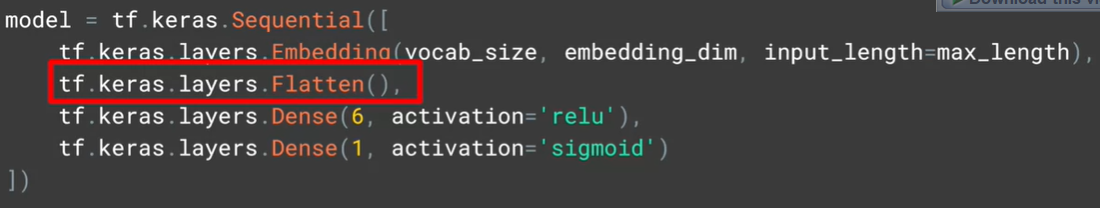
## Usage of Embeddings Layer in Keras :

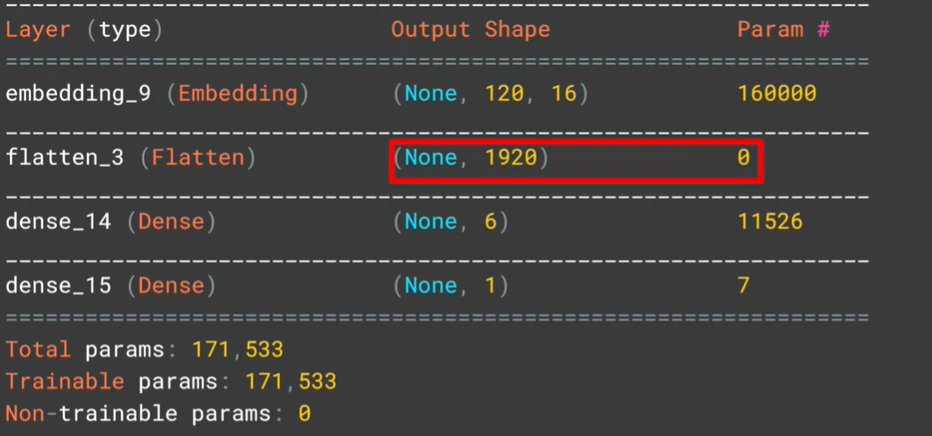
#### Quick reminder about word embeddings:

* Instead of the classic tokenization of the words by a random meaningless number , each word will be encoded by a vector of N-dimension , This vector will present the semantic of the word in a way of the similar words will have a close projection in the plan to that word ( for example , words like “Sad” , “horrible” and “awful” will have similar vectors )

