Assignment 3

Goal

The main goal of this assignment is to be able to predict in real-time to which Twitch channel each chat message belonged. To achieve this, we divided the work into four main steps: data collection, text preprocessing, construction of a predictive model and live prediction, which will be all explained in detail in this report.

Files

- `A3-Save.py.ipynb`: Data collection notebook, explained in Step 1
- `A3-Pyspark.ipynb`: Preprocessing and model notebook, in Steps 2 and 3
- `A3-Predict.ipynb`: Live prediction notebook, explained in Step 4
- `messages.csv`: csv file with the messages used for training and validation
- `messages_test.csv`: csv file with messages used for testing

Libraries

For this assignment we decided to use almost exclusively the `pyspark` library.

```
# Step 1
import threading
import os, re
import pandas as pd
from operator import itemgetter
from pyspark.streaming import StreamingContext

# Steps 2 and 3
import pyspark
from pyspark.sql import SparkSession
import pyspark.sql.functions as f
from pyspark.ml.feature import Tokenizer, StopWordsRemover, StringIndexer,
HashingTF, IDF, Word2Vec
from pyspark.ml.classification import LogisticRegression, NaiveBayes
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

Step 1: Data collection

In order to collect the necessary data for this assignment we first needed to choose the two Twitch channels we would be using. We decided to look for channels that had a large number of followers and streamed regularly, therefore having more comments per stream. We also looked for channels that had different types of content and target audiences, as their comment section would probably look different.

The first channel we chose was **loltyler1**[1]. This channel has 5 million followers and streams every day, between 10 and 16 hours per day. Each stream has between 500 thousand and 1.5 million viewers. The main activity he carries out is playing League of Legends.

The second channel we chose was **jinnytty**[2]. This channel has more than 700 thousand followers and streams regularly, although not every day. Each stream has between 100 and 300 thousand viewers. She mainly streams outdoors and plays games, but also has other types of content.

To collect the data, we created a notebook called `A3-Saving.ipynb` by using the proposed code in the original `spark_streaming_example_saving.py.ipynb` file, although changing some of the parameters. As the data collected is saved in multiple folders, files and is formatted as a JSON dictionary, we created a csv file with all the messages to make it more readable. To do this, we iterated through the different directories to go through the "part-00000" documents and retrieve the usernames, channels and messages while creating a dataframe, then exported as a csv. We collected a total of 107,013 messages in different days. 75,311 messages are used for training and validation, and 31,702 for testing.

There is a possibility that running the second part of the notebook results on an error (see image). This is because at least one of the "part-00000" files inside the folders is empty. It is usually the last two folders. To solve this, we can simply delete those folders and run again this part of the code.

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-9-2f8af983d0f9> in <module>
     28                 if (str(file).find(fileType) != -1) and (str(file).find('crc') == -1):
     29                     df = pd.DataFrame(pd.read_json(file, orient='records', lines=True, encoding = 'ISO-8859
-1'))
---> 30                     df.drop(df.columns[[0]], axis = 1, inplace = True)
     31                     dfAll = pd.concat([dfAll, df])
     32                     print(dfAll.shape)

~/opt/anaconda3/lib/python3.8/site-packages/pandas/core/indexes/base.py in __getitem__(self, key)
   5053             key = np.asarray(key, dtype=bool)
   5054
-> 5055         result = getitem(key)
   5056         # Because we ruled out integer above, we always get an arraylike here
   5057         if result.ndim > 1:

IndexError: index 0 is out of bounds for axis 0 with size 0
```

[1] https://www.twitch.tv/loltyler1
[2] https://www.twitch.tv/jinnytty

Step 2: Text preprocessing

Text preprocessing consists of a series of steps that are needed in order to make the data suitable for a prediction model. The order of these steps is very important, as it depends on which libraries are used (`pyspark` vs. `scikit-learn`).

