Master in Artificial Intelligence
Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

Money Tracker Technical Manual

Gilles Charlier
Jason Klimack

2021-01-14

Contents

1 Introduction **0**

2 Functional Architecture of the system **0**

2.1 Input 1

2.2 Preprocessing 1

2.2.1 Description 1

2.2.2 Implementation (`pic_to_text.py`) 1

2.3 Image to Text 2

2.3.1 Description 2

2.3.2 Implementation (`pic_to_text.py`) 3

2.4 Text to Object 4

2.4.1 Description 4

2.4.2 Implementation (`text2obj.py`) 5

2.5 Machine Learning Model 5

2.5.1 Database & preprocessing (`process_database.py`) 5

2.5.2 Training of the model (`process_database.py`) 6

2.5.3 Implementation (multiple files) 7

2.6 User Interface 8

1 Introduction

In this technical manual, we first explain our architecture design. The architecture is displayed in figure 1. Then we explain in detail how each part of the architecture works by first giving an explanation and then implementation details. The first component of the architecture is the input, that is essential for creating the output. Then the **Preprocessing module** adapts input images to make them more suitable for the **Image to Text module**. Then the **Text to Object** takes the output of the **Image to Text module** and filters product names for the **Machine Learning trained model** that has been trained with Amazon reviews dataset. At the end, the UI coordinates everything and outputs some graphs thanks to predictions of the model.

2 Functional Architecture of the system

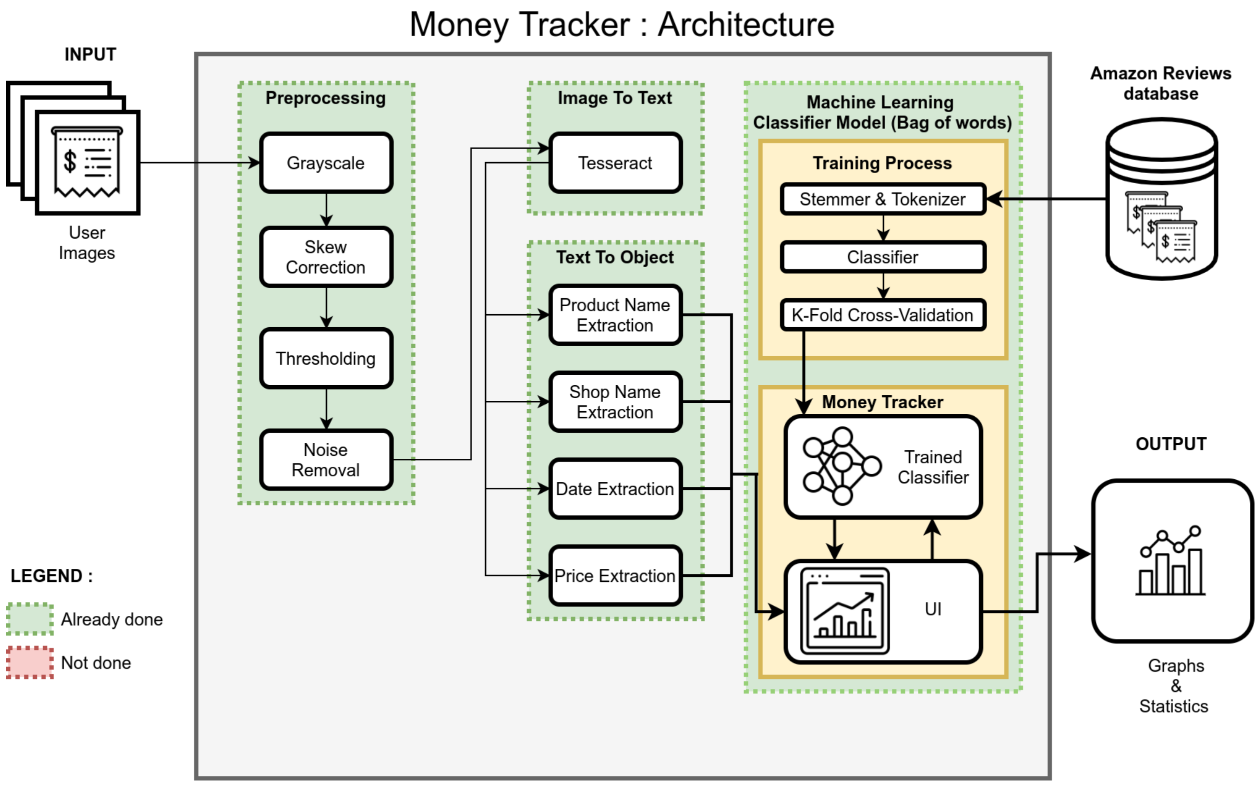


Figure 1: Functional architecture of the intelligent system

2.1 Input

Input images are images that have been taken by the user with any camera. We assume that the images taken were done so using a modern smartphone with high resolution since almost everyone has a smartphone. Our system can process JPG or PNG pictures.