#### Keras Embedding Layer :

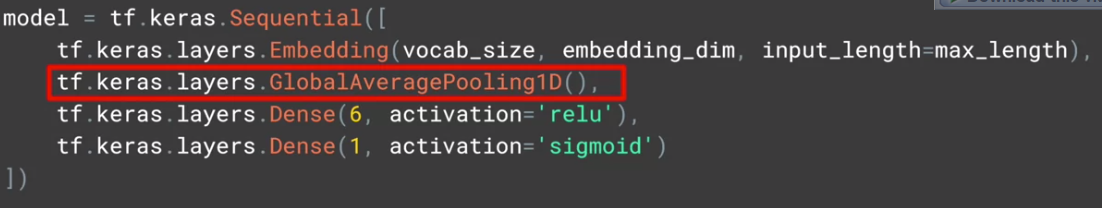


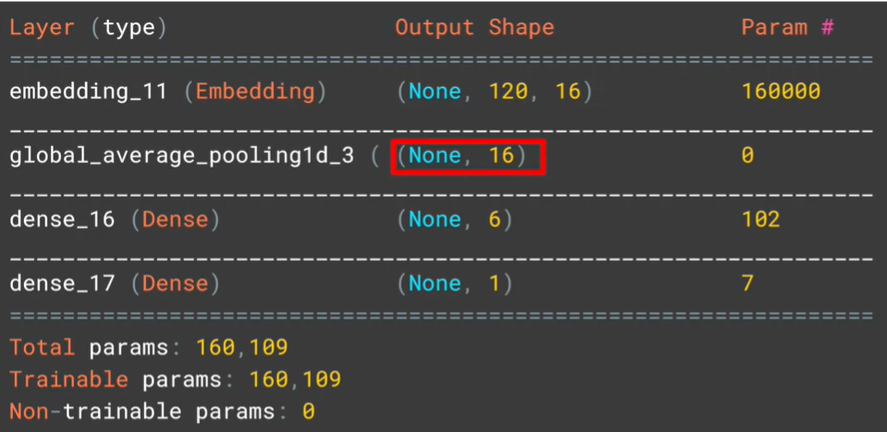
* The role of the Embedding layer is to to give for the words appearing in the comments of the same labels: a close representations , so for words like “ joyful” , “amazing” , “wonderful” : they all appeared in positiove comments labeled by 1, the Embedding layer detects that and do the necessary work to know that they have a similar semantics
* The first argument is the vocabulary\_size : how much distinct words are tokenized
* The second argument is the dimension of the embedding vector of each word
* The input\_length arguments specify the number of words in a sequence (which is the length of the maxsimum sequence after padding the shorten ones)
* The role of the Flatten Array is to pass from 2D array (embedding dim \* input\_length ) to 1D flat vector with embedding dim\*input\_length elements , the image below shows the impact of Flatten layer



* The last layer has only one output with sigmoid as an activation function because we are dealing with a Binary classification problem ( given a IMDB comment , guess if it’s a negative or a positive comment )

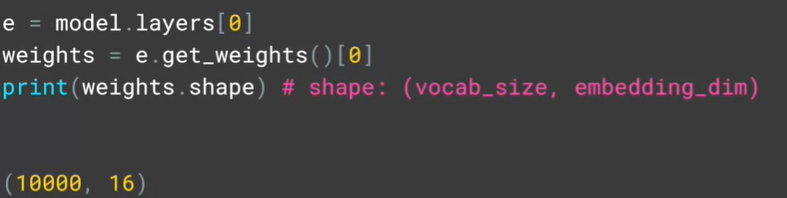
#### GlobalAveragePooling1D as an alternative to Flatten Layer:





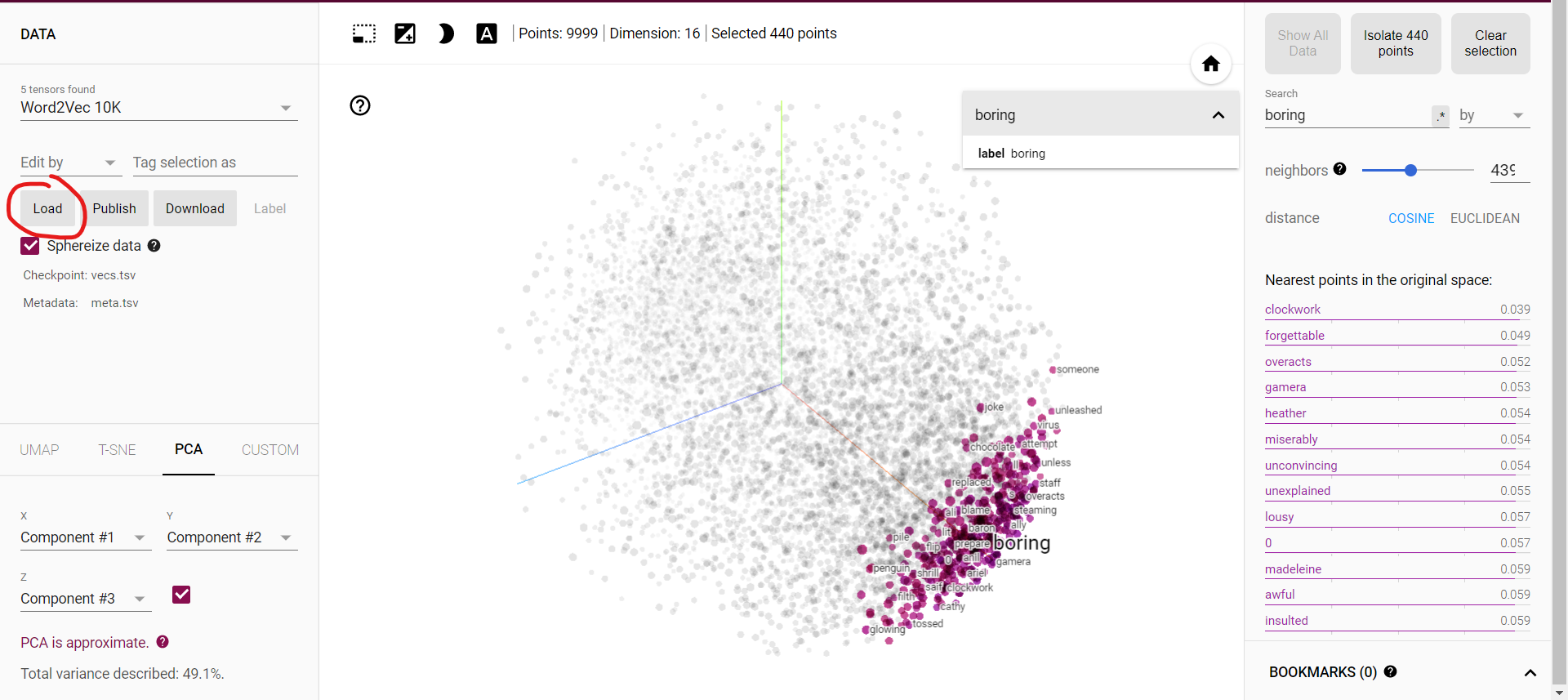
* Instead of just flattening the 2D array , GlobalAveragePooling1D() will return a 1D vector with a dimensions equals to the last dimension of the input ( which is input\_length=16 )
* For each word , he will take the average of its embedding vector ( or generally , doing the average based on the last axis : 16 )
* The number of parameters has a significant decrease compared to the first model ( which had the Flatten layer instead )