Firstly, we substituted the categorical channel variable by a numeric target. In this case, **jinnytty** is assigned label 0 and **loltyler1** label 1. Secondly, we decided to create a new variable with both the username and the message. As we assumed that the target audiences of the channels are different, keeping the usernames could be useful in the training and prediction steps. Whether the username is finally kept or not will be discussed in Step 3.

```
# Create label variable
df = StringIndexer(inputCol='channel',
outputCol='label',handleInvalid='keep') .fit(df).transform(df)

# Create column with both the username and the message
df = df.withColumn('message0', f.concat(f.col('username'), f.lit(' '),
f.col('message')))
```

Then, we removed symbols, extra spaces and missing values using `regex_replace` in order to clean the text. Next, we used the `Tokenizer` function to tokenize the text, which also makes it lowercase. Lastly, we removed stopwords with `StopWordsRemover`. As we said, the order of these steps is important as with `pyspark` it is necessary to have the words tokenized to be able to remove the stopwords, while with `scikit-learn` it is the other way around.

```
# Remove symbols
df = df.withColumn("words1", f.regexp_replace(f.col("message0"),
"[\$#,<>+@=?!]", ""))

# Remove extra spaces
df = df.withColumn("words2", f.regexp_replace(f.col("words1"), "  +", " "))

# Remove missing values
df = df.dropna()

# Tokenize (also makes everything lowercase)
tokenizer = Tokenizer(inputCol="words2", outputCol="words3")
df = tokenizer.transform(df)

# Remove stopwords
remover = StopWordsRemover(inputCol="words3", outputCol="words")
df = remover.transform(df)
```

After this preprocessing step, we are left with a pyspark dataframe with three columns: `label`, `message0` (or `message`, depending on whether we keep the username or not) –the original message and username– and `words` –with the tokenized words after all preprocessing–.

```
+-----+--------------------------------------------+--------------------------------------------------+
|label|message0                                    |words                                             |
+-----+--------------------------------------------+--------------------------------------------------+
|0.0  |zay_ih yyjPog Another Arcade                |[zay_ih, yyjpog, another, arcade]                 |
|0.0  |kipseu take the cable car up there YEP      |[kipseu, take, cable, car, yep]                   |
|0.0  |yvan_nix POGCRAZY  arcade                   |[yvan_nix, pogcrazy, arcade]                      |
|1.0  |darpyyyy monkerS                            |[darpyyyy, monkers]                               |
|1.0  |utamago NA > KR KEKW NA > KR KEKW NA > KR KEKW|[utamago, na, kr, kekw, na, kr, kekw, na, kr, kekw]|
+-----+--------------------------------------------+--------------------------------------------------+
only showing top 5 rows
```

Step 3: Build predictive model

TF-IDF and Word2Vec

To vectorize the data we used TF-IDF and Word2Vec. Whereas with `scikit-learn` we could directly use `TfidfVectorizer`, in `pyspark` this is done in two steps: `HashingTF`, which maps a sequence of terms to their term frequencies, and `IDF`, which calculates the inverse document frequency. One of the main differences between them is that Word2Vec takes into account the context of the words, while TF-IDF does not. Both outputs are saved in new columns in the dataframe. As we said earlier, we tried with and without the username,

```
# TF-IDF
hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=20)
featurizedData = hashingTF.transform(df)
idf = IDF(inputCol="rawFeatures", outputCol="tfidf")
idfModel = idf.fit(featurizedData)
df = idfModel.transform(featurizedData)

# Word2Vec
word2Vec = Word2Vec(vectorSize=10, minCount=0, inputCol="words",
outputCol="w2v")
w2vmodel = word2Vec.fit(df)
df = w2vmodel.transform(df)
```

```
+-----+--------------------+--------------------+--------------------+--------------------+
|label|            message0|               words|               tfidf|                 w2v|
+-----+--------------------+--------------------+--------------------+--------------------+
|  0.0|zay_ih yyjPog Ano...|[zay_ih, yyjpog, ...|(20,[6,11,17],[1....|[0.03237535967491...|
|  0.0|kipseu take the c...|[kipseu, take, ca...|(20,[14,15,17],[1...|[-0.2900261521339...|
|  0.0|yvan_nix POGCRAZY...|[yvan_nix, pogcra...|(20,[8,11,15],[1....|[0.01768894338359...|
|  1.0|    darpyyyy monkerS| [darpyyyy, monkers]|(20,[4,6],[1.9376...|[0.07023973017930...|
|  1.0|utamago NA > KR K...|[utamago, na, kr,...|(20,[10,12,17],[5...|[0.23133342787623...|
+-----+--------------------+--------------------+--------------------+--------------------+
only showing top 5 rows
```