2.2 Preprocessing

2.2.1 Description

For preprocessing the images, we have used grayscaling, thresholding, skew correction, opening and canny steps. **Grayscale** images consist of only a single color channel representing the pixel intensity of the image, and are computed from RGB images, which have three color channels. This also reduces the amount of data to be processed. **Thresholding** means that each pixel will be changed to either white or black (a binary value), based on its shade of gray and a predefined threshold of the algorithm. As shown in figure 2, the grayscale and thresholding techniques removes much of the noise from the image (eg. wrinkles, minor shadows, and background noise) so that the text on the receipt is clearer, and easier to extract. The worst results occur when there is a shadow on the image. As we can see in figure 3, shadows are more black than white which means that each pixel of the shadow will be translated to black. Thus, the text will not be found in the shadow regions of the image, resulting in the image being unusable. To fix this problem, we can use **adaptive thresholding**, which will adapt itself to the brightness of the image, resulting in more noise, but the content of the shadows will be readable. **Skew correction** simply detects the angle of the text in the image, and make it horizontal. **Opening** is a method used to remove noise in the image (that can maybe caused by adaptive threshold for example, that's why it is a the end of the pipeline). On Figure 4 we can see that noise has been removed with the opening process (on the left). As a last image preprocessing steps, we used **canny edge detector** process, which basically returns a black image with most important edges in white. This step allows us to more easily detect text on tickets. All these steps are essential the text extraction step.

2.2.2 Implementation (`pic_to_text.py`)

For this step, we have mainly used the library called OpenCV. For that we first import the module `cv2`. Then we create methods that first creates a kernel that will be passed in argument when we call OpenCV's functions. A kernel is a kind of matrix that is used in computer vision when processing pixels of an image [1]. We are not really concerned about customisation of that kernel so we just give a matrix of 5x5 of unsigned integers. Each function that we use (except the one that reads the image from the path) only takes as parameter an image object and returns a modified image object. We will therefore not explain parameters of the following function. Here are the functions we use in our preprocessing task :

