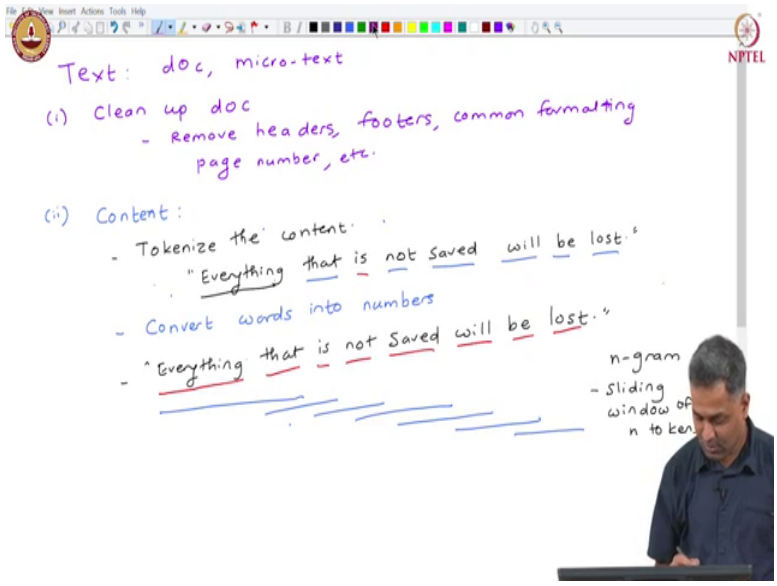
Practical Machine Learning
Dr. Ashish Tendulkar
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

Lecture - 15
Text Processing with TensorFlow

(Refer Slide Time: 00:16)



We get the text which could be a text document or a micro text which is a small text of few words. So, typically the first step that we do is if we are getting text in a document we generally try to clean up the document in the cleanup process we remove headers, footers and any other common formatting like page numbers etc. After cleaning up the document you are left with the main content of the document.

We try to first tokenize the content. So, by tokenizing what we mean is we want to break the string across the white spaces or tabs. So the first operation is tokenization.

After tokenization we get individual tokens. So, this tokens are words and we know that machine learning algorithms cannot work with words. So, we have to convert these words into numbers. So, we need to also see how to convert these words into numbers and there are few schemes that are implemented in TensorFlow text package. So, second

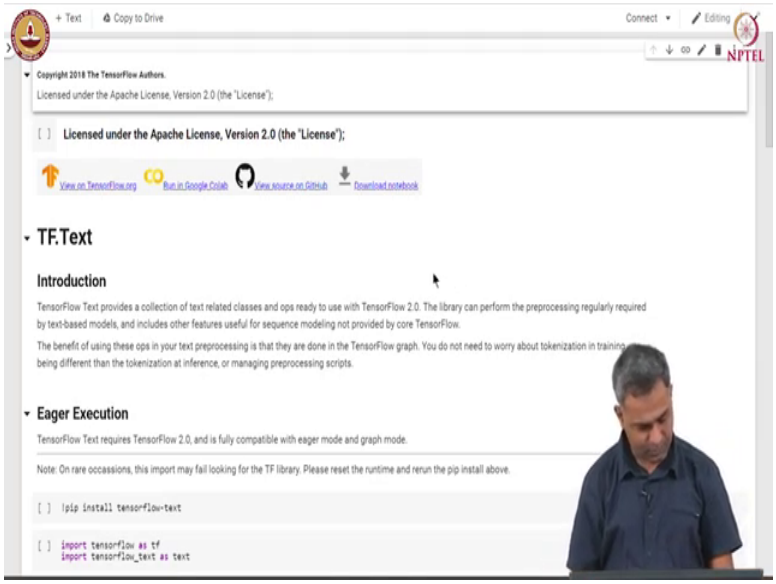
is convert words into numbers. Additionally we might be interested in getting bigrams or in general n-grams from the token from the string.

So, the tokens are special case of n-gram tokens are 1-grams. So, let us take an example here we have everything that is not saved will be lost as a string and in order to get n-gram we define first a sliding window of *n* tokens. So, for bigrams for let us say unigrams or 1-grams we have a sliding window of one token and for each instance of a sliding window we record the token and then slide the window by one token.

