Isaia Elena
Lăzărescu Anda-Monica
Mihăilă Andra

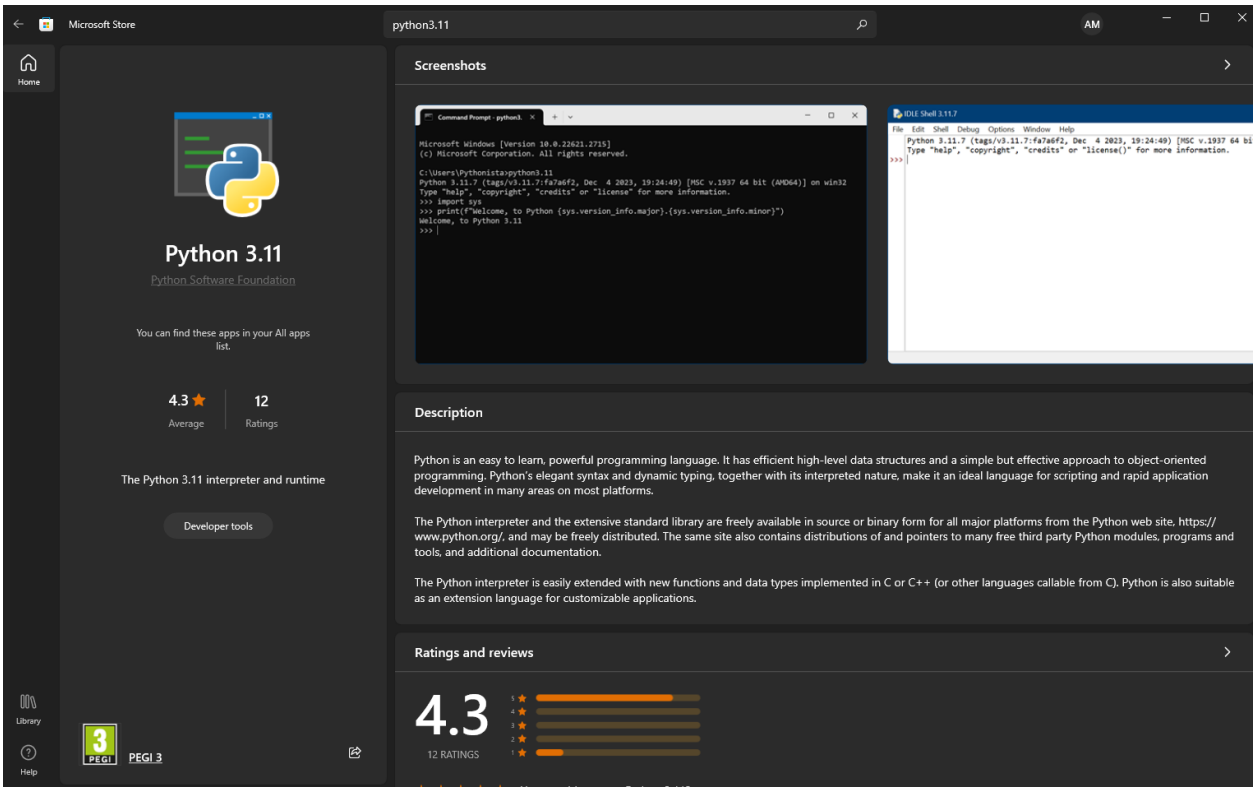
Sentiment analysis in text

1. Short description

Our project is focusing on sentiment analysis in text, specifically movie reviews written in Romanian. Our implementation is capable of classifying the overall sentiment behind a review as either positive or negative, based on the words used. Additionally, a scale between 0 and 1 is in place to determine how positive or negative a sentence is, this way also classifying the neutral sentences.

2. Setup

Our project is written in Python, using *Jupyter Notebook*. The main requirement is for Python 3.11 to be installed, being available in the Microsoft Store.



Additionally, Jupyter Notebook needs to be downloaded via cmd, using “pip install jupyter notebook”. Furthermore, we can launch the application using “python -m notebook”.

