KU LEUVEN

Advanced Analytics in a Big Data World

Group 32

Joani Radhima [r0823581] - joani.radhima@student.kuleuven.be
Michalopoulos Tryfon [r0829272] - tryfon.michalopoulos@student.kuleuven.be
Yash Mahesh Mathradas [r0823209] - yashmahesh.mathradas@student.kuleuven.be
Nazmi Serkan Atalay [r0825341] - nazmiserkan.atalay@student.kuleuven.be

Assignment 1

Objective

The objective of the 1st assignment is to train a model that can classify insurance claims as fraudulent or not. The training data consists of claims during the 2017 year and the test data has claims from year 2018. The dataset consists of 80+ features, including information about the claim, the driver, the expert that examined the claim, the location, etc.

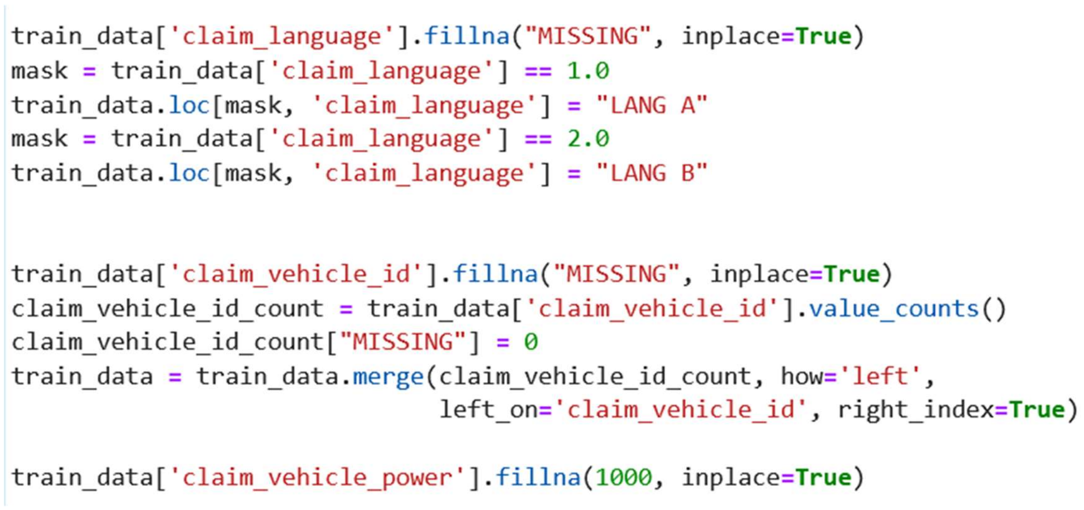
The model will be assessed by the total claim amount of fraudulent cases, for the 100 cases the model predicts are the most probable to be fraudulent. This makes for an interesting optimization criterion, as it also considers variables that are not easily used in the training process of algorithms like Random Forests. In addition, it makes ROC curves and thresholds ineffective since the cases are considered relative to each other and not relative to a predetermined threshold. The large class imbalance in the dataset is not an issue, since we do not care about classification but just probability.

Data Cleaning and pre-Processing

Since the data are a product of a manual process, they are not ready for the model but need cleaning and preprocessing. In addition, a lot of empty values exist that need to be encoded. The main strategy used for these steps was:

1. Empty categorical values are encoded as “MISSING”.
2. Empty numerical values are encoded using an extreme outlier. For example, if a case does not have the vehicle’s year of first use or horsepower, values of 1800 and 5000 are used, respectively. This option was preferred to imputation with the mean/median and the creation of a new column encoding whether the data were missing or not, since a Random Forest can easily deal with outlying values and thus, we could encode everything in only 1 column.
3. High cardinality variables are encoded using frequency encoding, with missing data getting a frequency of 0. New values in the test data are treated as missing and are given a frequency of 0 as well.
4. For variables that we think are not very useful, such as 3rd party, 2nd and 3rd attributes, we encode them using 0 and 1, based on if they exist or not. The thought is that it is important if these values exist or not, the actual values should not be important enough to warrant further detail.
5. Temporal data was featurized into Month, Day, DayOfWeek, (Date claimed - Date Happened), DaysLeftInContract, DaysSinceContract, etc. Model validation showed that including these features made predictions better, however it worsened the metric we are optimizing for (maximum return for the 100 first cases) and thus these features were not included in the model.

For the validation of the model the train-test split of the data was done before starting the cleaning procedure, to ensure that no data leakage happens. This leads to a more robust procedure. Code fragments are presented below. Using this pipeline we ensure that the data partitions are separate from each other, missing or new values are handled by encoding them as either 0, “MISSING”, or using an outlying value that our model will easily partition away.



```python
train_data['claim_language'].fillna("MISSING", inplace=True)
mask = train_data['claim_language'] == 1.0
train_data.loc[mask, 'claim_language'] = "LANG A"
mask = train_data['claim_language'] == 2.0
train_data.loc[mask, 'claim_language'] = "LANG B"


train_data['claim_vehicle_id'].fillna("MISSING", inplace=True)
claim_vehicle_id_count = train_data['claim_vehicle_id'].value_counts()
claim_vehicle_id_count["MISSING"] = 0
train_data = train_data.merge(claim_vehicle_id_count, how='left',
                              left_on='claim_vehicle_id', right_index=True)

train_data['claim_vehicle_power'].fillna(1000, inplace=True)
```

Modeling and Validation

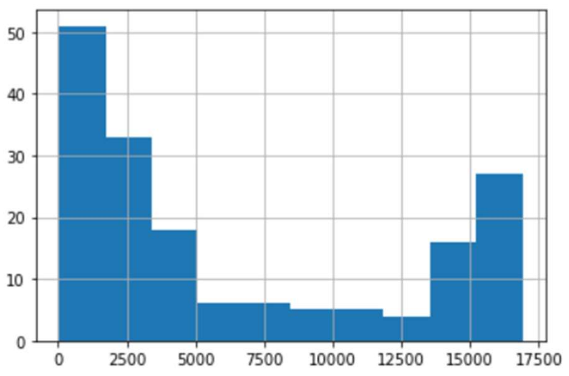
During the modeling and validation procedure, we set up a pipeline where we split the train data into a training and a validation split. We preprocess the resulting dataframes using the pipeline described above and compare different models based on the metric described, on the validation set. In order to optimize for the specific metric needed, we need to work in a smart way since Random Forests can optimize classification accuracy and they do not easily extend to custom metrics.

The first important realization is that there is a large discrepancy in the claim amount between the cases in the dataset. More specifically, the distribution of the amount for fraudulent cases is bimodal, with a concentration of cases with low amounts, less than 5000 euros, and another with high amounts, more than 13000. Values in between are rare and this indicates that these cases are not normally distributed and are driven by different phenomena. We want to maximize our return and thus, naturally, we want our model to learn to predict the high value cases.

To find out what the difference is, we split the fraudulent cases by their median into two groups, high and low value cases. We then train two Random Forest classifiers, each one containing fraudulent cases from only one of these two categories. In the end, we compare the feature importances by subtracting the feature importance vectors and looking at the highest and lowest values. We find out that cases with a low claim are mostly car accidents, while high value cases are mostly car theft and fire claims. In addition, we learn that the temporal variables are more important for the low value claims, which explains why including them in the model lead to better classification but worse returns.

Comparing the claim values we see that 1 correctly classified high value claim of 15.000 euros, like a fake car theft claim, nets a return equal to 15 correctly classified fraudulent car accidents of 1.000 euros each. Understanding this, we realize that it is better for our classifier to have a 20% precision on high value cases, than it having a 100% precision on low value cases. For that reason, our strategy will be centered around making sure that the model does not learn to classify low claim cases and focuses on high value ones instead.

```
train_data[train_data.fraud=='Y'].claim_amount.hist();
```



Another realization is that this is not a ROC/threshold problem. What we care about is that the cases the model predicts as most probable are high value ones. Thus, generally oversampling fraudulent cases does not help our performance. What we want is to oversample high value cases, forcing our classifier to learn one group over the other. We proceed to do exactly that:

1. We choose a threshold under which we discard all fraudulent cases.
2. We choose other thresholds over which we duplicate the fraudulent cases, leading to some being doubled, some tripled, and some quadrupled.

