Artificial Intelligence II Homework 1

Comments & Model Performance results on Question 2

Pavlos Spanoudakis (sdi1800184)

Basic Execution flow

1. Reading train set & validation set data from the input files, into `DataFrame`'s. The file paths can be modified on the first notebook cell.

2. Checking if all the samples have the expected format without missing values.

3. Trimming the created `DataFrame`'s, keeping just the required data (tweets & labels).

4. Sample vectorization using one of the available vectorizers in `sklearn.feature_extraction.text`.

   The model performs better when `TfidfVectorizer` is used. With `CountVectorizer`, the results are very similar, with a slight performance reduction.

   For better generalization and based on this discussion, the train set samples are first used to create the vectorizer vocabulary, and both data sets are vectorized on it.

   Several vectorizer parameters were used for the vocabulary creation:

   - `min_df = 0.1`: Ignores any words that occur in less than 1% of the given samples.
   - `max_df = 0.85`: Ignores any words that occur in more than 85% of the given samples.
   - `preprocessor = customPreprocessor`: Using the `customPreprocessor` function in the notebook, it removes URL's, `@` mentions and digits from each tweet before it is being processed.
   - `stop_words = "english"`: Ignores english stop words, based on the `sklearn` built-in stop words set.
   - `ngram_range = (1, 3)`: Add each phrase with 1, 2 or 3 words to the vocabulary (with respect to the above parameters).

   Many other `min_df`, `max_df` and `ngram_range` combinations were used, but the ones specified had the best results.

5. Instantiating the classifier with the desireable parameters.

   `LogisticRegression` from `sklearn.linear_model` is used, with `max_iter = 1000` and `mutliclass = "multinomial"`.
   All the available solvers have been used, but `lbfgs` (default) has the best performance.
   The default `penalty` parameter is set to `l2`, which adds a L2 penalty term.
   `penalty = "none"`, with `max_iter = 10000` was also tested: The learing curves are visibly "smoother" but the final results are basically the same as with `penalty = "l2"`.

6. Making predictions to create the learning curves.

   - We start by creating different train set sizes, equally increasing the current size each time, and for each size, we take a portion of the train set of that size to train the classifier.

◦ We make predictions using the classifier we just trained, on that particular portion of the train set, as well as 100% of the validation set.

7. Calculating the predictions scores.

   For each different training set size used, we calculate the **F1**, **Precision** and **Recall** scores of the predictions made by the classifier on the corresponding training set batch and the validation set, after training with that size.

8. Creating & displaying the learning curve.

   Using the F1 scores (as `y` values) and the training set sizes (as `x` values) we plot the curves for the Train and Validation sets respectively.
   If desired, we can plot curves for the Precision and Recall scores as well.

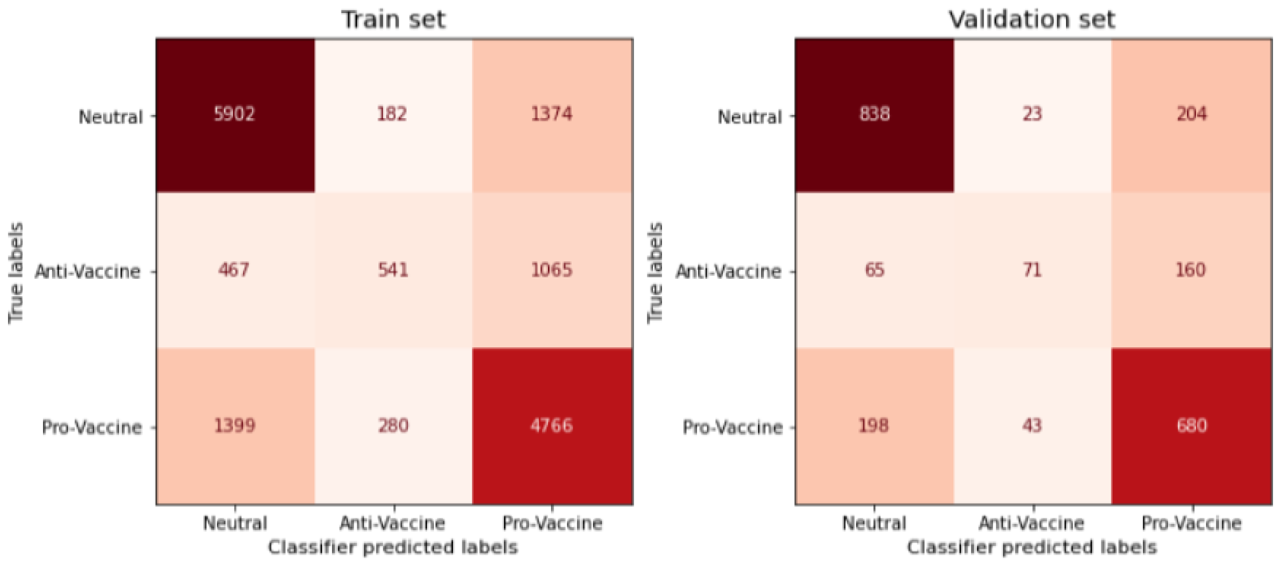
9. Displaying final scores.