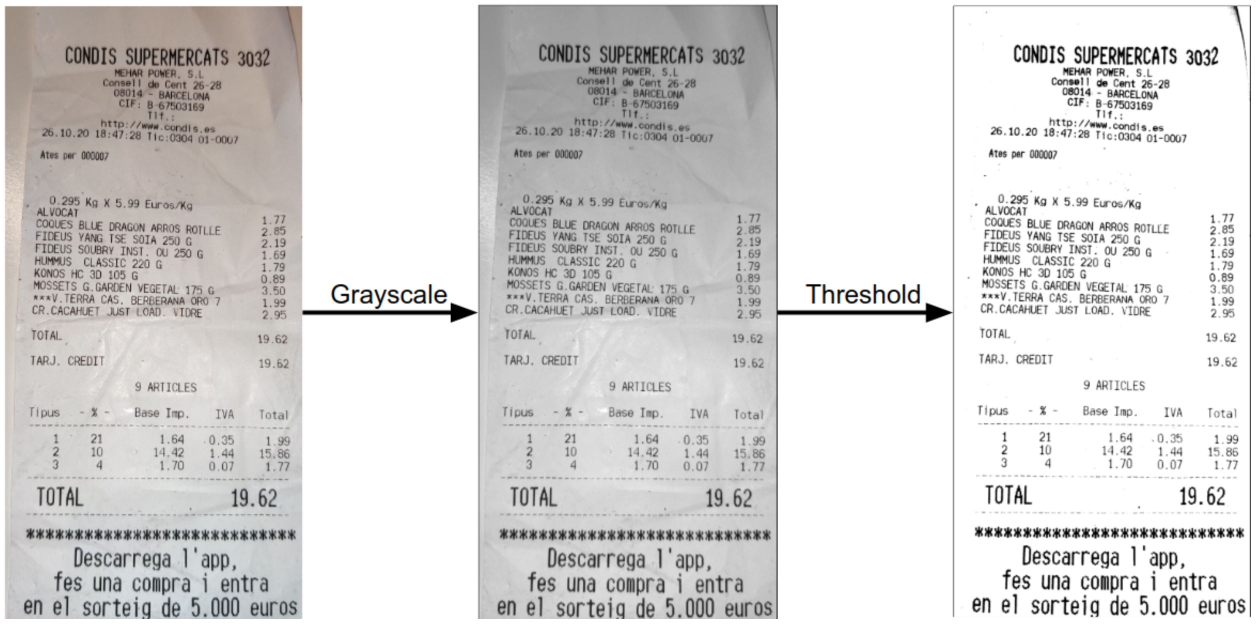


Figure 2: Preprocessing applied to an image

- `cv2.imread(image)` for reading the image from a specific path and returning it in a convenient format for the next functions.
- `cv2.get_grayscale(image,kernel)` that transforms images of multiple colours channels (usually 3) into mono colour channel image. As its name indicates, the following image will be gray after the call of this function.
- `cv2.thresholding(img,kernel)` that applies a threshold to the image. In this case we use only normal thresholding since we tested that it is the most effective one in most cases. Noises caused by adaptive thresholding can be fatal for text detection afterwards.
- `cv2.morphologyEx(img,cv2.MORPH_OPEN,kernel)` that applies the opening process described above in the description section.
- `cv2.canny(img,threshold1,threshold2)` that applies an edge detection of the image `img` and returns a black image with white edges.

2.3 Image to Text

2.3.1 Description

After preprocessing the image, the next step is to extract the text from the image. This is done quite simply using the library Tesseract-OCR, which is an open source engine for extracting text from images.

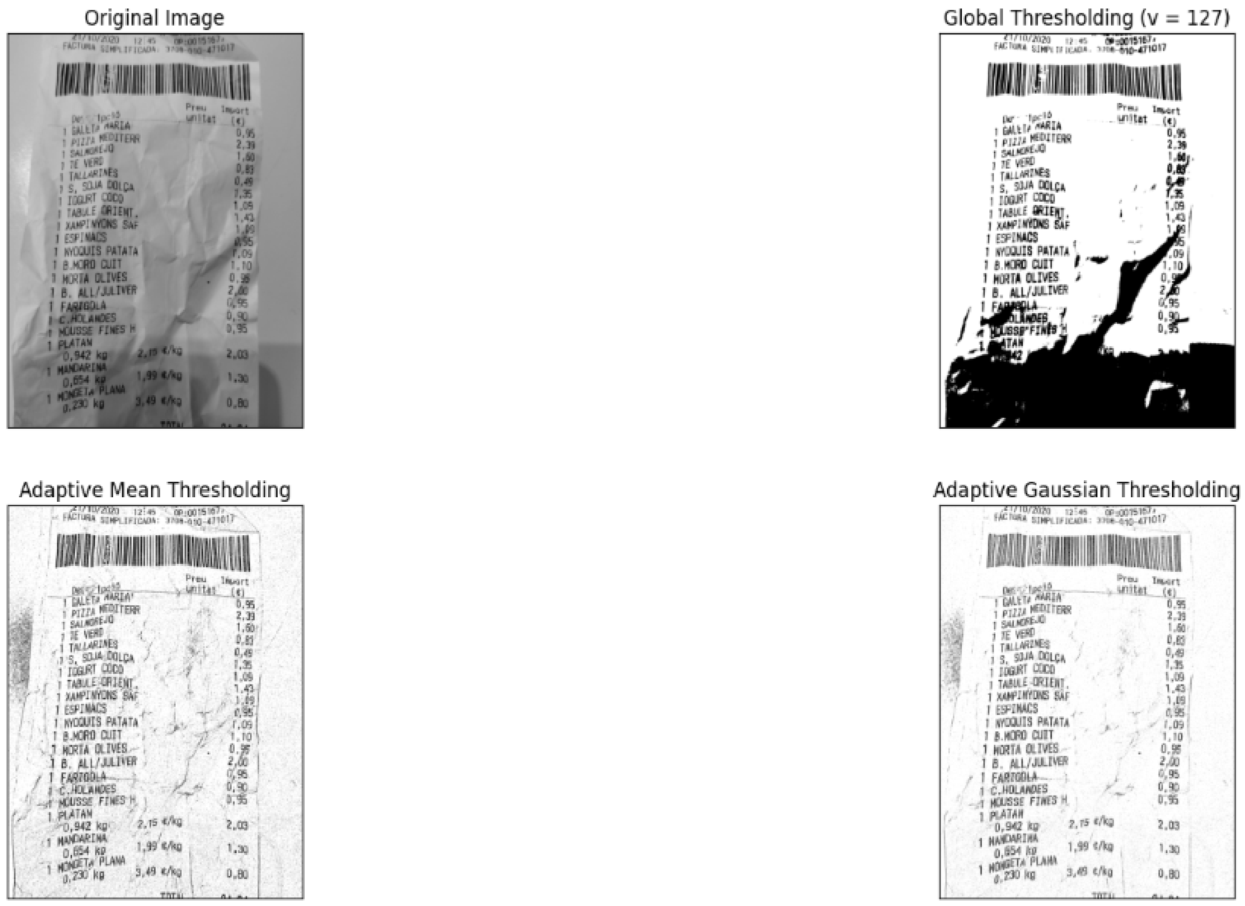
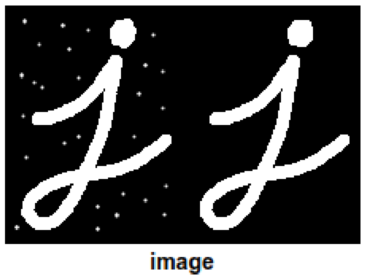