For bigram we have a sliding window of size 2 which we keep sliding one token at a time to get bigrams. Then we have for trigrams we will define a sliding window of size 3 and so on. So, these are some of the preprocessing steps that we undertake for any of the text content.

In addition to that we use embedding to represent the tokens with numbers. Besides embedding there are other methods like one hot encoding or a numeric representation based on mapping of the token to the string can also be used.

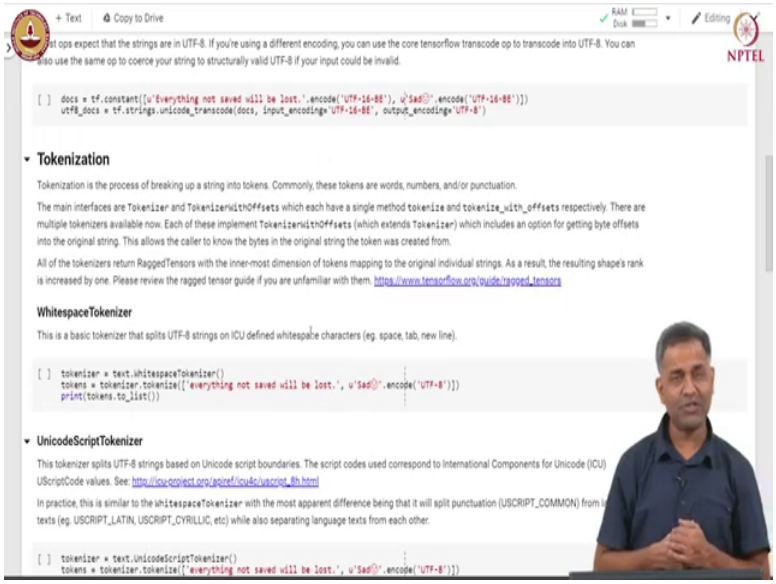
(Refer Slide Time: 06:29)



Let us explore tf.text library provided by TensorFlow 2.0. The tf.text library provides a collection of classes related to text and operations that can be used readily with TensorFlow 2.0. The library can perform preprocessing that is regularly required for text

based models and it includes other features for sequence modeling not provided by core TensorFlow.

(Refer Slide Time: 07:14)



Most of the operation expect the strings are in UTF-8. If you are using different encoding we can use transcode operation to transcode into UTF-8. We studied transcode operation while back while exploring how to handle unicode characters in TensorFlow. Let us take a string and encode it to UTF-16 and then we will use unicode transcode to convert from UTF-16 to UTF-8. Given the line of text the first operation is tokenization. Tokenization is a process of breaking up a strings into tokens.

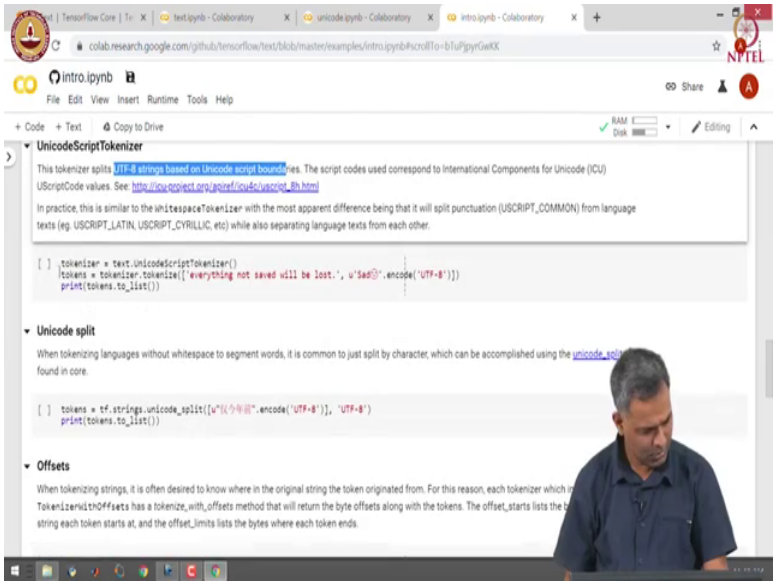
Commonly these tokens are words numbers and punctuations. The main interfaces are tokenizer and tokenizer with offset which each have a single method called tokenize and tokenize with offsets respectively. There are multiple tokenizers available. Each of these implement tokenizer with offsets which includes an option for getting byte offset into the original string this allows the caller to know the bytes in the original string the token was created from and this can also be used for variety of downstream analysis.

