapt(texts)
# You have to fit the vectorize_layer only on train set.
```

```
2022-01-21 15:34:08.056246: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimiza
tion Passes are enabled (registered 2)
```

In [ ]:

```
vectorize_layer(texts)
```

Out[ ]:

```
<tf.Tensor: shape=(5, 10), dtype=int64, numpy=
array([[ 5,  3,  6,  0,  0,  0,  0,  0,  0,  0],
       [ 5,  3,  4,  0,  0,  0,  0,  0,  0,  0],
       [ 2, 10,  8,  7,  0,  0,  0,  0,  0,  0],
       [ 2,  3,  6,  0,  0,  0,  0,  0,  0,  0],
       [ 2,  9,  4,  0,  0,  0,  0,  0,  0,  0]])>
```

In [ ]:

```
vectorize_layer.get_vocabulary()
```

Out[ ]:

```
['',
 '[UNK]',
 'you',
 'like',
 'tea',
 'i',
 'chocolate',
 'paris',
 'know',
 'hate',
 'didnt']
```

In [ ]:

```
vectorize_layer.get_config()
```

Out[ ]:
```
{'name': 'text_vectorization',
 'trainable': True,
 'batch_input_shape': (None,),
 'dtype': 'string',
 'max_tokens': 5000,
 'standardize': 'lower_and_strip_punctuation',
 'split': 'whitespace',
 'ngrams': None,
 'output_mode': 'int',
 'output_sequence_length': 10,
 'pad_to_max_tokens': False}
```

In [ ]:
```
# define the model
input_ = Input(shape=(1,), dtype=tf.string)
x = vectorize_layer(input_)
x = Embedding(vocab_size, 256, name="Embedding")(x)
x = Flatten()(x)
output_ = Dense(1, activation='sigmoid')(x)
model = Model(input_, output_)
# summarize the model
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         [(None, 1)]               0         
_________________________________________________________________
text_vectorization (TextVect (None, 10)                0         
_________________________________________________________________
Embedding (Embedding)        (None, 10, 256)           1280000   
_________________________________________________________________
flatten (Flatten)            (None, 2560)              0         
_________________________________________________________________
dense (Dense)                (None, 1)                 2561      
=================================================================
Total params: 1,282,561
Trainable params: 1,282,561
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
# compile the model
```

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# fit the model
model.fit(texts, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(texts, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

```
Accuracy: 100.000000
```

Glove/Word2Vec/FastText embedding

Traditional word embedding techniques (Glove/Word2Vec/FastText) learn a global word embedding. They first build a global vocabulary using unique words in the documents by ignoring the meaning of words in different context. Then, similar representations are learnt for the words appeared more frequently close each other in the documents. The problem is that in such word representations the words' contextual meaning (the meaning derived from the words' surroundings), is ignored. For example, only one representation is learnt for "left" in sentence "I left my phone on the left side of the table." However, "left" has two different meanings in the sentence, and needs to have two different representations in the embedding space.

For example, consider the two sentences:

1. I will show you a valid point of reference and talk to the point.
2. Where have you placed the point.

The word embeddings from a pre-trained embeddings such as word2vec, the embeddings for the word 'point' is same for both of its occurrences in example 1 and also the same for the word 'point' in example 2. (all three occurrences has same embeddings).

In [ ]:

```
# Same steps as Keras Embedding
vocab_size = 5000  # Maximum vocab size.
max_len = 10  # Sequence length to pad the outputs to.
vectorizer = TextVectorization(max_tokens=vocab_size, output_sequence_length=max_len)
vectorizer.adapt(texts)
texts_vec = vectorizer(texts)
```

In [ ]:

```
# Build word dict
voc = vectorizer.get_vocabulary()
word_index = dict(zip(voc, range(len(voc))))
word_index
```

Out[ ]:
```
{'': 0,
 '[UNK]': 1,
 'you': 2,
 'like': 3,
 'tea': 4,
 'i': 5,
 'chocolate': 6,
 'paris': 7,
 'know': 8,
 'hate': 9,
 'didnt': 10}
```

In [ ]:
```
# If necessary, download glove matrix
#!wget http://nlp.stanford.edu/data/glove.6B.zip
#!unzip -q glove.6B.zip
```

In [ ]:
```
# Make a dict mapping words (strings) to their NumPy vector representation:
path_to_glove_file = "/users/riveill/DS-models/glove.6B.50d.txt"

embeddings_index = {}
with open(path_to_glove_file) as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

print("Found %s word vectors." % len(embeddings_index))
```

```
Found 400000 word vectors.
```

Let's prepare a corresponding embedding matrix that we can use in a Keras Embedding layer. It's a simple NumPy matrix where entry at index i is the pre-trained vector for the word of index i in our vectorizer's vocabulary.

In [ ]:
```
num_tokens = len(voc) + 2
embedding_dim = 50
hits = 0
misses = 0

# Prepare embedding matrix
embedding_matrix = np.zeros((num_tokens, embedding_dim))
for word, i in word_index.items():
```

```
    print(word, i)
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # Words not found in embedding index will be all-zeros.
        # This includes the representation for "padding" and "OOV"
        embedding_matrix[i] = embedding_vector
        hits += 1
    else:
        misses += 1
print("Converted %d words (%d misses)" % (hits, misses))
```

```
 0
[UNK] 1
you 2
like 3
tea 4
i 5
chocolate 6
paris 7
know 8
hate 9
didnt 10
Converted 9 words (2 misses)
```

In [ ]:

```
embedding_layer = Embedding(
    num_tokens,
    embedding_dim,
    embeddings_initializer=tf.keras.initializers.Constant(embedding_matrix),
    trainable=False,
)
```

In [ ]:

```
# define the model
input_ = Input(shape=(max_len,), dtype=tf.int32)
x = embedding_layer(input_)
x = Flatten()(x)
output_ = Dense(1, activation='sigmoid')(x)
model = Model(input_, output_)
# summarize the model
model.summary()
```

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 10)]              0
_________________________________________________________________
embedding (Embedding)        (None, 10, 50)            650
_________________________________________________________________
flatten_1 (Flatten)          (None, 500)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 501
=================================================================
Total params: 1,151
Trainable params: 501
Non-trainable params: 650
_________________________________________________________________
```

In [ ]:

```
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# fit the model
model.fit(texts_vec, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(texts_vec, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

```
Accuracy: 100.000000
```

Train your own model with gensim

In [ ]:

```
#!pip install gensim
```

```
Requirement already satisfied: gensim in /Users/riveill/opt/miniconda3/lib/python3.9/site-packages (4.1.2)
Requirement already satisfied: numpy>=1.17.0 in /Users/riveill/opt/miniconda3/lib/python3.9/site-packages (from gensi
m) (1.19.5)
Requirement already satisfied: smart-open>=1.8.1 in /Users/riveill/opt/miniconda3/lib/python3.9/site-packages (from g
ensim) (5.2.1)
Requirement already satisfied: scipy>=0.18.1 in /Users/riveill/opt/miniconda3/lib/python3.9/site-packages (from gensi
m) (1.7.1)
```

To begin with, you need data to train a model. We will call this data set "corpus ».

Even if the corpus is small enough to fit entirely in memory, we thus implement a user-friendly iterator that reads the corpus line by line to demonstrate how one could work with a larger corpus.

In [ ]:
```
# Avalaible in your gensim installation
corpus_path="/Users/riveill/opt/miniconda3/lib/python3.9/site-packages/gensim/test/test_data/"
corpus="lee_background.cor"
```

In [ ]:
```
from gensim.test.utils import datapath
from gensim import utils

class MyCorpus:
    """An iterator that yields sentences (lists of str)."""

    def __iter__(self):
        for line in open(corpus_path+corpus):
            # assume there's one document per line, tokens separated by whitespace
            yield utils.simple_preprocess(line)
```

Let's go ahead and train a model on our corpus. Don't worry about the training parameters much for now, we'll revisit them later.

In [ ]:
```
import gensim.models

sentences = MyCorpus()
model = gensim.models.Word2Vec(sentences=sentences, vector_size=150)
```

Once we have our model, we can use it.

The main part of the model is model.wv\ , where "wv" stands for "word vectors".

In [ ]:
```
vec_king = model.wv['king']
```

Training non-trivial models can take time. Once the model is built, it can be saved using standard gensim methods:

In [ ]:
```
import tempfile

with tempfile.NamedTemporaryFile(prefix='gensim-model-', delete=False) as tmp:
    temporary_filepath = tmp.name
```