The code to do that is presented below. The validation procedure selected this model and methodology as the best performing one. We tested many other alternatives, including SMOTE, ADASYN, Gradient Boosting, using all the data, and ignoring all low values, however most results were noisy and way too dependent on correctly classifying 1 more high value claim. Thus, we selected a simple RF classifier with the duplicate high value cases, using the code we show below, as our final model. The results prove that our strategy was robust, as we rose in the rankings once the final dataset was revealed despite trying out different models and using the one with the best result. In other words, even though we chose the best model on the test set it still had not overfit, proving the robustness of our entire pipeline.

```
# target and covariates

indices = clean_data[(clean_data['fraud'] == 'Y') & (clean_data['claim_amount'] < 1000)].index
double = clean_data[(clean_data['fraud'] == 'Y') & (clean_data['claim_amount'] > 2000)]
triple = clean_data[(clean_data['fraud'] == 'Y') & (clean_data['claim_amount'] > 5000)]
quadruple = clean_data[(clean_data['fraud'] == 'Y') & (clean_data['claim_amount'] > 10000)]


oversampled_data = pd.concat([clean_data, double, triple, quadruple])

# target and covariates
y = oversampled_data.drop(indices)['fraud']
X = oversampled_data.drop(indices).drop(columns=['fraud', 'claim_amount'])
```

Assignment 2

Introduction

In the second assignment the goal is to build a deep learning model that given an image can as accurately as possibly predict the tags that the uploader has assigned to it. The framework that was used is *fasta.ai* by Jeremy Howard and Rachel Thomas, which basically is a layer on top of pytorch that wraps up pytorch's low level code and functionality so that we could focus more on the problem itself.

Reforming (slightly) the task

The dataset at our disposal consists of more than 11696 images scraped from Instagram, that have the hashtag *#recipe* and each of these images is expected to have at least one tag related to its content which is not always the case since 3822 images have no tags assigned to them reducing the total amount of useful images to 7874. The total unique values of labels that were found in the dataset was 97, the distribution of which can be seen in *Figure 1*, but heavy preprocessing was applied on them as an attempt to reduce that number to at least half its value. The raw labels were, at the first phase of the preprocessing, lemmatized so that words like *cake* and *cakes* were grouped together that led to the count of distinct values of the labels to be significantly reduced to a total of 62 labels. As a side effect of the lemmatization process, tags like *vegan* and *vegetarian* that are semantically different were also grouped together. Taking into account the ratio of the total count of distinct labels and the total number of images it felt like we should reduce that number even further; this was actually confirmed when we run a baseline model, after doing some minor image-augmentation of course, in order to get an idea of what to expect as default accuracy and the results where a lot worse than what was initially expected. Other than the problem itself being extremely difficult, if solvable at all, the target feature, the tag label, is as noisy as it could be, since it is highly dependent on the uploader's perception or likings or personality in general, the overall trend that takes place at the specific place and time that the image was uploaded etc. Simply put, there is no guarantee that certain that the tags that the uploader chooses are the 'correct' ones or that there is an underlying pattern associated with these tags that the deep learning model will be able to uncover. Further grouping the labels in a way that is as balanced as possible, not only by lemmatizing them, but also grouping them together based on their meaning, even though sometimes they used to be quite different, and eventually throwing away the labels with low count, that were considered to add noise to an already rather noisy dataset, we ended up working with 24 labels, the distribution of which is shown in *Figure 2.*

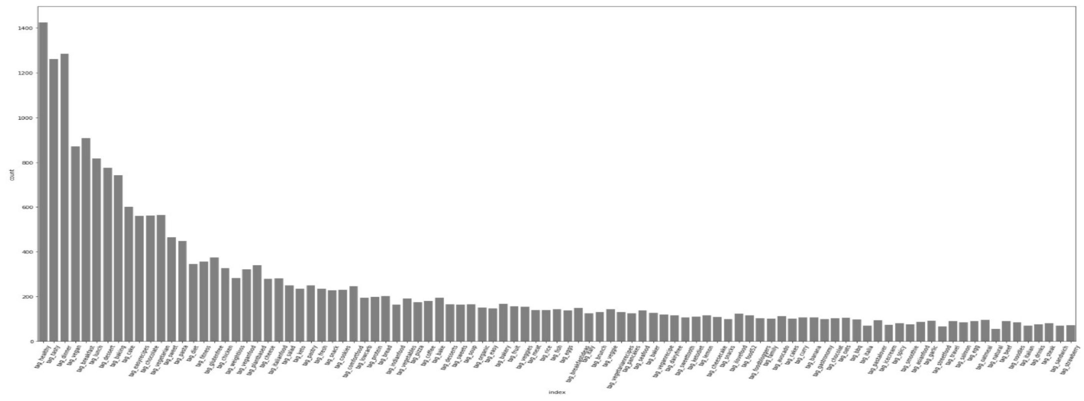


Figure 1. Distribution of distinct labels in the dataset

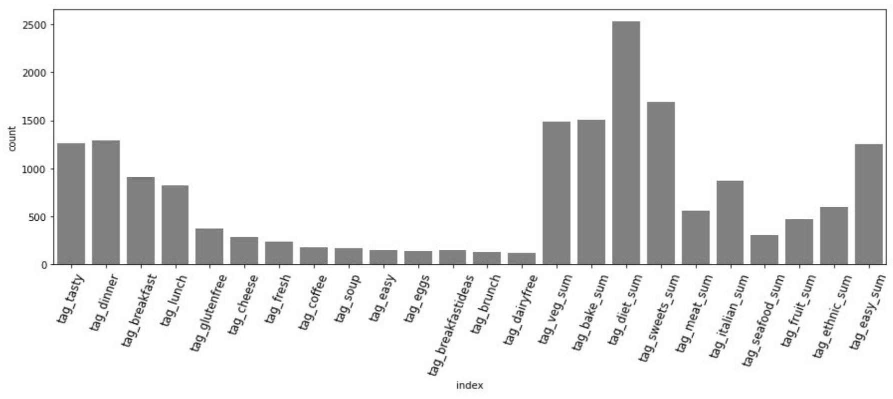


Figure 2.Distribution of distinct labels after lemmatizing and groupping tags together. Tags having the suffix 'sum' indicates that they were the product of the groupping procedure.

Data preprocessing

There are a few preprocessing steps that need to be done in order get our data ready for the *fastai* model. *Fastai* offers a convenient class (or *Container* as the documentation states) that wraps up all the steps needed for this, the *DataBlock.* Other than trivial procedures like properly loading up the dataset and splitting it to train and validation set, the *DataBlock* make the distinction between two transformations that are being applied on the dataset, the *item_tfms* which is applied on an item basis and the *batch_tfms,* with which we can define the augmentations to be applied on the dataset at a batch level.

In short, he dimensionality of the input space has to be implicitly defined, meaning that all images that are fed into the fastai model need to have the same total number of features, which is equal to the number of pixels an image has. However, it is expected that not all of the pictures have the same pixel count which is taken care of with *DataBlock*'s parameter *item_tfms.* Instead of just stretching or squeezing the data so that it fits some specified height and width pixel count, we used the *RandomResizedCrop* function that picks a random scaled crop of an image and resizes it to the size specified, 64 in our case. Such transformations are shown in *Figure 3.*

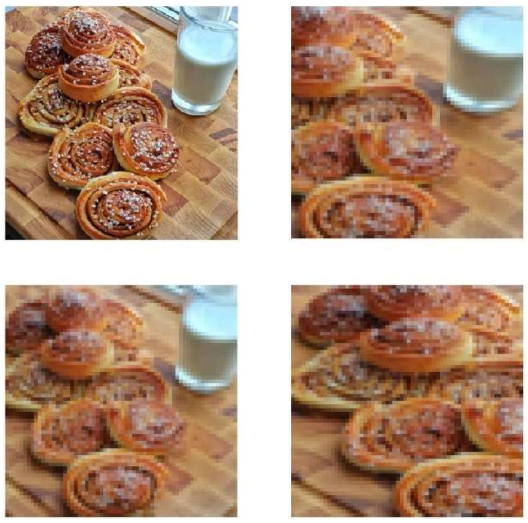


Figure 3. Three output images of RandomResizedCrop for the image at the top left.

As for the *batch_tfms,* it was clear from the start that heavy augmentations would have to be applied given the small size and the complexity of the data set. In a way it can be thought of artificially expanding the size of our training dataset, which as already mentioned above is quite limited, and thus can improve generalization. The augmentations chosen for the project are:

- Randomly flip an image with $p = 0.5$ probability
- Rotate the image and fill in with black pixels the unallocated parts of the rotated image
- Randomly adjust the overall lighting of the image at a different level.

The model

Training a model from scratch for this project would be infeasible, so we chose the 'transfer learning' path and trained on a couple of the most popular pretrained models for computer vision tasks, out of which the *resnet18* seemed to fit our needs. What initially made the *resnet* stand out from the rest of the models that existed at the time period that it came out, I the way that it dealt with the vanishing gradients; it introduced the idea of the so-called "identity shortcut connection" that skipped on or even more layers. It was claimed that going deeper with this architecture would not lead to degrading of the net as compared to swallower ones of the same architecture, since was possible to just stack identity mappings that would essentially make it at least equivalent to them. This means that, theoretically speaking at least, a *resnet50* architecture should perform at least equally with the *resnet18* that we chose as the final model of this project, given that there is enough computational power to train the deeper architecture. Using the *fastai* framework on could access the pretrained model's architecture with

```
model.summary()
```

and parameter weights with

```
[layer for layer in model.named_parameters()]
```

More interesting than looking at raw tensors would be to visualize what the feature maps and network activations corresponding to the first convolutional layer of the network, shown in the following figure.