All of the tokenizers return ragged tensors with the innermost dimension of token mapping to the original individual strings. As a result the resulting shapes rank is increased by one. Let us look at the basic tokenizer which is whitespace tokenizer. It

splits UTF-8 strings on icu defined white space characters such as space, tabs or new line. Let us see what kind of tokens we get with whitespace tokenizer on some of these examples where we have one string which is in English other string which is which has English word followed by an emoji and both these strings are encoded in UTF-8 and we apply tokenize operation from the whitespace tokenizer. Let us print the list of tokens with tokens.to_list.

We can see that after we apply tokenizer the first string got tokenized into words. The last token also includes punctuation which is ".". So, we can see that the first string got tokenized into ['everything', 'not', 'saved', 'will', 'be', 'lost']. The second string got tokenized into a single token. There are 6 tokens in the first string and a single token in the second string. We can use more sophisticated unicode script tokenizer that splits UTF-8 strings based on unicode script boundaries.

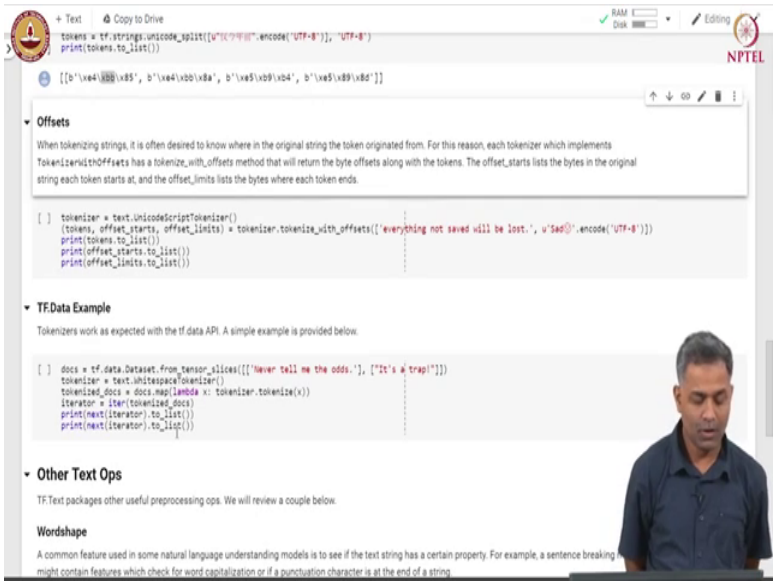
(Refer Slide Time: 11:28)



So, let us look at how the strings get tokenize with unicode script tokenizer. So, if you can compare these 2 outputs you will see that the first string has now an additional token corresponding to the "." where as the second string got tokenized into 2 tokens one corresponding to "sad" and second corresponding to the emoji. When tokenizing languages without whitespace to segment words it is common to just split by characters which can be accomplished using unicode_split operation found in the TensorFlow core.

So, let us try to split this Chinese string encoded in to UTF-8. You can see that the string got segmented into 4 tokens. Each token corresponds to a single character in the original string.

(Refer Slide Time: 12:59)