The “ipynb” file can be imported into the notebook, where we can find the runnable code. The dataset can be found under the following Github: https://github.com/katakonst/sentiment-analysis-tensorflow/tree/master/datasets. The dataset needs to be downloaded from this repository, and unzipped, the path to the folder needing to be added in the first cell.

```
path = pathlib.Path('C:/Users/mandr/Desktop/Andra Mihaila/Uni/DataMining/sentiment-analysis-tensorflow-master/datasets/ro/train') # modify to correct pat
```

Every cell in the notebook needs to be executed in order to see the results. The following libraries need to be installed in order to have no errors when running the code:

- pip install pandas
- pip install matplotlib
- pip install scikit-learn
- pip install lime
- nltk.download(‘stopwords’)
- nltk.download(‘punkt’)
- pip install tensorflow

3. Code description

3.1. Loading the dataset

The first step of our implementation is reading the dataset. In order to have the dataset ready for the classification, we need to remove all newlines and only select the lines that have some text. To store the dataset in our code we are using *pandas Dataframe*.

```
# loading the dataset
data = load_files(path, encoding="utf-8", decode_error="replace", random_state=500)

# remove newlines
data['data'] = [it.lower().replace('\n\n', ' ') for it in data['data']]

# convert dict to Pandas dataFrame
df_raw = pd.DataFrame(list(zip(data['data'], data['target'])), columns=['text', 'label'])

# select only rows with non empty text
df = df_raw[df_raw['text'] != '']
```

The following is a sample of our dataset:

	text	label
0	acest film a fost cel mai rau film pe care l-a...	0
1	calitatea de nostri a lui foley in acest film ...	1
3	creativitatea acestui film a fost pierduta de ...	0
4	cand am inchiriat acest lucru, speram ca ceea ...	0
5	acesta este un film de familie care incalzeste...	1
6	recomand\nraportul calitate-pret unul foarte bun	1
7	acest film parea promitator, dar de fapt era d...	0
8	este foarte amuzant. are o distributie mare, c...	1
9	am jurat mult timp in urma ca niciodata, sa ma...	0
10	unul dintre filmele cele mai nihiliste si brut...	1

3.2. Pre-processing the dataset

Then, the pre-processing of the dataset starts. At first, we convert every label from string format to binary values as our model will perform a binary classification. The next operation performed is making all characters from the sentences lower case.

```
# convert the labels from strings to binary values for our classifier
df['label'] = df.label.map({l: idx for idx, l in enumerate(set(df.label))})

# convert all characters to lower-case
df['text'] = df.text.map(lambda x: x.lower())
```

The following step is the tokenization of the sentences. The tokenization process implies dividing a string into substring by splitting on a specified character or string. For example, the sentence “Filmul a fost grozav!” will be tokenized to [filmul, a, fost, grozav, !]. In order to do this transformation, we used *nltk* library, which returns a list of words from the sentence based on the whitespaces and the punctuation of the phrase.

```
# apply tokenization: divide a string into substrings by splitting on the specified string (defined in subclasses)
df['text'] = df['text'].apply(nltk.word_tokenize)
```

After tokenization, we replaced every non-alphanumeric word, as well as stopwords and every punctuation sign.

```
STOPWORDS = set(stopwords.words('romanian'))
PUNCTUATION_SIGNS = [c for c in string.punctuation] + ['```', "'''", '...', '..']
```

```
# remove non-alphanumeric words
df['text'] = df.text.map(lambda txt: [word for word in txt if word.isalpha()])

# remove stopwords + punctuation signs
df['text'] = df.text.map(lambda txt: [word.strip() for word in txt if word not in STOPWORDS])
df['text'] = df.text.map(lambda txt: [word.strip() for word in txt if word not in PUNCTUATION_SIGNS ])
```

As the final step of our preprocessing, we performed stemming using *SnowballStemmer* for Romanian text, from the *nltk* library. Stemming refers to transforming variations of the same word to their stem base.

```
# perform word stemming: remove all variations of words carrying the same meaning (eg: filmul -> film)
stemmer = SnowballStemmer(language="romanian")
```

This is a sample of how the top 10 sentences in the dataset look like after these processes.

	text	label
0	[film, rau, film, vazut, vreodat, misiun, domi...	0
1	[calitat, foley, film, satur, intens, celuloz,...	1
3	[creativ, film, pierdut, incep, scriitor, regi...	0
4	[inchir, lucru, sper, reign, of, fir, nast, ci...	0
5	[film, famil, incalzest, inim, absol, straluc,...	1
6	[recomand, raport, bun]	1
7	[film, par, promit, fapt, dest, rau, premis, c...	0
8	[amuz, distribut, mar, perform, grozav, ales, ...	1
9	[jurat, timp, urma, niciod, uit, vreodat, film...	0
10	[film, nihilist, brutal, vazut, vreodat, tragi...	1

In addition, we needed to balance the dataset so that we didn't have more labels of one type, as that might create biases in the model, forcing it to learn to classify one label better than the other one.