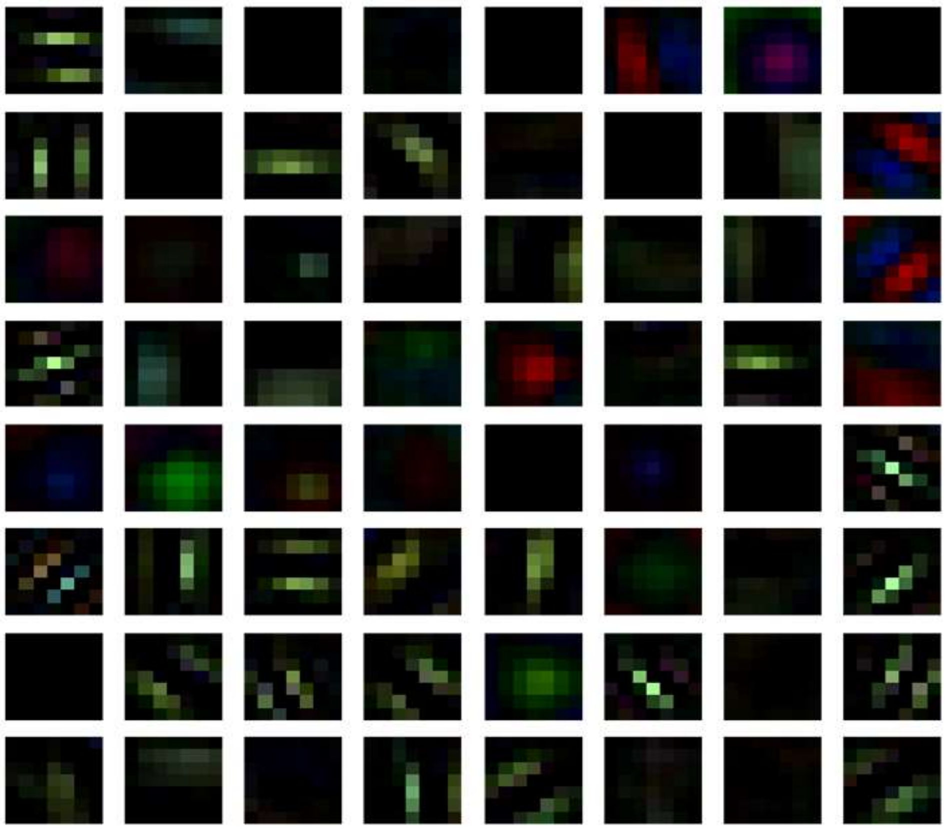


Figure 4. Feature maps / Filters of the 'conv1' layer of the pretrained resnet18. Takes in 3 channels and outputs 64 to the next layer, with a kernel size of 7, stride 2 and padding 3.

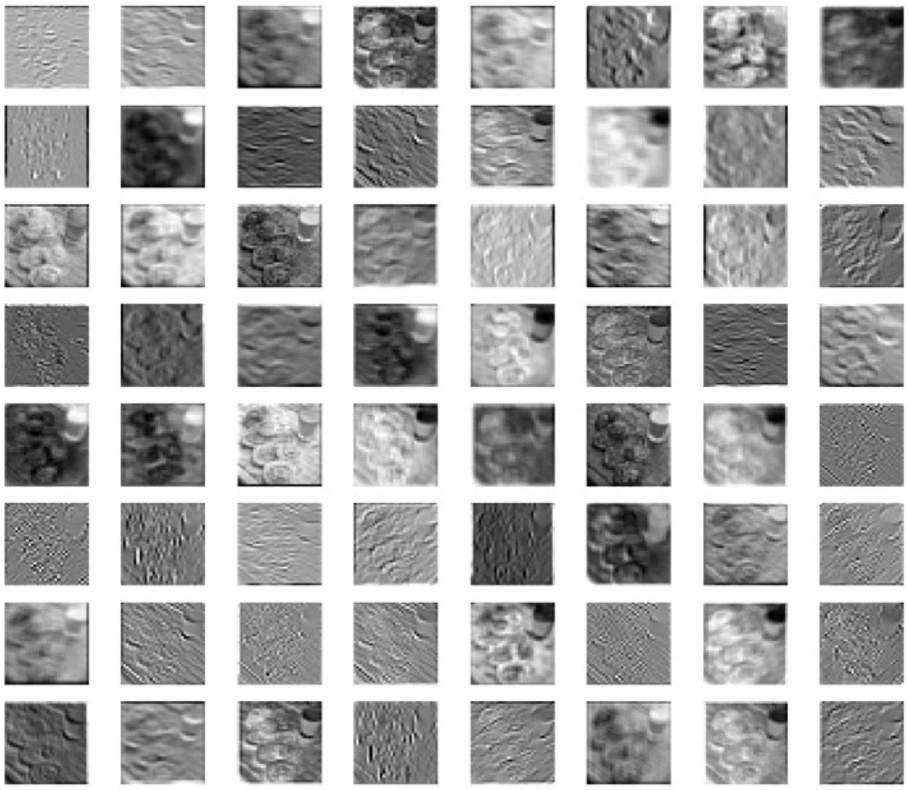


Figure 5. Outputs from the 'conv1' layer of the pretrained resnet18. Takes in 3 channels and outputs 64 to the next layer, with a kernel size of 7, stride 2 and padding 3. The same image as that of Figure 3 was fed through the layer.

Training

The essence of "transfer learning" lies in the fact that a model, having been trained for quite a long time on millions of images that it has been fed with, it has "learned" meaningful patterns and thus, it can exploit the knowledge gained from a previous task to improve generalization for another. In fact, only the very last few fully connected layers are needed to be trained, with the rest of the network being "frozen". As a sidenote, these last few layers are implicitly transformed so that the output of the last layer coincides with the count of classes, when we define the model as

```
model = cnn_learner(dls, resnet18, pretrained=True)
```

by taking into account the target variable that was previously defined and is now stored in the *dls* parameter. In other words, although our model might have been trained in a dataset, let's say cifar-10 that consists of 10 classes, we could as well use it to on completely different dataset and make predictions over a different number of classes; only thing that is needed is that the last layer (typically with softmax activation) is redefined so that the output is what it needs to be and the pretrained model gets fine_*tuned.* This is exactly the strategy we followed for this project with the pretrained *resnet18* architecture.

Possibly the most important hyper-parameter to be tuned for a neural net is the *learning rate* (*LR*); Picking extreme values would lead to a net that is either trained too slowly for a low value of *LR* or make the loss fluctuate around the optimal value, without ever getting closer to it or even causing outright divergence. *Fastai* offers a quite convenient function, *model.lr_find( ),* which seeks for a well-performing candidate for the *LR* parameter, based on the paper by *Leslie N. Smith - Cyclical Learning Rates for Training Neural Networks.* The output of the function is shown in *Figure 4.* It should be noted that once a pretrained model is loaded in, fastai automatically freezes all the pretrained layers, which means that the $LR \approx 0.02$ that was found previously (one of the two that are reported back by the *lr_funtcion( ) that corresponds to the steepest loss*) refers to the model when all the pretrained layers except for the classification-specific ones at the very end are frozen. So, we start training the model, at the current frozen state, with the suggested *LR,* so that only the fully connected layers at the end, that would otherwise be too difficult to train, are tuned for our task. The results for 15 epochs, that were requested are shown in *Figure 5.*