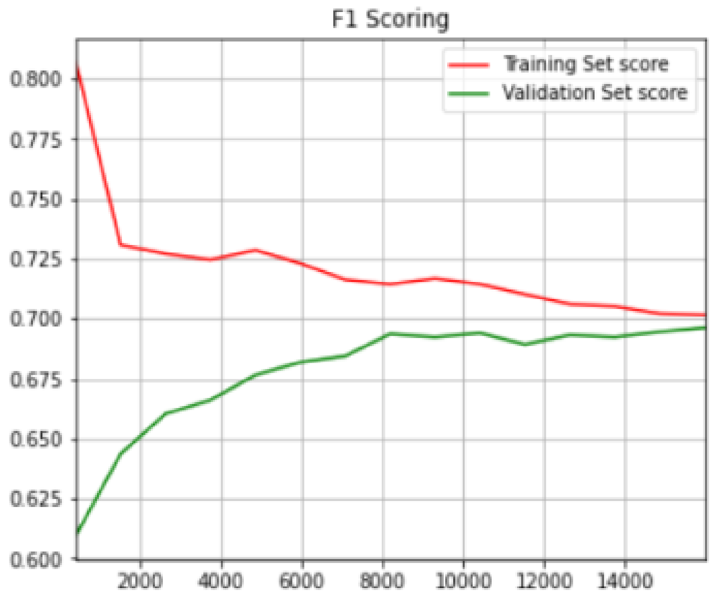
   We display the F1, Precision & Recall scores for the final predictions, after training the classifier using 100% of the train set.
   We also display confusion matrixes on both set predictions.

Different models performace comparison

For all the results below, `solver = "lbfgs"` was used in the Classifier.

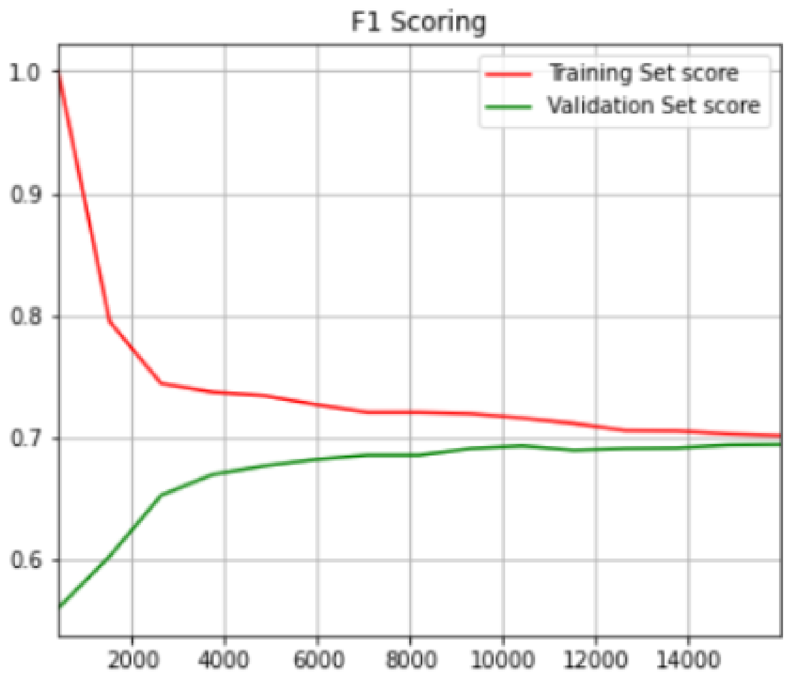
- Using **`TfidfVectorizer`**:
  - Using `min_df`, `max_df` & `ngram_range` in the Vectorizer:



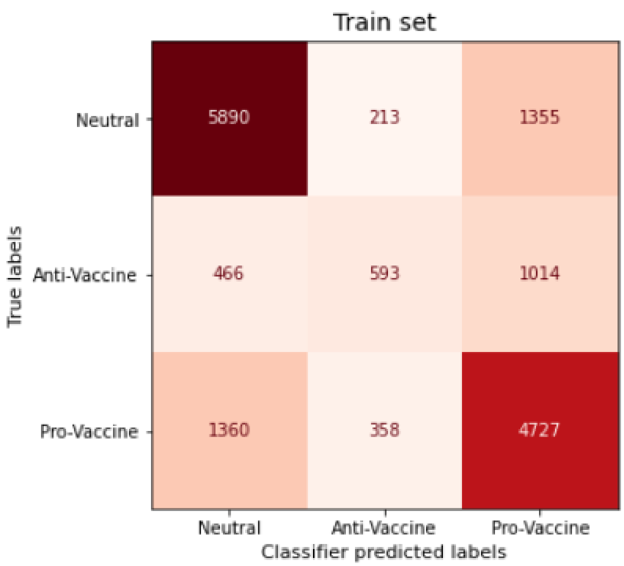


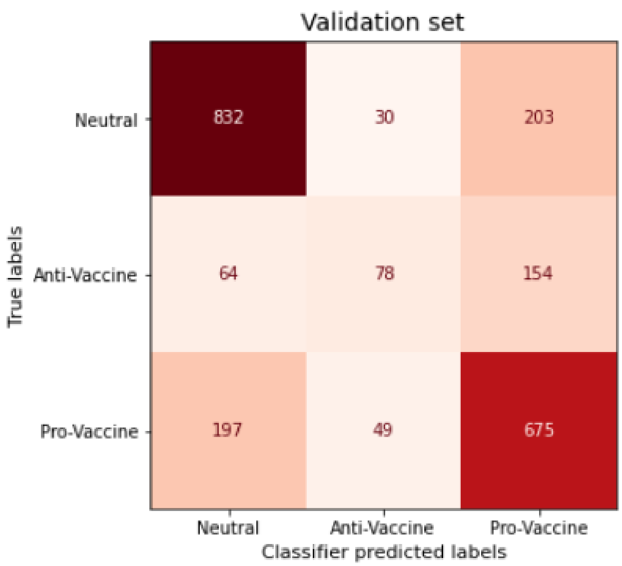
	F1	Precision	Recall
Train	0.7016	0.6536	0.5973
Validation	0.6963	0.6436	0.5883

- Using `min_df`, `max_df` & `ngram_range` in the Vectorizer & `penalty="none"` in the Classifier:



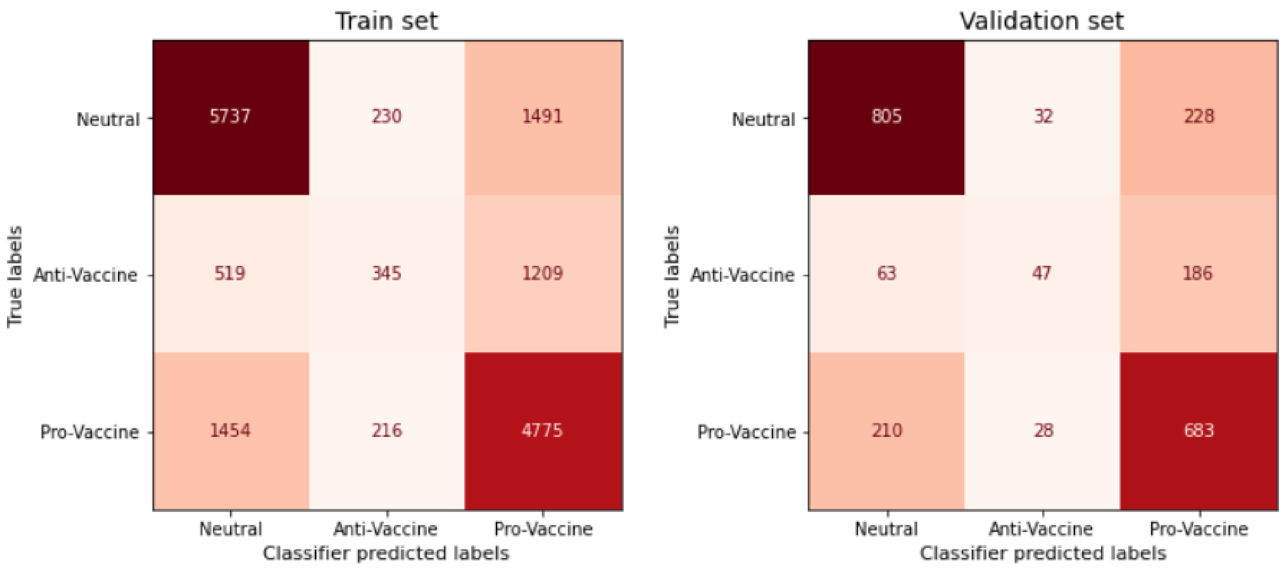
	F1	Precision	Recall
Train	0.7017	0.6463	0.6031
Validation	0.6946	0.6374	0.5925



