# Awful Meme Rater

Dorin Heroi

January 19, 2022

## 1 Introduction

The awful meme rater was developed, as its name shows, to rate memes and doing an awful job while at it. Its motivation is that you could probably save some precious time of your life testing a meme that you just created with this algorithm before throwing it on Reddit and getting *downvoted into oblivion*. It most probably won't be able to save you from the downvoting just yet, but further tests and improvements may help with this issue. The idea behind it is pretty simple. Based on image content and some metadata that is extracted and created, there are two clusters created with K-Means, one cluster being the popular memes and the other one being the unpopular memes. K-Means is used as it is an easy method to do unsupervised learning on a dataset, which is exactly what is needed in this case.

The dataset is created by scraping Reddit r/memes and programatically downloading the pictures that are `.jpg, .jpeg` or `.png` and creating a new file `imagename.attrs` where metadata is stored. This metadata consists of the title, score, date when the post was created and text that is extracted from the image. Some images may not have text in them.

## 2 Creating the dataset

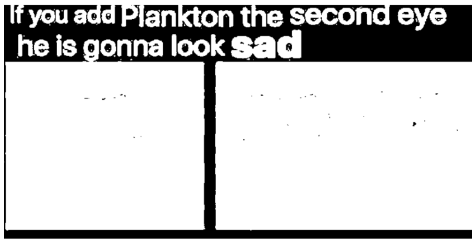
The dataset was created using PRAW: The Python Reddit API Wrapper. With it, you can choose one subreddit or more and iterate through the submissions. You can download the chosen submission and also check certain metdatata of the submission. I have iterated on through one subreddit, r/memes.

As stated in the previous section, the dataset `.attrs` file containts four types of metadata. They are title, score, creation date and extracted text. While you can easily get the `title, scrore` and `created_utc` attributes,

the text inside the image must be extracted with a specific OCR library. I have used Tesseract.

Before actually extracting text, the image must be pre-processed. I have tried different solutions for this. The first one I tried was applying canny edge detector on the grayscaled image to get only the edges of the text. This method usually resulted in many other edges besides the edges of the text. I tried applying a median blur on the image so that edges were smoother before applying the canny edge detector, but that didn't show good results either. Rotating the image also did nothing because memes are mostly in a good position. The last method used was using a bilateral filter, as it *can reduce unwanted noise very well while keeping edges fairly sharp*, as stated in the documentation. After this, we binarize the image using a threshold function.


(a) Original meme




(b) Filtered meme

Figure 1: Meme before and after

On this filtered image, we can apply `pytesseract.image_to_string` to get the text from the image. At first, it will not be very exact, so we do a little bit of tweaking. Tesseract has a three OEM(OCR Engine modes) and thirteen PSM(Page segmentation modes). After multiple combination, I chose OEM 3, that is default, and PSM 11, which means *"Sparse text. Find as much text as possible in no particular order"* as per the documentation. After applying `pytesseract.image_to_string`, we get some English words with some random letters, so we need to filter the words. I used a library named `enchant` to be used as an English US dictionary. After every word is checked, we remain only with the words available in this enchant dictionary.

```
12 https://i.redd.it/xqfj8i9f9kc81.jpg
he is gonna look sad
```

Figure 2: Filtered text from meme with corresponding image URI

After this, all the metadata is written into a file (with the title encoded UTF-8, due to the fact that some monsters like to put emojis in titles and ruin my script).

```
1  title=b'Plankton sad'
2  score=1262
3  created_utc=1642511274.0
4  text=he is gonna look sad
```

Figure 3: xqfj8i9f9kc81.attrs file contents

# 3 The algorithm

Having chosen the K-Means algorithm, we know that its input must be an array of floats. We know that the number of clusters is 2 (popular and unpopular), so now we have two problems: how to turn images into floats that actually mean something and how to turn attributes of the metadata in floats that actually mean something.