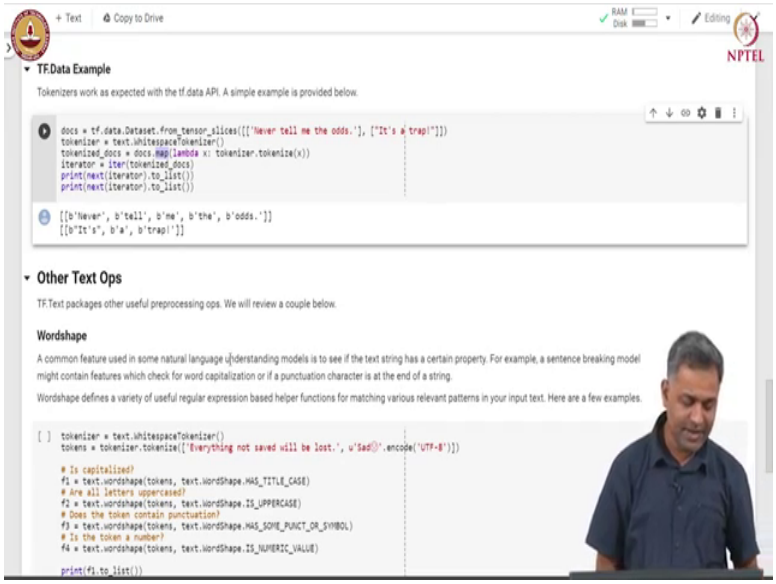
When tokenizing strings it is often desired to know where in the original string the token comes from. For this reason each tokenizer which implements tokenizer with offsets has tokenizer_with_offsets() method that returns the byte offset along with each of the token. You can see that tokenize with offset returns tokens offset_starts and offset_limits. Offset_starts lists offset starts lists the bytes in the original string where the token starts.

Offset_limits lists the bytes where each token ends. We can see that in "everything not saved will be lost" string “everything” started with byte 0 and ended at tenth byte and so on. So, this way we can keep track of whether token started and ended in the original string. We can use tokenizers with tf.data API. Here we create a data set called docs from tensor slices with 2 strings “Never tell me the odds.” and “It’s a trap!”. We use white space tokenizer on each string and we do that with a map method.

So, in map we apply the tokenize operation on each of the strings. Let us look at the tokenized version of each of the strings in the data set. The tokenizer works as expected

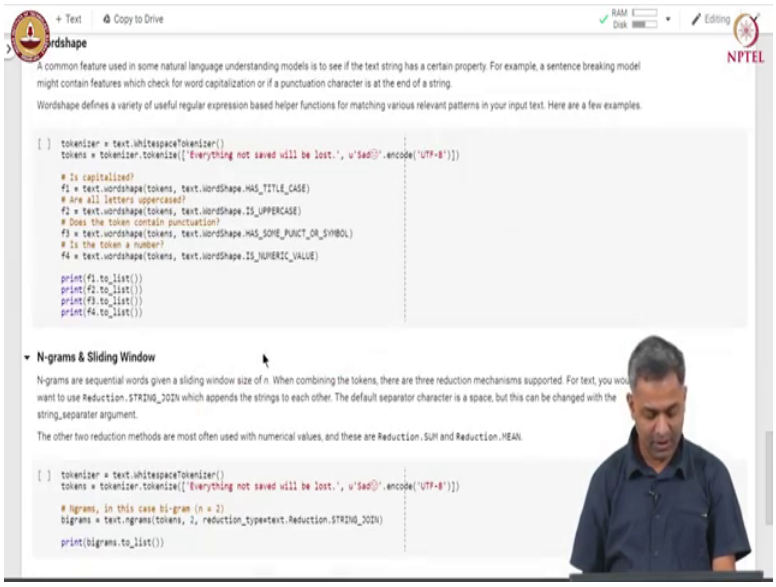
just like it was working on the strings which were not included in the data set. After including the strings in the data set and if you tokenize you also get the same result.

(Refer Slide Time: 15:45)



There are some other text operations that are also implemented in tf.text package.

(Refer Slide Time: 16:03)



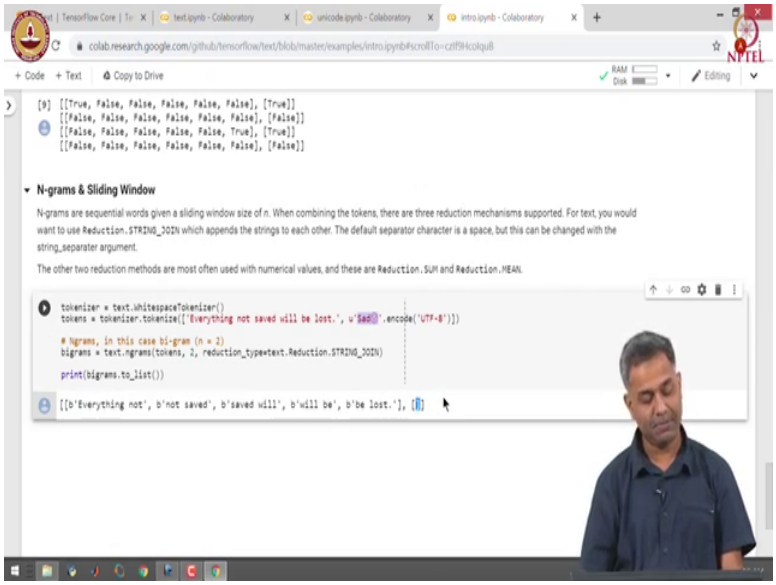
One of the commonly used feature is a word shape where you are interested in checking if the string has a particular property. As an example you want to know if the string starts with a capital letter or a string has all the uppercase letters or whether it has some