#### To check the shape of the first Layer : Embedding Layer :



## Tensorflow Embedding projector

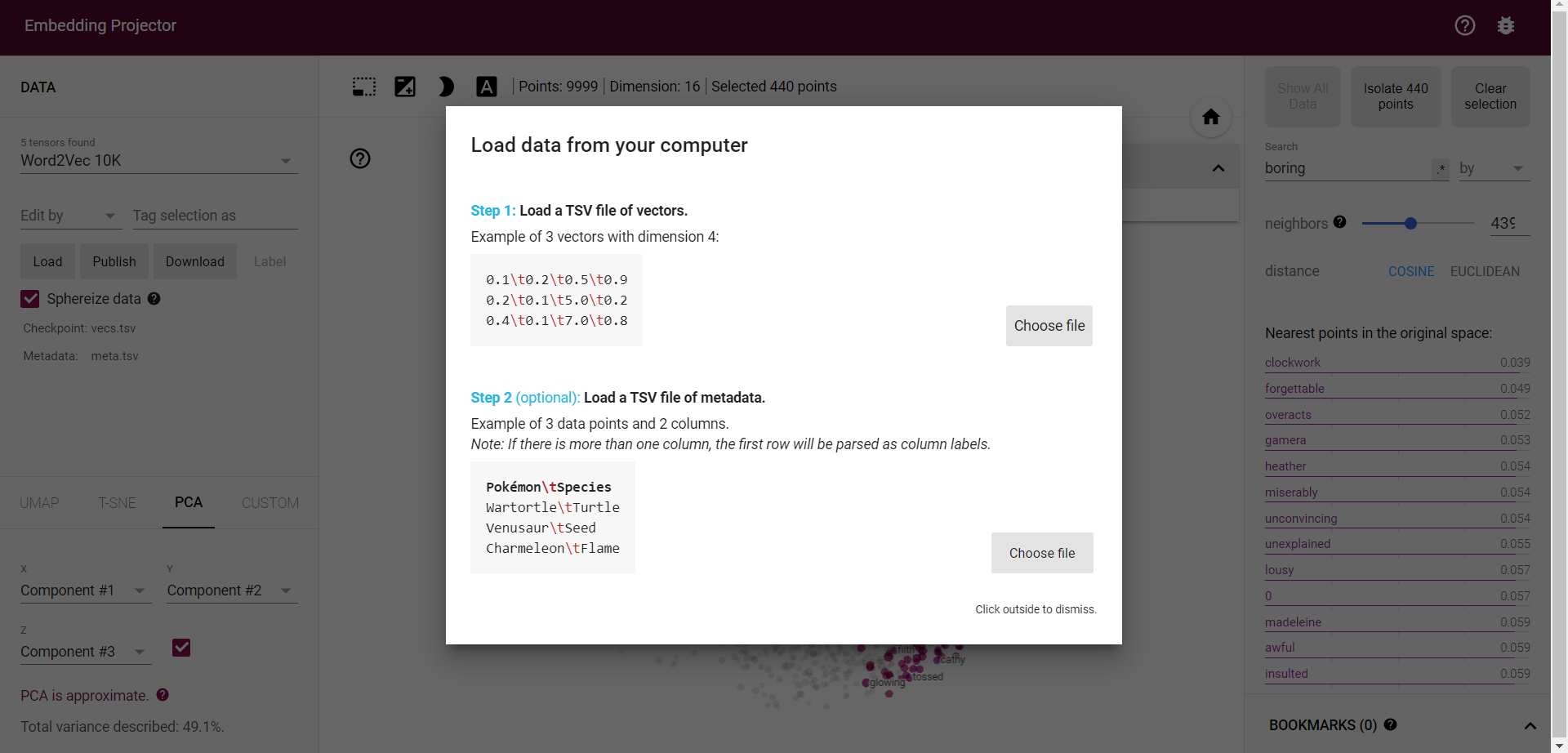
* TensorFlow had an interesting website : [https://projector.tensorflow.org/](https://projector.tensorflow.org/%20) which provides you a 2D visualization of the generated word embeddings vector by the Embedding layer by loading meta and vector files I should generate from the Embedding Layer