## 3.1 Turning images into floats

If we want to cluster images, we need to turn them into *numbers*. But putting just the pixels there would not mean much to the clustering algorithm as they would simply represent colors and would not be in a context. So, for solving this issue and getting a meaningful array of floats, we use feature extraction. We use a pre-trained model in Keras to extract the features. The model used

is VGG16. We can create a DataFrame based on these features and we are ready to go to the next step or we can create a cluster only with this data.

| | 25080 | 25081 | 25082 | 25083 | 25084 | 25085 | 25086 | **25087** |
|---|---|---|---|---|---|---|---|---|
| 129 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| 130 | 0.00000 | 0.00000 | 26.32618 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 30.87879 |
| 131 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| 132 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| 133 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 3.32359 | 0.00000 | 0.00000 | 0.00000 |
| 134 | 0.00000 | 0.00000 | 0.00000 | 7.81126 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| 135 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| 136 | 0.00000 | 0.00000 | 0.00000 | 15.21581 | 0.00000 | 0.00000 | 21.45491 | 0.00000 |
| 137 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 3.07853 | 0.00000 |
| 138 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 14.05489 | 0.00000 | 0.00000 | 0.00000 |
| 139 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| 140 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 8.65393 | 27.48213 |
| 141 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| 142 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 51.19298 | 0.00000 | 0.00000 | 0.00000 |
| 143 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| 144 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| 145 | 0.00000 | 3.29548 | 0.00000 | 0.00000 | 6.44276 | 0.00000 | 9.52805 | 0.00000 |
| 146 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 8.92938 | 0.00000 |
| 147 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| 148 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 | 19.89464 |

Figure 4: How the dataframe looks for 149 images

## 3.2 Turning text into floats

Turning text into floats is a bit trickier. To do this we must use TF-IDF (Term Frequency-Inverse Document Frequency). This is a numerical statistic that is intended to reflect how important a word is to a collection of words which fits perfectly to our needs. We use it as a weighting factor to calculate a score for each string of words gathered from OCR.

To go into more detail as to how it is used, a dictionary `attr_dict_expl` is created which holds the attributes as a list to each keyword. For example, we can access `attr_dict_expl["title"]` to get a list of all titles of the images stored. Let's call this list `title_list`. Applying function `get_tfidf_dict` on the category "title" will run `TfidVectorizer.fit()` and return a dictionary where, based on all the words in all of elements in the `title_list`, each is assigned an idf score. Based on this resulting dictionary, we can calculate the score of each title in `title_list`. The score is represented by a sum of the idfs of all **known** words in the dictionary. This sum is then

divided by the number of known words so as to not be biased towards longer titles/sentences.

## 3.3 Determining popular and unpopular memes

The determination of the popularity was made in a very naive way. Knowing that we already have two clusters, we need to determine which one is the cluster of popular images and which is the other. We calculate an average sum based on all the memes and then we check which clusters has more memes that have a score (read upvotes) greater that the average sum. Then, when we predict with our meme, we can see if it is popular or not. If prediction puts our meme in the unpopular cluster, it tells us to try another meme.

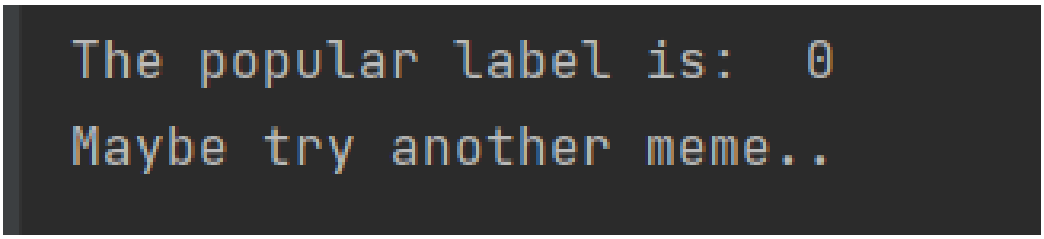



Figure 5: The determination of the popular meme - our was not popular

## 3.4 Other relevant results

This section is dedicated to other relevant results.

