AI for Global Health using Natural Language Processing

Project 2 for the Machine Learning for Health Care course (261-5120-00L)

May 2023

Sofie Daniëls, Aishwarya Melatur, Juraj Micko
Group: 3
Department of Computer Science, ETH Zurich, Switzerland

PART 1: DATA EXPLORATION AND PRE-PROCESSING

Q1: PREPROCESSING

As a first step in our preprocessing pipeline, we enforced the data types of each column and dropped rows (or tweets) that did not follow such a table schema, or were duplicates. We found that about 126K tweets were duplicates, while about 53K tweets had invalid TweetIDs or invalid indices. Finally, we dropped about 113K tweets that contained the value 'NaN' in any of the columns, as a 'NaN' in a column can be indicative of data corruption for that tweet (as we had a very large dataset, we deemed it appropriate to have this stringent procedure). We thus end up with approximately 382K tweets out of an original 675K.

The dataset has quite a lot of challenges, even after dropping invalid rows as above: indeed, the TweetText column is very unstructured by nature, as it can contain URLs, emojis, user mentions, hashtags, misspellings, abbreviations, numbers, and variations in capitalisation. In order to counter this, we used a combination of regex and string functions to very efficiently strip the tweets of URLs, mentions, numbers, capitalisation, emojis, and to split hashtags into separate words, done in a few seconds. An example of this procedure is shown in Figure 1, and the code snippet to achieve this is shown as well listing 1.

We then tokenize each of the tweets by using Nltk's tokenize package, and then remove common English stopwords using Nltk's corpus package. The column 'UserLocation' is used in Part 3, where we filtered tweets based on location, which we don't consider to be a part of preprocessing, however this is fully explained in Part 3.

Finally, we choose to split our labels into two different columns, one for positive and for negative, as this is the obvious choice in order to use supervised models with two output heads, instead of being boxed into carrying out classification over 25 possible output labels (5*5).

Q2: EXPLORATORY DATA ANALYSIS

The most common unigrams and bigrams after preprocessing are shown in Figure 2, and the ones before preprocessing are shown in Figure 3 (after removing invalid rows as explained above). We show the two words in a bigram separated by an underscore, for better visualization in the word cloud. As expected, the observed unigrams and bigrams are worthless before preprocessing.

We then follow exactly the same procedure, but for extremely negative tweets, shown in Figure 4, and for extremely positive tweets, shown in Figure 5.

One can observe the unigram 'amp' in every unigram word cloud, however this is a keyword in social media websites (Accelerated Mobile Pages) and is thus not significant. We observe that the negative tweets have many terms with negative connotations on social media ('hate', 'sad', 'killed', 'Trump') but also neutral words like 'people', 'coronavirus', 'covid',

```
THE TWEET BEFORE:
  @realDonaldTrump You are such a dick.

#MoscowMitch
#ImpeachTrumpPenceBarr
#ImpeachTrump
#ImpeachmentNow
#NepotismBarbie
#TrumpCrimeFamily
#TruthExposed
#ImpeachmentInquiry
#TrumpResign https://t.co/DteWeU126h
 THE TWEET AFTER:
   you are such a dick moscow mitch impeach trump pence barr impeach trump impeachment now
nepotism barbie trump crime family truth exposed impeachment inquiry trump resign
```

Fig. 1

Fig. 2: Most common unigrams (left) and most common bigrams (right) **after tweet preprocessing** shown as word clouds.

Fig. 3: Most common unigrams (left) and most common bigrams (right) **before tweet preprocessing** shown as word clouds.

Fig. 4: Most common unigrams (left) and most common bigrams (right) for **extremely negative tweets** (negative_sentiment $<$ -3).

Fig. 5: Most common unigrams (left) and most common bigrams (right) for **extremely positive tweets** (positive_sentiment $>$ 3).

```python
def rep(m):
    s=m.group(1)
    return ' '.join(re.split(r'(?=[A-Z])', s))


df_tweets['TweetText'] = [re.sub('((www\.[^\s]+)|(https?://[^\s]+))', '', x) for x in df_tweets['TweetText']] #remove url
df_tweets['TweetText'] = [re.sub('@[^\s]+', '' , x) for x in df_tweets['TweetText']] #remove mentions
df_tweets['TweetText'] = [x.encode("ascii", "ignore").decode() for x in df_tweets['TweetText']] # remove emojis
df_tweets['TweetText'] = [re.sub(r'#(\w+)', rep, x) for x in df_tweets['TweetText']] # split hashtags that are stuck together if they have case change: #iLovePotatoes becomes i love potatoes
df_tweets['TweetText'] = [re.sub(r'\W+', ' ', x) for x in df_tweets['TweetText']] # remove all characters that aren't alphanumerical - hashtag signs, punctuation, slashes...
df_tweets['TweetText'] = df_tweets["TweetText"].astype(str).str.replace('\d+', '') # remove numbers entirely
df_tweets['TweetText'] = df_tweets['TweetText'].str.lower() # remove all caps
```

Listing 1

Fig. 6: Bubble chart of the number of positive and negative sentiments in the dataset.

which are words that do not appear in the extremely positive unigram word cloud. This shows that **tweets that mention the COVID-19 pandemic explicitly tend to be more overtly negative**. The extremely positive tweets' word clouds show very positive terms like 'loved', 'loving', 'stay home', 'stay safe', 'one', 'protect loved', and thus refer more to enforcing a positive attitude during the pandemic rather than the events of the pandemic themselves.

We show the proportion of each sentiment label as a bubble chart along a grid defined by the labels (we take the absolute value of the negative sentiments, plotted here on the y-axis), in Figure 6. We observe that **most of the labels are fairly neutral** (leaning towards the low end of the positive and negative values), however, the **more extreme tweets tend to be more negative than positive**: for example, (1, 4) is larger than (4, 1), where (x, y) refers to the bubble size of positive_sentiment = x and negative_sentiment = y. The dataset thus contains a very large proportion of mild tweets that are not particularly charged in sentiment.

Q3: Metric choice

First, we introduce our interpretation of the problem. Predicting a value for either positive or negative sentiment is inherently a regression problem since the integer value is ordinal rather than categorical. In other words, the *positive sentiment* value of 3 semantically lies between the values of 2 and 4. The fact that these five categories of values (for both sentiments) are ordered brings the potential for models to perform better. However, in the problem description, we are given the task to solve a classification problem. Hence, we decided for the following approach:

1) We are **solving a classification problem** as is asked for, in that our models return two integers in the suitable ranges (one for positive and negative sentiment), and during evaluation, we will treat each sentiment variable as categorical, therefore only allowing categorical metrics.

```python
from sklearn.metrics import f1_score
from scipy.stats import hmean

def f1w_hm(pos_true, pos_pred, neg_true, neg_pred):
    """Harmonic mean of the weighted f1 scores for positive and negative sentiment variables.

    Args:
        pos_true (np.array): True labels of the positive sentiment variable.
        pos_pred (np.array): Predicted labels of the positive sentiment variable.
        neg_true (np.array): True labels of the negative sentiment variable.
        neg_pred (np.array): Predicted labels of the negative sentiment variable.
    """
    pos_f1 = f1_score(pos_true, pos_pred, average='weighted')
    neg_f1 = f1_score(neg_true, neg_pred, average='weighted')
    return hmean([pos_f1, neg_f1])
```

Listing 2: The "Weighted-F1 HM" metric chosen for the two-output classification problem of predicting the positive and negative sentiment variables.

Fig. 7: Average of positive labels over time, over 200 bins.

2) Under the hood, we allow the models to capture the intrinsic regression problem, even though their output for evaluation will be one of five classes. This means that if it is suitable for a given model, we will naturally choose a regression loss function such as the mean squared error.

Metric choice. At first, we observe from fig. 6 that classes are highly imbalanced, with classes closer to zero being orders of magnitude more likely (for both sentiment variables). Therefore an appropriate metric has to take the class imbalance into account. For a given class, a natural choice is first to evaluate the standard F1 score, calculated from Precision and Recall, and then calculate the Weighted F1 score which takes class imbalance into account. However, considering all 25 classes individually (= all combinations of permissible values for positive and negative sentiment variables) would not lead to a representative metric because a model that predicts both variables incorrectly would be rated the same as another model which predicts one of the variables correctly. Therefore, we decided to split the problem into two separate classification problems, one for each sentiment variable. We then calculate the Weighted F1 score for both variables and report their harmonic mean as the final metric. The harmonic mean, as opposed to the arithmetic mean, penalises models whose performance on the two variables is imbalanced.

Listing 2 shows the primary metric that we call *Weighted-F1 HM* that we will use to evaluate a model's performance on the given classification task, as well as to compare the predictive performance of different models.

Q4: Dataset splitting

We choose to split our dataset in the following manner: we reserve 33% of the dataset for the test set, and keep 67% of the dataset for training and validation (where validation is typically 20% of the training set, however this has been varied in our experiments). We also split by randomly shuffling the dataset, as we want the training to be agnostic to the timeframe. Indeed, in Figure 7, we plot the average of the positive labels in time (sorted timestamps, then averaged over 200 bins), and observe that this average isn't constant: there is a sharp fall towards the middle of the dataset. Thus, if we split our dataset without shuffling, we risk ending up with dataset splits that have different label proportions with respect to each other.

```python
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
sid_obj = SentimentIntensityAnalyzer()

def map_sentiment(compound_val):
    if compound_val >= 0.05:
        return 'positive'
    if compound_val <= - 0.05:
        return 'negative'
    else:
        return 'neutral'
df_tweets['vader_polarity'] = [sid_obj.polarity_scores(sentence) for sentence in df_tweets['TweetText']]
df_tweets['vader_decision'] = [map_sentiment(sentiment_dict['compound']) for sentiment_dict in
↪  df_tweets['vader_polarity']]

# convert our dataset to positive, negative and neutral labels by adding pos_sent and neg_sent
df_tweets['overall_sent_GT'] = df_tweets['pos_sent'].astype('int') + df_tweets['neg_sent'].astype('int')
def map_int(val):
    if val > 0:
        return 'positive'
    if val < 0:
        return 'negative'
    else:
        return 'neutral'
df_tweets['overall_sent_GT'] = [map_int(val) for val in df_tweets['overall_sent_GT']]

from sklearn.metrics import confusion_matrix
confusion_matrix(df_tweets['overall_sent_GT'], df_tweets['vader_decision'])
```

Listing 3: Using VADER for our dataset, and converting our labels to three classes to measure the performance of VADER.

PART 2: NLP LEARNING BASED METHODS

VADER (5 PTS)

Q1: WORKINGS OF VADER

According to VADER's repository [24], VADER (Valence Aware Dictionary and sEntiment Reasoner) is a rule-based sentiment analysis tool which uses a combination of lexicon-based and rule-based approaches to determine the sentiment polarity of a given text, and is especially designed for analyzing social media texts.

VADER uses a pre-built lexicon of words (validated by human raters) that have associated 'valence' scores: for each text, we find the words within that text that belong to the lexicon, sum their valence scores, adjust to VADER's rules, and normalize to range between -1 and 1. This gives the compound score of the whole text, which we then threshold to get a ternary classification: positive, negative, or neutral.

Q2: CODE

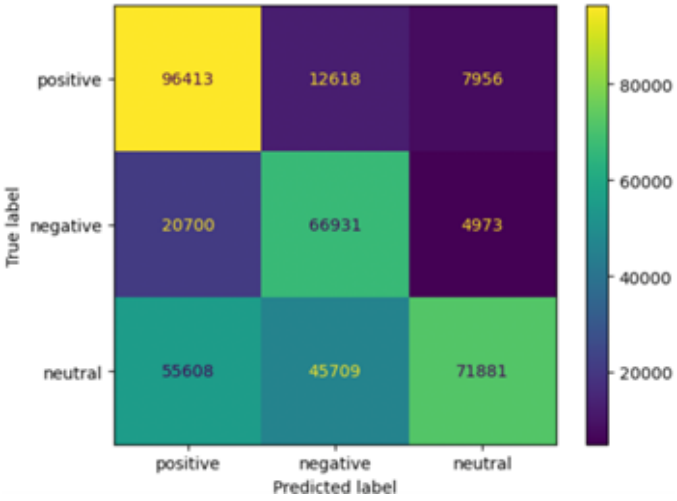
Code snippet listing 3 shows how we have used VADER for our use case. The lexicon that VADER uses is especially suitable for social media, and thus, there is no need to tokenize tweets and remove stopwords. However, it is still important to preprocess the tweet strings into clean text (by the methods shown in code snippet listing 1).

Q3: APPLICATION TO TWEETSCOV19 DATASET, PERFORMANCE

We convert our ground truth labels into VADER-compatible labels by summing the positive and negative values, and taking the sign of the output. If the value is 0, it is neutral. While this approach is straightforward, it is quite naive (for example, a tweet that is labeled as (5, -5) classified as neutral in the same manner as a tweet labeled as (1, -1)). However, since extreme tweets (whose sentiment is higher than 3 for either positive or negative sentiments) are quite rare in the dataset, and there is no other straightforward heuristic for conversion, we choose this naive solution.

In Figure 8, we plot our confusion matrix between our VADER-converted ground truth labels, and the VADER predictions.

While the performance evaluation of the VADER method might not seem impressive, it is important to remember that our ground truth labels are not adapted to the ternary classification task, and thus it is unwise to cast doubts on VADER's predictions on this basis. Furthermore, the F1-score of every class is between 56% and 67%, and while these are not very high figures, they do show some agreement between our ground truth labels and the VADER method.



	precision	recall	f1-score	support
positive	0.56	0.82	0.67	116987
negative	0.53	0.72	0.61	92604
neutral	0.85	0.42	0.56	173198
accuracy			0.61	382789
macro avg	0.65	0.65	0.61	382789
weighted avg	0.68	0.61	0.60	382789

Fig. 8: Confusion matrix and classification report of VADER's predictions on the basis of our converted ground truth labels.

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer(lowercase=False, max_features=250)
bow_matrix = vectorizer.fit_transform(df_tweets_preprocessed['unigram'].apply(" ".join))
```

Listing 4: Short code snippet showing how to get the BoW vectors for each token.

```
from sklearn.feature_extraction.text import TfidfTransformer
transformer = TfidfTransformer()
bow_matrix_tfidf = transformer.fit_transform(bow_matrix)
```

Listing 5: Extension of the previous code snippet, applying tf-idf to the BoW vectors.

WORD EMBEDDINGS

Q1: BAG OF WORDS (BOW)

Applying Bag of Words (BoW) [12] to tokens is the exact same as one-hot encoding of tokens. We implemented our version of BoW using 'CountVectorizer' from scikit-klearn [6, 17], as can be seen in Code Snippet 4. To keep our implementation low in memory consumption, we choose to work with a vocabulary of size 250, which limits our token vectors to a size of 250.

Q2: TF-IDF

Similar to BoW, using term frequency–inverse document frequency (TF-IDF) [21] results in an embedding vector that has the same size as the vocabulary. Instead of one-hot encoding, the entry of a word in a vector is directly related to its frequency in that tweet, and inversely related to the commonality of that word in general (i.e. the occurrence of that word in all tweets). In Code Snippet 5, we apply TfidfTransformer from scikit-learn [6, 17] to the BoW matrix we obtained in the previous question.

Q3: WORD2VEC WITH CBOW OR SKIP-GRAM

Continuous Bag of Words (CBOW) [15] is a word embedding model that predicts a word by looking at the context (i.e. the surrounding words). Skip-gram [15], on the other hand, does the opposite: it predicts the context of a word, by looking at the word itself. In general, Skip-gram is used more often, but CBOW is much faster, which makes sense, since you only predict a single word for the latter. Since we would like for our model to take context into account, we opt to use Skip-gram. Still, we will compare this with the performance of CBOW later on in question 8 and 9. For computational reasons (related to Q8), we limit the vector size of a token to 20 and therefore train the model from scratch instead of using a pre-trained model. In addition, we choose a context window of 3. We use the same vector and context window size for all subsequent models. The code for the respective functions can be found in Code Snippets 6 and 7, and was written using functions from Gensim [20].

```
from gensim.models import Word2Vec
from gensim.models.callbacks import CallbackAny2Vec

# following class function from https://stackoverflow.com/questions/54888490/gensim-word2vec-print-log-loss
class callback(CallbackAny2Vec):
    def __init__(self):
        self.epoch = 0
        self.loss_to_be_subed = 0

    def on_epoch_end(self, model):
        loss = model.get_latest_training_loss()
        loss_now = loss - self.loss_to_be_subed
        self.loss_to_be_subed = loss
        print('Loss after epoch {}: {}'.format(self.epoch+1, loss_now))
        self.epoch += 1

context_size = 3 #hyperparameter
vec_size = 20 #300 most commonly used, according to https://arxiv.org/pdf/1812.04224.pdf

model_cbow = Word2Vec(min_count=1, vector_size=vec_size, window=context_size, sg=0, workers=20)
model_cbow.build_vocab(df_tweets_preprocessed['unigram'])
model_cbow.train(df_tweets_preprocessed['unigram'], epochs=15, total_examples=model_cbow.corpus_count,
                 compute_loss=True, callbacks=[callback()])
```

Listing 6: Code snippet showing how the CBOW model was constructed and trained. The loss function was printed after every epoch.