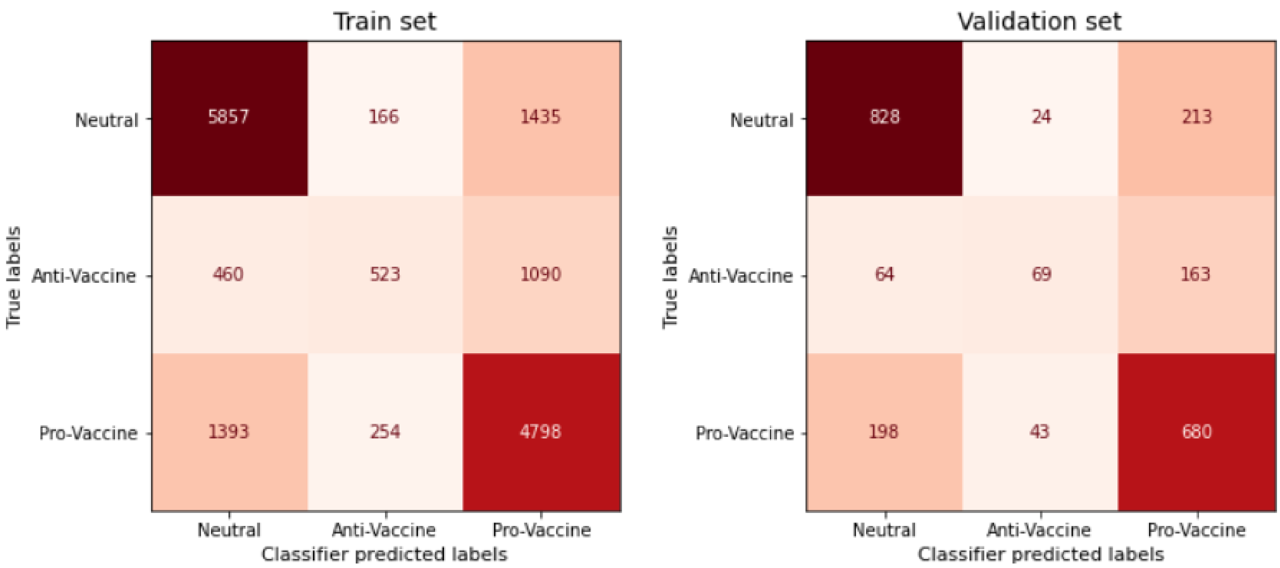


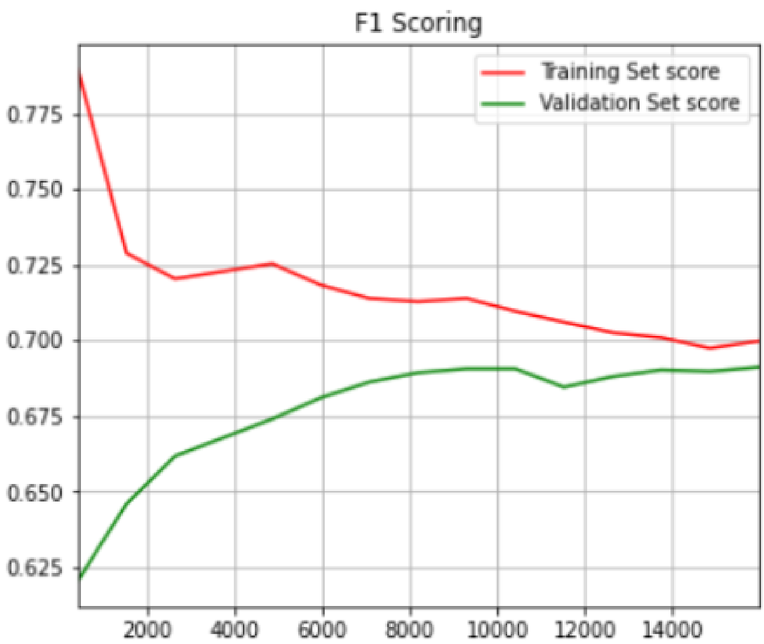
- Using `min_df`, `max_df`, `customPreprocessor` & `stop_words` in the Vectorizer:

	F1	Precision	Recall
Train	0.6796	0.6064	0.5589
Validation	0.6727	0.6029	0.5521



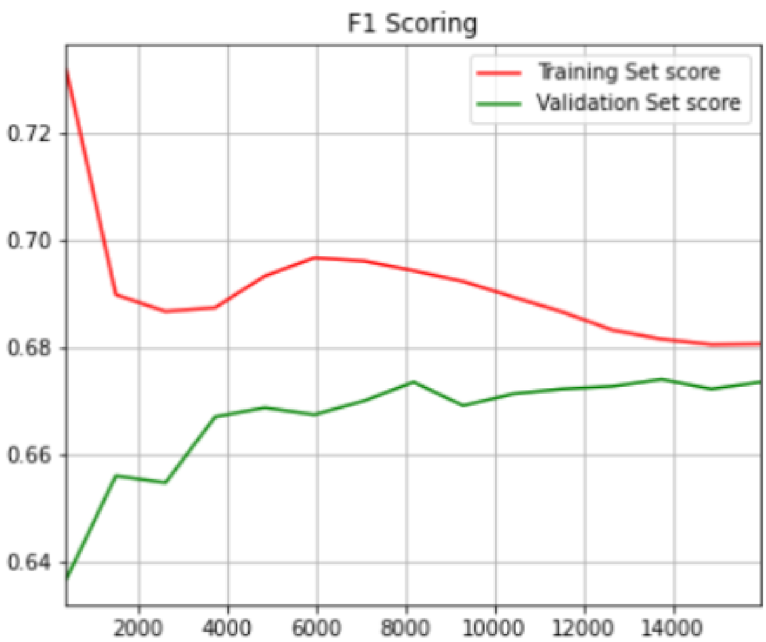
- Using `min_df` & `max_df` in the Vectorizer:



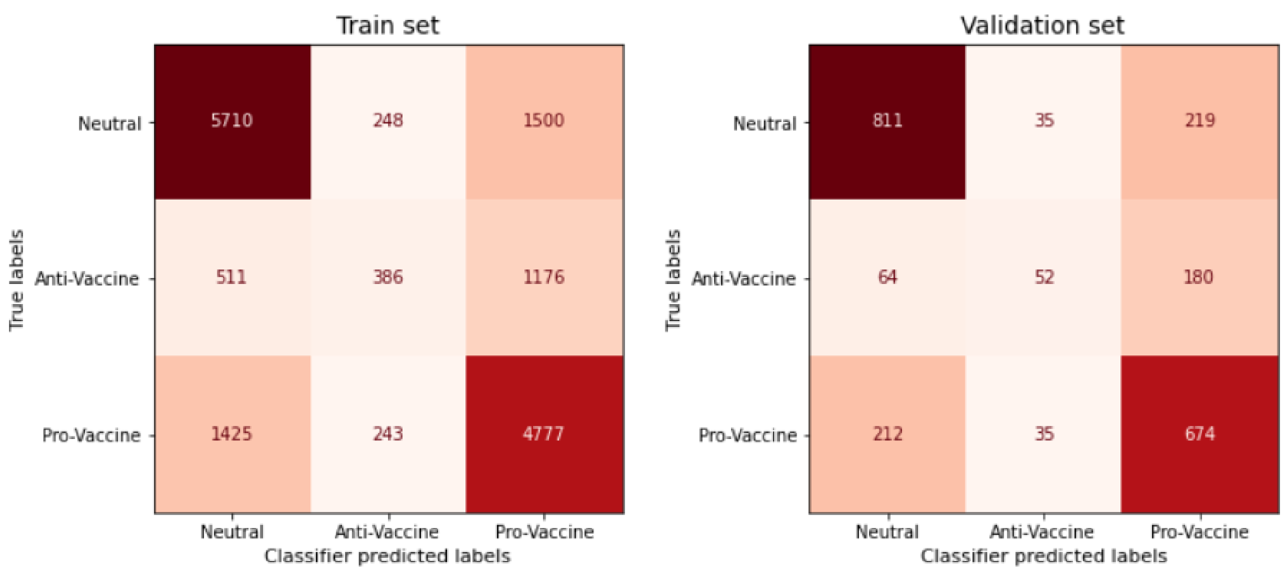


	F1	Precision	Recall
Train	0.6997	0.6565	0.594
Validation	0.6911	0.6370	0.583

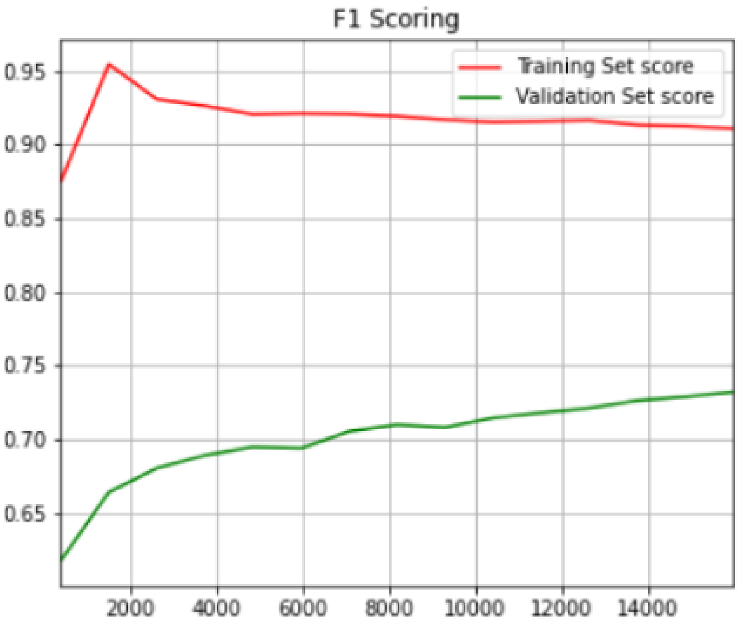
- Using `min_df`, `max_df`, `customPreprocessor`, `stop_words` & `ngram_range` in the Vectorizer:



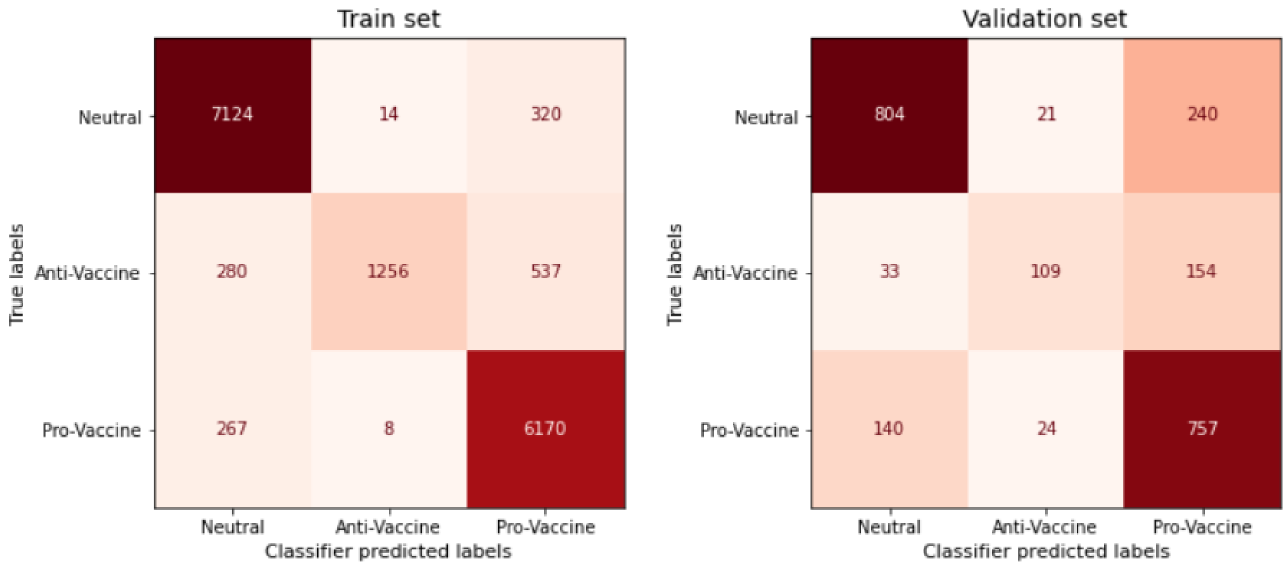
	F1	Precision	Recall
Train	0.6806	0.6093	0.5643
Validation	0.6735	0.6002	0.5563