Split train and validation sets

```
df_train, df_val = df.randomSplit([0.7, 0.3], seed = 234)
```

```
Training Dataset Count: 52864
Validation Dataset Count: 22447
Test Dataset Count: 31702
```

Models

We decided to compare three different models to check which one had a higher accuracy: logistic regression with TF-IDF, logistic regression with Word2Vec and Naive Bayes with TF-IDF. The full code and output is available in the `A3-Pyspark.ipynb` notebook.

At first, we used the data from `messages.csv` and split it into train and validation sets to see how the model would behave. We tried this two ways: deleting and keeping the username. In both cases, the logistic regression with Word2Vec worked has a higher accuracy. Keeping the username increased the accuracyfor Word2Vec, while it remained stable for TF-IDF.

In addition, we knew that the data we would be getting in the live prediction would be completely new. Not only new, but different users would be commenting, and probably the streamers would be doing different activities than when we collected the original data, therefore the topics discussed in the chats could also differ. Therefore, we collected new data to use as test set in `messages_test.csv`. Again, the highest accuracy values were seen in logistic regression with Word2Vec. The Naive Bayes model has worse performance than the other two models, and keeping the username or removing it doesn't have a big impact either, so it is not a useful model in this case. Therefore, the obvious choice was to use this model with username.

	Logistic regression with TF-IDF			**Logistic regression with Word2Vec**			**Naive Bayes with TF-IDF**		
	train	**valid**	**test**	**train**	**valid**	**test**	**train**	**valid**	**test**
Without username	0.641	0.640	0.619	0.676	0.670	0.623	0.605	0.609	0.575
With username	0.637	0.641	0.617	0.813	0.815	**0.764**	0.611	0.611	0.582

```
# Apply logistic regression model
lr_w2v = LogisticRegression(featuresCol = 'w2v', labelCol='label').fit(df_train)

evaluator_acc = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="accuracy")

# Train Results
train_results_w2v = lr_w2v.evaluate(df_train).predictions
accuracy = evaluator_acc.evaluate(train_results_w2v)

# Validation Results
val_lr_w2v = lr_w2v.transform(df_val)
accuracy = evaluator_acc.evaluate(val_lr_w2v)

# Test Results
pred_lr_w2v = lr_w2v.transform(df_test)
accuracy = evaluator_acc.evaluate(pred_lr_w2v)
```

```
+-----+----------+--------------------------------------------------------------+
|label|prediction|probability                                                   |
+-----+----------+--------------------------------------------------------------+
|0.0  |0.0       |[0.7927476905908345,0.20725230940875863,4.0688827966079027E-13]|
|0.0  |0.0       |[0.7541796706691013,0.24582032932888845,2.010404189421244E-12] |
|0.0  |0.0       |[0.9881052292058464,0.011894770763522123,3.063144260542358E-11]|
|0.0  |0.0       |[0.7451198215158286,0.2548801784822799,1.891342121408406E-12]  |
|0.0  |0.0       |[0.9677191430591785,0.032280856939991706,8.298317777190093E-13]|
+-----+----------+--------------------------------------------------------------+
only showing top 5 rows

Train set accuracy = 0.8130864104116223
+-----+----------+--------------------------------------------------------------+
|label|prediction|probability                                                   |
+-----+----------+--------------------------------------------------------------+
|0.0  |0.0       |[0.6167718688500925,0.38322813114883925,1.0683698235591442E-12]|
|0.0  |0.0       |[0.6239824723086594,0.37601752768882296,2.517694632788934E-12] |
|0.0  |0.0       |[0.7805626873724874,0.21943731262549288,2.0197937566171048E-12]|
|0.0  |1.0       |[0.41336987766049177,0.5866301223382321,1.2761062509628436E-12]|
|0.0  |0.0       |[0.9809002353617897,0.019099764635763895,2.446262751778833E-12]|
+-----+----------+--------------------------------------------------------------+
only showing top 5 rows

Test set accuracy = 0.8148973136722056
+-----+----------+--------------------------------------------------------------+
|label|prediction|probability                                                   |
+-----+----------+--------------------------------------------------------------+
|0.0  |0.0       |[0.5565197277410503,0.44348027225781117,1.1385856927363257E-12]|
|1.0  |1.0       |[0.4654594165829706,0.5345405834156725,1.357051257666222E-12]  |
|0.0  |1.0       |[0.4783296167808523,0.5216703832181151,1.03269021004537E-12]   |
|0.0  |0.0       |[0.9715862574298688,0.028413742565301336,4.829887747295278E-12]|
|0.0  |0.0       |[0.9999876358177159,1.2364130074522217E-5,5.2209602126535314E-11]|
+-----+----------+--------------------------------------------------------------+
only showing top 5 rows

Test set accuracy = 0.7649675099362816
```