```
model_sgram = Word2Vec(min_count=1, vector_size=vec_size, window=context_size, sg=1, workers=20)
model_sgram.build_vocab(df_tweets_preprocessed['unigram'])
model_sgram.train(df_tweets_preprocessed['unigram'], epochs=15, total_examples=model_sgram.corpus_count,
                  compute_loss=True, callbacks=[callback()])
```

Listing 7: Code snippet showing how the Skip-gram model was constructed and trained. Using the function from the previous code snippet, the loss function was printed after every epoch.

```
! pip install glove-python-binary

from glove import Glove, Corpus
from gensim.models.keyedvectors import KeyedVectors
from gensim.scripts.glove2word2vec import glove2word2vec

corpus = Corpus()
corpus.fit(df_tweets_preprocessed['unigram'], window=context_size)
model_glove = Glove(no_components=vec_size, learning_rate=0.05, alpha=0.75, max_count=100, max_loss=10.0)
model_glove.fit(corpus.matrix, epochs=15, no_threads=16, verbose=True)
model_glove.add_dictionary(corpus.dictionary)

#from https://stackoverflow.com/questions/55693318/encoding-problem-while-training-my-own-glove-model
with open("word_embedding/results_glove.txt", "w") as f:
    for word in model_glove.dictionary:
        f.write(word)
        f.write(" ")
        for i in range(0, vec_size):
            f.write(str(model_glove.word_vectors[model_glove.dictionary[word]][i]))
            f.write(" ")
        f.write("\n")


glove2word2vec(glove_input_file="word_embedding/results_glove.txt",
↪  word2vec_output_file="word_embedding/gensim_glove_vectors.txt")

model_glove_gensim = KeyedVectors.load_word2vec_format("word_embedding/gensim_glove_vectors.txt",
↪  binary=False)
```

Listing 8: Code snippet showing how the GloVe embedding model was trained. We used the glove-python-binary library [19] in combination with Gensim [20].

Q4: GloVe

GloVe [18] is an embedding method that counts how often a word occurs in a certain context. It combines these findings into a so-called co-occurrence matrix, which is then approximated by a lower-dimensional matrix to limit the vector size of the embeddings. We use the library 'glove-python-binary' for our code [19], as can be seen in Code Snippet 8.

Q5: FastText

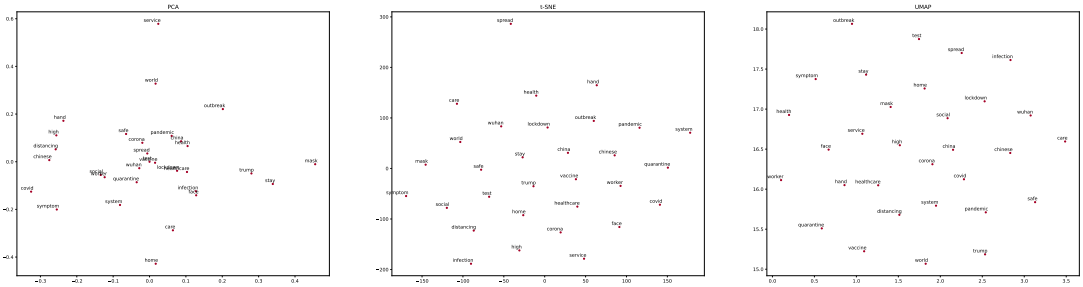
Different from the previous embedding methods, FastText [3] uses n-grams as input for training to ensure that uncommon and unseen words can still be embedded. Previous methods try to find a unique embedding for each word, but since FastText cares about the morphological similarity of words through n-grams, it can handle out-of-vocabulary (OOV) words. Training follows a similar fashion to the word2vec models mentioned earlier [15]. The code can be found in Code Snippet 9

Q6: Visualization of embeddings

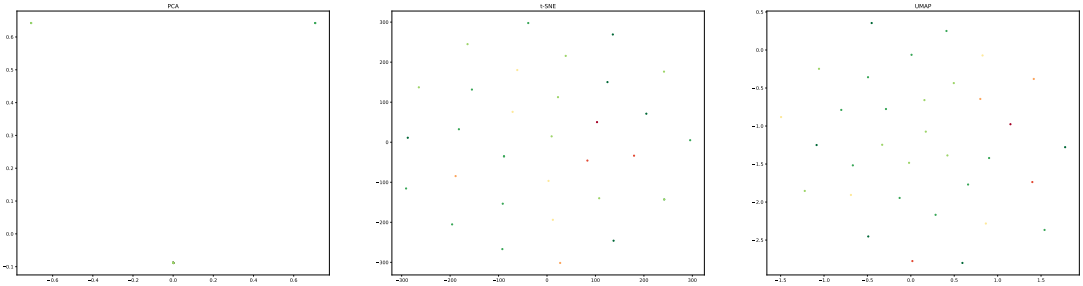
We visualize our word embeddings both by the positive/negative score of tokens, but also by the keywords used for filtering the tweets.

```
from gensim.models import FastText

sg = 1 #cbow=0, skipgram=1
model_ft = FastText(min_count=1, vector_size=vec_size, window=context_size, workers=20, sg=0)
model_ft.build_vocab(df_tweets_preprocessed['unigram'])
model_ft.train(df_tweets_preprocessed['unigram'], epochs=15, total_examples=model_ft.corpus_count)
model_ft.save("word_embedding/fasttext.model")
```

Listing 9: Code used for training the FastText model. Like with CBOW and Skip-gram, the Gensim library was used [20].



(a) Distribution of keywords used to select tweets for the TweetsCOV19 dataset.



(b) Distribution of positive and negative words from the AFINN-96 dataset.

Fig. 9: Visualization of BoW embedding of keywords and positive/negative words with PCA, t-SNE, and UMAP.

For the first method, we us a list of over 1400 positive and negative words from [16], each with a rank, and map their embeddings into the latent space. We hope to see that all of the previously implemented embedding methods allow us to easily discern words with a positive meaning from words with a negative meaning.

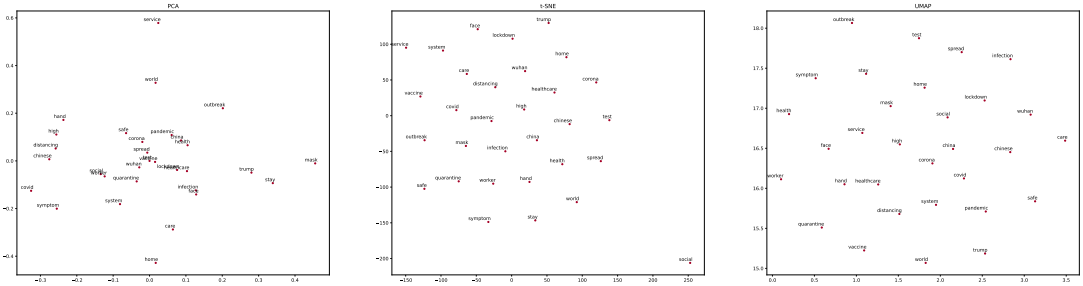
For the second method, we use the keywords used for filtering the TweetCov19 dataset [8], and again map the embeddings into the latent space. This time, we hope to see similar words clustered or located close to one another, and words with completely different meanings further away from each other.

Note that we only use keywords and positive/negative words that are in the vocabulary of the trained embedding model. This means that for both our BoW and TF-IDF model, most words are filtered out (since the entire vocabulary only has size 250), as you can see in Figure 9 and 10. This will impede our downstream classifiers in finding the correct sentiment scores, but instead greatly speeds up training, since the feature size is significantly reduced. Furthermore, dimensionality reduction and clustering of one-hot encoded vectors from BoW proves to be difficult, and visualizations do not give much additional information, as most words are distributed evenly throughout the space. Interestingly, the PCA visualization for TF-IDF embedding of keywords shows strong similarity with those of the BoW embeddings (see Figure 10a and 9a), which reveals that the keywords present in the vocabulary likely have similar term frequencies.

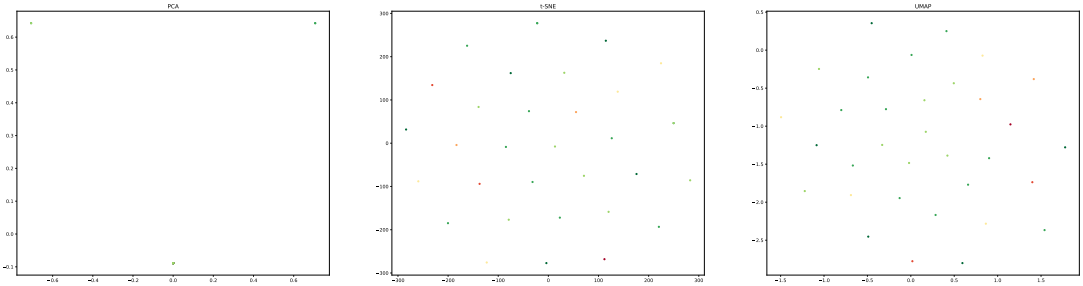
After embedding of keywords with FastText, we can see a clear grouping of similar words when visualized with t-SNE and especially UMAP, as can be seen in Figure 13a. This makes sense, since FastTexts looks at the morphological similarity of words by working with n-grams. For example, in the UMAP visualization, 'distancing' and 'distance' are grouped together, but 'physicaldistancing', 'socialdistancing', and 'socialdistance' are also located nearby. This will likely make it easier for FastText to assign similar scores to similar words, e.g. to 'bad' and 'badly'.

Embeddings with GloVe, Skip-grams, and CBOW seem to capture more subtle relationships, and UMAP visualization is able to group words such as 'facemask', 'kn', and 'ffp' together. The visualizations can be found in Figures 14a, 12a, and 11a. Classification results using any of these embedding models will therefore in all likelihood be relatively similar.

Application of PCA on the embedding with CBOW, Skip-grams, and FastText seems to show a reasonable separation of positive and negative words, although the t-SNE and UMAP visualizations do not noticeably confirm this finding (see Figure 11b, 13b, and 12b). GloVe embedding, on the other hand, seems to not be easily separable in two dimensions after application of PCA, but shows similar results for t-SNE and UMAP, as can be seen in Figure 14b. In general, we can see that most of embedding models are able to pick up on the sentiments of certain words, but not on the relative weights of that sentiment. Therefore, it might be difficult for our downstream classifiers to pick up on these nuances, and especially so when using GloVe embeddings.

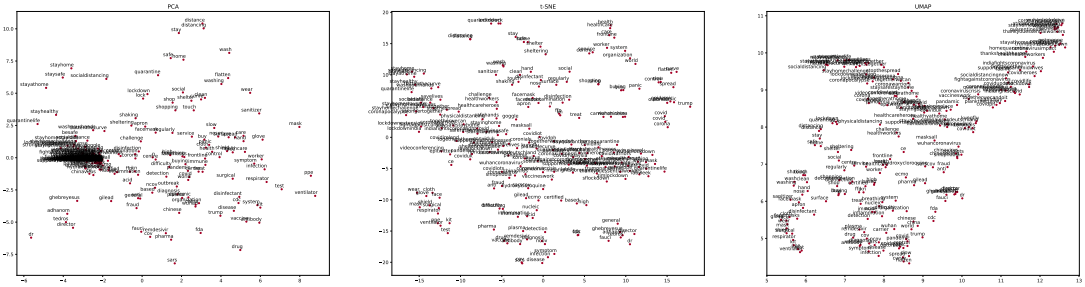


(a) Distribution of keywords used to select tweets for the TweetsCOV19 dataset.

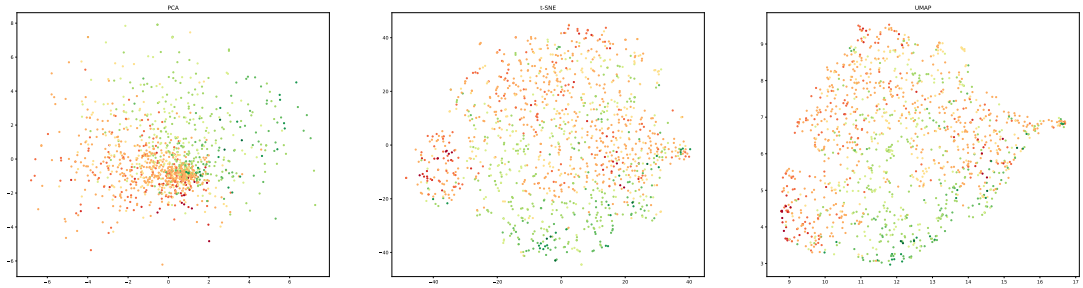


(b) Distribution of positive and negative words from the AFINN-96 dataset.

Fig. 10: Visualization of TF-IDF embedding of keywords and positive/negative words with PCA, t-SNE, and UMAP.

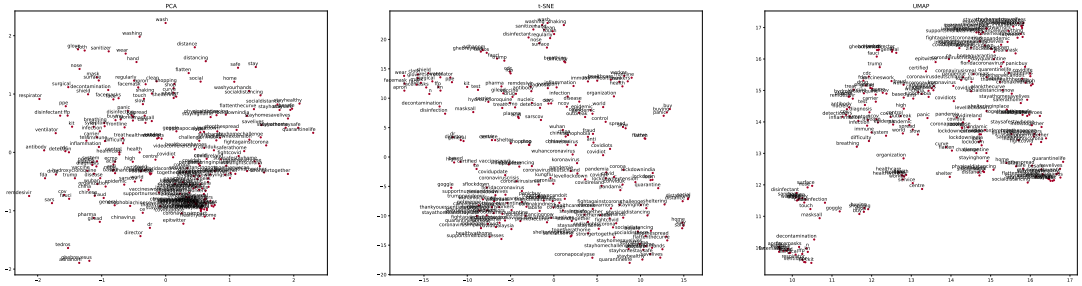


(a) Distribution of keywords used to select tweets for the TweetsCOV19 dataset.

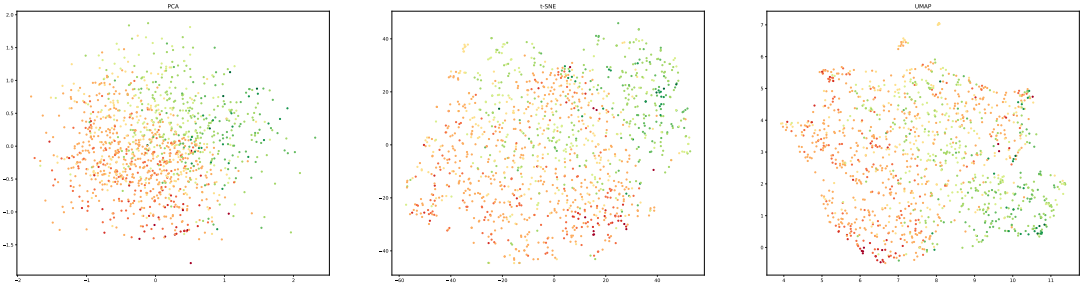


(b) Distribution of positive and negative words from the AFINN-96 dataset.

Fig. 11: Visualization of CBOW embedding of keywords and positive/negative words with PCA, t-SNE, and UMAP.

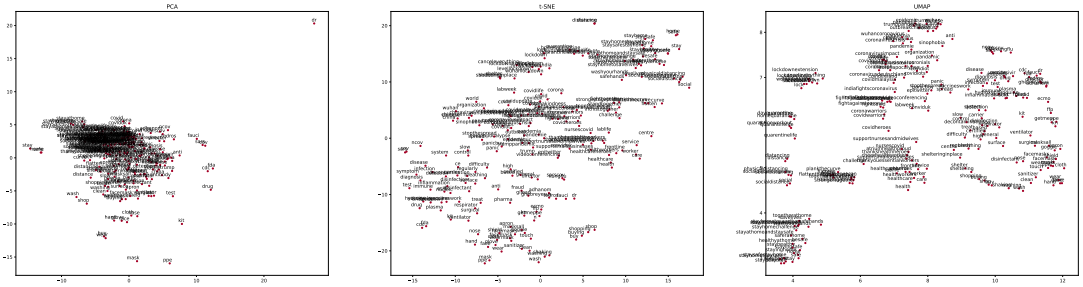


(a) Distribution of keywords used to select tweets for the TweetsCOV19 dataset.

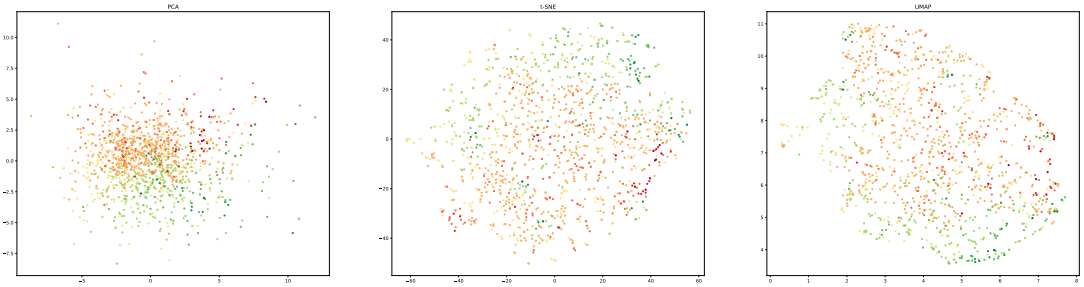


(b) Distribution of positive and negative words from the AFINN-96 dataset.

Fig. 12: Visualization of Skip-gram embedding of keywords and positive/negative words with PCA, t-SNE, and UMAP.

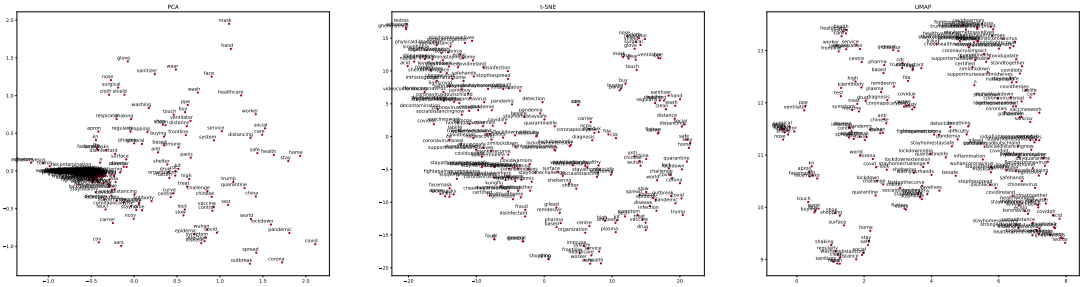


(a) Distribution of keywords used to select tweets for the TweetsCOV19 dataset.

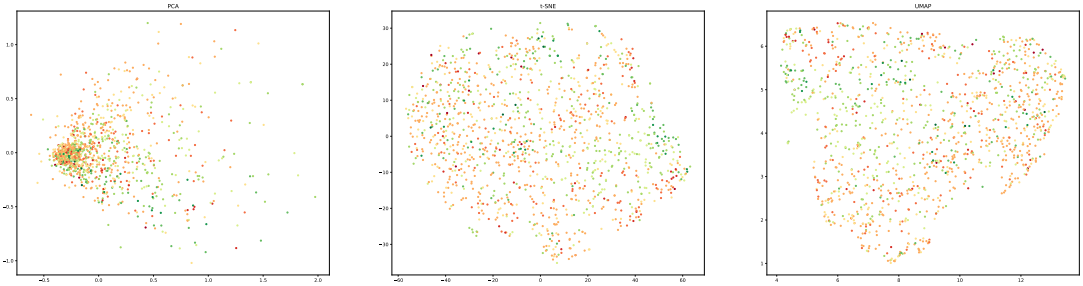


(b) Distribution of positive and negative words from the AFINN-96 dataset.

Fig. 13: Visualization of FastText embedding of keywords and positive/negative words with PCA, t-SNE, and UMAP.



(a) Distribution of keywords used to select tweets for the TweetsCOV19 dataset.



(b) Distribution of positive and negative words from the AFINN-96 dataset.

Fig. 14: Visualization of Glove embedding of keywords and positive/negative words with PCA, t-SNE, and UMAP.

Q7: Tweet embeddings

For this question, we compare summing, averaging, and concatenation of the minimum and maximum of the word embeddings.

Our **'summing' method** simply consists of summing up the embeddings of every word in the tweet. If a word cannot be found in the vocabulary, we do not include it in the sum (this is done because we limited vocabulary size in some of our methods for computational reasons).

The **'averaging' method** uses the results from the first 'summing' method, but divides it by the number of valid (i.e. not OOV) words found in the tweet.

The code for the first two methods can be found in Code Snippet 10.

Finally, we designed a **'minmax' method**, where we find the respective minimum and maximum values (per index) of all word embeddings and concatenate the results. Again, as can be seen in Code Snippet 11, if a word is OOV, we simply ignore it.

Q8: Classifier

Instead of classifiers, we implement **three regression models** to perform sentiment analysis of the tweets, since the labels 'positive sentiment score' and 'negative sentiment score' are ordered discrete values. We compare our results using the **mean harmonic F1 score** (i.e. the mean of the harmonic F1 score for positive and negative sentiments respectively).

Before calculating the harmonic F1 score, we round the regression output to the nearest integer, and then clip all values outside of the sentiment score range (1 to 5 for positive sentiments, and -5 to -1 for negative sentiments), to ensure consistency with classifier outputs. This details of this preprocessing step can be found in Code Snippet 12

All hyperparameter tuning was performed using GridSearchCV from the scikit-learn library [6, 17]. Still, to speed up training, we tune the models with limited parameters and train on only two folds.

For our first 'classifier', we use **Random Forests** [5]. Random Forests is a model that combines the results of multiple decision trees, returning the most likely class for classification, or the mean prediction for regression. The code can be found in Code Snippet 13.

For the second 'classifier', we use **GradientBoosting** [4, 10, 11]. Gradient boosting is a method that combines weak models to create a strong predictive model. It does so by iteratively improving upon the loss from the weak model of the previous round. We again use a method from scikit-learn [6, 17], as you can see in 14.

Finally, for our third 'classifier', we implemented **AdaBoost** [9] in Code Snippet 15. Adaptive boosting is an iterative training method that in every iteration adjusts the weights of the data points that were predicted badly in the previous iteration. The final output is the weighted average of all previous models, the weights depending on their overall performance.

The train and test scores of the classifiers for each word embedding model and aggregation method can be found in Table I.