3.3. Implementing the models

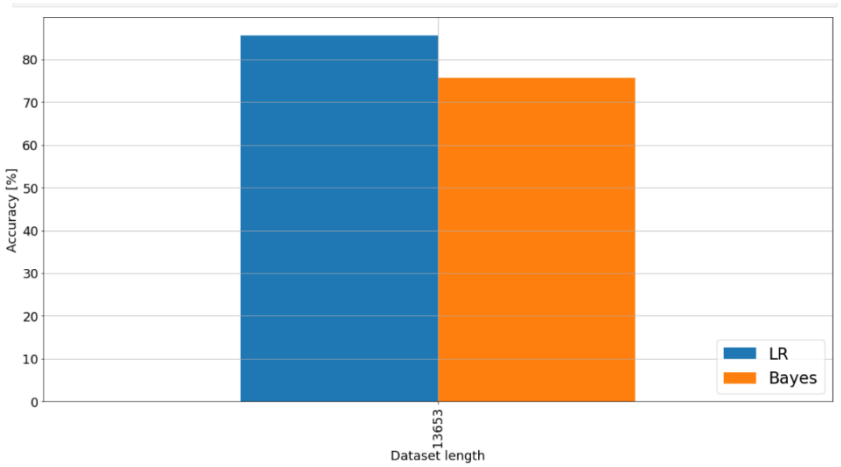
At first, we implemented the naïve Bayes model, as we thought that for some smaller datasets, it would work better. This classifier is built on the assumption that the features provided are independent of each other, given the specific target class. In other words, it operates under the condition that the presence or absence of one feature does not influence the presence or absence of another feature when the target class is known. After analyzing the accuracy score for the Bayes model, we also implemented the Logistic Regression method, which proved to give better results.

In order to run the experiments, we first split the dataset into train and test. For both models, our implementation used the *scikit-learn* library. For the Bayes model, we needed to choose which function was needed and the best fit for our problem. Therefore, we selected the Bernoulli method, which is designed for boolean/binary problems. After creating the instances for the two classifiers, we needed to fit the models to the training data and use the prediction methods from scikit-learn.

```python
def classify_logistic_regression(x_test, x_train, y_test, y_train):
    # crate Logistic Regression classifier
    model_lr = LogisticRegression()

    # fit classifier on training set
    model_lr.fit(x_train, y_train)

    # predict on test dataset
    y_pred_lr = model_lr.predict(x_test)

    # use 'accuracy_score()' function to compute the prediction accuracy
    _lr_accuracy_score = metrics.accuracy_score(y_pred_lr, y_test) * 100

    # compute classification report
    classification_report = metrics.classification_report(y_test, y_pred_lr)

    return model_lr, _lr_accuracy_score, y_pred_lr, classification_report


def classify_bayes(x_test, x_train, y_test, y_train):
    # create Naive Bayes classifier
    model_bayes = naive_bayes.BernoulliNB()  # designed for binary/boolean features (other options MultinomialNB, GaussianNB and ComplementNB)

    # fit Bayesian classifier on training set
    model_bayes.fit(x_train, y_train)

    # predict on test dataset
    y_pred_bayes = model_bayes.predict(x_test)

    # compute accuracy score by using accuracy_score() function
    _bayes_accuracy_score = metrics.accuracy_score(y_pred_bayes, y_test) * 100

    # compute classification report
    classification_report = metrics.classification_report(y_test, y_pred_bayes)

    return model_bayes, _bayes_accuracy_score, y_pred_bayes, classification_report
```

4. Results and performance measuring

After classifying the input data, we used the accuracy score to plot and highlight the difference in performance between Naïve Bayes model and the Logistic Regression.



