# Pride Month Classifier:

You work as a social media moderator for your firm. Your key responsibility is to tag uploaded content (images) during Pride Month based on its sentiment (positive, negative,or random) and categorize them for internal reference and SEO optimization.

## Task

Your task is to build an engine that combines the concepts of OCR and NLP that accepts a .jpg file as input, extracts the text, if any, and classifies sentiment as **positive** or **negative**. If the text sentiment is neutral or an image file does not have any text, then it is classified as **random**.

## Data

You must use an external dataset to train your model. The attached dataset link contains the sample data of each category [Positive | Negative | Random] and test data.

**Data files**

|  |  |
| --- | --- |
| **File name** | **Description** |
| Test.zip | Contains image files to be classified |
| Sample.zip | Contains sample image files belonging to each category |
| Test.csv | Predictions file containing indices of test data and a blank target column |
| sample\_submission.csv | Submission format to be followed for uploading predictions |

**Data description**

|  |  |
| --- | --- |
| **Column name** | **Description** |
| Filename | File name of test data image |
| Category | Target column [values: 'Positive'**/**'Negative'**/**'Random'] |

My approach:

Not having enough experience in the sector of natural language processing, I decided to go through how to perform simple spam filtering in Tensorflow to later expand that into the current problem.

# Sentiment Analysis:

Reading the mood from text with machine learning is called [sentiment analysis](https://en.wikipedia.org/wiki/Sentiment_analysis), and it is one of the prominent use cases in text classification. This falls into the very active research field of [natural language processing (NLP)](https://en.wikipedia.org/wiki/Natural_language_processing). Other common use cases of text classification include detection of spam, auto tagging of customer queries, and categorization of text into defined topics. So how can you do this?

Before we start, let’s take a look at what data we have. Go ahead and download the data set from the [Sentiment Labelled Sentences Data Set](https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences) from the UCI Machine Learning Repository.

By the way, this repository is a wonderful source for machine learning data sets when you want to try out some algorithms. This data set includes labeled reviews from IMDb, Amazon, and Yelp. Each review is marked with a score of 0 for a negative sentiment or 1 for a positive sentiment.

Extract the folder into a data folder and go ahead and load the data with [Pandas](https://pandas.pydata.org/):

# On to the Script:

import pandas as pd

filepath\_dict = {'yelp': 'data/sentiment\_analysis/yelp\_labelled.txt',

'amazon': 'data/sentiment\_analysis/amazon\_cells\_labelled.txt',

'imdb': 'data/sentiment\_analysis/imdb\_labelled.txt'}

df\_list = []

for source, filepath in filepath\_dict.items():

df = pd.read\_csv(filepath, names=['sentence', 'label'], sep='\t')

df['source'] = source # Add another column filled with the source name

df\_list.append(df)

df = pd.concat(df\_list)

print(df.iloc[0])

The result will be as follows:

sentence Wow... Loved this place.

label 1

source yelp

Name: 0, dtype: object

One way you could do this is to count the frequency of each word in each sentence and tie this count back to the entire set of words in the data set. You would start by taking the data and creating a vocabulary from all the words in all sentences. The collection of texts is also called a corpus in NLP.

The vocabulary in this case is a list of words that occurred in our text where each word has its own index. This enables you to create a vector for a sentence. You would then take the sentence you want to vectorize, and you count each occurrence in the vocabulary. The resulting vector will be with the length of the vocabulary and a count for each word in the vocabulary.

The resulting vector is also called a feature vector. In a feature vector, each dimension can be a numeric or categorical feature, like for example the height of a building, the price of a stock, or, in our case, the count of a word in a vocabulary.

use the CountVectorizer provided by the scikit-learn library to vectorize sentences. It takes the words of each sentence and creates a vocabulary of all the unique words in the sentences. This vocabulary can then be used to create a feature vector of the count of the words:

>>> sentences = ['John likes ice cream', 'John hates chocolate.']

from sklearn.feature\_extraction.text import CountVectorizer

>>> vectorizer = CountVectorizer(min\_df=0, lowercase=False)

>>> vectorizer.fit(sentences)

>>> vectorizer.vocabulary\_

{'John': 0, 'chocolate': 1, 'cream': 2, 'hates': 3, 'ice': 4, 'likes': 5}

This vocabulary serves also as an index of each word. Now, you can take each sentence and get the word occurrences of the words based on the previous vocabulary.

When you take the previous two sentences and transform them with the CountVectorizer you will get a vector representing the count of each word of the sentence:

>>> vectorizer.transform(sentences).toarray()

array([[1, 0, 1, 0, 1, 1],

[1, 1, 0, 1, 0, 0]])

This is considered a Bag-of-words (BOW) model, which is a common way in NLP to create vectors out of text. Each document is represented as a vector.

When you work with machine learning, one important step is to define a baseline model. This usually involves a simple model, which is then used as a comparison with the more advanced models that you want to test. In this case, you’ll use the baseline model to compare it to the more advanced methods involving (deep) neural networks, the meat and potatoes of this tutorial.