```python
def av_sum_sentence(row, model, vec_size=vec_size, av=True):
    if row != []:
        red = 0
        sentence = 0
        for word in row:
            try:
                sentence += model.wv[word]
            except:
                red += 1
        if av:
            sentence = sentence/(1+len(row)-red)
        if red == len(row):
            return np.zeros(vec_size).tolist()
        else:
            return sentence.tolist()
    else:
        return np.zeros(vec_size).tolist()


def av_sum_sentence_bow(row, vectorizer, transformer=None, av=True):
    sentence = vectorizer.transform([row])
    if transformer is not None:
        sentence = transformer.transform(sentence)
    sentence = sentence.toarray()[0]
    if av:
        sentence = sentence/(1+sentence.sum())
    return sentence.tolist()
```

Listing 10: Code of our 'summing' and 'averaging' word-to-sentence embedding functions. We created two functions to account for the individual differences between Gensim [20] and scikit-learn [6, 17].

```python
def minmax_sentence(row, model, vec_size=vec_size):
    sentence_min = np.ones(vec_size)*1000
    sentence_max = np.ones(vec_size)*(-1000)
    if row != []:
        red = 0
        for word in row:
            try:
                sentence_min = np.minimum(np.asarray(model.wv[word]), sentence_min[idx])
                sentence_max = np.maximum(np.asarray(model.wv[word]), sentence_max[idx])
            except:
                red += 1
        if red == len(row):
            sentence_min = np.ones(vec_size)*1000
            sentence_max = np.ones(vec_size)*(-1000)
    return np.concatenate((sentence_min, sentence_max), axis=0).tolist()

def minmax_sentence_bow(row, vectorizer, vocab_size, transformer=None):
    sentence_min = np.ones(vocab_size)
    sentence_max = np.zeros(vocab_size)

    if row != []:
        res = vectorizer.transform([row])
        if transformer is not None:
            res = transformer.transform(res)
        sentence_min = np.minimum(res.toarray()[0], sentence_min)
        sentence_max = np.maximum(res.toarray()[0], sentence_max)
    return np.concatenate((sentence_min, sentence_max), axis=0).tolist()
```

Listing 11: Code of our 'minmax' word-to-sentence embedding function. There are two functions, in order to account for the differences between the Gensim [20] and scikit-learn libraries [6, 17].

```python
from sklearn.metrics import f1_score, make_scorer

def weighted_harmonic_f1_score(y_true, y_pred):
    y_pred_pos = np.clip(np.around(y_pred[:,0]), 1, 5).astype(int)
    y_pred_neg = np.clip(np.around(y_pred[:,1]), -5, -1).astype(int)
    f1_scores_pos = f1_score(y_true[:,0], y_pred_pos, average='weighted')
    f1_scores_neg = f1_score(y_true[:,1], y_pred_neg, average='weighted')
    weighted_f1 = (f1_scores_pos + f1_scores_neg)/2
    return weighted_f1

weighted_harmonic_f1_scorer = make_scorer(weighted_harmonic_f1_score)

```

Listing 12: Code snippet showing the calculation of the weighted harmonic F1 score used in Q8.

```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.multioutput import MultiOutputRegressor


def rand_forest(sentence, sentence_test):
    params = {'estimator__n_estimators':[5,15],
              'estimator__n_jobs':[-1], 'estimator__random_state':[42],
              'estimator__criterion':['squared_error'], 'estimator__max_depth':[4,5],
              'estimator__verbose':[0]}
    regr = GridSearchCV(MultiOutputRegressor(RandomForestRegressor()), param_grid=params,
                        n_jobs=-1, cv=2, scoring=weighted_harmonic_f1_scorer, verbose=0)
    start_time = time.time()
    regr.fit(sentence, np.asarray(y_train_val[['pos_sent', 'neg_sent']].values.tolist()))
    elapsed_time = time.time() - start_time

```

Listing 13: Code snippet for the first classifier, Random Forests.

```python
from sklearn.ensemble import GradientBoostingRegressor

def grad_reg(sentence, sentence_test):
    params = {'estimator__learning_rate': [0.1, 1], 'estimator__n_estimators': [5, 15],
              'estimator__max_depth': [4, 5], 'estimator__n_iter_no_change':[7]}
    regr = GridSearchCV(MultiOutputRegressor(GradientBoostingRegressor()), param_grid=params,
                        n_jobs=-1, cv=2, scoring=weighted_harmonic_f1_scorer, verbose=0)
    start_time = time.time()
    regr.fit(sentence, np.asarray(y_train_val[['pos_sent', 'neg_sent']].values.tolist()))
    elapsed_time = time.time() - start_time
```

Listing 14: Code snippet for the second classifier, Gradient Boosting.