By analyzing the accuracy, precision, recall, f1-score of the two implemented methods, the logistic regression produced better results in its predictions. The accuracy is clearly better for the LR model, since the two models have some differences that influence the performance of them such as:

- Bayes is a model in which the probability between two events is calculated using P(event2|event1), explaining that event1 has happened when event2 has occurred. On the other hand, the Logistic Regression is a model that directly calculates the link between event1 and event2, being able to catch intricate connections.
- In Bayes, all the features are considered to be independent, this meaning that, given two features that are correlated, the classification doesn't perform as well. However, in Logistic Regression, the features are split linearly.
- While Naïve Bayes has a higher bias and a lower variance, Logistic Regression is the opposite, with lower bias and higher variance. This means that LR is able to fit more closely the training data, enabling it to capture more complex relationships.

	precision	recall	f1-score	support
0	0.85	0.86	0.85	1343
1	0.86	0.86	0.86	1388
accuracy			0.86	2731
macro avg	0.86	0.86	0.86	2731
weighted avg	0.86	0.86	0.86	2731

	precision	recall	f1-score	support
0	0.77	0.72	0.74	1343
1	0.74	0.79	0.77	1388
accuracy			0.76	2731
macro avg	0.76	0.76	0.76	2731
weighted avg	0.76	0.76	0.76	2731

5. Error analysis

The first error that can be observed in either one of these models is that neutral sentences cannot be evaluated correctly. By only having two possible labels, positive and negative, we automatically cannot have the perfect answer for neutral sentences. Therefore, these types of sentences can produce inconsistent results, sometimes being evaluated to positive, sometimes to

negative, depending on the splitting of the initial dataset and the way the models learned the classification.

In order to test this, we used a tiny set of sentences, with one positive review, one negative review and a neutral one.

```
test_sample1 = 'Filmul este fantastic! Imi place foarte mult!'
test_sample2 = 'Filmul este groaznic! Nu l-as viziona!'
test_sample3 = 'Filmul este so-so. As merge mai degraba la teatru.'
```

At every run of this test, the first sentence was classified as positive and the second one as negative. But the third sentence was sometimes classified as positive and sometimes as negative due to the fact that the models couldn't classify accurately a neutral sentence.

6. Improved implementation

In order to solve the problem presented previously, we also implemented a classifier based on Neural Networks, which besides classifying the sentences to positive or negative, can give a score on a scale from 0 to 1 based on the positivity of each sentence, with 1 being the maximum positivity and 0 the maximum negativity.

For building the model we used a neural network from the *keras* library with the following architecture:

a) Embedding layer: Word Embedding is a textual representation in words that have similar meanings, have a similar representation. This way, depending on the relationships between words, the ones that have a related meaning, are placed closer. This aspect is typically managed through an embedding layer, which maintains a lookup table that maps words, denoted by numeric indexes, to their respective dense vector representations.

b) GRU: We used Gated Recurrent Units (GRU), a type of recurrent neural network architecture, designed to address the problems that might appear in other simpler models. It is a useful architecture in our case, as it has the ability to capture short-term dependencies in sequential text.

c) Batch Normalization: This is a technique used to normalize the input for every layer by adjusting the parameters through mini-batches.

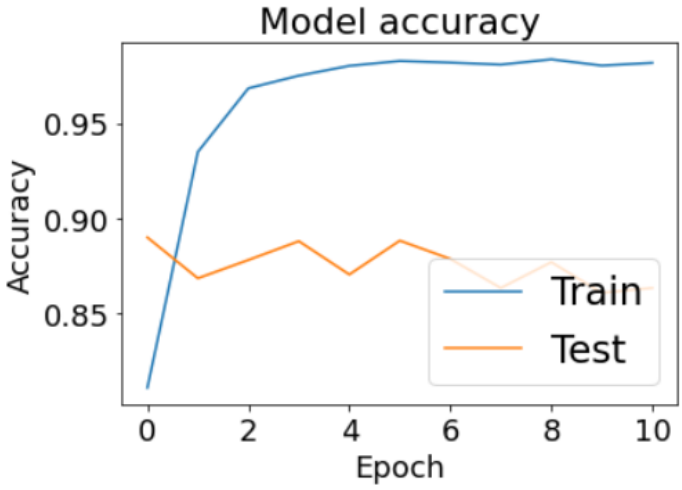
d) Dense layer: Our implementation used this type of layer as the problem involves learning complex patterns and relationships between words. We used the **sigmoid** activation function as we needed to produce a probability score for the classification of our sentences.