First, you are going to split the data into a training and testing set which will allow you to evaluate the accuracy and see if your model generalizes well.

We start by taking the Yelp data set which we extract from our concatenated data set. From there, we take the sentences and labels. The .values returns a NumPy array instead of a Pandas Series object which is in this context easier to work with:

>>> from sklearn.model\_selection import train\_test\_split

>>> df\_yelp = df[df['source'] == 'yelp']

>>> sentences = df\_yelp['sentence'].values

>>> y = df\_yelp['label'].values

>>> sentences\_train, sentences\_test, y\_train, y\_test = train\_test\_split(

... sentences, y, test\_size=0.25, random\_state=1000)

Here we will use again on the previous BOW model to vectorize the sentences. You can use again the CountVectorizer for this task. Since you might not have the testing data available during training, you can create the vocabulary using only the training data. Using this vocabulary, you can create the feature vectors for each sentence of the training and testing set:

>>> from sklearn.feature\_extraction.text import CountVectorizer

>>> vectorizer = CountVectorizer()

>>> vectorizer.fit(sentences\_train)

>>> X\_train = vectorizer.transform(sentences\_train)

>>> X\_test = vectorizer.transform(sentences\_test)

>>> X\_train

<750x1714 sparse matrix of type '<class 'numpy.int64'>'

with 7368 stored elements in Compressed Sparse Row format>

You can see that the resulting feature vectors have 750 samples which are the number of training samples we have after the train-test split. CountVectorizer performs tokenization which separates the sentences into a set of tokens as you saw previously in the vocabulary. It additionally removes punctuation and special characters and can apply other preprocessing to each word.

The classification model we are going to use is the logistic regression which is a simple yet powerful linear model that is mathematically speaking in fact a form of regression between 0 and 1 based on the input feature vector.

>>> from sklearn.linear\_model import LogisticRegression

>>> classifier = LogisticRegression()

>>> classifier.fit(X\_train, y\_train)

>>> score = classifier.score(X\_test, y\_test)

>>> print("Accuracy:", score)

Accuracy: 0.796

You can see that the logistic regression reached an impressive 79.6%, but let’s have a look how this model performs on the other data sets that we have. In this script, we perform and evaluate the whole process for each data set that we have:

for source in df['source'].unique():

df\_source = df[df['source'] == source]

sentences = df\_source['sentence'].values

y = df\_source['label'].values

sentences\_train, sentences\_test, y\_train, y\_test = train\_test\_split(

sentences, y, test\_size=0.25, random\_state=1000)

vectorizer = CountVectorizer()

vectorizer.fit(sentences\_train)

X\_train = vectorizer.transform(sentences\_train)

X\_test = vectorizer.transform(sentences\_test)

classifier = LogisticRegression()

classifier.fit(X\_train, y\_train)

score = classifier.score(X\_test, y\_test)

print('Accuracy for {} data: {:.4f}'.format(source, score))