Lastly, we saved this model to be able to import it to the notebook for step 4.

```
lr_w2v.save("A3.model")
```

Step 4: Live predicting

For this fourth step we used the structure of the proposed notebook `spark_streaming_example_predicting.py.ipynb`. The original idea was to create a pipeline with the preprocessing and vectorization made in steps 2 and 3, however, we run into multiple errors which we weren't able to solve, especially when trying to add Word2Vec to the pipeline. Therefore, we ended up adding all the preprocessing steps to the notebook, although we knew that this was not ideal, as the model will be relearn every time we get a new dataframe. This is something that can and should be improved in the future. Then, we imported the logistic regression model from step 3. This way, we managed to get new comments and their respective predictions. Unfortunately, we were not able to try this prediction notebook when both streamers were active at the same time, so the output in the following image is not very useful, as all comments were posted to the same channel.

```
+--------------+-----+--------------------+--------------------+----------+
|       message|label|       rawPrediction|         probability|prediction|
+--------------+-----+--------------------+--------------------+----------+
|    WAYTOODANK|  0.0|[8.95171278152332...|[0.55265136062115...|       0.0|
|           lol|  0.0|[9.03273209312361...|[0.58421914092609...|       0.0|
|   Kappa do it|  0.0|[9.12555048295419...|[0.66344502390647...|       0.0|
|          LULW|  0.0|[9.17968738325061...|[0.63921715099595...|       0.0|
|    PepoDetect|  0.0|[8.95737463362953...|[0.45092616311109...|       1.0|
| -1 phone KEKW|  0.0|[8.94319720710894...|[0.51618760177720...|       0.0|
|          KEKW|  0.0|[9.09520783285095...|[0.59020230862581...|       0.0|
|fast lane KEKW|  0.0|[9.11252503300617...|[0.62883163252057...|       0.0|
|        KEKBye|  0.0|[8.96921881966335...|[0.53022916467239...|       0.0|
|          KEKW|  0.0|[9.02568570365385...|[0.55591492413571...|       0.0|
|     PepePoint|  0.0|[9.09112176287228...|[0.66508322716081...|       0.0|
|      OMEGALUL|  0.0|[8.99260740947693...|[0.52179265874539...|       0.0|
|         Kappa|  0.0|[9.00506524138595...|[0.60473150711288...|       0.0|
|           WRF|  0.0|[8.83113280455416...|[0.38670114667487...|       1.0|
|     PepeLaugh|  0.0|[8.97158975064368...|[0.56006841034234...|       0.0|
|     PepeLaugh|  0.0|[8.93637695825765...|[0.50700023387909...|       0.0|
|         uh oh|  0.0|[9.08385492780065...|[0.60551415935059...|       0.0|
|   NotLikeThis|  0.0|[9.00370533638617...|[0.49298761941781...|       1.0|
|     PepeLaugh|  0.0|[8.89031332180363...|[0.53752634304209...|       0.0|
|           LOL|  0.0|[8.97761315383213...|[0.53953083261603...|       0.0|
+--------------+-----+--------------------+--------------------+----------+
only showing top 20 rows
```