```python
from sklearn.ensemble import AdaBoostRegressor

def ada_reg(sentence, sentence_test):
    params = {'estimator__loss': ['square', 'exponential'], 'estimator__n_estimators': [5, 15, 30],
              'estimator__learning_rate': [0.1, 1, 10], 'estimator__random_state':[42]}
    regr = GridSearchCV(MultiOutputRegressor(AdaBoostRegressor()), param_grid=params,
                        n_jobs=-1, cv=2, scoring=weighted_harmonic_f1_scorer, verbose=0)
    start_time = time.time()
    regr.fit(sentence, np.asarray(y_train_val[['pos_sent', 'neg_sent']].values.tolist()))
    elapsed_time = time.time() - start_time
```

Listing 15: Code snippet for the third classifier, AdaBoost.

TRAIN SCORE	RandomForests	GradientBoosting	AdaBoost
CBOW average	0.48	0.54	0.48
CBOW summing	0.52	0.56	0.5
CBOW minmax	0.08	0.08	0.48
Skip-Grams average	0.49	0.55	0.48
Skip-Grams summing	0.51	0.56	0.51
Skip-Grams minmax	0.08	0.08	0.48
FastText average	0.47	0.54	0.48
FastText summing	0.52	0.56	0.49
FastText minmax	0.08	0.08	0.48
GloVe average	0.08	0.08	0.48
GloVe summing	0.08	0.08	0.48
GloVe minmax	0.08	0.08	0.48
BoW average	0.32	0.65	0.25
BoW summing	0.32	0.65	0.48
BoW minmax	0.32	0.64	0.48
TF-IDF average	0.32	0.65	0.48
TF-IDF summing	0.32	0.65	0.5
TF-IDF minmax	0.32	0.65	0.48

TEST SCORE	RandomForests	GradientBoosting	AdaBoost
CBOW average	0.48	0.53	0.48
CBOW summing	0.52	0.56	0.5
CBOW minmax	0.08	0.08	0.48
Skip-Grams average	0.49	0.54	0.47
Skip-Grams summing	0.51	0.55	0.51
Skip-Grams minmax	0.08	0.08	0.48
FastText average	0.47	0.53	0.48
FastText summing	0.52	0.55	0.49
FastText minmax	0.08	0.08	0.48
GloVe average	0.08	0.08	0.48
GloVe summing	0.08	0.08	0.48
GloVe minmax	0.08	0.08	0.48
BoW average	0.32	0.65	0.25
BoW summing	0.32	0.65	0.48
BoW minmax	0.32	0.64	0.48
TF-IDF average	0.32	0.65	0.48
TF-IDF summing	0.32	0.65	0.5
TF-IDF minmax	0.32	0.65	0.48

TABLE I: Training and test results of all classifiers. The scoring was performed using the weighted harmonic F1 score.

EXECUTION TIME	RandomForests	GradientBoosting	AdaBoost
CBOW average	22.63	189.48	15.94
CBOW summing	34.42	188.15	18.28
CBOW minmax	3.42	232.97	228.01
Skip-Grams average	34.16	186.13	20.1
Skip-Grams summing	16.32	178.82	18.34
Skip-Grams minmax	3.05	7.09	7.68
FastText average	271.31	179.63	13.47
FastText summing	32.63	178.91	21.94
FastText minmax	3.06	4.42	6.2
GloVe average	2.2	2.48	236.42
GloVe summing	1.9	3.62	4.33
GloVe minmax	3.76	225.92	223.53
BoW average	45.85	66.99	78.58
BoW summing	248.67	59.41	47.88
BoW minmax	96.62	196.91	120.23
TF-IDF average	61.06	106.1	54.52
TF-IDF summing	57.18	115.27	83.28
TF-IDF minmax	102.66	234.46	118.18

TABLE II: Execution time of the best model found with GridSearchCV for all classifiers, measured in seconds.

Q9: Performance comparison

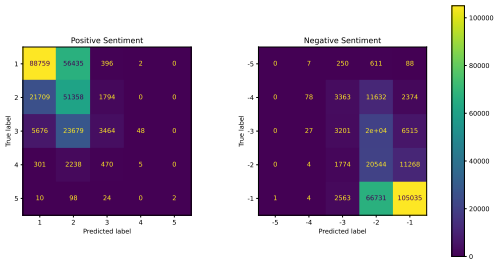
As can be seen in Table II, the execution time varies a lot, and is difficult to compare between classifiers and embedding/aggregator models due to external interference and computer usage. Still, in general, it seems like using the 'minmax' aggregator greatly increases training time, likely due to the increased size of the feature vectors.

In addition, when looking at the weighted harmonic F1 score (Table I, the 'summing' aggregator seems to produce the best results when comparing aggregator models within each embedding model. Similarly, our GradientBoosting 'classifier', generally outperformed other classifier models with the same aggregator method and embedding model. The only exceptions were first of all the GloVe models, which ended up producing better results when trained with AdaBoost, but secondly also the 'minmax' aggregator, which produced better results (for the Gensim-based models) when trained with AdaBoost.

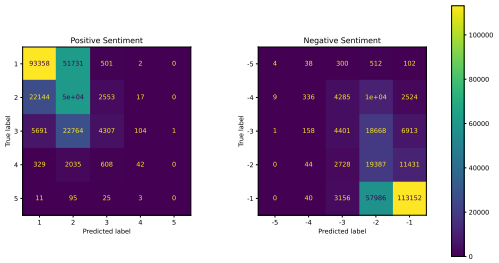
Somewhat suprising is the good results that were obtained using the simplest embedding methods: BoW and TF-IDF. This might be partially caused by the limitation of the vocabulary size, which affected the aggregator models differently than the Gensim-based models and also allowed the classifier to focus on the most frequently occurring words.

If we look at the confusion matrices of the models that have a final training score of ≥ 0.54 (Figures 15 and 16), we see that (likely due to class imbalance), very positive and very negative tweets get misclassified most often. Therefore, we would like a model that prevents this somewhat. Our final choice, taking our computational restrictions into account, is therefore to use TF-IDF for word embedding, averaging or summing as aggregation method, and Gradient Boosting as the training method. Although not the fastest, both these options strongly outperform most of the other models, even though the vocabulary size is limited. Since OOV words cannot be accounted for with TF-IDF, a Gradient Boosting model trained with FastText embeddings aggregated by summing might still perform reasonably, and is thus considered a strong runner up for our final choice, especially since the vector size was limited in the embedding model.

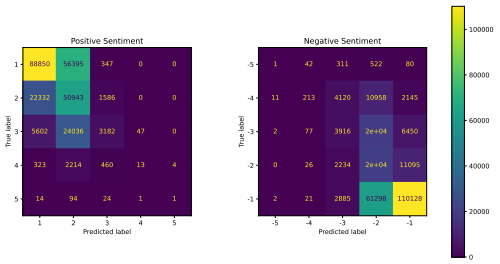
As mentioned in the previous paragraph, one major thing we limited, was the final vector size of our embedding models. This was done to facilitate training. Still, for the Gensim models, the vector size we chose (v=20) vastly differs from the ones mentioned in the original papers (v=300), so using a larger vector size can potentially improve our results. The same holds for



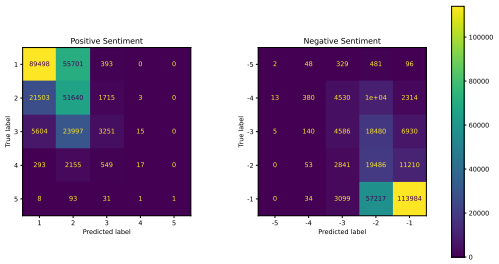
(a) Confusion matrix for the Gradient Boosting 'classifier' trained on CBOW-embeddings that were aggregated by averaging.



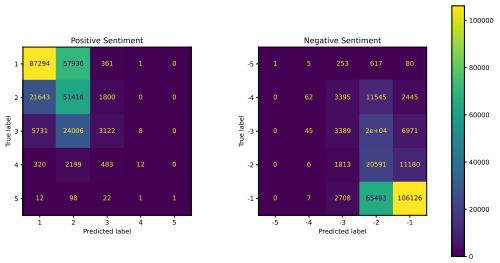
(b) Confusion matrix for the Gradient Boosting 'classifier' trained on CBOW-embeddings that were aggregated by summing.



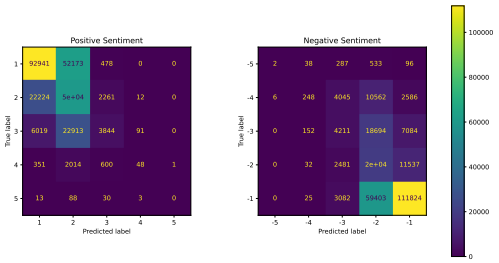
(c) Confusion matrix for the Gradient Boosting 'classifier' trained on Skip-gram embeddings that were aggregated by averaging.



(d) Confusion matrix for the Gradient Boosting 'classifier' trained on Skip-gram embeddings that were aggregated by summing.



(e) Confusion matrix for the Gradient Boosting 'classifier' trained on FastText embeddings that were aggregated by averaging.



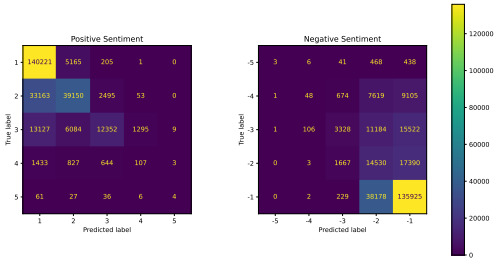
(f) Confusion matrix for the Gradient Boosting 'classifier' trained on FastText embeddings that were aggregated by summing.

Fig. 15: Confusion matrices of six models trained with the Gradient Boosting algorithm. The models were selected based on a cutoff of ≥ 0.54 for the training score (weighted harmonic F1 score). The confusion matrix was constructed with the test set.

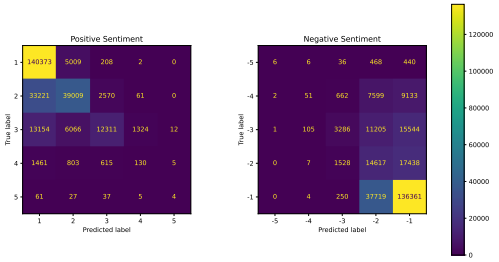
vocabulary size in the TF-IDF and BoW models, which we set at v=250. We expect some classifier results to improve when using larger feature vectors as input for the classification model.

In addition, we could try to train our very own convolutional neural network (CNN) for sentiment classification, which gives us much more flexibility than the usage of predefined models from existing libraries like scikit-learn.

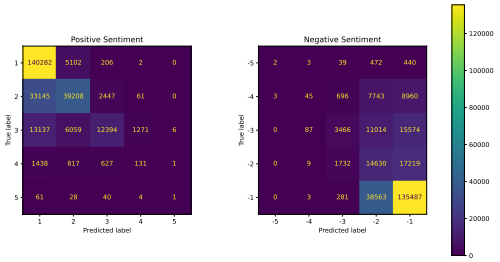
Finally, we could look into classification vs. regression more in depth, and compare how 'real' classifiers perform, which would be especially useful to correct for class imbalance.



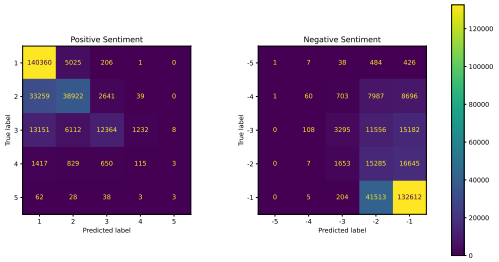
(a) Confusion matrix for the Gradient Boosting 'classifier' trained on BoW-embeddings that were aggregated by averaging.



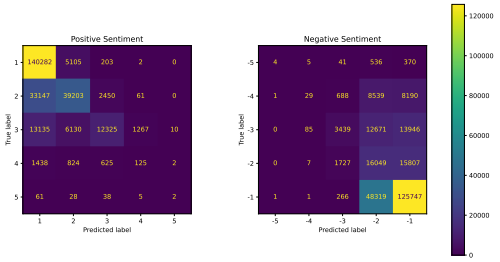
(b) Confusion matrix for the Gradient Boosting 'classifier' trained on TF-IDF-embeddings that were aggregated by averaging.



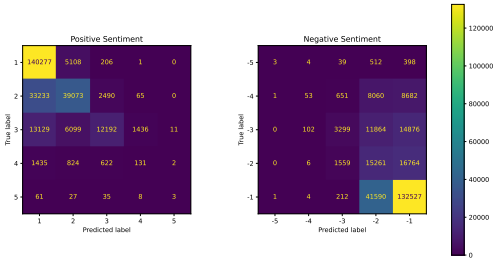
(c) Confusion matrix for the Gradient Boosting 'classifier' trained on BoW-embeddings that were aggregated by summing.



(d) Confusion matrix for the Gradient Boosting 'classifier' trained on TF-IDF embeddings that were aggregated by summing.



(e) Confusion matrix for the Gradient Boosting 'classifier' trained on BoW-embeddings that were aggregated by concatenating the minimum and maximum over all word embeddings.



(f) Confusion matrix for the Gradient Boosting 'classifier' trained on TF-IDF-embeddings that were aggregated by concatenating the minimum and maximum over all word embeddings.

Fig. 16: Confusion matrices of six models trained with the Gradient Boosting algorithm. The models were selected based on a cutoff of ≥ 0.54 for the training score (weighted harmonic F1 score). The confusion matrix was constructed with the test set.

	split (%)	# of samples	15% of the samples
Train	66% * 95% = 63%	246,397	36,960
Validation	66% * 5% = 3%	12,968	1,945
Test	33%	128,585	19,288

TABLE III: Proportions of dataset splits and their sizes used in transformer-based models.

TRANSFORMERS

For the section below, we split the dataset into three parts (training, validation, test). Due to constraints on computational resources especially with transformer-based models, we only consider 15% of each of the splits. Detailed proportions and dataset sizes are shown in table III.

Q1. TRANSFORMER-BASED LANGUAGE MODELS

Transformer-based models are characteristic for using self-attention, which solves several bottlenecks in other models processing longer sequences of words/tokens. Unlike pure recurrent neural networks without self-attention mechanisms, self-attention heads in transformers allow the inference of the representation of input sequence tokens to focus on various other tokens of the input sequence equally. Without attention, the information from neighbouring tokens would vanish more easily with increasing distance between the tokens. Overall, this allows the representation of each token to be highly contextualised in the token sequence (such as a sentence), also if important relationships between input sequence tokens involve words that are far apart.

Some important architectural differences, which determine the number of parameters, are the following:

- Training objectives. Language models differ in their objective functions, which determine the kind of tasks they would perform well.
- Building blocks in the model's architecture, such as self-attention heads, uni- or bi-directional long short-term memory units, gated recurrent units, etc.
- Size of the model, including number of hidden layers, number of connections between layers, sizes of embedding, sizes of hidden states of recurrent units, and the like.
- Training data and where they come from.

Below, we briefly compare the three models BERT, RoBERTa and GPT.

BERT and RoBERTa are similar models in that both are usually used in downstream tasks such as classification, language inference, question answering, and similar, with RoBERTa being a refined BERT model trained on a larger corpus. On the other hand, GPT is a generative model with the purpose of generating larger amounts of text.

BERT-based models are usually trained with masked language modelling and next-sentence prediction.

- Masked language modelling means that the model is trained to correctly guess masked words from their context in both directions. Using a bidirectional transformer, the model can base the prediction of a masked token also on future tokens in the sequence.
- Next sequence prediction – for a pair of sentences, the model learns to predict whether the sentences are consecutive.

In contrast to this, GPT models are trained to predict the next word in a sequence. Given a large corpus of human-generated text, training objectives are generated such that the model is given some prefix of a token sequence and trained to predict the next word of the sentence. The model uses a unidirectional transformer and so cannot make use of future tokens in the sentence for the prediction of the target token.

Q2. SCALABILITY

Embedding-based models usually work on the word level and have a fixed embedding vector for each word. The embedding vectors are calculated during a much simpler training procedure, have a smaller number of parameters, and require smaller corpora. As opposed to word embedding methods, transformer-based language models can produce contextualised embedding vectors for each input token which may lead to higher performance for a given task, but that entails much larger model sizes. Due to increased model complexity, transformer-based models require more computation resources and larger training datasets.

As for the trade-offs, embedding-based models can be more suitable for applications that require faster prediction, smaller memory usage, or lower computation resources. However, transformer-based models, if they can be afforded in terms of computation resources and available corpora, usually perform better.

Q3. CODE

We chose to use the `distilbert-base-uncased-finetuned-sst-2-english` model which is a lightweight version of the BERT model with fewer parameters (only 6 hidden layers). Further, the model is fine-tuned for sentiment classification, so it should perform well even without much training.

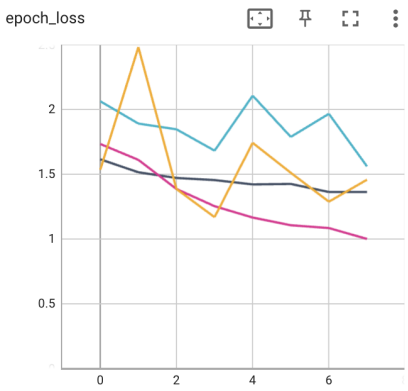


Fig. 17: MSE loss values over the training of 8 epochs, on the training and validation datasets.

- **training** (grey ■ for the `fixed` model, rose ■ for the `refined1` model),
- **validation** (blue ■ for the `fixed` model, orange ■ for the `refined1` model).

The `bottom1` model is not included for brevity.

However, our task is different from the usual sentiment analysis: we need two outputs, one for positive and one for negative sentiment variable. This will require at least some training, since we are constructing new layers with a different number of parameters than the standard model.

Further, we chose to model this as a regression problem, even though the reported metric will be the classification metric *Weighted-F1 HM*. Therefore we use the *MSE* loss for both sentiment variables, adding the loss values together, to optimise the regression.

First, as shown in listing 16, we create a new class `TFDistilBertForSentiStrengthRegression`, subclassing from `TFDistilBertForSequenceClassification`. Our code, as opposed to its super class, modifies the last two layers. We use the following last layers:

- A hidden layer with the size of 768 (the default hidden state size for distilbert) with the ReLU activation function, similar to previous hidden layers of the model. This layer takes as input the embedding corresponding to the `[CLS]` token, i.e. the embedding that is trained to capture the sentiment of the whole sentence. This is done in the same way as the `TFDistilBertForSequenceClassification` model.
- An output layer with two outputs, one for each sentiment variable. Since we are modelling the problem as regression directly of the given values, we apply no activation function on the two output neurons (i.e. *linear*).

Further, in listing 17 we demonstrate how to build the model, with a custom number of layers that are set as trainable. The included `build_model` function does the following:

- Initialize the `TFDistilBertForSentiStrengthRegression`
- Based on the provided argument, the function iterates through the layers of the pre-trained model and sets them as trainable respectively. By default, none of the pre-trained layers will be set as trainable.
- Compiles the model with the Adam optimizer, using the `mse_sum` loss function that adds the MSE loss value of the two sentiment variables.

In listing 18 we show how to call the `build_model` function to construct the model with different sets of layers set to be fine-tuned. Finally, in listing 19 we show how to train the models.

Q4, Q5: Performance analysis, Transfer learning details

As per listing 19, we trained the three models (one fixed, one with the last layer trainable, and one with the first (bottom) layer trainable) for 8 epochs. In all cases the cumulative loss continued decreasing, and the validation Weighted-F1 HM continued increasing overall. However, we didn't train it for more epochs due to computational constraints. The training and validation loss function are shown in fig. 17; validation metrics are shown in fig. 18.

The `fixed` model (without fine-tuning pre-trained layers) achieved a score (*Weighted-F1 HM*) of **0.48**, the `refined1` model (with the last pre-trained layer fine-tuneable) achieved a score of **0.60**, and the `bottom1` model (with the first pre-trained layer fine-tuneable) achieved a score of **0.63**.

The fine-tuned models were able to specialize for the task better in terms of the predictive performance. Not only the training loss was able to decline lower, but also the `f1w_hm` metric on the validation dataset rose higher over the training epochs. This confirms that fine-tuning the last layer of the method helps increase the performance.

Suggested approaches to improve the performance if computational resources were not a bottleneck:

- Train on a larger dataset. As described at the beginning of the section Transformers, we only consider 15% of the provided dataset.
- Train for more epochs. As clearly seen from fig. 18a (harmonic weighted F1 grows) and fig. 17 (loss decreases), continuing the training for longer has a high predisposition for leading to better performance.
- Use a larger transformer model or allow fine-tuning of more layers of the selected model. Here, we opted for DistilBERT, which is a smaller, distilled version of BERT, but options with more layers (requiring more computing resources) have

Listing 16: A code snipped showing our implementation of `TFDistilBertForSentiStrengthRegression`.

```python
from transformers import TFDistilBertMainLayer
from transformers.modeling_tf_utils import get_initializer, unpack_inputs, TFModelInputType
from transformers.utils.generic import ModelOutput
from typing import Optional, Union, Tuple
from dataclasses import dataclass

@dataclass
class TFSentiStrengthRegressionOutput(ModelOutput):
    loss: Optional[tf.Tensor] = None
    values: Optional[tf.Tensor] = None
    hidden_states: Optional[Tuple[tf.Tensor]] = None
    attentions: Optional[Tuple[tf.Tensor]] = None

class TFDistilBertForSentiStrengthRegression(TFDistilBertForSequenceClassification):
    def __init__(self, config, *inputs, **kwargs):
        super(TFDistilBertForSequenceClassification, self).__init__(config, *inputs, **kwargs)

        self.distilbert = TFDistilBertMainLayer(config, name="distilbert")
        self.pre_regression = tf.keras.layers.Dense(
            config.dim,
            kernel_initializer=get_initializer(config.initializer_range),
            activation="relu",
            name="pre_regression",
        )
        self.regression = tf.keras.layers.Dense(
            units=2, kernel_initializer=get_initializer(config.initializer_range), name="regression",
            activation='linear'
        )
        self.dropout = tf.keras.layers.Dropout(config.seq_classif_dropout)

    @unpack_inputs
    def call(
            self,
            input_ids: Optional[TFModelInputType] = None,
            attention_mask: Optional[Union[np.ndarray, tf.Tensor]] = None,
            head_mask: Optional[Union[np.ndarray, tf.Tensor]] = None,
            inputs_embeds: Optional[Union[np.ndarray, tf.Tensor]] = None,
            output_attentions: Optional[bool] = None,
            output_hidden_states: Optional[bool] = None,
            return_dict: Optional[bool] = None,
            labels: Optional[Union[np.ndarray, tf.Tensor]] = None,
            training: Optional[bool] = False,
    ) -> Union[TFSentiStrengthRegressionOutput, Tuple[tf.Tensor]]:
        distilbert_output = self.distilbert(
            input_ids=input_ids,
            attention_mask=attention_mask,
            head_mask=head_mask,
            inputs_embeds=inputs_embeds,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=return_dict,
            training=training,
        )
        hidden_state = distilbert_output[0]  # (bs, seq_len, dim)
        pooled_output = hidden_state[:, 0]  # (bs, dim)
        pooled_output = self.pre_regression(pooled_output)  # (bs, dim)
        pooled_output = self.dropout(pooled_output, training=training)  # (bs, dim)
        values = self.regression(pooled_output)  # (bs, dim)

        loss_fn = tf.keras.losses.MeanSquaredError(reduction=tf.keras.losses.Reduction.NONE)
        loss = None if labels is None else loss_fn(labels, values)

        if not return_dict:
            output = (values,) + distilbert_output[1:]
            return ((loss,) + output) if loss is not None else output

        return TFSentiStrengthRegressionOutput(
            loss=loss,
            values=values,
            hidden_states=distilbert_output.hidden_states,
            attentions=distilbert_output.attentions,
        )
```

Listing 17: A code snipped showing how to build and compile the tensorflow model.

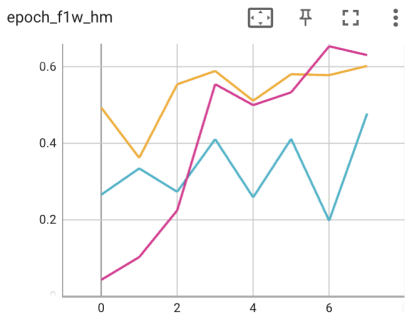
```
def build_model(trainable_last_layers=0, trainable_last_layers_set=[]):
    custom_bert = TFDistilBertForSentiStrengthRegression.from_pretrained(MODEL_NAME)
    if trainable_last_layers > 0 or len(trainable_last_layers_set) > 0:
        custom_bert.distilbert.trainable = True
        custom_bert.distilbert.embeddings.trainable = False
        if len(trainable_last_layers_set) > 0:
            for layer in custom_bert.distilbert.transformer.layer:
                layer.trainable = False
            for layer in trainable_last_layers_set:
                custom_bert.distilbert.transformer.layer[layer].trainable = True
        else:
            for layer in custom_bert.distilbert.transformer.layer[0:-trainable_last_layers]:
                layer.trainable = False
    else:
        custom_bert.distilbert.trainable = False

    input_ids = tf.keras.layers.Input(shape=(None,), dtype=tf.int32, name="input_ids")
    outputs = custom_bert(input_ids).values

    model = tf.keras.models.Model(
        inputs=[input_ids],
        outputs=[outputs],
        name=MODEL_NAME_SHORT,)

    def mse_sum(y_true, y_pred):
        return tf.keras.metrics.mean_squared_error(y_true[0], y_pred[0]) +
        ↪  tf.keras.metrics.mean_squared_error(y_true[1], y_pred[1])
    def mae_sum(y_true, y_pred):
        return tf.keras.metrics.mean_absolute_error(y_true[0], y_pred[0]) +
        ↪  tf.keras.metrics.mean_absolute_error(y_true[1], y_pred[1])
    def mae_pos(y_true, y_pred):
        return tf.keras.metrics.mean_absolute_error(y_true[0], y_pred[0])
    def mae_neg(y_true, y_pred):
        return tf.keras.metrics.mean_absolute_error(y_true[1], y_pred[1])