* We can see our embedding layer give a close presentation to words “boring” , “forgettable” , “miserably” and “awful” which is pretty good !

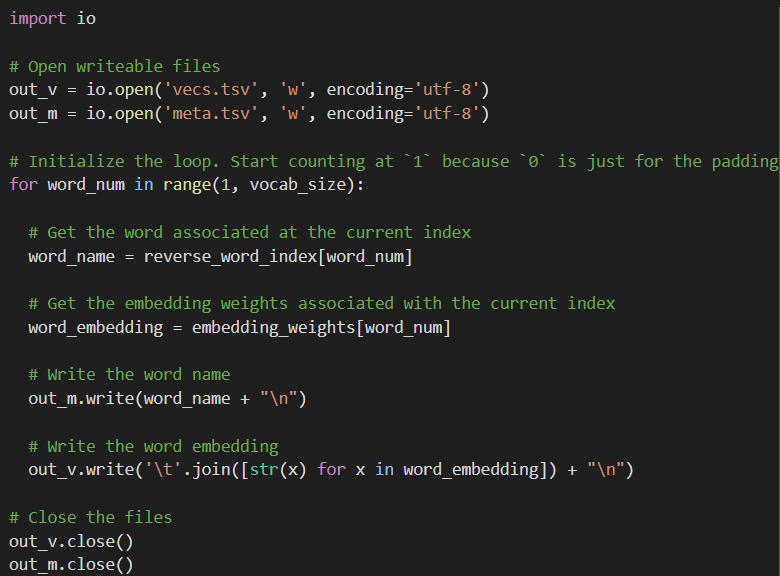
#### Loading vector.tsv and meta.tsv files in the website :

* By clicking in the load button I should get this dialog :

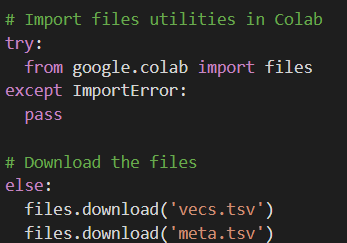


* From the dialog we see that the TSV file of vectors is a text file , each row represents a word embedding vector , its components are separated by /t
* TSV file of meta data contains is a text file , each row contains the vector name

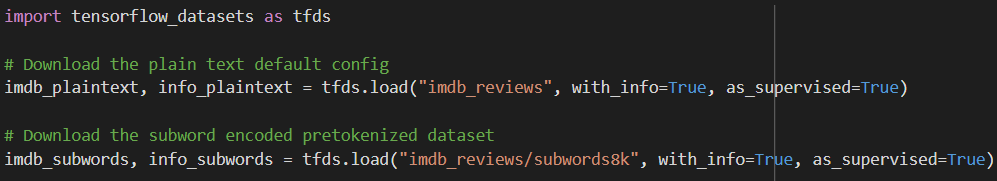
#### How to generate those two files in Python :