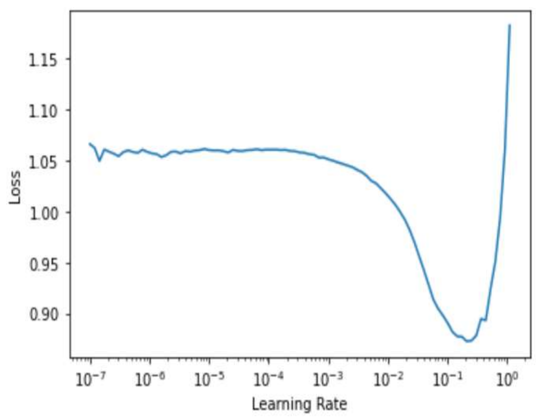
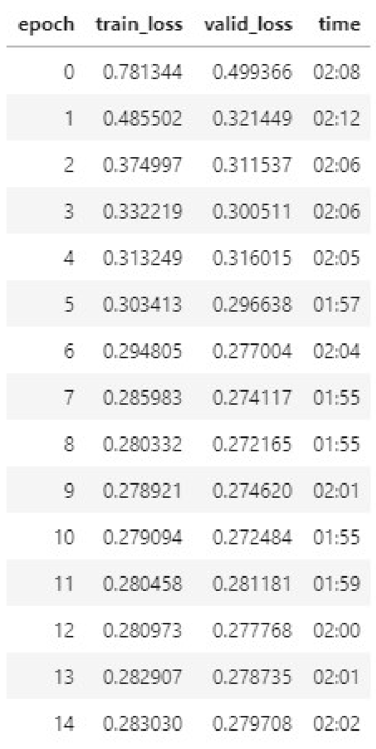
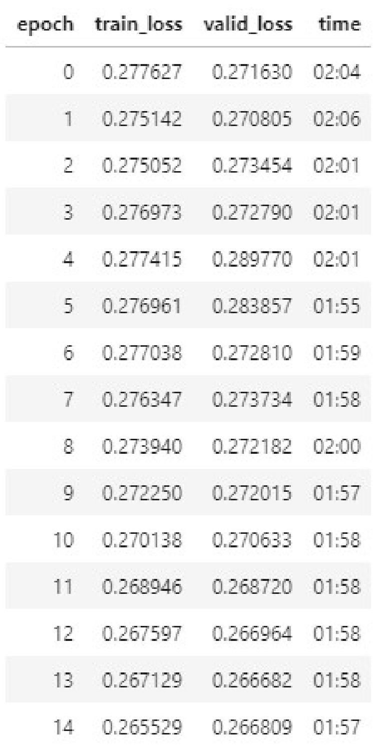


Figure 6. model.lr_find( ) visual output



epoch	train_loss	valid_loss	time
0	0.781344	0.499366	02:08
1	0.485502	0.321449	02:12
2	0.374997	0.311537	02:06
3	0.332219	0.300511	02:06
4	0.313249	0.316015	02:05
5	0.303413	0.296638	01:57
6	0.294805	0.277004	02:04
7	0.285983	0.274117	01:55
8	0.280332	0.272165	01:55
9	0.278921	0.274620	02:01
10	0.279094	0.272484	01:55
11	0.280458	0.281181	01:59
12	0.280973	0.277768	02:00
13	0.282907	0.278735	02:01
14	0.283030	0.279708	02:02

Figure 7. Fastai output while at the 'frozen' state



epoch	train_loss	valid_loss	time
0	0.277627	0.271630	02:04
1	0.275142	0.270805	02:06
2	0.275052	0.273454	02:01
3	0.276973	0.272790	02:01
4	0.277415	0.289770	02:01
5	0.276961	0.283857	01:55
6	0.277038	0.272810	01:59
7	0.276347	0.273734	01:58
8	0.273940	0.272182	02:00
9	0.272250	0.272015	01:57
10	0.270138	0.270633	01:58
11	0.268946	0.268720	01:58
12	0.267597	0.266964	01:58
13	0.267129	0.266682	01:58
14	0.265529	0.266809	01:57

Figure 8. Fastai output while at the 'unfrozen' state

As expected, the *train_loss* constantly keeps decreasing till a point, around the 9th epoch, where it reaches a plateau, while the *valid_loss* keeps decreasing till the 15th epoch, where it shows some uprising trend (although it is not shown here). That is an indication that the model has started learning the training images' specific features and its ability of generalizing to unseen images starts decreasing, meaning that it has started overfitting the data. At that point we stop the training procedure, "unfreeze" the model, find a new *LR* that corresponds to this unfrozen state and retrain every single parameter of the *resnet18.* It was requested that the model would be trained for 15 epochs as well and the results of this second training part are shown in *Figure 6.*

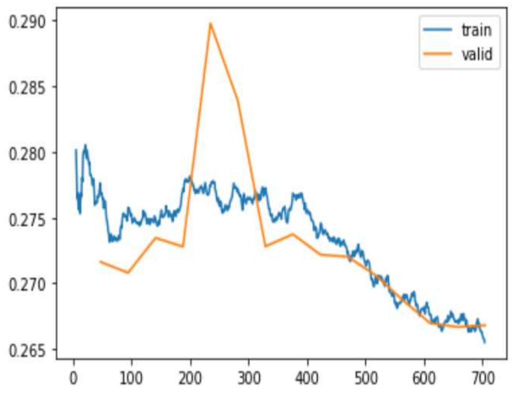


Figure 9. Loss vs images processed for the train and validation set.

The training process for the "unfrozen" state of the model seemed to plateau at around the 15th epoch for both the training and the validation loss, which could be considered as an indication that a larger *LR* should be used , but that led to no improvement in our case. A handy function offered by *fastai* that wraps up the steps of *i)* training the last layers of the frozen model for a specified number of epochs, *ii)* unfreezing the model and *iii)* training the rest of the network for a specified number of epochs with the use of *discriminative LR* is the

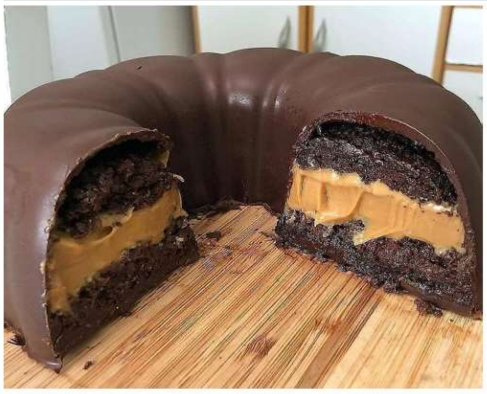
```
model.fine_tune(epochs=15, base_lr=lr_[0], freeze_epochs=15),
```

that was used to produce the outputs of *Figure 7, 8* and *9.*

The verdict

The model that accompanies this report was somewhat able to capture the underlying patterns of the data, given the nature of the problem, that even a human would have a hard time with and the hugely imbalanced and small-sized dataset. For demonstration purposes two images were fed into the model and the predicted tags are shown in the following Figures. For the prediction part of the project, we implemented a function that could easily be configured on the fly, although we could have used *fastai*'s implementation with appropriately tuned thresholds for classification. Moreover, the multilabel metrics offered by the *fastai,* like *fbeta_score*, *f1* and *recall*, did not seem to fit the needs of the task so we implemented and used one that has similarities with a *multilabel-Recall* kind of metric, but focuses on the dominant labels, completely ignoring the noise introduced by less important labels.

```
# Visualize the img_test (raw/unprocessed)
img_test.resize((500,400))
```

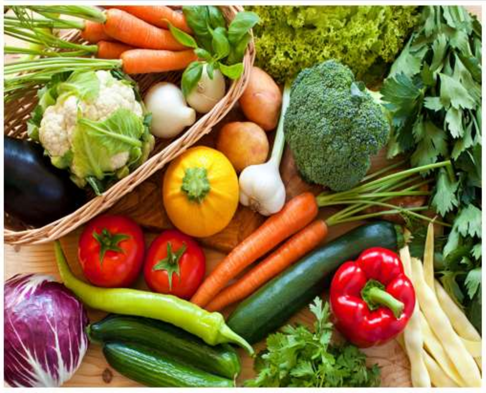


And our model's predictions:

```
my_predict_func(preds, request_number=2)
```

```
['tag_sweets_sum', 'tag_bake_sum']
```

```
# Visualize the img_test (raw/unprocessed)
img_test.resize((500,400))
```



And our model's predictions:

```
my_predict_func(preds, request_number=2)
```

```
['tag_diet_sum', 'tag_veg_sum']
```

Figure 10. Testing the final model on random images found on instagram.

Assignment 3

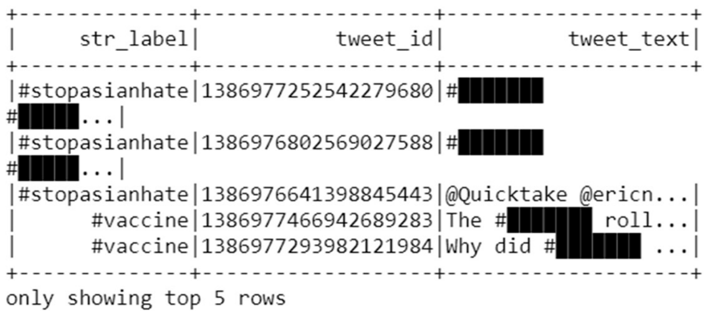
Objective

The goal of the task is to collect tweet data, train a model using the collected data and load the trained model to predict real-time streaming tweets using Apache Spark and Spark MLlib.