    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
        loss=mse_sum,
        metrics=[mae_sum, mae_pos, mae_neg],)
    return model
```

Listing 18: A code snipped showing how to call the `build_model` function.

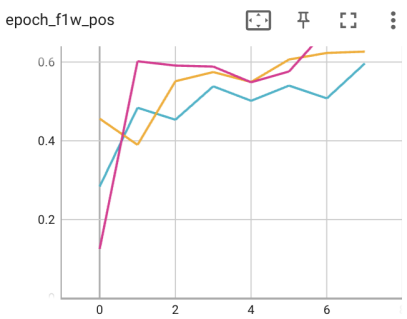
```
# DistilBERT with fixed pre-trained layers, only set to train the added regression layers
model_fixed = build_model(trainable_last_layers_set=[])
# DistilBERT also allowing the last pre-trained layer to be fine-tuned
model_refined1 = build_model(trainable_last_layers_set=[5])
# DistilBERT allowing the first (bottom) pre-trained layer to be fine-tuned
model_bottom1 = build_model(trainable_last_layers_set=[0])
```

Listing 19: A code snipped showing how to train the model.

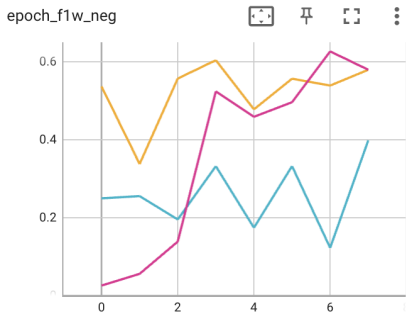
```
def train_model(name, model, epochs=10, train_data=train_data, val_data=val_data):
    # Create callbacks
    metrics = ... # A callback that calculates our metrics (including val_f1w_hm) on the validation dataset
    model_checkpoint_callback = ... # Save the model after every epoch
    tensorboard_callback = ...

    history = model.fit(
        x=train_data,
        validation_data=val_data,
        epochs=epochs,
        callbacks=[
            ...
        ],
    )
    return history