Original Image
Global Thresholding (v = 127)
Adaptive Mean Thresholding
Adaptive Gaussian Thresholding

Figure 3: Variants of thresholding applied to a complex image

We use Tesseract to extract the text by providing an image, and the output results are stored in a string variable that will be given back to the UI. We have already achieved quite good results with these previous steps. We have tried to use language packages from Tesseract, but they are less trained than the standard one, and words in receipts are not always complete and findable in a dictionary. We can clearly say that the preprocessing can improve a bit the quality of the images provided by the user but overall the initial quality and exposure of the original picture will have a huge impact on the performance of the system.

2.3.2 Implementation (`pic_to_text.py`)

When clicking the button `Extract Products`, the `UI.py` script will call the `pic2text(img_path)` method from the `pic_to_text.py` script to apply the load the image, apply the preprocessing, and then request the whole text of the ticket as a string from Pytesseract with the function call `txt= pytesseract.image_to_string(thresh)`. This string is then returned to the script `UI.py`.



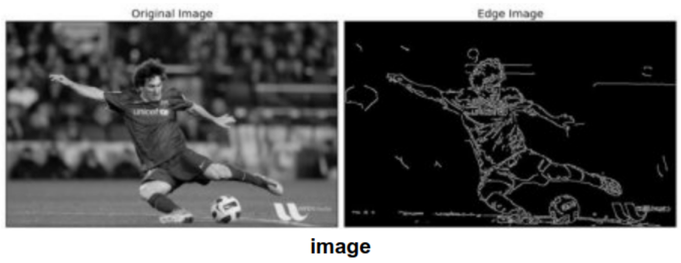


Figure 4: Opening (left) and Canny (right) preprocessing on example images

2.4 Text to Object

2.4.1 Description

The text extracted from the images in the previous steps comes in the form of a single string value composed of all of the characters and symbols found in the receipt. The majority of this information is not useful, especially in its current format. Thus, we need to only extract the useful information from the text, and create an object from this information.

There are a few points that we observe for most receipt images that we can utilize: 1) the first piece of text found on the receipt is usually the name of the store, 2) Each line that contains a product ends in the price (most of the time), which is in the form of a number with two decimal places, and 3) the total price of all of the products will be listed after all of the products with their associated prices.

To utilize the first point, we can take the first line of text that is not blank as the name of the shop that the receipt comes from. For the second point, we can comb through the receipt line-by-line applying a regular expression to each line for checking if the line ends in a decimal number. If the line ends in a decimal number with two decimal places, then the line contains a product. Currently, the regular expression that we are using is:

$$r = \text{"}\hat{}.*[0-9][\backslash.,][0-9][0-9](\ .*)?\$\text{"} \tag{1}$$

The third point can be used as a stop line in the text-parsing. If the word 'Total' is present in the line, then likely the parser has reached the end of the products, and we can stop searching for product lines.