* In Collab I should call the cell below to download the files locally



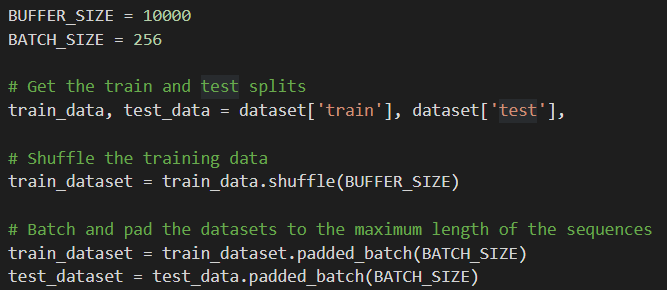
## Prepackaged Datasets and the pre-tokenization

* Tensorflow has a prepackaged Datasets called “Tensorflow DataSets” : tfds
* This kind of prepackaged datasets usually the word are pre-tokenized , that means we will have already the tokenization of the sentences in the dataset in a particular variable :
* In the code above we see that the path “imdb\_reviews/subwords8k” contains the data pretokenized
* Subwords8K means that the technique used is Sub-Word Tokenization with vocab\_size=8K=8000

## Padded\_batch() an interesting method :

In tfds library, we have an interesting method called padded\_batch() :

* Its role is to Batch and pad the datasets to the maximum length of the sequences



The more the BATCH\_SIZE Is big , the more the training will be quicker ( per epoch )

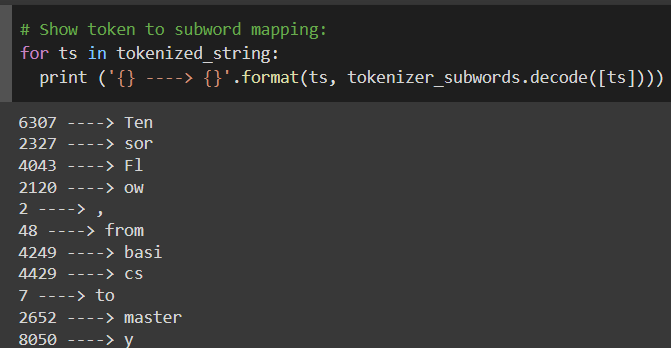
## Sub-Word Tokenization:

* Subword-based tokenization is a solution between word and character-based tokenization.
* It’s born to solve the character-based tokenization issues ( very long sequences and meaningless tokens ) and the word-based tokenization ( a high risk to get <OOV> token due to the difficulty to cover all the vocabulary words )
* For binary classifiers, this might not have a big impact but you may have other applications that will benefit from avoiding OOV tokens when training the model (e.g. text generation). If you want the tokenizer above to not have OOVs, then the vocab\_size will increase to more than 88k. This can slow down training and bloat the model size. The encoder also won't be robust when used on other datasets which may contain new words, thus resulting in OOVs again.

#### Description of the technique used:

* It keeps the most-frequently used words, but it splits the rare words and the composed ones into smaller meaningful words.
  + As Example: we keep “boy” but we split “boys” into “boy” and “s” , This helps the model learn that the word “boys” is formed using the word “boy” with slightly different meanings but the same root word.
  + The word “tokenization” for example will be splitted into “token” and “ization”
* As we may guess , the length of the tokenized sentence will be longer than the original size of the sentence ( because word = token or more than one token )
  + A concrete example :
    - Comparison about word and sub-word tokenization of the sentence **“TensorFlow, from basics to mastery**” :
      * Word-tokenization:



* We got <OOV> in the place of Tensorflow , basics and mastery because our Tokenizer didn’t meet those words in the training set
  + - * Sub-Word Tokenziation :
* Thanks to splitting the rare words : we can tokenize and decode the whole sentence without getting <OOV> tokens

#### Conclusion about Sub-Word tokenization:

* We saw how subworld text encoding can be a robust technique to avoid out-of-vocabulary tokens. It can decode uncommon words it hasn't seen before even with a relatively small vocab size. Consequently, it results in longer token sequences when compared to full word tokenization