history_fixed = train_model('fixed', model_fixed, epochs=8)
```



(a) The harmonic mean of the Weighted-F1 scores of the two (positive, negative) sentiment variables.



(b) Weighted-F1 score of the positive sentiment variable.



(c) Weighted-F1 score of the negative sentiment variable.

Fig. 18: Three reported metrics for the positive (fig. 18b) and negative (fig. 18c) sentiment variables evaluated on the validation dataset, and their harmonic mean (fig. 18a, the main metric), after each of 8 training epochs.
Blue ■ corresponds to the `fixed` model, orange ■ corresponds to the `refined1` model, and rose ■ corresponds to the `bottom1` model.

1	@TheMomKind This has happened to a few families...scary stuff...
2	Joe Biden presents symptoms that ought to concern everyone. @DrBenChavis @NCPoliticalSpin @ChathCoLineNews @afaduln2 @moby-grapefan @Flwrgirl66x @Laura78703 @MaRobbie @shannondjackson @TulsiGabbard @ChuckRocha @TheBernReport @davidsirota @daviddoel @KyleKulinski @cenkuygur
3	Coronavirus: Scientists say anti-parasite drug could kill virus https://t.co/BkRLwU12IR
4	In these trying times when so many are testing positive for coronavirus we should have tremendous gratitude that Hillary Clinton still tests negative for president of the United States.
5	Study out of New York may change everything we thought we knew about fevers and coronavirus https://t.co/70yUs5HGNa
6	Manchester City are approaching the situation with a "clear, calm head", according to senior sources, and have been laying the ground work for months in the event of a ban from UEFA. A robust legal challenge is expected to be lodged on appeal with CAS. [@TeleFootball]
7	I'm a little disappointed but shit happens
8	I have been struggling to breathe with this Coronavirus. But I have been listening non stop to this worship song Unstoppable God by Sanctus Real. It has lifted my soul as Coronavirus attacked my lungs. Where does my help come, my help comes from the Lord. #conronavirus #COVID19 https://t.co/OEJrZvTv0J
9	44 migrants on one US deportation flight tested positive for coronavirus https://t.co/YqZDhacOWE
10	I can't wait for that day when this pandemic is over. Hoping for better days!

TABLE IV: The full unprocessed text of ten selected tweets for visualisation purposes.

the potential to perform better. However, this approach would mostly make sense if the training reaches the bottleneck given by the complexity of the model.

Q6. Embedding analysis

In this question, we investigate what effect fine-tuning of the last layer of the pre-trained DistilBERT transformer model has on tweet embeddings. Therefore, we study the embeddings of tweets produced by two models, `fixed` (with all pre-trained layers fixed), and `refined1`, with the last pre-trained layer fine-tuned. First, we explain and clarify a few concepts used in this section.

Tweet embeddings model. Given a trained transformer model, we modify it such that it returns the embedding of the first token (`[CLS]`) of a given tweet. The transformer-based language models are trained such that the embedding of the `[CLS]` token (i.e. the first token after tokenisation) has information about the sentiment of the whole input sequence, that is why we choose to visualise the embedding of this token. Given a tweet, evaluating the model on the pre-processed and tokenized version of the tweet results in a 768-dimensional embedding of the tweet.

Selected tweets: 1000 + 10. We randomly sampled 1000 tweets from the test set (disjoint from the training set) to use for dimensionality reduction; not more due to computational constraints. For some visualisations in this section, we manually selected an additional set of ten tweets (also distinct from the 1000), which we manually verified are related to Covid-19. The embedding of those ten tweets can be visualised and annotated with the text. These ten tweets' whole (unprocessed and not tokenized) text is shown in table IV.

Dimensionality reduction functions: PCA, UMAP. When visualizing the ten selected tweets (table IV), we need to use a dimensionality reduction function trained on a bigger sample set since 10 tweets alone would not be enough to capture the relationships of tweets in the whole test dataset. Therefore, we train the dimensionality reduction functions on the set of 1000 tweets. Further, we argue that the training of dimensionality reduction should be independent from the ten tweets that we later visualize, to avoid bias e.g. when setting the hyperparameters of the dimensionality reduction function. Thus, we opt for the methods PCA and UMAP [14], which give us a function that we can apply to new data (as opposed to t-SNE [13], which cannot be applied to new data).

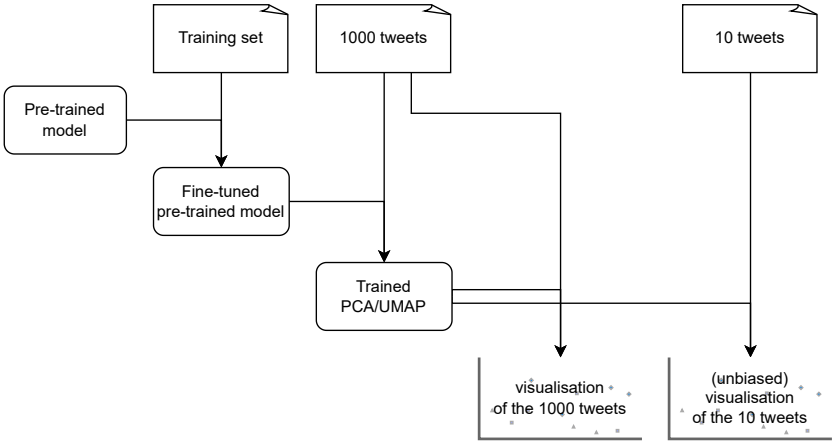


Fig. 19: Diagram showing how plots visualising tweet embeddings are obtained from three splits of the tweet dataset.

Figure 19 shows the procedure to obtain visualisations of the two sets of selected tweets.

Positive sentiment. In this section, we show plots where samples are colored by the positive sentiment variable. However, similar results and conclusions can be derived from plotting the negative sentiment variable; we don't include the plots for brevity.

As the first try, we tried to compare the embeddings of the 10 selected tweets, output by the two models `fixed` and `refined1`. This is visualized in fig. 20. However, from inspecting the plots, we conclude that the fine-tuning of the model changes the meanings of the embedding dimensions enough such that combining tweet embeddings produced by the two models in the same embedding space doesn't convey useful information. In other words, the embedding space of the `refined1` model seems to have a different representation than the embedding space of the `fixed` model.

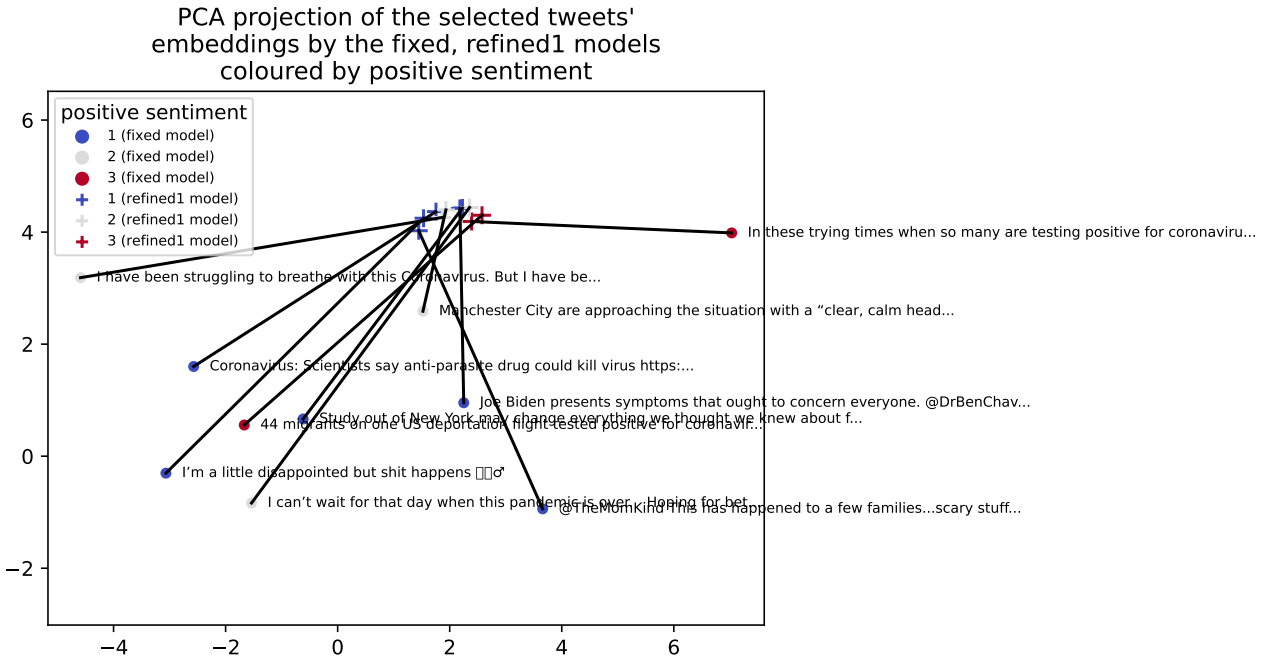
On our second try, we visualised the embeddings separately for the two models. The result is shown in fig. 21 (for the `fixed` model) and fig. 22 (for the `refined1` model). Finally, we also visualised all 1000 embedding vectors that were used to train the PCA and UMAP. Results are shown in fig. 23.

The parameters set for UMAP in all visualisations in this section are `n_neighbors=30`, `min_dist=0.2`.

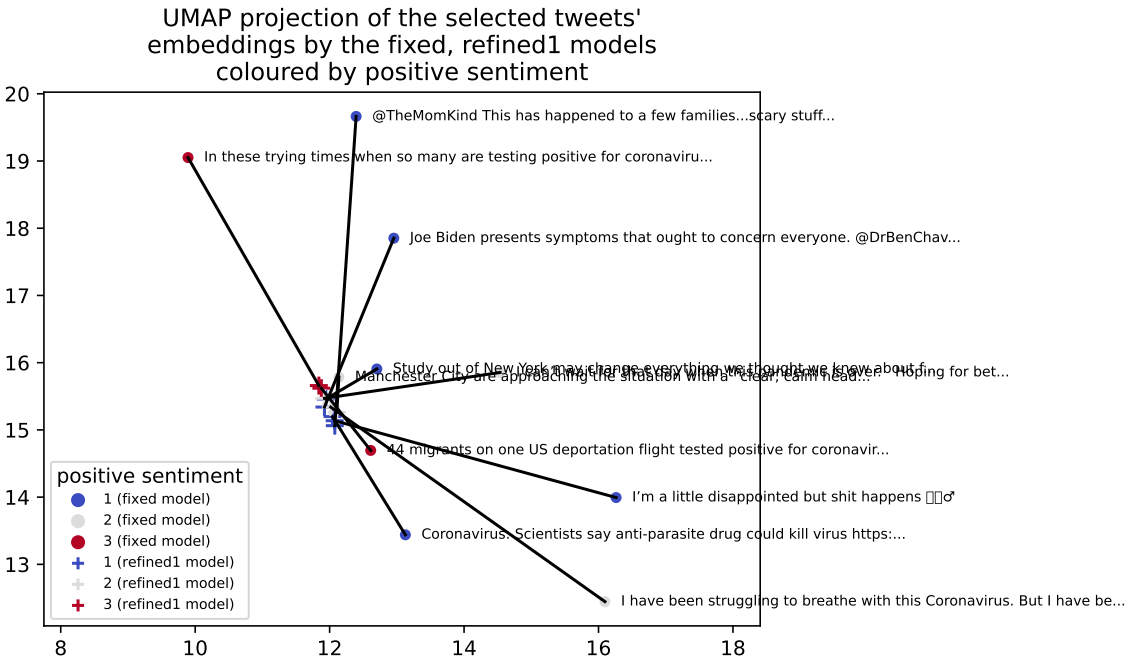
Conclusions. In the scatter plots, we can observe the following: None of the PCA and UMAP plots of the embeddings from the `fixed` model (i.e. fig. 21, fig. 23a, fig. 23b) show any clustering related to the positive sentiment, in that tweets of all positive-sentiment values, are distributed relatively uniformly across the embedding space. This may be a natural consequence of the fact that these embeddings are produced by a pre-trained model. During training for our task (regression of two sentiment values), we only trained the weights of two layers built on top of the tweet embedding. Thus, no weights that affect the embedding were changed during this training, and so the embedding of tweets is exactly as would be output by the original pre-trained model. One might still expect the embedding to convey information relevant to sentiment prediction since the used model "`distilbert-base-uncased-finetuned-sst-2-english`" is pre-trained on the Stanford Sentiment Treebank [22] corpora. However, apparently, the original problem and the problem in this project are different enough.

However, in scatter plots of embeddings from the `refined1` model (fig. 22, fig. 23c, fig. 23d), we see the separation of tweets based on their positive sentiment value, while the only connection between those scatter plots and the positive sentiment variable is that the last pre-trained layer of the `refined1` model was fine-tuned to predict the positive and negative sentiment variables. As a specific example, the y-axis of the PCA plot in fig. 23c is highly correlated with the positive sentiment variable: tweets labelled with high positive sentiment are mostly present at higher y values in the plot. Further, in the UMAP in fig. 23d, tweets with higher positive sentiment values are nicely separated from the low-positive-sentiment tweets, and so even a clustering algorithm on the 2D UMAP would likely lead to better-than-random performance. As demonstrated, it suffices to fine-tune the last layer (the ultimate pre-embedding layer) to make the embeddings specialised and convey information about the positive sentiment variable. We conclude that fine-tuning the model does result in the embeddings being more specialized for the given task.

Compared to visualising word embeddings in Part 2, Q6, we see that the separation of positive and negative tweets using transformer-based models is visually stronger than the separation of positive and negative words as shown in Part 2, Q6, where the performance of a simple clustering algorithm on the 2D embedding would likely be poor.

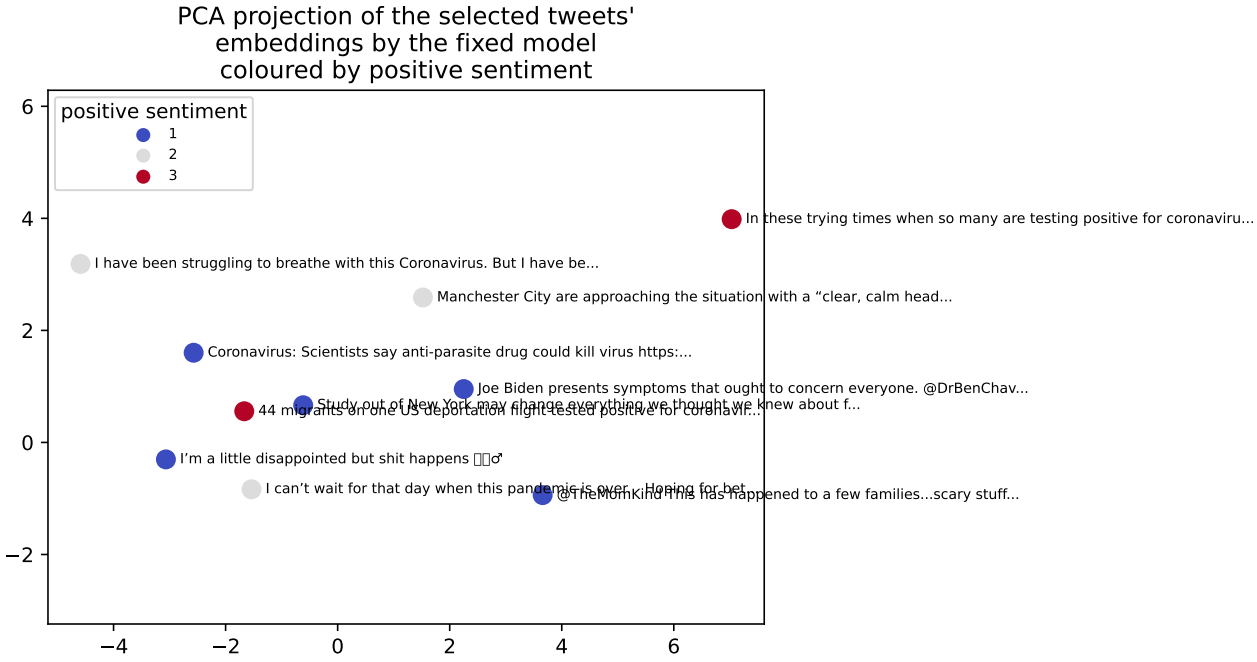


(a)

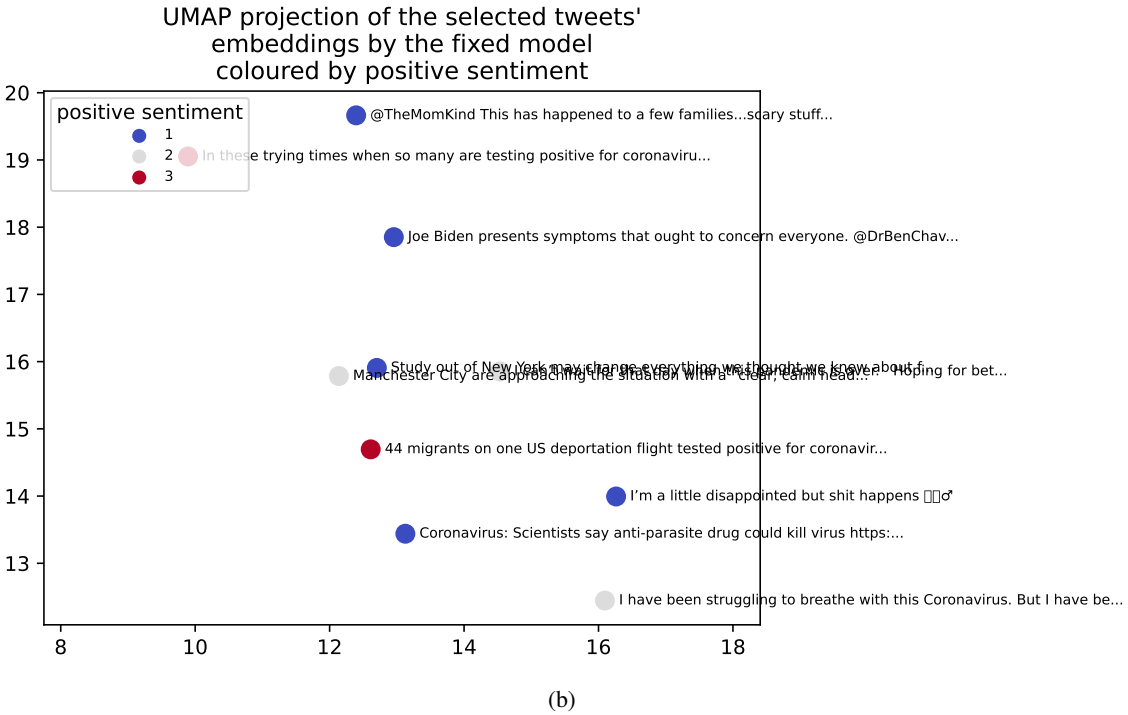


(b)

Fig. 20: Visualisation of the embeddings of the 10 selected tweets, output by the two models `fixed`, `refined1`. The dimensions are reduced to 2D using the same dimensionality reduction function (fig. 20a PCA, fig. 20b UMAP), trained on the 1000 embedding-tweets set output by the `fixed` model.

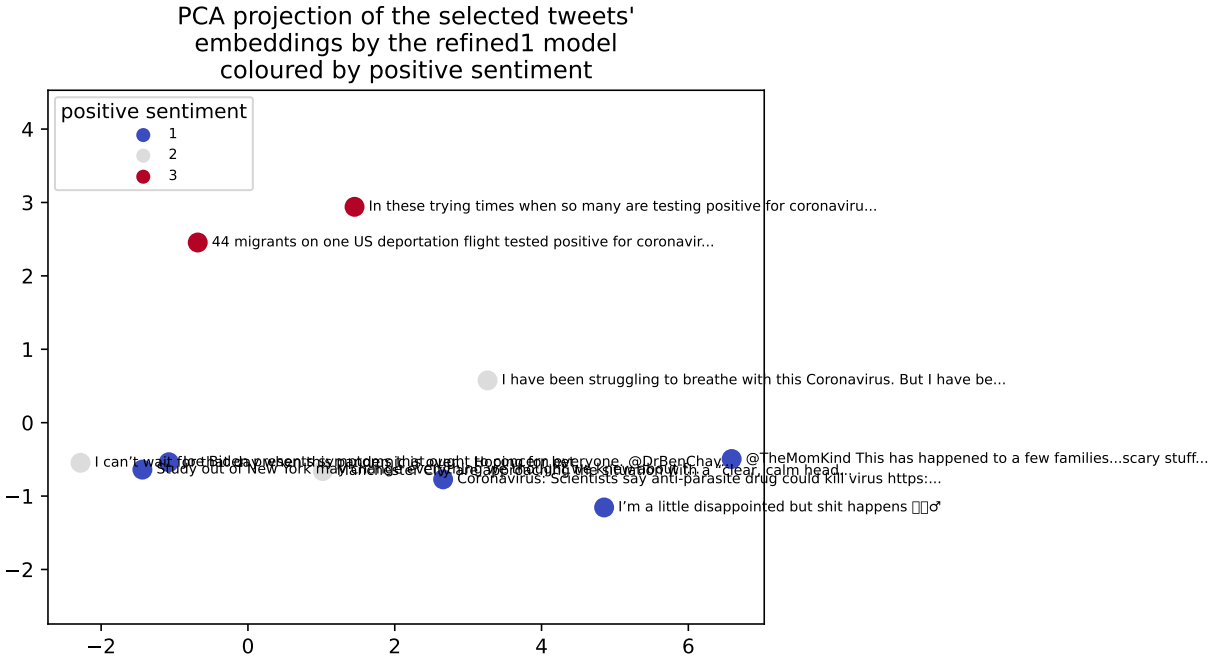


(a)

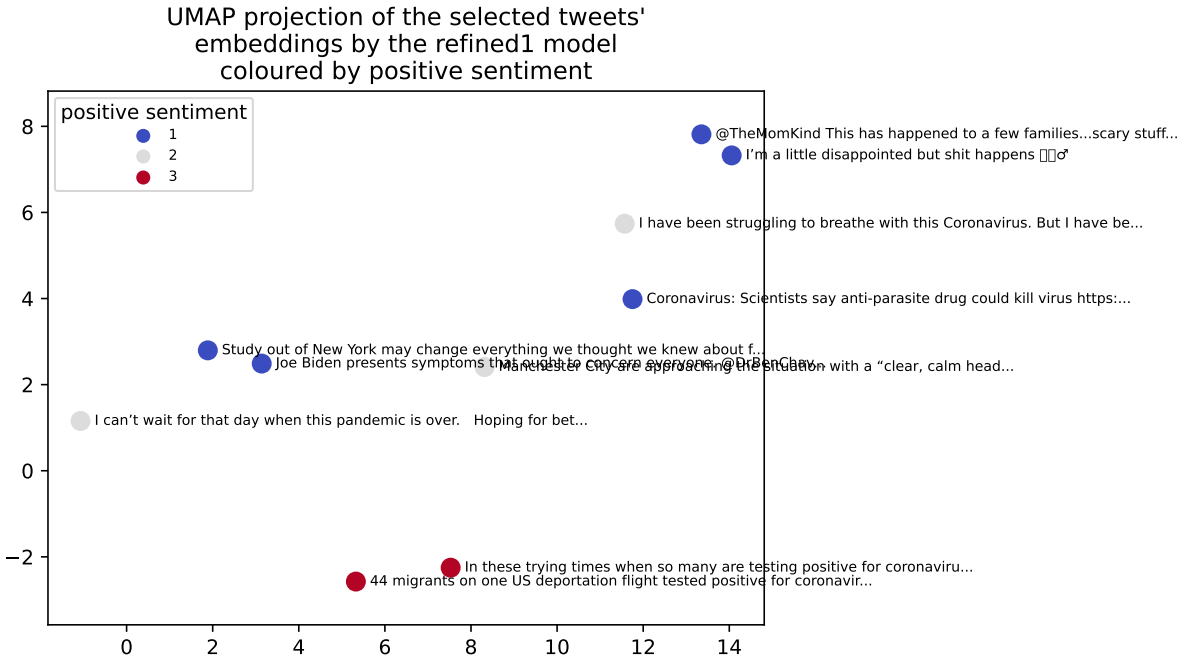


(b)

Fig. 21: Visualisation of the embeddings of the 10 selected tweets, output by the `fixed` model, using fig. 21a PCA and fig. 21b UMAP dimensionality reduction functions trained on the 1000 embedding-tweets set output by the `fixed` model.



(a)



(b)

Fig. 22: Visualisation of the embeddings of the 10 selected tweets, output by the `refined1` model, using fig. 22a PCA and fig. 22b UMAP dimensionality reduction functions trained on the 1000 embedding-tweets set output by the `refined1` model.

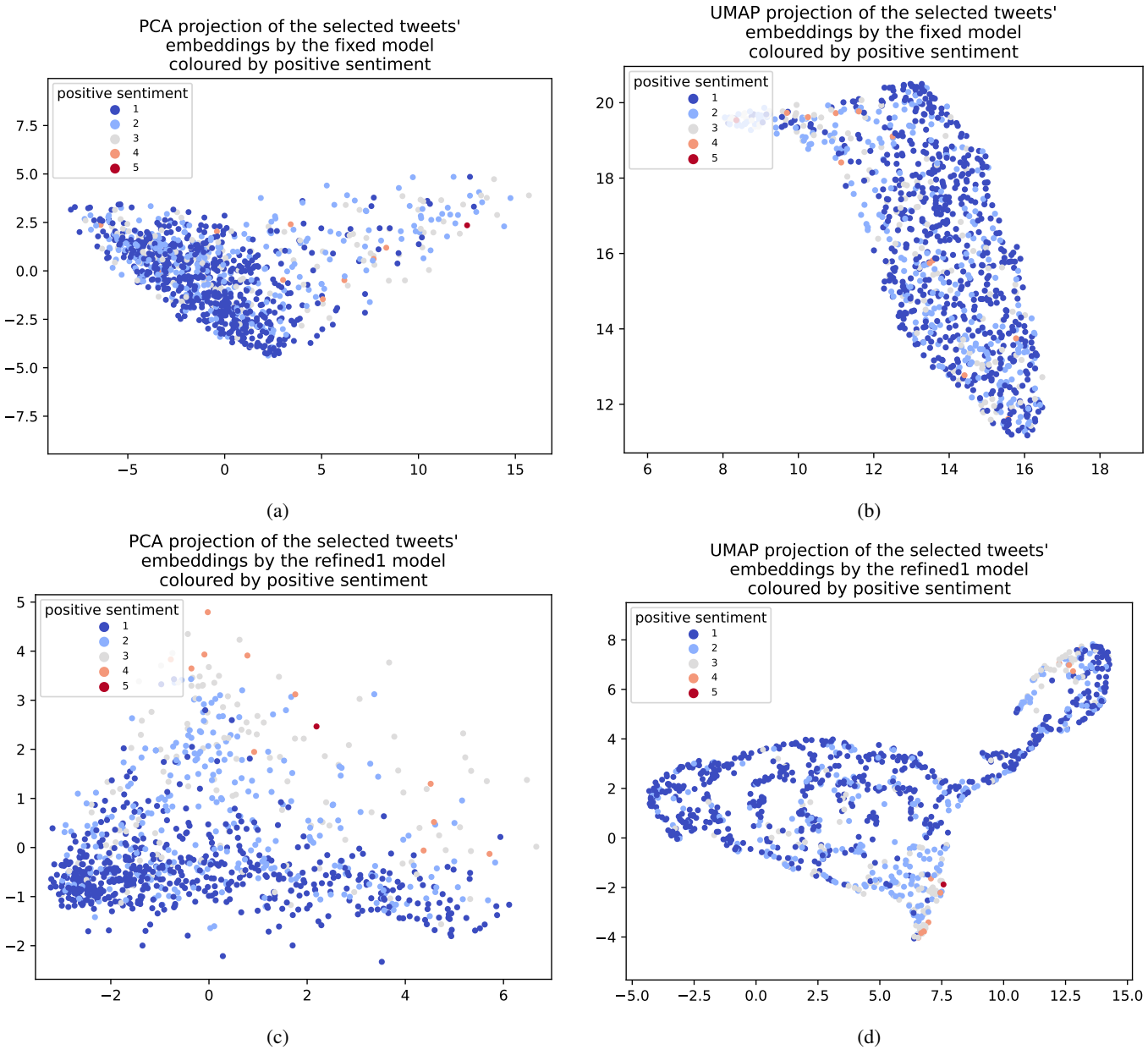


Fig. 23: Visualisation of PCA and UMAP values of the 1000 embeddings that were used to train the dimensionality reduction functions. Results are shown for models `fixed` (fig. 23a PCA, fig. 23b UMAP), `refined` (fig. 23c PCA, fig. 23d UMAP).

Listing 20: Timestamp sorting, resampling weekly, normalizing by number of tweets.