Data Acquisition & Loading

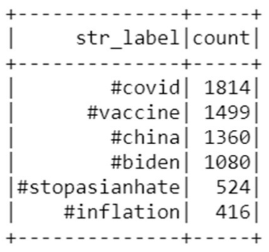
The initial step of the assignment was to collect enough data to train our model. A stream was setup for a few days that collected approximately 15000 tweets from the ("seppe.net", 7778) server which were then saved in JSON format. In order to load the tweets which are located in multiple folders we use following code.

```
tweets = spark.read.json("/path/tweets-*")
```



```
+--------------+-------------------+--------------------+
|     str_label|           tweet_id|          tweet_text|
+--------------+-------------------+--------------------+
|#stopasianhate|1386977252542279680|#[redacted]
#[redacted]...|
|#stopasianhate|1386976802569027588|#[redacted]
#[redacted]...|
|#stopasianhate|1386976641398845443|@Quicktake @ericn...|
|      #vaccine|1386977466942689283|The #[redacted] roll...|
|      #vaccine|1386977293982121984|Why did #[redacted] ...|
+--------------+-------------------+--------------------+
only showing top 5 rows
```

As seen in the figure, the data had three columns, each containing label, tweet_id and tweet_text. Our task was to use the tweet_text to predict the label associated with that text.



```
+--------------+-----+
|     str_label|count|
+--------------+-----+
|        #covid| 1814|
|      #vaccine| 1499|
|        #china| 1360|
|        #biden| 1080|
|#stopasianhate|  524|
|    #inflation|  416|
+--------------+-----+
```

The collected data that we have, has the following class frequencies as shown in the picture on the left. This data will then be split into a train and test set for training and testing our model performance.

Preprocessing

Firstly, we dropped the tweet_id column and removed the duplicate tweets. To make the text usable for the training our model, we must subject the text through some preprocessing steps for better results.

```
distinctDF = df.distinct()

tweets = tweets.drop('tweet_id')
```

Our next step is to apply the string indexer on our response variable. This provides an index to each class of our label.

```
labeler = StringIndexer(inputCol="label", outputCol="target")
```

Next, we tokenize our tweet_text, remove commonly occurring stop words. Then we use both count vectorizer and IDF feature to vectorize our text to make it suitable for training with our chosen algorithms

```
regexer = RegexTokenizer(inputCol='tweet_text', outputCol="tokens",
pattern="((https).+)|[^0-9a-z#+]+", minTokenLength=3)

stopworder = StopWordsRemover().setInputCol('tokens').setOutputCol('words')

vectorizer = CountVectorizer(inputCol='words', outputCol="countFeatures")

idf = IDF(inputCol='countFeatures', outputCol="features")
```

Training and Model Evaluation

Since we are dealing with a textual data, we decide that we train a few models using Logistic Regression (using Cross Validation to tune the hyperparameters) for our classification task.

Logistic Regression

```
lr = LogisticRegression(featuresCol='features', labelCol="target", maxIter=20)

pipeline = Pipeline(stages=[labeler, regexer, stopworder, vectorizer, idf, lr])
```

We first apply the pipeline including the logistic regression trainer and apply 4-fold cross validation to tune the hyperparameters.

Among our hyperparameters, the elastic net parameter chooses a balance between L-1 and L-2 penalization (i.e., either weighing more towards ridge or LASSO regression [shrinkage vs selection]). Lambda parameter controls the learning rate, we set a 3x4 grid containing all the possible combinations for our parameters.

```
paramGrid = (ParamGridBuilder()
   .addGrid(lr.elasticNetParam, [0.0, 0.2, 0.5, 0.8])
   .addGrid(lr.regParam, [0.001, 0.01, 0.1])
   .build())
```

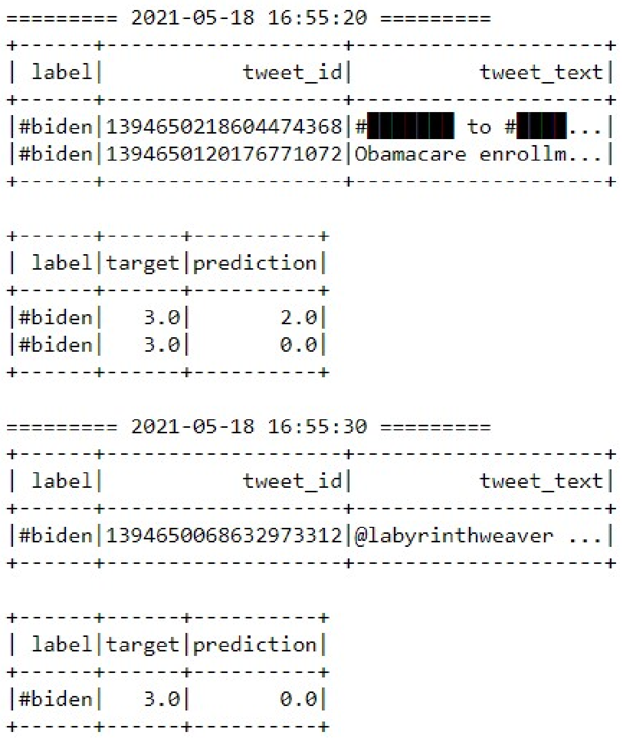
To evaluate the performance of our model, we use the F-1 score metric to quantify it.

```
evaluator = MulticlassClassificationEvaluator(labelCol="target",
predictionCol="prediction", metricName="f1")


crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=evaluator,
                          numFolds=4)

cvmodel = crossval.fit(trainDF)

predictions = cvmodel.transform(testDF)

evaluator.evaluate(predictions)
```

Best model turned out to be the model with elastic net parameter of 0.5 which represents both L1 and L2 regularization is applied partially. The learning rate is 0.01. This logistic regression achieved an accuracy rate of 63.22% on average.

Live Prediction

After tuning the parameters and obtaining the best performing cross validated model, we saved it to use for predictions in streaming data. Below, you can find the function loads the logistic regression model and predicts the tweet labels as it streams.



```
========= 2021-05-18 16:55:20 =========
+------+-------------------+--------------------+
| label|           tweet_id|          tweet_text|
+------+-------------------+--------------------+
|#biden|1394650218604474368|#[redacted] to #[redacted]...|
|#biden|1394650120176771072|Obamacare enrollm...|
+------+-------------------+--------------------+

+------+------+----------+
| label|target|prediction|
+------+------+----------+
|#biden|   3.0|       2.0|
|#biden|   3.0|       0.0|
+------+------+----------+

========= 2021-05-18 16:55:30 =========
+------+-------------------+--------------------+
| label|           tweet_id|          tweet_text|
+------+-------------------+--------------------+
|#biden|1394650068632973312|@labyrinthweaver ...|
+------+-------------------+--------------------+

+------+------+----------+
| label|target|prediction|
+------+------+----------+
|#biden|   3.0|       0.0|
+------+------+----------+
globals()['models_loaded'] = False
globals()['my_model'] = None

def process(time, rdd):
    if rdd.isEmpty():
        return

    print("========= %s =========" % str(time))

    # Convert to data frame
    df = spark.read.json(rdd)
    df.show()

    # Load in the model if not yet loaded:
    if not globals()['models_loaded']:
        # load in your models here
        from pyspark.ml.tuning import CrossValidatorModel
        globals()['my_model'] = CrossValidatorModel.load('cvmodel')
        globals()['models_loaded'] = True

    # And then predict using the loaded model:
    df_result = globals()['my_model'].transform(df)
    df_result.select('label', 'target', 'prediction').show()
```

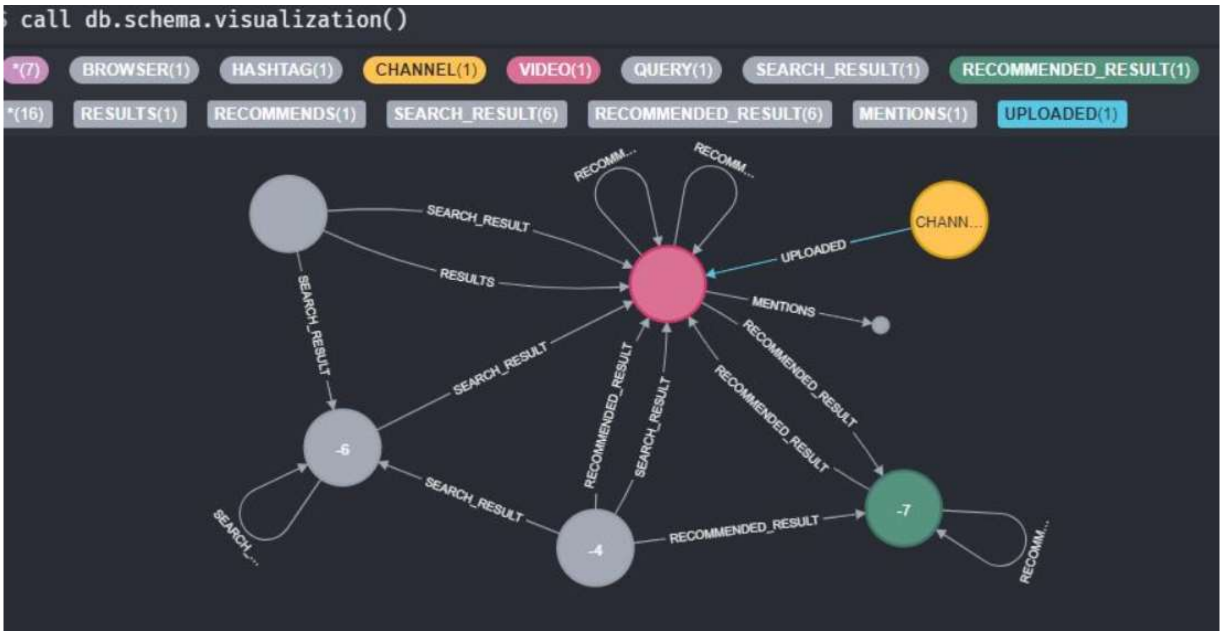
Python code are accessible in *saving*, *model* and *live prediction*.