```
    print(temporary_filepath)
    model.save(temporary_filepath)
    #
    # The model is now safely stored in the filepath.
    # You can copy it to other machines, share it with others, etc.
    #
    # To load a saved model:
    #
    new_model = gensim.models.Word2Vec.load(temporary_filepath)
```

```
/var/folders/l4/ptzjj6dn0_x5trgl90dc98g40000gn/T/gensim-model-__mtorja
```

You can then use the template exactly as if it were a Glove/Word2Vec/FastText template retrieved from the Internet.

In [ ]:

```
# Same steps as Keras Embedding
vocab_size = 5000  # Maximum vocab size.
max_len = 10  # Sequence length to pad the outputs to.
vectorizer = TextVectorization(max_tokens=vocab_size, output_sequence_length=max_len)
vectorizer.adapt(texts)
texts_vec = vectorizer(texts)
```

In [ ]:

```
# Build word dict
voc = vectorizer.get_vocabulary()
word_index = dict(zip(voc, range(len(voc))))
```

In [ ]:

```
# Load gensim model
new_model = gensim.models.Word2Vec.load(temporary_filepath)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/var/folders/l4/ptzjj6dn0_x5trgl90dc98g40000gn/T/ipykernel_42288/2646765101.py in <module>
      1 # Load gensim model
----> 2 new_model = gensim.models.Word2Vec.load(temporary_filepath)

NameError: name 'gensim' is not defined
```

In [ ]:

```
num_tokens = len(voc) + 2
embedding_dim = 150
hits = 0
```

```
misses = 0

# Prepare embedding matrix
embedding_matrix = np.zeros((num_tokens, embedding_dim))
for word, i in word_index.items():
    try:
        model.wv[word]
        # Words not found in embedding index will be all-zeros.
        # This includes the representation for "padding" and "OOV"
        embedding_matrix[i] = model.wv[word]
        hits += 1
    except :
        misses += 1
print("Converted %d words (%d misses)" % (hits, misses))
```

```
Converted 4 words (7 misses)
```

In [ ]:

```
embedding_layer = Embedding(
    num_tokens,
    embedding_dim,
    embeddings_initializer=tf.keras.initializers.Constant(embedding_matrix),
    trainable=False,
)
```

In [ ]:

```
# define the model
input_ = Input(shape=(max_len,), dtype=tf.int32)
x = embedding_layer(input_)
x = Flatten()(x)
output_ = Dense(1, activation='sigmoid')(x)
model = Model(input_, output_)
# summarize the model
model.summary()
```

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 10)]              0
_________________________________________________________________
embedding_1 (Embedding)      (None, 10, 150)           1950
_________________________________________________________________
flatten_2 (Flatten)          (None, 1500)              0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 1501
=================================================================
Total params: 3,451
Trainable params: 1,501
Non-trainable params: 1,950
_________________________________________________________________
```

In [ ]:

```
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# fit the model
model.fit(texts_vec, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(texts_vec, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

```
Accuracy: 100.000000
```

Contextual word embedding

However, **contextual embeddings** (are generally obtained from the transformer based models: BERT, ELMO, GPT3). The emeddings are obtained from a model by passing the entire sentence to the pre-trained model. Note that, here there is a vocabulary of words, but the vocabulary will not contain the contextual embeddings. The embeddings generated for each word depends on the other words in a given sentence. (The other words in a given sentence is referred as context. The transformer based models work on attention mechanism, and attention is a way to look at the relation between a word with its neighbors). Thus, given a word, it will not have a static embeddings, but the embeddings are dynamically generated from pre-trained (or fine-tuned) model.

For example, consider the two sentences:

1. I will show you a valid point of reference and talk to the point.

2. Where have you placed the point.

The embeddings from BERT or ELMO or any such transformer based models, the the two occurrences of the word 'point' in example 1 will have different embeddings. Also, the word 'point' occurring in example 2 will have different embeddings than the ones in example 1.

We won't be seeing the transformers this year, but if you want to see how it works here is a great presentation

In [ ]:

```

```