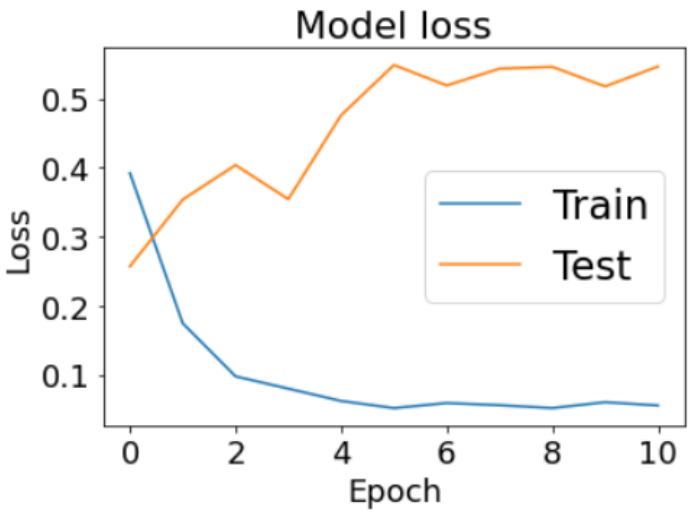
We then fit the model to our dataset and tested the classification accuracy values.

```
EMBEDDING_DIM = 100

print('Started building model structure...')

rnn_model = Sequential()
rnn_model.add(Embedding(vocab_size, EMBEDDING_DIM, input_length=max_length))
rnn_model.add(GRU(units=64, dropout=0.1, recurrent_dropout=0.1))
rnn_model.add(BatchNormalization())
rnn_model.add(Dense(1, activation='sigmoid'))

_optimizer = optimizers.RMSprop(learning_rate=0.01)

rnn_model.compile(loss='binary_crossentropy', optimizer=_optimizer,
              metrics=['accuracy'])

print('Done building model structure...')
print(rnn_model.summary())
```

7. Evaluation for the improved model

After training our model, we used the accuracy score and the loss value for analyzing the new implementation.





Overall, we can determine that our model has a good accuracy, fluctuating between 0.85 and 0.90 for our test data.

```
# Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='lower right')
plt.show()

# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='center right')
plt.show()
```

In addition, we used sample data in order to view the results produced by this model for the classification. We can see that sentences 1, 2, 6, 9, 10 are classified with values very close to 1, as they are truly positive, sentences 4 and 8 have values closer to 0, these sentences being negative ones. We can also easily observe that the neutral sentence problem was solved by this implementation, as in the new model, sentence 5, which is a neutral one has a value close to 0.5.

Isaia Elena
Lăzărescu Anda-Monica
Mihăilă Andra

```
test_sample1 = 'Filmul este fantastic! Imi place foarte mult!'
test_sample2 = 'Filmul este spectaculos! Mi-a placut foarte, foarte mult!'
test_sample3 = 'Filmul a fost teribil. As merge mai degraba la teatru.'
test_sample4 = 'A fost un film plictisitor.'
test_sample5 = 'Filmul nu a fost preferatul meu'
test_sample6 = 'Sunt foarte multumita. '
test_sample7 = 'Nu m-a impresionat... a fost o pierdere de timp'
test_sample8 = 'Se putea mai bine, actorii nu au jucat bine'
test_sample9 = 'Cel mai bun film pe care l-am vazut anul acesta!'
test_sample10 = 'Un desen foarte bun pentru copii, fetitei mele i-a placut mult'
test_samples = [test_sample1, test_sample2, test_sample3, test_sample4, test_sample5, test_sample6, test_sample7, test_s

test_samples_tokens = tokenizer_obj.texts_to_sequences(test_samples)
test_samples_tokens_pad = pad_sequences(test_samples_tokens, maxlen=max_length)

# predict
rnn_model.predict(x=test_samples_tokens_pad)
```

```
1/1 [==============================] - 0s 48ms/step
array([[0.9996122 ],
       [0.9996785 ],
       [0.3526308 ],
       [0.02519246],
       [0.5597522 ],
       [0.9999763 ],
       [0.13173172],
       [0.070989  ],
       [0.999658  ],
       [0.999889  ]], dtype=float32)
```