Assignment 4

Objective

The objective of this assignment is to use network analysis and gain insights into the Youtube recommendation algorithm, specifically on whether the algorithm is biased for videos with misleading content, like conspiracy theories. The thought behind these claims is that controversial videos lead to greater engagement, as people might view them for amusement even if they disagree with the premise of the video. These theories claim that Youtube can lead people to a recommendation loop, where every video links to other controversial videos, spreading misinformation and helping with the rise of extremism in society.

The dataset consists of a mixture of videos, queries, recommendations, channels, and hashtags. Directed edges connect these nodes, as described in the figure below. We will query the data using Neo4J, a graph database best suited for network data, and analyze and visualize the results with Gephi, a tool created for that purpose.



```
call db.schema.visualization()
```

Thought Process

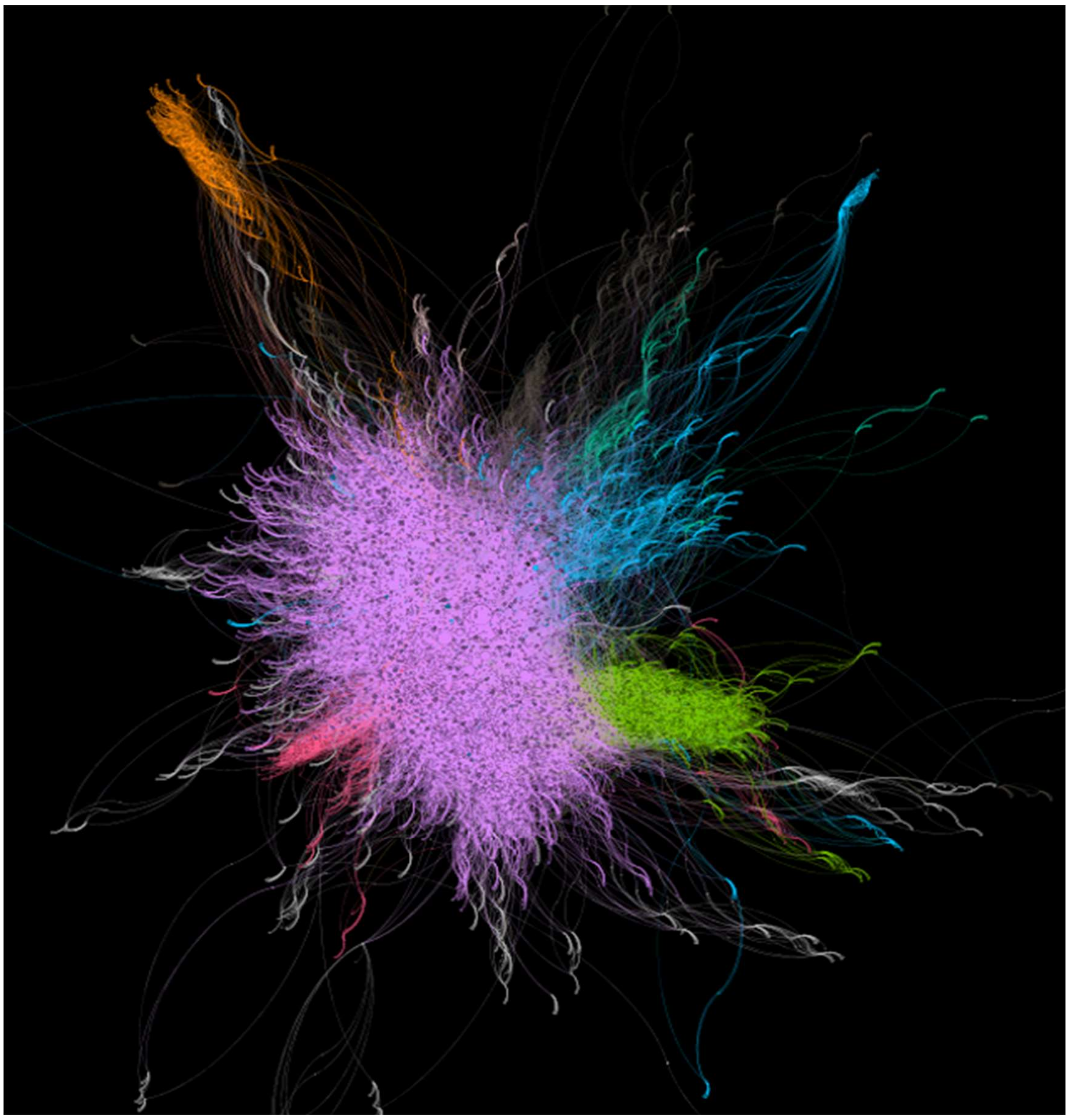
Our thoughts on this problem were simple; if a bias is present in the algorithm, it should be apparent in the data as well. Furthermore, it should be apparent easily. Thus, we decided to pursue the matter in a wholistic and general way: look at the entire video recommendation network and try to find if conspiracy or misinformative communities exist, that link mostly to each other and can trap users in an endless cycle of misinformation.

Analysis of 1st Query

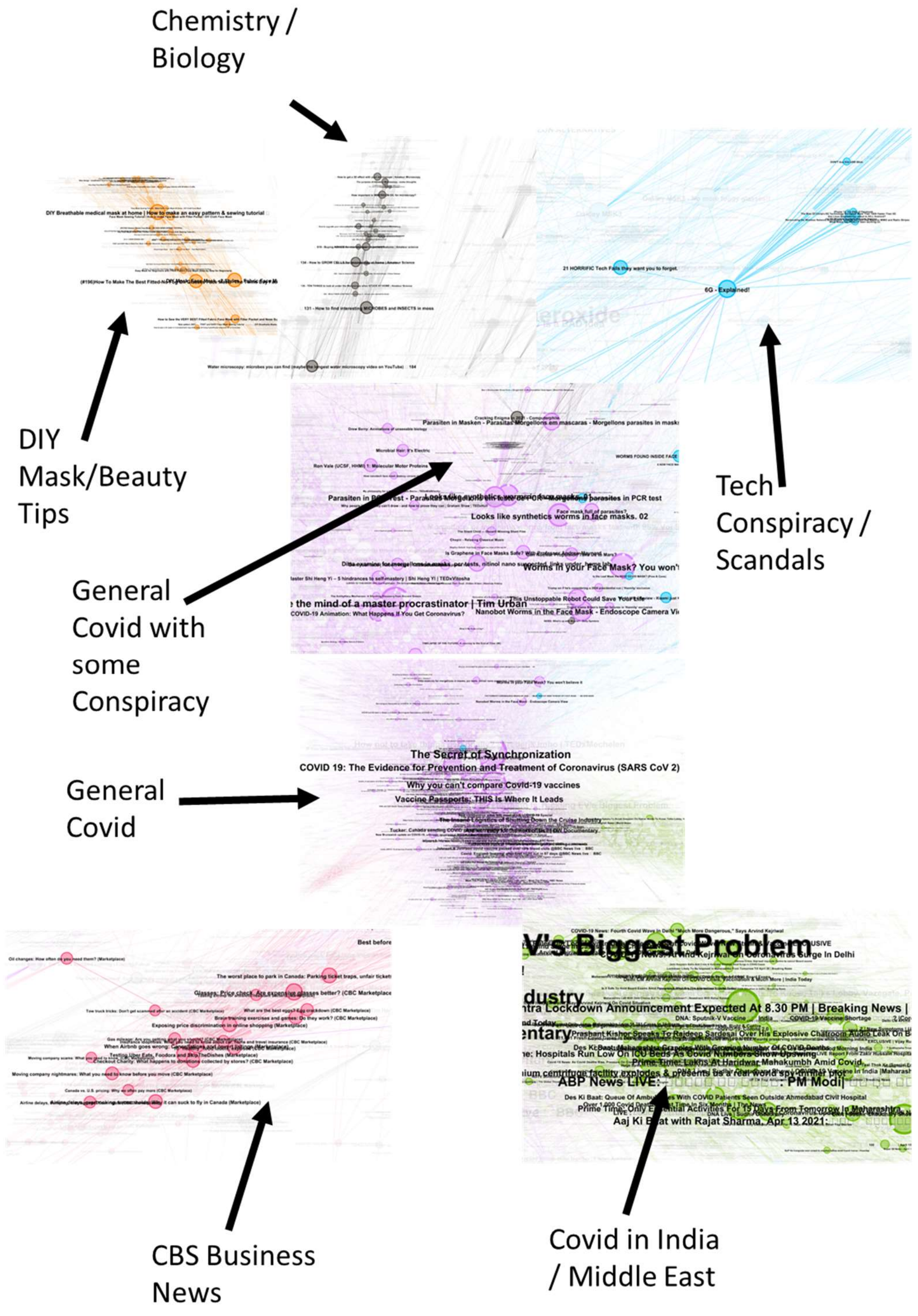
Our first query is simple enough: find all videos and their recommendations.

```
match (v1:VIDEO)-[r:RECOMMENDS]-(v2:VIDEO) return *
```

