**Intro to Text Processing by Predicting Sentiments of Tweets using Azure Machine Learning Studio and R**

In this tutorial we will be building a sentiment classification model to predict, from the words used in a given tweet, whether the tweet will be positive or negative.

**What is sentiment?**

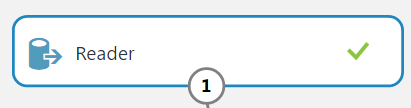
Sentiment in respect text is the emotional character, tone, or “feel” of the given message or text. For example, was this an angry Tweet? Was it happy, sad, solemn, or slanderous? Now defining the exact rules for sentiment is not so easy and in the supervised learning context, it will be humans who have to read the text and classify as such.

Since this is an introductory tutorial, we will be building a binary classification model for sentiment. Wherein a tweet can only be “positive” or “negative” sentiment with no option of any other sentiment such as “neutral”. As such this model may not be practical for deployment in real world applications, but incremental development states that we all must start from a firm foundation.

**The Dataset**

We will be using a public dataset with 160,000 pre-classified tweets. There are two columns in the dataset; the text of the tweet and the text’s corresponding sentiment score with “0” being negative and “4” being positive. The classification itself was scored by human judges (my guess is an army of linguistic grad students) where a judge would read the tweet and give it a score between 0 and 4; with 4 being highly positive and 0 being highly negative. This is a sampled down dataset where “1”, “2”, and “3” sentiments have been filtered out to scale the machine learning problem down to a binary classification problem. Then the dataset was sampled down further so that there would be an even representation of “0” and “4” sentiment (the internet tends to be very negative place).

We will be reading in the dataset from a public blob storage using the “reader module”.



Reader Module Parameters:

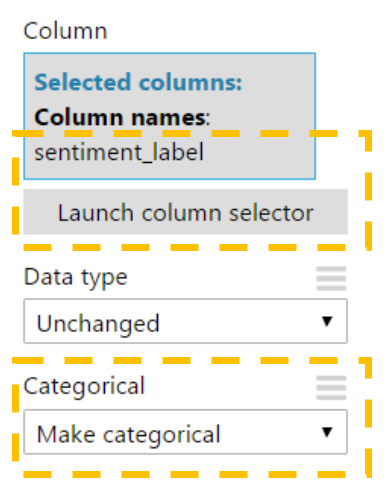
Data source: Azure Blob Storage  
Authentication type: PublicOrSAS  
URI: <http://azuremlsampleexperiments.blob.core.windows.net/datasets/Sentiment140.tenPercent.sample.tweets.tsv>  
File format: TSV  
URI has header row: Checked

From here you may right-click on the output and “save as dataset”. You may delete the reader module and drag in the newly saved dataset which is now listed under “My Datasets”.



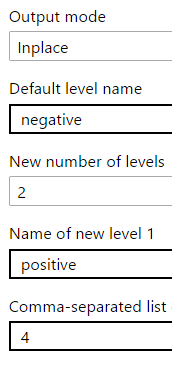
Categorical Casting & Transformation

The “sentiment\_label” column is currently a numeric feature and needs to be casted into a discrete or categorical feature or else this will become a regression problem rather than a binary classification problem. To do that, drag in the meta-data editor, select “sentiment\_label” using the column selector and set “categorical” to “make categorical” in the dropdown menu.



Right it’s not so clear what “0” and “4” means without an external explanation or an accompanying legend or key. What we will do now is convert all instances of “0” to say “negative” and “4” to say “positive”. That way anyone who uses our predictive model out of context will at least know which is which. This step is entirely optional but will help to facilitate clarity from our responses to external users if we were to ever share this model. To do this we will use the “group categorical values” module. We will use it to specify two separate categorical levels to be grouped and have the transformed values replace the old values.

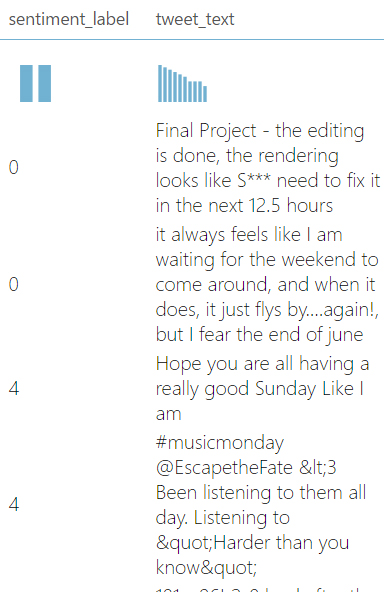




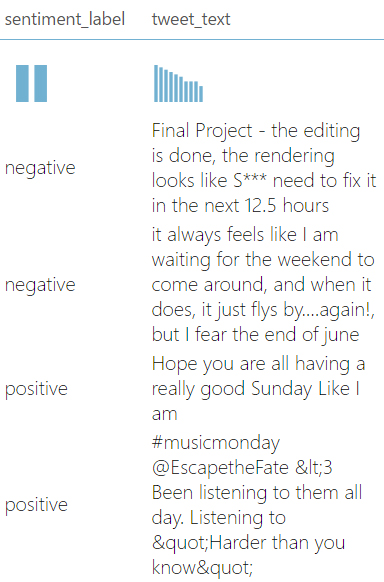
Group Categorical Values Parameters:

Selected Columns: **“sentiment\_label”** Output mode: **inplace**  
 Default level name: **negative**  
 New number of levels: **“2”** Name of new level 1: **positive**  
 Comma-separated list of old levels to map to new level 1: **“4”**

Before:



After:



**Text Processing Overview**

Now we enter the realm of text processing. A tweet message body is unstructured and mathematical operations cannot be performed on unstructured data. So in order for our machine learning models to operate we must apply transformations that give the message body structure. Usually this is done by transforming words into a sparse matrix, aka somehow give each word its own column and track whether or not a word appears in the text. For example, consider the two documents below when transformed into a simple term frequency matrix. Each word gets a distinct column and the frequency of occurrence is tracked. In the example below each document can be represented on a six dimensional hyperplane and distance measurements can be calculated, from distance we can figure out things like similarity or distinction.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Document | twinkle | little | star | all | the | night |
| Twinkle, twinkle, little star. | 2 | 1 | 1 | 0 | 0 | 0 |
| Twinkle, twinkle, all the night. | 2 | 0 | 0 | 1 | 1 | 1 |

The problem: However, getting to this term frequency matrix can be a little tricky. Imagine turning every word in the English dictionary into a matrix, that’s 171,476 columns. Now imagine adding everyone’s names, every corporation or product or street name that ever existed. Now feed it slang. Feed it every rap song. Feed it fantasy novels like Lord of the Rings or Harry Potter so that our model will know what to do when it encounters “The Shire” or “Hogwarts”. Good, now that’s just English. Do the same thing again for Russian, Mandarin, and every other language. After this is accomplished, we are approaching a trillion columns and two problems arise. First, it becomes computationally infeasible to perform permutations over this matrix. Secondly, the matrix becomes very sparse and any distance measurements becomes so absurdly distant in scale that they all seem the same (France and Poland are very far apart from each other, but from the perspective of a New Yorker, they in relatively the same place by distance measurements). Most of the research that goes into natural language processing is less about the syntax of language (which is important) but more about how to reduce the size of this matrix. We will introduce you to 3 concepts, conforming, stemming, and stop word removal.

**Text Processing Step 1: Cleansing and Conforming the Text**

Before we can get to this matrix we have to conform our text or else the computer will mistaken the same words for other words. For example, in the document above, “twinkle” and “Twinkle” are two separate words to a computer, same with “star.” and “star” and would be given two distinct columns each even though they are the same word. We will also be removing all numbers from our documents. As a beginning guide to text processing, we will apply a sweeping general transformation where we remove all numbers, special characters, and lowercase every word.

**Special Considerations:** I will now talk about a few considerations that are beyond the scope of this introductory tutorial but is **very** important for real-life production systems. By removing special characters, we are omitting them as data points that could have been used and learned from by the model. For example, text based emoticons like “=)” or “>:(” are very indicative of sentiment. We are also removing digits, so consider the infinitely gridlocked freeway in Washington State, “I-405”. By normal model standards anytime someone talks about “I-405”, more likely than not the document should be classified as “negative”. However, by removing numbers and special characters, the word now becomes “I”. We have **completely** changed the meaning of the word. The same goes for capitalization, so “trump”, which is usually a positive word, may have completely different meaning when used as “Trump” with a capital T, representing someone’s last name.

**Text Processing Step 2: Stemming and Lemmatization**

Now we must do what’s called [stemming](https://en.wikipedia.org/wiki/Stemming). Stemming is where we group together the same representation of the word in its various forms such as past tense, present tense, future tense, progressive, simple, or perfect forms. For example, the words “argue”, “argued”, “argues”, “arguing”, “argumentative” and “argument” all carry with them the same information, that there is a disagreement, and should be represented as a single column rather than 5 different columns within the matrix. Stemming will reduce all words down to their English root. So all instances of the words “argue”, “argued”, “argues”, “arguing”, “argumentative” and “argument” would be replaced to just simply “argu”. So in our matrix example above, if in future documents there were the words “twinkling”, “twinkler”, “twinkles” and “twinkle”, they would all be stemmed to the root of “twinkl”.

**Text Processing Step 3: Cleaning & Filtering Stop Words**

<https://raw.githubusercontent.com/datasciencedojo/meetup/master/real-time_sentiment/AzureML%20Code/Stop%20Words%20List.csv>