Finally, we are using a dictionary to store the different values of the receipt. The key-value pairs in the dictionary are as follows:

- PRODUCTS : *list of product names found on the receipt*
- PRICES : *list of prices as float, associated to each product*

Further improvements will be to include the date that the receipt was made, as well as account for products that span multiple lines in the receipt, as the current model does not work for these features.

2.4.2 Implementation (`text2obj.py`)

The UI script calls the function `text2object(text)` with as argument the text provided by the picture to text extraction. In this function, first we put the whole string in uppercase, then we replace $ signs and with spaces as a simple preprocessing before the parsing. Then we split the string by lines. For each line we check if it ends with price pattern as explained above with regex expressions. If it is the case, we parse the current line and add the result in a dictionary. At the end of the function, we return the dictionary to the UI script that will handle it.

2.5 Machine Learning Model

2.5.1 Database & preprocessing (`process_database.py`)

In order to train a machine learning model, we were convinced that the more data we had the better it was. To train our model, were initially convinced that the french database from french students [2] was perfect for our tasks. But we were wrong. This database is more suitable to train a special text extractor for receipts, which was not our main goal. Categories of products are not provided in this database, and we can't really train the Tesseract picture to text extractor. Instead of using this database, we searched online for another one that was bigger, and that was containing product titles as well as their category. We found an Amazon product-review database [3] that contains millions of reviews that include the product title as well as its category. The database was heavy (around 15GB compressed in `json.gz` format) but we managed to only keep products that can be purchased in stores. For example, softwares and kindle books are not really meaningful for our trainer. After having filtered the dataset, we construct a dataset on CSV format (`database.CSV`) that contains only category name and title. This dataset contains 20 000 products for each category, resulting in more than 200 000 products. We weren't able to add more products in the training process since it was already requesting too much RAM to our laptops (around 13.5GB). To filter the database and train our model, we used the script `process_database.py` that is inspired from the amazon dataset user manual (for the database processing), as well as the amazing blog of Sara Robinson [4] that explains how to train and use the bag of words from the library Keras. To explain shortly what is in this script, the method `folder_to_csv()` was used to go through all the databases (because the main database was downloadable as multiple smaller databases sorted by categories). For each database, we collect a certain amount of products (20K in this case) and store

them in a bigger dataframe. At the beginning each product belonged to multiple categories and subcategories, but we have chosen to only keep one main category per product. We also shuffle the indices in order to not overfit for some categories/products.
After having added each product with its own category, we decided to apply some preprocessing to the product's titles. First, we removed unwanted characters thanks to regular expression matching with the module `re` and the instruction `symbols.search(word)` where symbols represent a compiled regex expression. Then we applied a stemming process to each title. Stemming is a technique in natural language processing (NLP) that crops the end of a word in the goal of retrieving the singular/infinitive form of the word. This is really necessary otherwise two words like "train" and "training" would be recognised as different. This is really useful to reduce the complexity of the model. Stemming is method of the library *Natural Language Toolkit (NLTK)* [5] and used with these instrutions :

```
stemmer = SnowballStemmer("english")
stemmer.stem(word)
```

2.5.2 Training of the model (`process_database.py`)

As a machine learning model, we searched a lot online to see which algorithm would suit the most for our use case. After many searches and tests, we decided to use a bag of words approach using the library Keras. To recall, the goal was to find the category of a product based on its title or name.
Let's see how a it works :
Bag of word is a technique used to represent the text in a numerical format, like vector and matrix. This method is particularly useful if we want to extract features from this text or guess its context or category. This method contains two elements : a dictionary of known words and a frequency counter attached to this dictionary. After having processed a whole text or a list of product's titles in this situation, we will have a huge dictionary. If we give an unknown title to the model, it will recognise only a fraction of words of it. For example, if we have a dictionary of length 5 `["desk", "lamp", "chair", "table", "mouse"]` and the unknown title `"Desk with Lamps 20cm x 7cm 200 nits"`, then we will have the following vector of the same length as the dictionary:

```
[1, 1, 0, 0, 0]
```

were 1's represent words that are in the current new title and 0's represent known words that are not in the new title. Since the sentence's array is in the order of the dictionary, we shuffle words like in a "bag".
Now that we have a huge matrix of titles, we can use this data to predict from which category each product is. To do that we first have to do the same process for our product's categories. For example, the unknown title `"Desk with Lamps 20cm x 7cm 200 nits"` with the categories "dictionary" `["Pet & supplies","Garden products", "Home & Kitchen"]` will have the category vector `[0,`