Accuracy for yelp data: 0.7960

Accuracy for amazon data: 0.7960

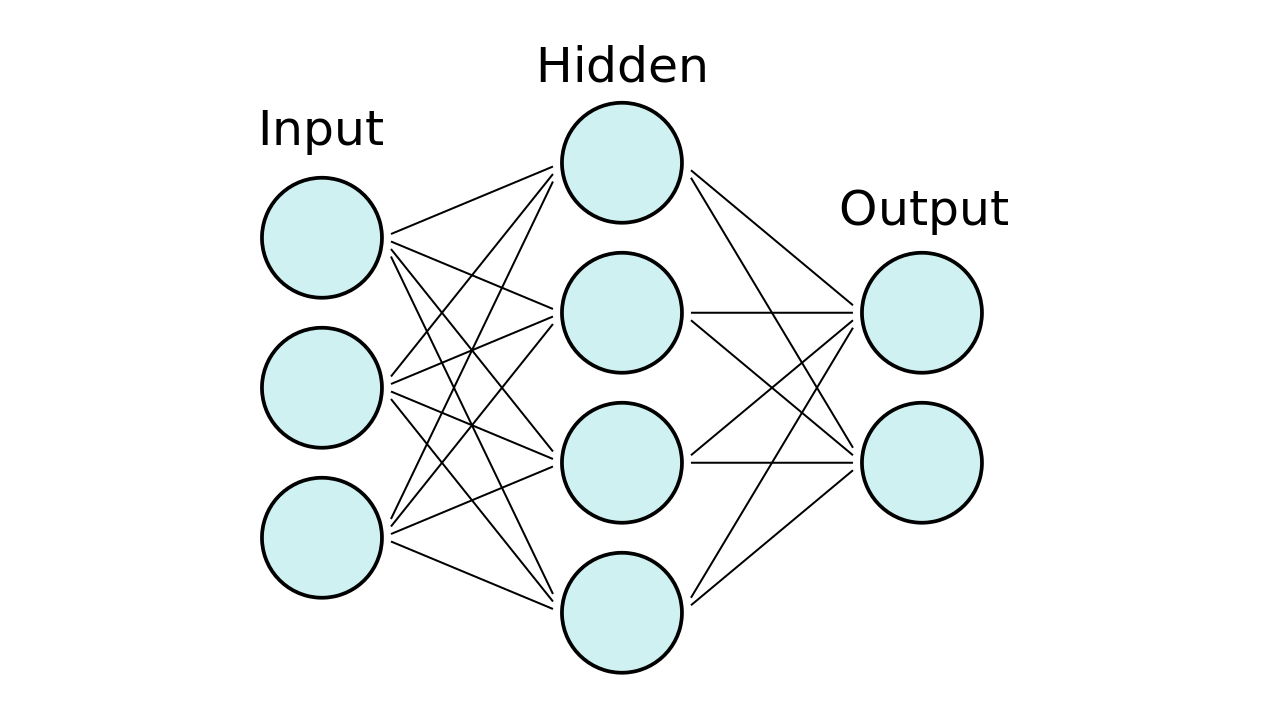
Accuracy for imdb data: 0.7487

This was a simple application of logistic regression, however now to expand this further we will try to implement neural networks and deep learning to make this better.

Neural networks, or sometimes called artificial neural network (ANN) or feedforward neural network, are computational networks which were vaguely inspired by the neural networks in the human brain. They consist of neurons (also called nodes) which are connected like in the graph below.

You start by having a layer of input neurons where you feed in your feature vectors and the values are then feeded forward to a hidden layer. At each connection, you are feeding the value forward, while the value is multiplied by a weight and a bias is added to the value. This happens at every connection and at the end you reach an output layer with one or more output nodes.

If you want to have a binary classification you can use one node, but if you have multiple categories you should use multiple nodes for each category:



We will be using keras as our framework to make this system work,

Let’s see if we can achieve some improvement to our previous logistic regression model. You can use the X\_train and X\_test arrays that you built in our earlier example.

Before we build our model, we need to know the input dimension of our feature vectors. This happens only in the first layer since the following layers can do automatic shape inference. In order to build the Sequential model, you can add layers one by one in order as follows:

The back end used for keras is tensorflow.

>>> from keras.models import Sequential

>>> from keras import layers

>>> input\_dim = X\_train.shape[1] # Number of features

>>> model = Sequential()

>>> model.add(layers.Dense(10, input\_dim=input\_dim, activation='relu'))

>>> model.add(layers.Dense(1, activation='sigmoid'))

Using TensorFlow backend.

Before you can start with the training of the model, you need to configure the learning process. This is done with the .compile() method. This method specifies the optimizer and the loss function. Additionally, you can add a list of metrics which can be later used for evaluation, but they do not influence the training.

model.compile(loss='binary\_crossentropy',

... optimizer='adam',

... metrics=['accuracy'])

>>> model.summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense\_1 (Dense) (None, 10) 17150

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 1) 11

=================================================================

Total params: 17,161

Trainable params: 17,161

Non-trainable params: 0

from \_\_future\_\_ import print\_function  
import os  
import zipfile  
import plaidml.keras  
plaidml.keras.install\_backend()

add this always to use tensorflow backend on amd cpu

Another parameter you have to your selection is the batch size. The batch size is responsible for how many samples we want to use in one epoch, which means how many samples are used in one forward/backward pass. This increases the speed of the computation as it need fewer epochs to run, but it also needs more memory, and the model may degrade with larger batch sizes.

>>> history = model.fit(X\_train, y\_train,

... epochs=100,

... verbose=False,

... validation\_data=(X\_test, y\_test)

... batch\_size=10)

Now you can use the .evaluate() method to measure the accuracy of the model. You can do this both for the training data and testing data. We expect that the training data has a higher accuracy then for the testing data. Tee longer you would train a neural network, the more likely it is that it starts overfitting.

Note that if you rerun the .fit() method, you’ll start off with the computed weights from the previous training. Make sure to compile the model again before you start training the model again. Now let’s evaluate the accuracy model:

>>> loss, accuracy = model.evaluate(X\_train, y\_train, verbose=False)

>>> print("Training Accuracy: {:.4f}".format(accuracy))

>>> loss, accuracy = model.evaluate(X\_test, y\_test, verbose=False)

>>> print("Testing Accuracy: {:.4f}".format(accuracy))

Training Accuracy: 1.0000

Testing Accuracy: 0.7960

You can already see that the model was overfitting since it reached 100% accuracy for the training set. But this was expected since the number of epochs was fairly large for this model.

If you wish to use matplot lib and plot the following training procedure use the snippet below in your code

Note : if you are using pycharm or any other ide which does not provide block compile, consider saving the model to avoid retraining again and again

Use the model.save() function for that, find more in the following web link: <https://www.tensorflow.org/tutorials/keras/save_and_load>

You can see that we have trained our model for too long since the training set reached 100% accuracy. A good way to see when the model starts overfitting is when the loss of the validation data starts rising again. This tends to be a good point to stop the model. You can see this around 20-40 epochs in this training.