```
# end up with 103k tweets we are sure are in the US
df_tweets_US = df_tweets_US.set_index('Timestamp').sort_values(by='Timestamp', ascending=True)
df_tweets_US.index = df_tweets_US.index.astype('datetime64[ns]')
df_tweets_US[['pos_sent', 'neg_sent']] = df_tweets_US[['pos_sent', 'neg_sent']].astype('int')
df_tweets_US['neg_sent'] = df_tweets_US['neg_sent'].abs()
df_tweets_US = df_tweets_US.rename(columns={'US_or_not': 'TweetCount'})

df_US_weekly = df_tweets_US.resample('W').sum() # sum of positive and negative
df_US_weekly['avg_pos'] = df_US_weekly['pos_sent']/df_US_weekly['TweetCount']
df_US_weekly['avg_neg'] = df_US_weekly['neg_sent']/df_US_weekly['TweetCount']
```

PART 3: DOWNSTREAM GLOBAL HEALTH ANALYSIS

Q1: RESEARCH QUESTION

We have chosen to look at the paper "How epidemic psychology works on Twitter: evolution of responses to the COVID-19 pandemic in the U.S." by Aiello, L.M., Quercia, D., Zhou, K. et al. [1]. We choose to look at the evolution in time of sentiment and emotion with regards to COVID-19, in the United States.

The paper focuses on a theory called "Epidemic Psychology" by Philip Strong [23], which is a sociological study of psycho-social 'epidemics' of fear, moralization and action that occur during major health epidemics. The authors wish to assess whether they can empirically observe these phases of emotion spreading as an epidemic through language, through the analysis of tweets, during the COVID-19 pandemic.

Analyzing the evolution of emotion and sentiment throughout the COVID-19 pandemic through the use of tweets is a very important undertaking, as it helps inform future public health policies. Indeed, as information dissemination is extremely rapid nowadays, the spreading of fear, panic, and the amplifying misinformation, are a grave danger to public health messaging. Understanding past trends can help prevent these feelings that ultimately have a big impact on policy. Finally, we also choose to focus on the U.S. as per the paper, as the U.S. has the highest proportion of Twitter users, and because it is easier to relate evolution of emotions in time to the events of a single country.

Q2: METHOD CHOICE AND DESIGN

We first select tweets whose User Location is in the U.S., and we do so by using regex-based methods of U.S. locations, as per the paper (matching to cities, states, common locations in the U.S., as well as variations on the name 'U.S.'). This is extremely efficient and takes no more than a few seconds to run on the entire dataset, returning approximately 103K U.S.-based tweets from a preprocessed dataset of 382K tweets.

We then run the VADER sentiment analysis on this subset, aggregate the scores of tweets over the course of a week, normalize these figures by the number of total tweets that week, and plot it. The result is shown in fig. 24. We do not provide a code snippet for VADER as it is almost identical to the one in Part 2.

For completeness' sake, we compare with the ground truth labels aggregated and plotted in a similar fashion (the positive labels and absolute values of negative labels are averaged over the course of a week). While this is not sentiment analysis, it provides a comprehensive view of the dataset. This is shown in fig. 25. For completeness, we also plot the number of tweets per week in fig. 26, as this puts our findings in context, and shows the evolution of the interest in the topic of COVID-19.

In listing 20, we show how we sort the time stamps, and aggregate any column into weekly counts and normalize by the number of tweets.

Just like in the paper, we then choose a lexicon, the NCR lexicon (also called Emolex), which categorizes words into 8 emotions (or none): Anger, Anticipation, Disgust, Fear, Joy, Sadness, Surprise, and Trust. For every token in the preprocessed, tokenized, stemmed tweets, we match to the lexicon and count the number of occurrences of each emotion for all tweets. This process is shown in listing 21. As above, we then aggregate our findings over the course of the week, normalize these counts by the number of tweets of that week, and plot our findings as a heatmap.

We then perform change point detection as per the paper, which we plot over this weekly emotion heatmap. The change points and the emotions heatmap are shown in fig. 27, and the code snippet is shown in listing 22. As this is an unsupervised task, we do not propose any quantitative performance metrics to measure the success of our approach. However, we can qualitatively assess our methods, by comparing changes in the plots to certain events occurring in the U.S. at that time that relate to the COVID-19 pandemic.

Q3: RESULTS & ANALYSIS

As stated above, we summarize our findings through the figures shown below.

Looking at the VADER analysis in fig. 24, we observe that while the normalized positive and negative sentiments fluctuate in time, we very clearly observe a clear difference in the week 23/02/2020. Every week always has always had a higher proportion

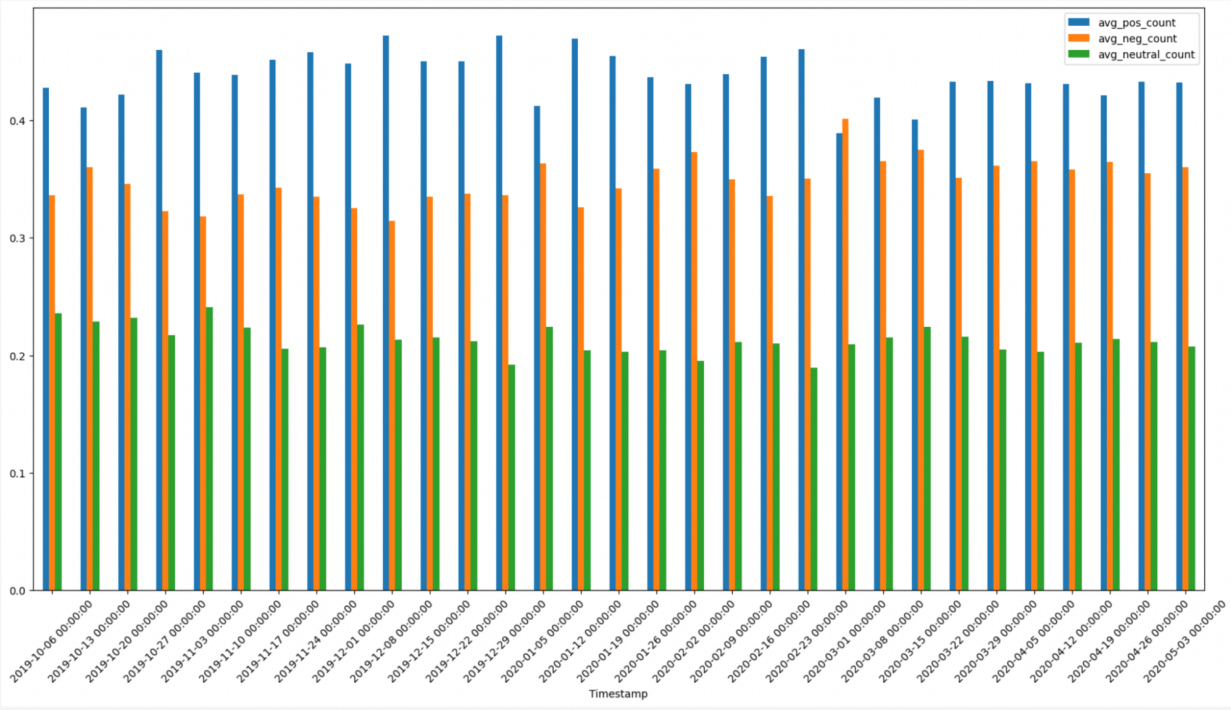


Fig. 24: VADER sentiment scores aggregated weekly and normalized by the number of tweets that week.

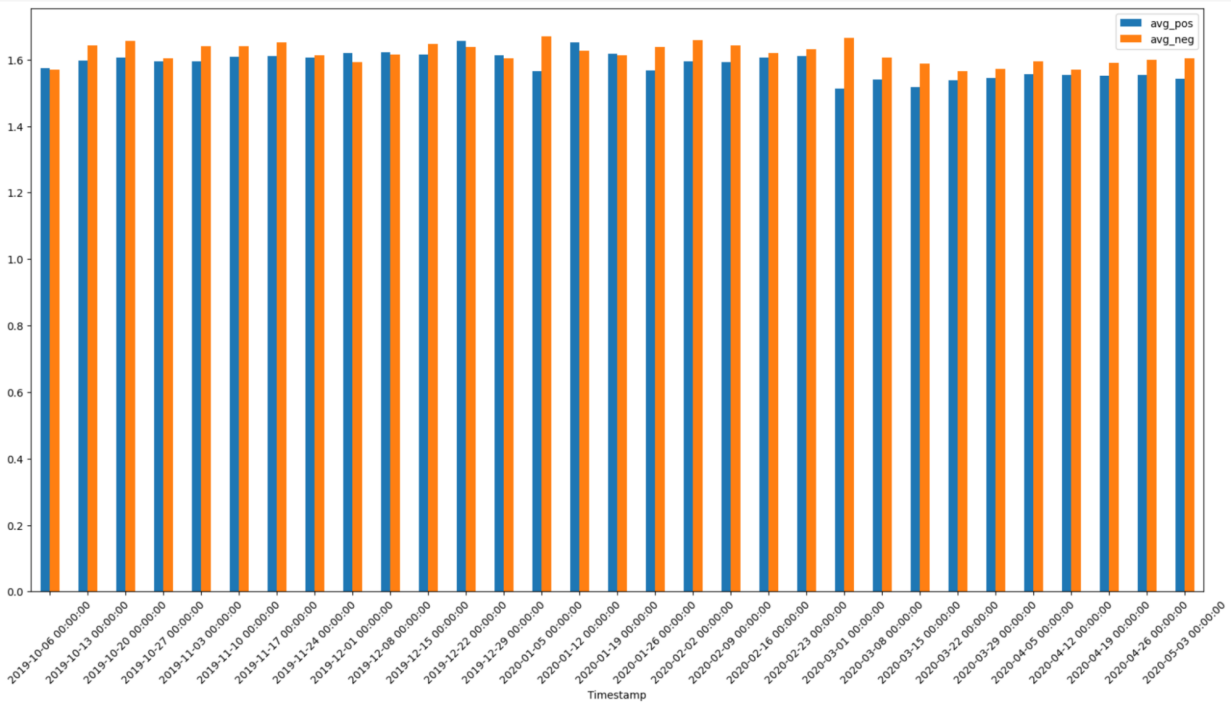


Fig. 25: Ground truth labels averaged over each week (the absolute values of the negative labels are taken).

Listing 21: Emotion analysis on our dataset using the NCR/Emolex lexicon, then resampling weekly and normalization of counts by number of weekly tweets as usual.

```
# Importing the data from the NCR lexicon
ncr = pd.read_csv('NCR-lexicon.csv', sep =';')
emotions = ['Anger', 'Anticipation','Disgust','Fear', 'Joy','Sadness', 'Surprise', 'Trust']
stemmer = SnowballStemmer("english")
df_emotions = pd.DataFrame(0, index=df_tweets_US.index, columns=emotions)

for i, row in df_tweets_US.iterrows(): # for each tweet
   tweet = row['unigram']
   for word in tweet: # for each token in tweet
       word_stemmed = stemmer.stem(word) # stem token
       # check if the word is in NRC
       result = ncr[ncr.English == word_stemmed]
       # we have a match
       if not result.empty:
           # update the tweet-emotions counts
           for idx, emotion in enumerate(emotions):
               df_emotions.loc[i, emotion] += result[emotion].values
df_emotions_weekly = df_emotions.resample('W').sum() # sum of positive and negative
df_emotions_weekly = df_emotions_weekly[['Anger','Anticipation', 'Disgust', 'Fear', 'Joy', 'Sadness',
↪  'Surprise', 'Trust']].div(df_US_weekly.TweetCount, axis=0)
```

Listing 22: Change point detection on weekly emotion heatmap.

```
from scipy.signal import find_peaks
grouped = df_emotions.groupby(pd.Grouper(freq='W'))

gradients_em = pd.DataFrame(grouped.apply(lambda x: np.mean(np.square(np.gradient(x, axis=0)), axis=0)),
↪  columns=['Values'])['Values'].apply(pd.Series)
gradients = pd.DataFrame(grouped.apply(lambda x: np.mean(np.square(np.gradient(x, axis=0)))))
change_points, _ = find_peaks(gradients[0], height=np.mean(gradients[0]) + 1.5 * np.std(gradients[0]))
print('Global change points detected at:', change_points)

# plot as heatmap as there are 8 values for each week
fig, ax = plt.subplots(figsize=(20,7))
ax = sns.heatmap(df_emotions_weekly.T, ax=ax, cmap='Blues')
ax.set_xticklabels(df_emotions_weekly.index.strftime('%d-%m-%Y'))
plt.xticks(rotation=45)
ax.vlines(change_points, *ax.get_ylim(), color='red', linewidth=7)

for cnt, i in enumerate(gradients_em.columns):
   change_points, _ = find_peaks(gradients_em[i], height=np.mean(gradients_em[i]) + 1.5 *
   ↪  np.std(gradients_em[i]))
   print('Local change points for emotion', emotions[i], 'detected at:', change_points)
   ax.vlines(change_points, cnt+1, cnt, color='black', linewidth=4)

plt.show()
```

of positive tweets than negative, except this week. After this week, every subsequent week has had a fewer proportion of positive tweets. This is also observed in the ground truth labels in fig. 25, albeit less notably.

The heatmap shown in fig. 27 shows that there is a sharp drop in Anticipation after, and including, the week of 23/02/2020. This change is also observed, albeit less markedly, in Fear (increase), Joy (decrease) and Sadness (increase), which agrees with our observations from the VADER plot. Since the week of 23/02/2020 approximately corresponds to when COVID-19 took over news coverage, with the constant publication of new COVID cases spreading across the country, we qualitatively judge our methods to be successful as we can see a marked change across the heatmap (Emolex) and our sentiment analysis (VADER) at approximately the same time. This is thus when the public truly became aware of the threat of the virus.

Using change point detection, we can confirm the change in Sadness at the start of the pandemic, and additionally see a peak in joy right around Christmas week, with Fear apparently at its height around New Year. We thus see that emotion analysis through a lexicon is a powerful tool to understand public perception of a major health crisis, as we can clearly see changes in sentiment and emotion occurring at approximately the times which correspond to international and nationwide events.

Q4: Comparison to literature

As shown above, we do see a change in emotions and sentiments corresponding to major events. However, it is difficult to make an analysis of actual phases as in the paper – the authors found phases of refusal, anger and acceptance. The difficulty

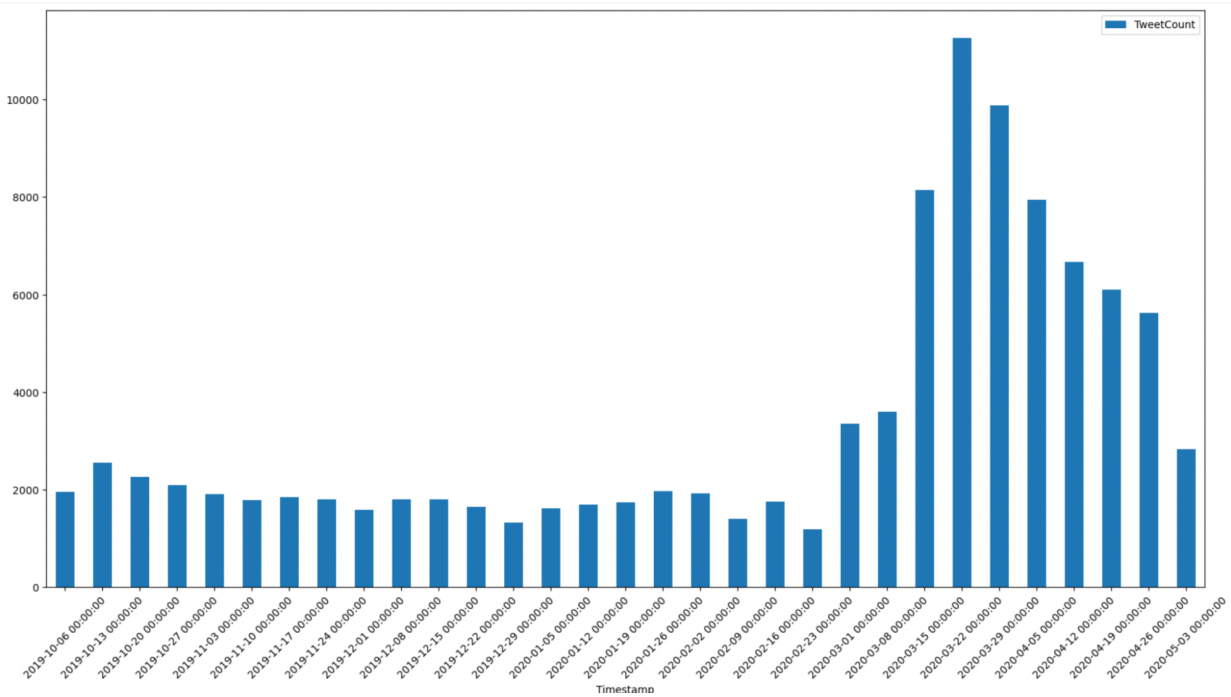


Fig. 26: Number of tweets per week pertaining to COVID-19 in the US (subset of given dataset).

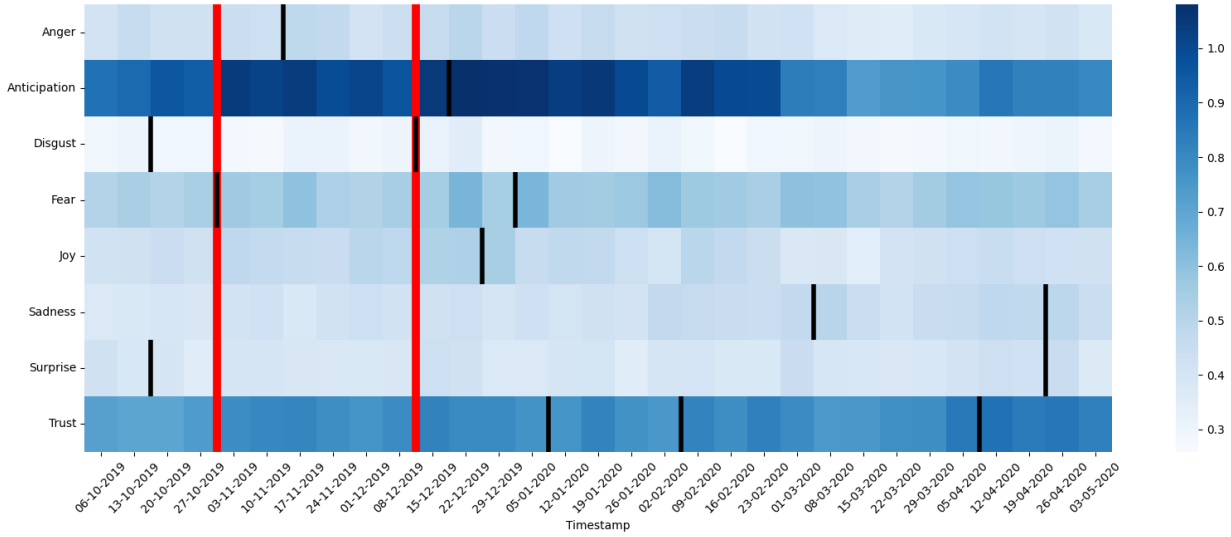


Fig. 27: Heatmap of emotion analysis, aggregated weekly, normalized over the number of tweets every week. Change points are overlaid on this heatmap (red: global change point, black: local change point).

that we face in order to replicate that in our case, is due to the fact that our dataset ends much earlier (theirs ends at the end of 2020, whereas ours ends in May 2020, a mere two months after discussions on the pandemic were widespread across the U.S.), and due to the fact that we have a lot less tweets (they have 122M, while we have 103K). It is thus not possible, due to the dataset, to carry out an analysis of phases or 'epidemic psychology' and seeing how they coincide with contagion waves.

Q5: Discussion

The authors have constructed their own methodology consisting of their own lexicon based on four lexicons, while we used a readily available one, the NCR/Emolex lexicon. This is due to the fact that we are running a study on a much smaller scale. We also implemented change-points just as in the paper, however, we added VADER sentiment analysis in order to understand the correlations between the changes in emotions and whether they are positive or negative.

The smaller scale of this study and the dataset limitations thus mean that it is difficult to compare our work to the paper's, however, our work shows that, even with severe limitations, it is possible to see significant changes in emotions and sentiment that coincide with major events across a country.

The advantage of our methods are that they are extremely simple and efficient, however this is also a disadvantage compared to the paper, as our techniques are not as extensive and fine-tuned as those of the authors.

Q6: Summary & Conclusion

Through the use of unsupervised sentiment analysis techniques, such as VADER, emotion lexicon-matching and change point detection, we are able to detect significant changes in emotion and sentiment in completely unstructured and brief social media texts. This is a significant finding, even as the dataset used was small and significantly shorter than the actual duration of the pandemic, as it indicates that unsupervised sentiment analysis techniques are powerful enough to actually detect human emotion in the absence of labels. Our simple methodology is hence a success, as we can demonstrate that these techniques are valuable to analyze public perception at a very large scale, and can help inform public health policy.