punctuations or symbols. So, we can tokenize the string and assess the word shape of each of the tokens.

We can that after tokenizing this particular string and let us say after applying has title case word shape. We can see that this particular word shape is true for the first token which is everything and it is false everywhere else in the first string.

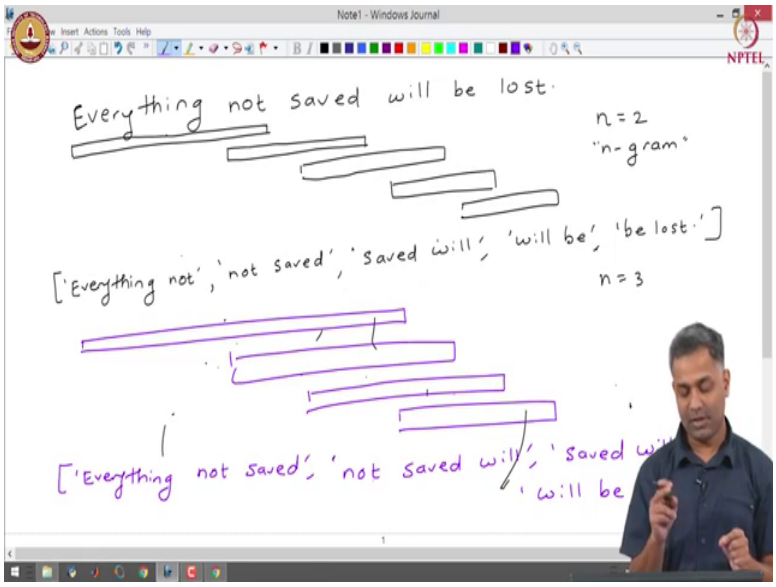
In the second string sad followed by emoji it is true because it starts with a title case. None of the strings none of the tokens in 2 strings are uppercase that is why we have false everywhere. We can see that the last token “lost.” with a punctuation mark and Sad emoji has some punctuations or symbol that is why they are *true*. Another token is a number that is why is numeric value is *false*.

(Refer Slide Time: 17:46)



Finally we want to sometimes convert the strings into n-grams. These n-grams could be bigrams or trigrams and the way we construct that is by defining a sliding window of the specific size.

(Refer Slide Time: 18:13)



To give you an example let us say so "Everything not saved will be lost." is let us say an example string and let us say if you want to construct bigrams. So, since you are interested in constructing bigrams n is equal to 2 in the context of n-gram. So, we define a sliding window of size 2 to begin with position the window at the first token and we record the bigram we position a window containing 2 tokens and position it at the first token and we record all the tokens that are in the window.

So, the first instance we get “everything” not as a bigram then what we do is we slide the window by one token and position it to the next token. So, in this case you slide it by one. So, now, the window is positioned at the second token and we note down the bigrams that we get. So, here we have “not saved” as a second bigram then we again slide the window and record.

So, we have saved will then we have will be the next bigram and finally, we have be lost as the last bigram. So, we get 5 bigrams from the string. If you put n=3 we need to construct sliding window of size 3 and records the word and record the trigrams or 3-rams. So, here the 3 grams will be “everything not saved” , “not saved will”, “saved will be” and “will be lost.”. So, there are 4 trigrams each correspond to the position of a sliding window of size 3 across every token. So, you can see the text.ngram where we set n=2 and we get bigram by using the reduction type of string join.

So, we got the bigram as we listed in we got the bigram same as the once that we worked out and incase of the second string “sad” it is a single token string. So, there are no bigrams. It is empty.