Using Gephi for visualization, network analytics, and community detection, we get the following network layout with the Force Atlas 2 algorithm. One can see 4 or 5 communities, colored with distinct colors in the figure. From these, the most separate community is the orange one, which looks like a proper echo chamber that could represent the conspiracy bubble we are looking for. We link a node's size and label size with its degree centrality, to see which videos in each cluster have the most links towards them. The assumption is that they should be the most representative videos for that community.



The resulting inspection shows no sign of conspiracy communities in our data. The orange community appears to be “DIY Mask / Covid Beauty” themed. Some communities, like the blue one, have a more controversial theme, with central videos titled “Tech scandals they don’t want you to know!” or “6G Explained!”. The top part of the central, purple community is where the “worms in masks” videos live. Even though they are linked together, they are also linked and close to fairly straightforward videos, thus the assumption that some bias exists cannot be verified by this simple analysis.



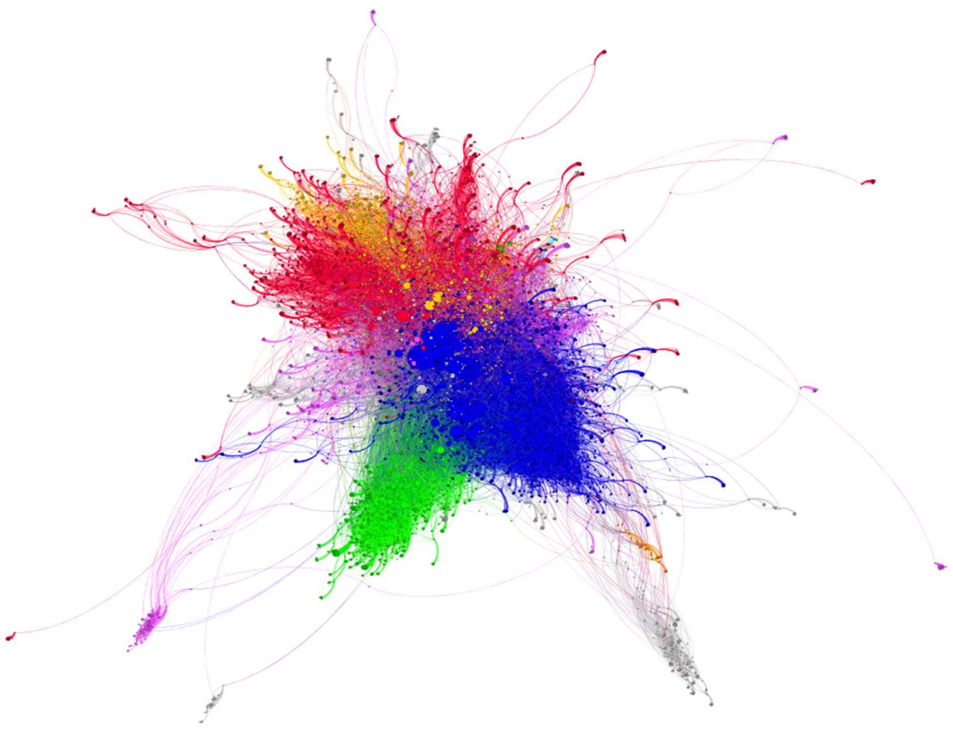
Chemistry / Biology
DIY Mask/Beauty Tips
Tech Conspiracy / Scandals
General Covid with some Conspiracy
General Covid
CBS Business News
Covid in India / Middle East

Analysis of 2nd Query

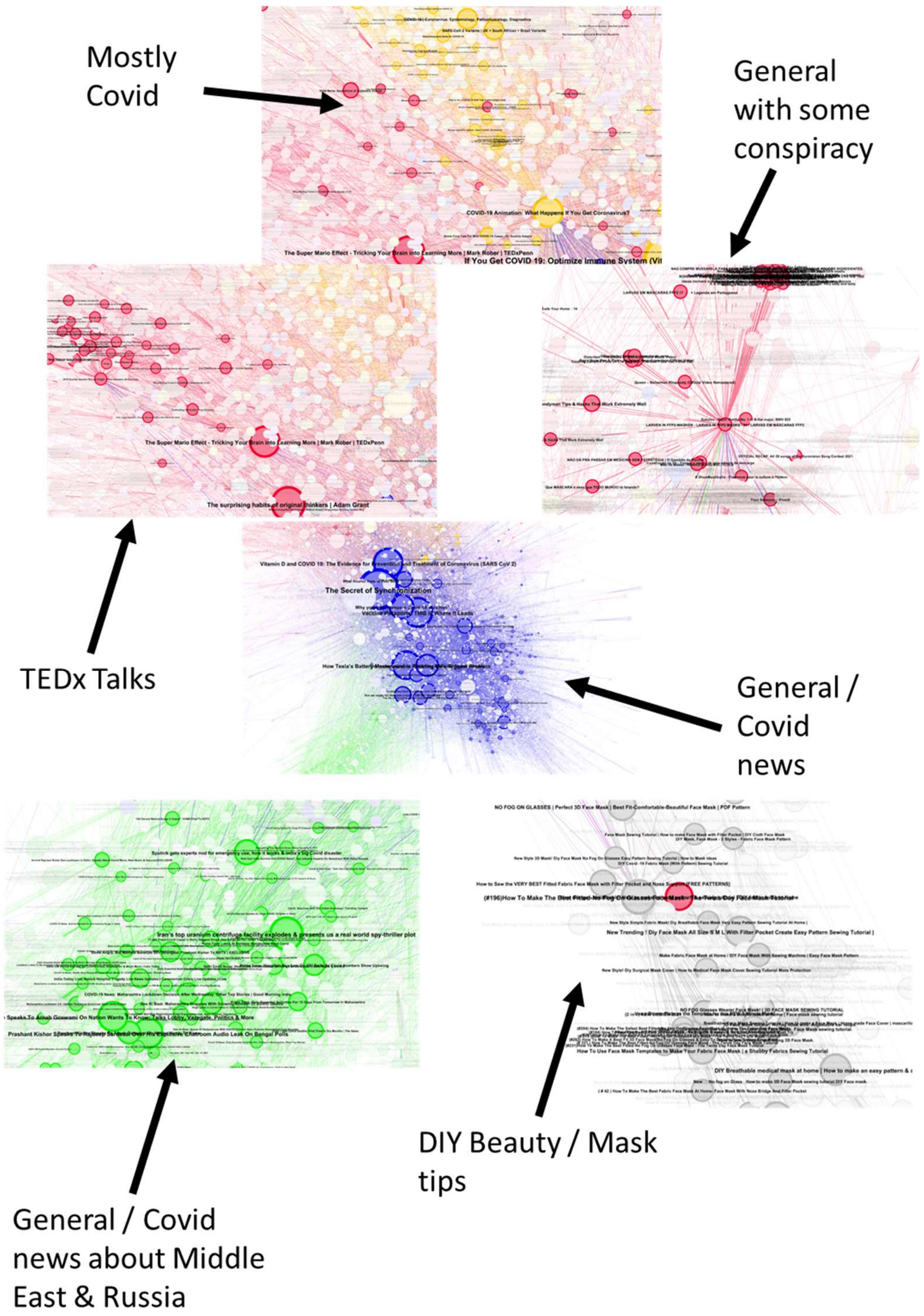
Our first query did not show any strong signs of bias in the algorithm, at least not at the level where we can consider it a problem and detrimental for public discourse. However, the dataset is rich enough that we can test more elaborate queries. We decide to include some controversy in the sampled data, by requiring that at least one video in each (video)—(recommendation)—(video) pair is a result in a query that has a negative connotation; such queries include "coronavirus is fake", "parasites in masks", and "bill gates is evil".

```
match (v1:VIDEO)-[r:RECOMMENDS]-(v2:VIDEO)←(q:QUERY)
where q.is_missleading=true return *
```

The same analysis steps were followed in Gephi and the resulting network is presented below, colorized by distinct communities. Inspection of the central labels in each community resulted in mainly the same thematic communities, with the grey community that seems to be the most isolated being again the "DIY" community.