```
! pip install pyLDAvis

from gensim.corpora.dictionary import Dictionary
from gensim.models import LdaModel,CoherenceModel
import pyLDAvis
import pyLDAvis.gensim

diction = Dictionary(df_tweets_preprocessed['unigram'])
corpus = [diction.doc2bow(i) for i in df_tweets_preprocessed['unigram']]
n_topics = 4
lda = LdaModel(corpus, id2word=diction, num_topics=n_topics, random_state=42, passes=2)

#inspired by https://www.machinelearningplus.com/nlp/topic-modeling-gensim-python/
# Compute Perplexity
print('\nPerplexity: ', lda.log_perplexity(corpus))  # a measure of how good the model is. lower the
↪  better.

# Compute Coherence Score
coherence_model_lda = CoherenceModel(model=lda, texts=df_tweets_preprocessed['unigram'],
↪  dictionary=diction, coherence='c_v')
coherence_lda = coherence_model_lda.get_coherence()
print('\nCoherence Score: ', coherence_lda)

pyLDAvis.enable_notebook()
vis = pyLDAvis.gensim.prepare(lda, corpus, diction,R=10)
vis

```

Listing 23

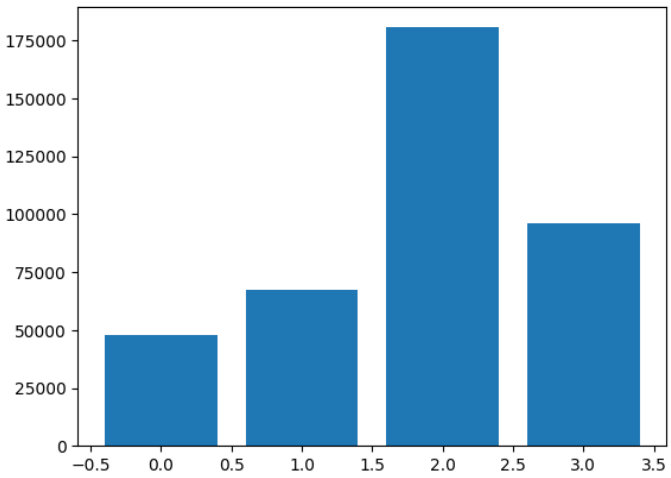


Fig. 28: Most common topics in the TweetsCov19 dataset. Topic 1 corresponds to the unknown topic, topic 2 to 'news reports', topic 3 to 'covid safety rules', and topic 4 to 'government and politics'.

Bonus: Topic & Emotion Analysis

Topic Modeling

Latent Dirichlet Allocation (LDA) [2] is a method used to infer topics from text. These topics are not known in advance, and are supposed to be inferred from the words that are grouped under them after training of the model.

We inferred the following topics from LDA: **covid safety rules** (containing words such as: 'stay' and 'home', 'social' and 'distancing', 'mask'), **government and politics** (containing words like: 'Trump', 'lockdown', 'president', and 'pm'), **news reports** (containing words like 'new' and 'cases', 'pandemic', 'death', 'China'), and a **covid-related topic** we could not find a common term for (containing words such as 'quarantine', 'know', 'like', 'thing', 'lockdown', 'day', and 'one'). The histogram of the topic frequencies can be found in Figure 28, and the words related to each topic can be found in Figure 29.

Different from the Nature article [7], we were unable to find more than 4 sensible topics, with topics beyond 3 often strongly overlapping with one another. This is likely due to the fact that we are using a different dataset and different preprocessing methods. Their dataset is almost double the size (800 000 tweets), and contains tweets from the UK only, while ours contains

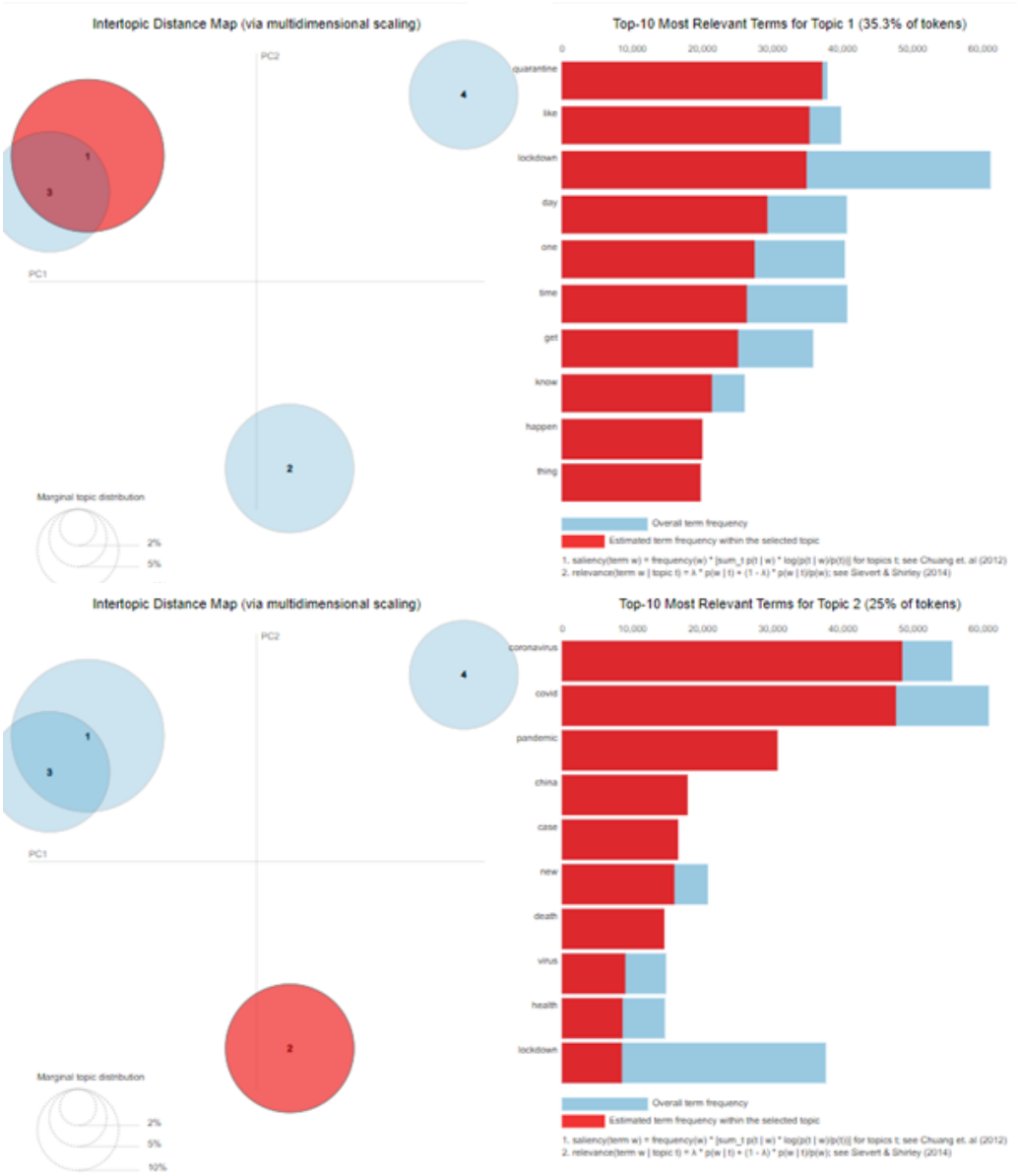


Fig. 29: Visualization of the topics inferred from LDA. Topic 1 corresponds to the unknown topic and topic 2 to 'news reports'.

tweets from across the world (the majority from the US, which is probably a big reason why Trump is so prominent in one of the topics).

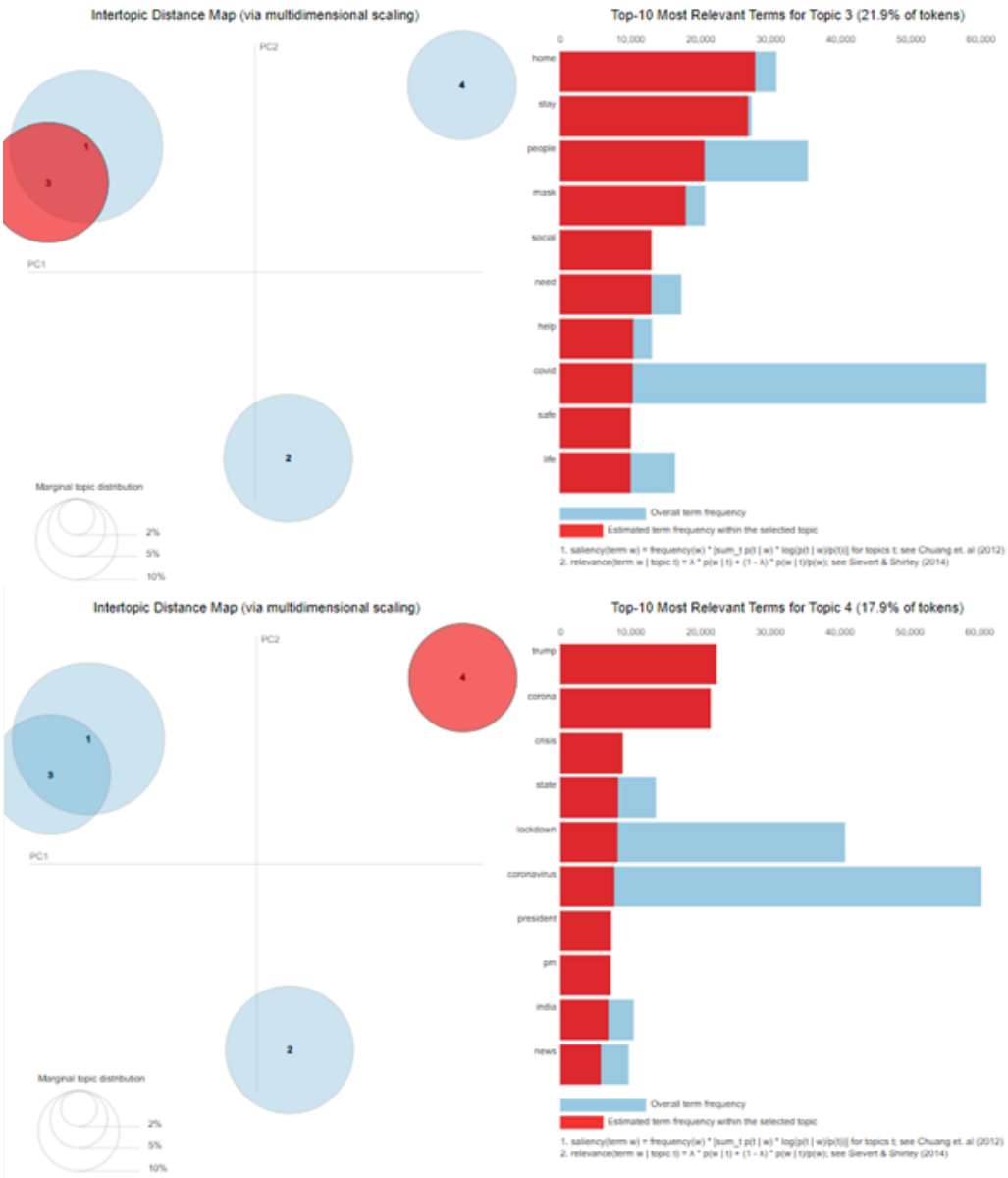


Fig. 29: (continued) Visualization of the topics inferred from LDA. Topic 3 corresponds to 'covid safety rules', and topic 4 to 'government and politics'.

REFERENCES

[1] Luca Maria Aiello et al. "How epidemic psychology works on Twitter: Evolution of responses to the COVID-19 pandemic in the US". In: *Humanities and social sciences communications* 8.1 (2021).

[2] David M Blei, Andrew Y Ng, and Michael I Jordan. "Latent dirichlet allocation". In: *Journal of machine Learning research* 3.Jan (2003), pp. 993–1022.

[3] Piotr Bojanowski et al. "Enriching Word Vectors with Subword Information". In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 135–146. DOI: 10.1162/tacl_a_00051. URL: https://aclanthology.org/Q17-1010.

[4] Leo Breiman. *Arcing the edge*. Tech. rep. Citeseer.

[5] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (Oct. 2001), pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

[6] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project". In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[7] I Cheng et al. "Evaluation of Twitter data for an emerging crisis: an application to the first wave of COVID-19 in the UK". In: *Scientific Reports* 11 (Sept. 2021). DOI: 10.1038/s41598-021-98396-9.

[8] Pavlos Fafalios et al. "TweetsKB: A Public and Large-Scale RDF Corpus of Annotated Tweets". In: *CoRR* abs/1810.10308 (2018). arXiv: 1810.10308. URL: http://arxiv.org/abs/1810.10308.

[9] Yoav Freund and Robert E. Schapire. "A desicion-theoretic generalization of on-line learning and an application to boosting". In: *Computational Learning Theory*. Ed. by Paul Vitányi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 23–37. ISBN: 978-3-540-49195-8.

[10] Jerome H Friedman. "Stochastic gradient boosting". In: *Computational statistics & data analysis* 38.4 (2002), pp. 367–378.

[11] Jerome H. Friedman. "Greedy function approximation: A gradient boosting machine." In: *The Annals of Statistics* 29.5 (2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451. URL: https://doi.org/10.1214/aos/1013203451.

[12] Zellig S. Harris. "Distributional Structure". In: *¡i¿WORD¡/i¿* 10.2-3 (1954), pp. 146–162. DOI: 10.1080/00437956.1954.11659520. eprint: https://doi.org/10.1080/00437956.1954.11659520. URL: https://doi.org/10.1080/00437956.1954.11659520.

[13] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605. URL: http://jmlr.org/papers/v9/vandermaaten08a.html.

[14] Leland McInnes, John Healy, and James Melville. *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. 2020. arXiv: 1802.03426 [stat.ML].

[15] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[16] F. Å. Nielsen. *AFINN*. Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, Mar. 2011. URL: http://www2.compute.dtu.dk/pubdb/pubs/6010-full.html.

[17] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[18] Jeffrey Pennington, Richard Socher, and Christopher Manning. "GloVe: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: https://aclanthology.org/D14-1162.

[19] *Python implementation of GloVe*. https://github.com/maciejkula/glove-python. Accessed: 2023-04-30.

[20] Radim Řehůřek and Petr Sojka. "Software Framework for Topic Modelling with Large Corpora". English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. http://is.muni.cz/publication/884893/en. Valletta, Malta: ELRA, May 2010, pp. 45–50.

[21] Gerard Salton and Christopher Buckley. "Term-weighting approaches in automatic text retrieval". In: *Information processing & management* 24.5 (1988), pp. 513–523.

[22] Richard Socher et al. "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank". In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. EMNLP 2013. Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 1631–1642. URL: https://aclanthology.org/D13-1170 (visited on 05/16/2023).

[23] Philip Strong. "Epidemic psychology: a model". In: *Sociology of health & illness* 12.3 (1990), pp. 249–259.

[24] *VADER Sentiment Analysis Scoring*. https://github.com/cjhutto/vaderSentiment#about-the-scoring. Accessed: 2023-04.