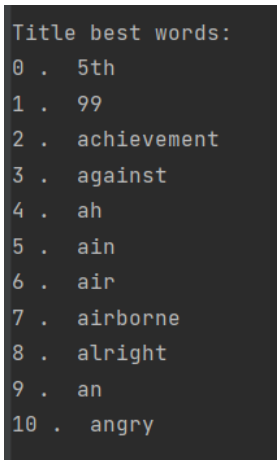



Figure 6: The most important words used in the title

```
Text best words:
0 .  120
1 .  16
2 .  182
3 .  183
4 .  2050
5 .  2nd
6 .  46
7 .  8th
8 .  99
9 .  ab
10 .  ace
```

Figure 7: The most important words used in the text

## 3.5 Errors encountered

An interesting bug was when I tried testing with a new meme. First of all, I tried fitting the dataset appended with this new meme. Due to a bug, it got added twice to the dataset so its lenght was now `len(dataset + 2)`. When I checked the labels, everything was labeled with 0, except the last 2 images. This is to show how biased you can turn the model with just adding another one of the same item is a 25088 dimensional space.

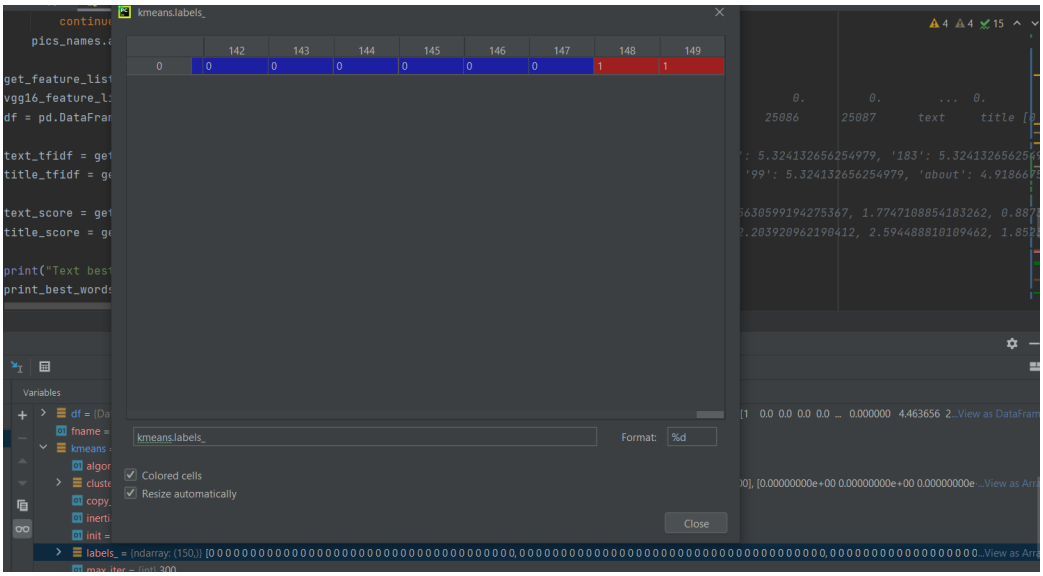



Figure 8: All labels 0 except the last two

# 4 Conclusions

Project was ***fun*** and it reached a place where the dataset can be automatically built and a model can be built on it so I can say it serves its purpose. You can feed it a meme and it predicts in what cluster it is.

## 4.1 Can it be done better?

Most probably yes. As stated before in this document, many implementations were *naive*. One of them would be the TF-IDF implementation which could probably have used the scores better than to calculate their sum.

## 4.2 How could it be developed in the future?

A more efficient way of doing things would be to "throw" it in a neural network that not only outputs features(I'm looking at you, VGG16). I am not yet aware of how that implementation would look like. One first step would be to feed the VGG16 outputs into a SVM for classification.

## 4.3 Does it *really* work?

I would say the project does what it was built for, but there is no way to test the actual accuracy of it other than actually watching the progress of the memes in their natural habitat, and that is Reddit.