Mostly, the communities stayed the same and even when beginning or ending to a video that can be the result of a misleading query, the recommendation network does not end up with isolated conspiracy communities. Videos with a controversial theme are always clustered together, however they are not isolated and do not constitute “conspiracy black holes”.

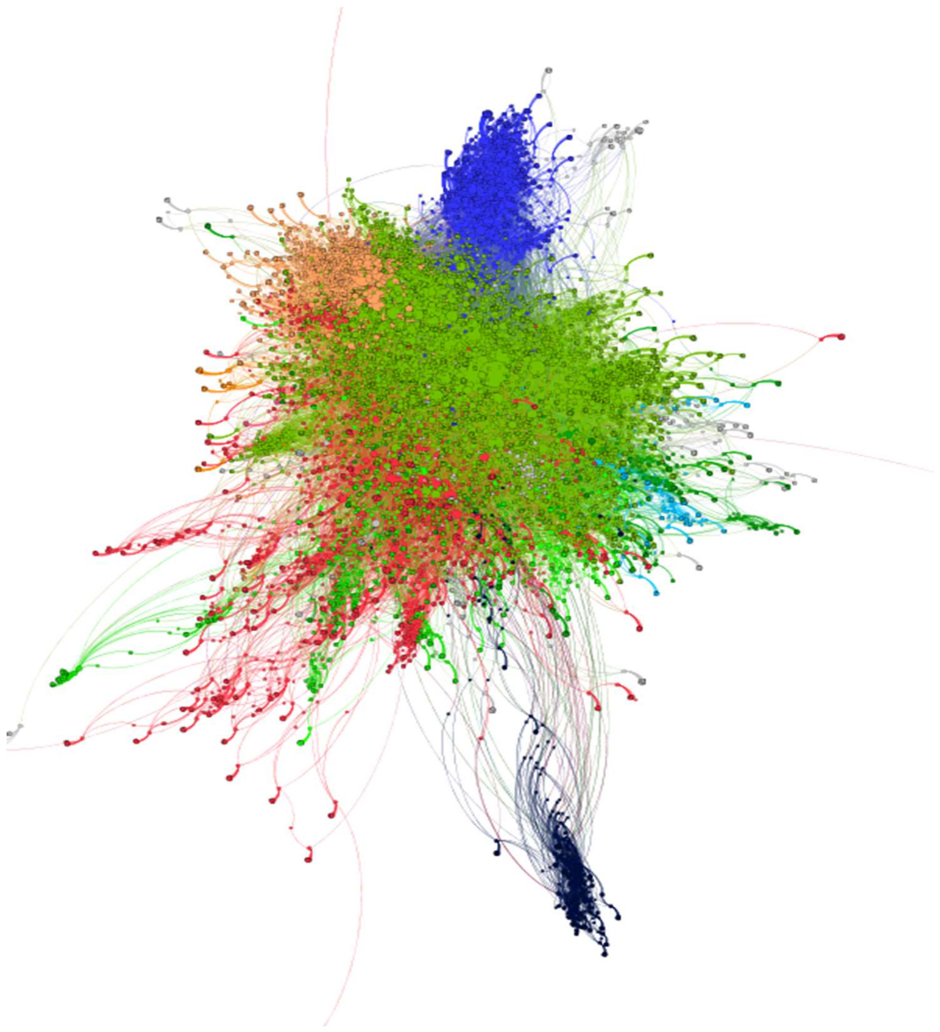


Analysis of 3rd Query

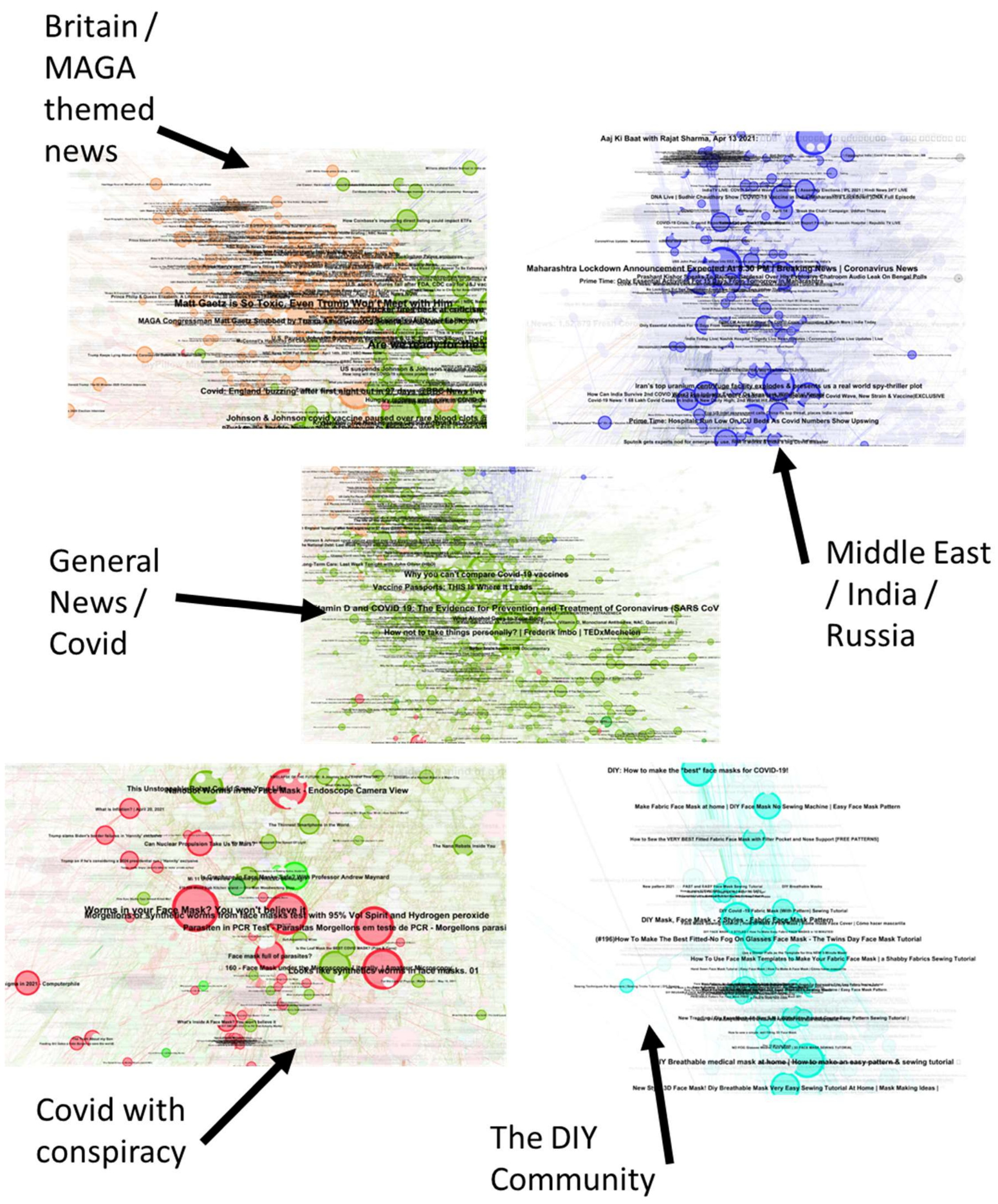
A final attempt to uncover a serious bias in the algorithm was attempted; starting or ending with a video that is known to be misleading, searching up to 3 edges deep, we reconstruct the video recommendation network and again visualize it to search for conspiracy communities.

```
match (v1:VIDEO)-[r:RECOMMENDS*0..3]-(v2:VIDEO)
where v1.is_known_misleading=true return *
```

The resulting network is visualized below, using the same methodology applied in the analysis of the previous queries. We find the same pattern as before with a central dominating community and smaller ones forming around it. Again, one community appears to be most isolated, colored here in black.



After inspecting the labels, we confirm the previous results. The main community is dominated by trustworthy and mainstream videos. The community about Middle Eastern countries, India, and Russia forms again as a distinct entity. We get a MAGA/Trump and British Royal Family cluster, which is more political and possibly conservative in theme but in no way conspiracy themed. The red cluster seems to be where the conspiracy videos are situated, however that community is spread out and could be considered part of the main community (green color). Finally, the isolated community is again the "DIY Mask and Beauty" themed one.



Conclusions

Judging from the results of this simplistic analysis, the Youtube algorithm does not show strong indications of algorithmic bias and recommendation manipulation for more extreme content. No strong and isolated community formed, capable of trapping users in endless loops of controversial videos. The only consistent community that could be categorized as isolated and looked like what we were searching for was the one about DIY masks and beauty tricks. We can thus conclude that if anything, Youtube is trying to sell makeup products and not to radicalize society.