0, 1]. We can then train a classifier do its tasks by first giving it labelled data to train with, and then unlabelled data to test. Let's see how it is implemented in our code. The following code is based on the blog of Sara Robinson and its Google video [4], [6] but we finetuned it with code in the script `classifier-option.py`.

2.5.3 Implementation (multiple files)

First, we decided to save our final dataframe separately in order to not process it every time we train the model. We trained our model in the script `process_database.py`. Once the dataset is loaded as a dataframe called `data` we binarize our categories thanks to the module `sklearn.preprocessing` and split them into train and test sets of 80% and 20% of the whole dataset as explained earlier. Then we do the same for the titles of our products but we keep the most 8000 relevant words using a custom class called `TextPreprocessor` in the script `textpreprocessing.py`. Since we need a Tokenizer aka a text splitter and formater to transform our titles into a bag-of-word matrix for both the training/testing and runtime, we need to save the instance of this tokenizer as a file, in this case `processor_state.pkl` with the library `pickle`. Once we will be at runtime, we will just load the model as well as the tokenizer.

Then we create our model, that will be in this case a deep model with two hidden layers in between the input and output layers. After fine-tuning we discovered that 100 for the first hidden layer and 50 for the second one was a good trade-off. Obviously the number of inputs is the size of the dictionary (in this case 8000 was a good value) and the number of outputs is the number of categories in our model, in this case 10. The activation function of our output layer is sigmoid since we want an accuracy for each prediction between zero and one. Once our model is created, we fit it on the training categories and titles with 3 epochs, a batch size of 128 and a validation split of 0.1. This validation split allows us to fine-tune the train model without overfitting. Then we evaluate it with the remaining 20% of our dataset to see if it performs well or not. We achieve 87% of accuracy overall, which is quite good. We tried other models, like convolutional neural networks but it wasn't performing as good as this simple one, and was way slower.

Once the model is trained and evaluated, it's time to save it on a file `keras_saved_model.h5`.

Once the UI requests a prediction to the model, the UI calls the class `classifier.py` that will stem the words received try to make predictions by first loading the Tokenizer as well as the model trained. This is done in the class `CustomModelPrediction` on the script `custom.py`. If at least a prediction has more than 33% certainty, the model will return the best prediction, otherwise it returns the category "Other" for that product. Then the category will be given back to the UI.

2.6 User Interface

Finally, we created the user interface to allows a user to input their own images, and create output graphs for visual display of their expenditures using `Tkinter`. This user interface explained in detail in the User manual document.

References

[1] Thiago Carvalho. *Basics of Kernels and Convolutions with OpenCV*. en. July 2020. URL: `https://towardsdatascience.com/basics-of-kernels-and-convolutions-with-opencv-c15311ab8f55` (visited on 01/13/2021).

[2] Chloe Artaud et al. "Find it! Fraud Detection Contest Report". en. In: *2018 24th International Conference on Pattern Recognition (ICPR)*. Beijing: IEEE, Aug. 2018, pp. 13–18. ISBN: 978-1-5386-3788-3. DOI: `10.1109/ICPR.2018.8545428`. URL: `https://ieeexplore.ieee.org/document/8545428/` (visited on 11/18/2020).

[3] *Amazon review data*. URL: `https://nijianmo.github.io/amazon/index.html` (visited on 01/13/2021).

[4] Sara Robinson. *Interpreting bag of words models with SHAP*. URL: `https://sararobinson.dev/2019/04/23/interpret-bag-of-words-models-shap.html` (visited on 01/13/2021).

[5] *Stemming and Lemmatization in Python*. Oct. 2018. URL: `https://www.datacamp.com/community/tutorials/stemming-lemmatization-python` (visited on 01/13/2021).

[6] Google. *Getting started with Natural Language Processing: Bag of words*. 2019. URL: `https://www.youtube.com/watch?v=UFtXy0KRxVI&t=206s&ab_channel=GoogleCloudPlatform` (visited on 01/13/2021).