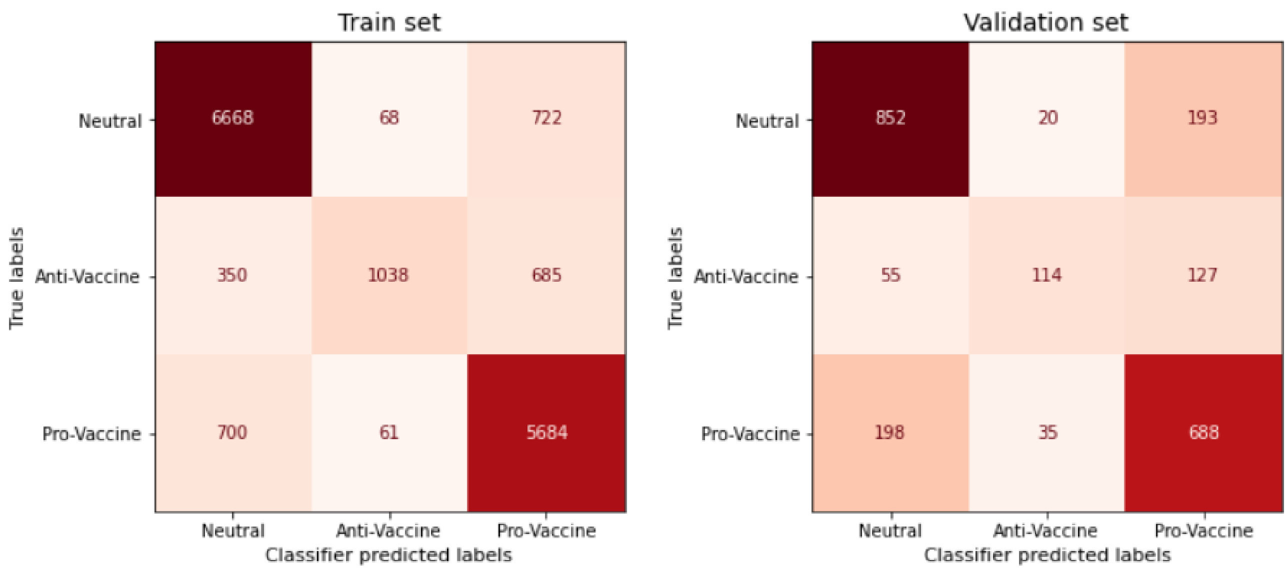
- Using `customPreprocessor`, `stop_words` & `ngram_range` in the Vectorizer:

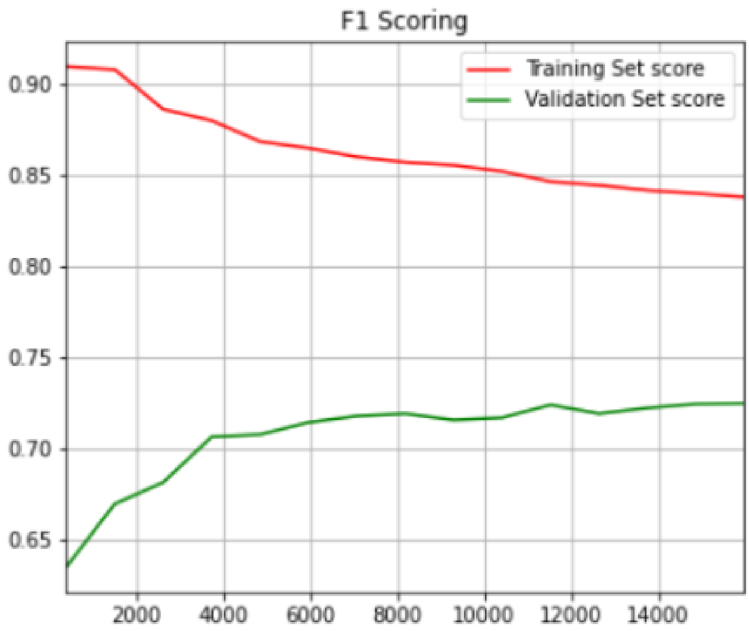


	F1	Precision	Recall
Train	0.9107	0.9298	0.8395
Validation	0.7318	0.7295	0.6484



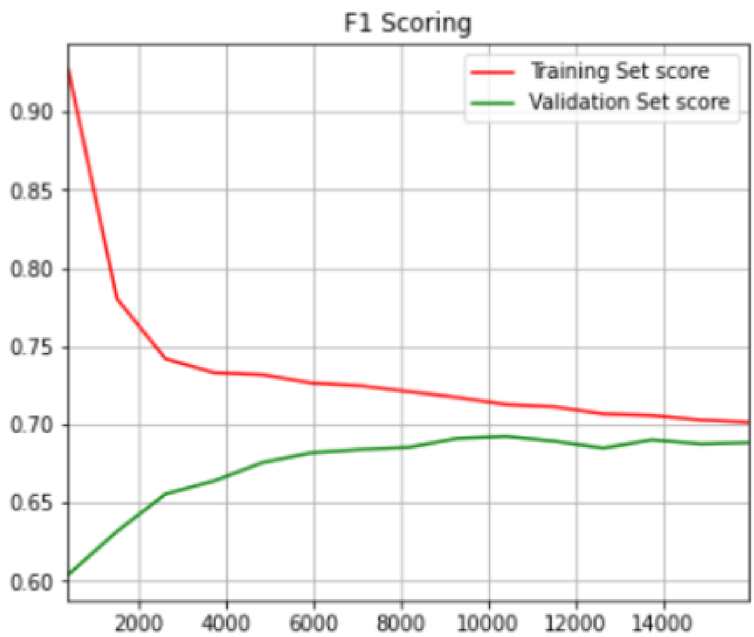
- Using `customPreprocessor` & `stop_words` in the Vectorizer:



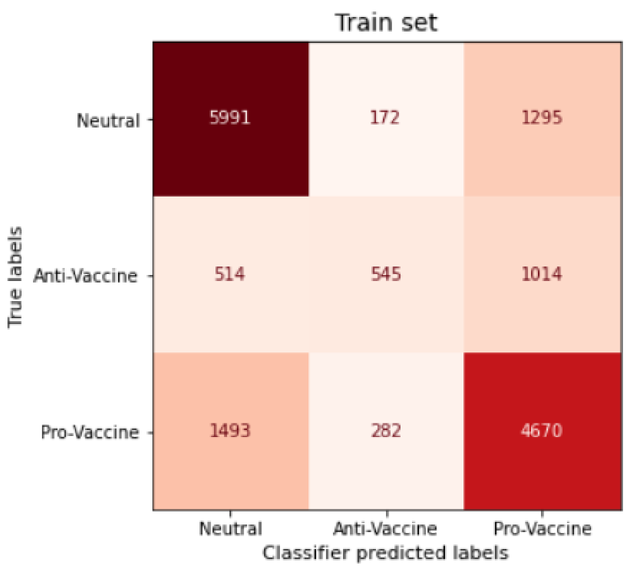


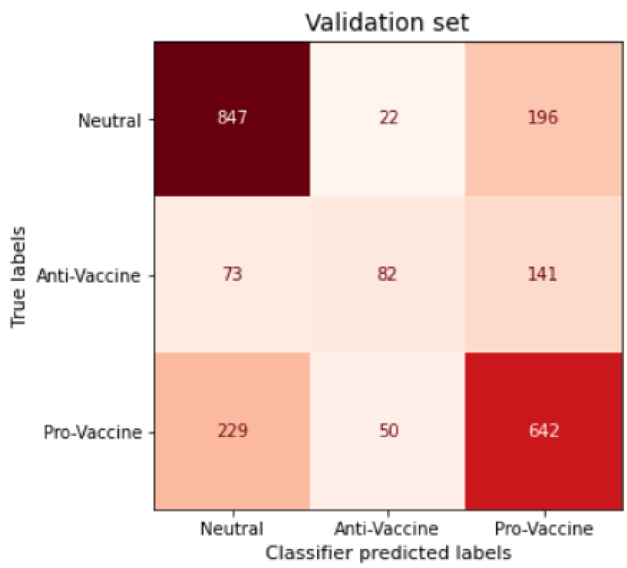
	F1	Precision	Recall
Train	0.8381	0.8517	0.7589
Validation	0.7248	0.7094	0.6440

- Using **CountVectorizer**:
    - Using `min_df`, `max_df` & `ngram_range` in the Vectorizer:

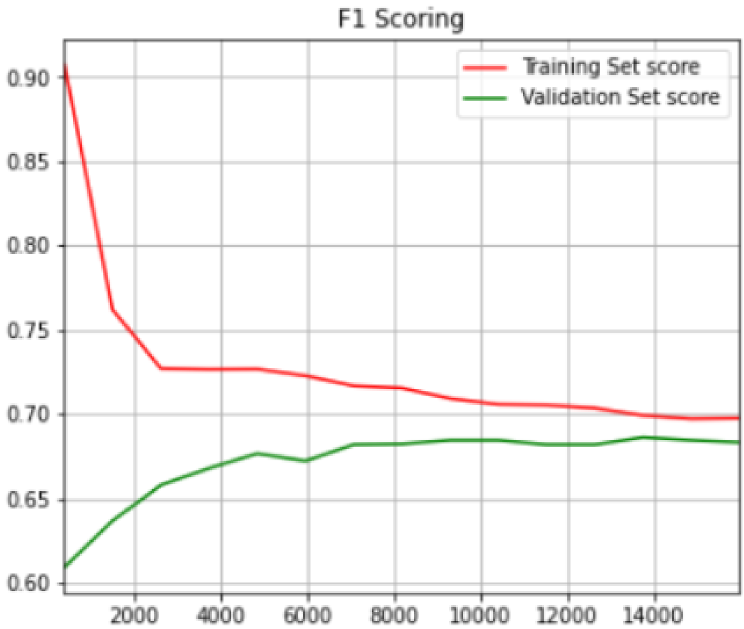


	F1	Precision	Recall
Train	0.7014	0.6546	0.5969
Validation	0.6884	0.6418	0.5898

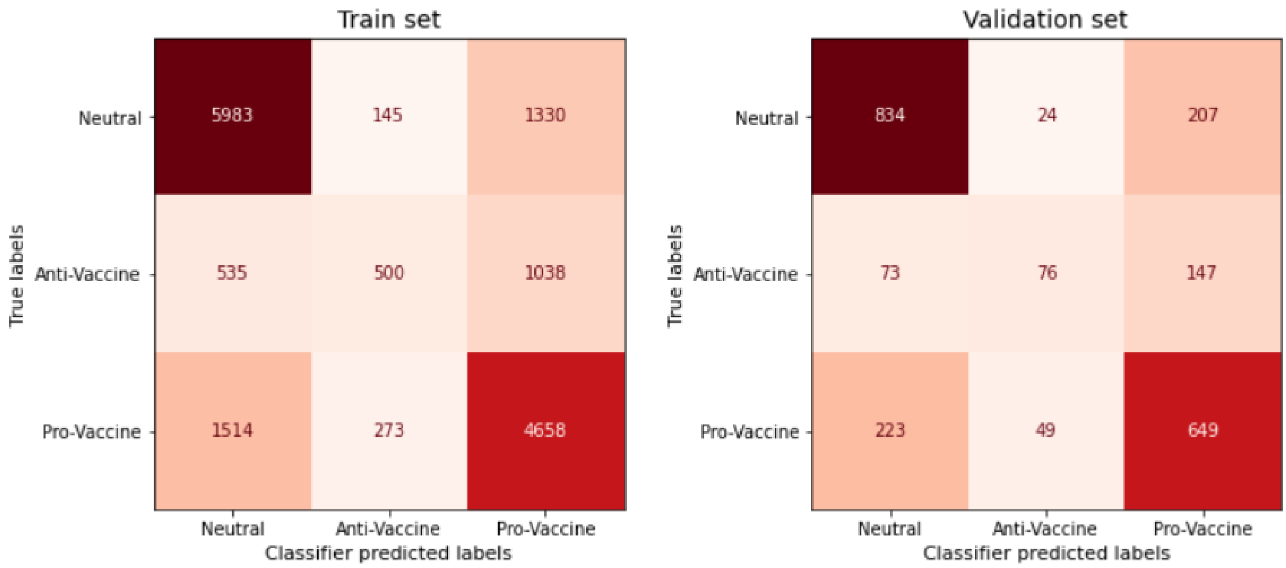




- Using `min_df`, `max_df` in the Vectorizer:



	F1	Precision	Recall
Train	0.6974	0.6508	0.5887
Validation	0.6832	0.6317	0.5815



Takeaways

- We see much better performance on `Neutral` and `Pro-Vaccine` tweets in all models, since a significant amount of train set tweets are labeled as such.
  Apparently, the majority of the `Anti-Vaccine` tweets are falsely predicted as `Pro-Vaccine`, which can be explained: Tweets from both labels are expected to have many common features ("vaccine", "virus" etc.). The number of `Pro-Vaccine` tweets in the train set is significantly greater, which confuses the model to increase these feature weights on the `Pro-Vaccine` class. We can improve the model performance on `Anti-Vaccine` tweets, by "feeding" it with more of them.
- When we do not specify `min_df` & `max_df` parameters, the model is overfitting, since the number of features is way too large and prevents our model predictions from generalizing. In these cases, we can see that all the metric scores are better, which means that, hypothetically, if we feed the model with more samples, it will achieve better scores (when the two curves eventually approach each other). Of course, the difference between the two curves is huge, therefore the required amount of extra samples may be extreme.
- We see that preprocessing, as well as using stop words list does not help the model. In fact it causes a slight performance reduction, and also increases the "sense" of possible overfitting, since the curves

become almost parallel at the end.
- Simply using `CountVectorizer` causes slight scores reduction, so `TfidfVectorizer` (which is essentially `CountVectorizer` & `TfidfTransformer`) seems to be the right choice.
- The final model that is selected in the notebook uses `TfidfVectorizer`, with custom `min_df`, `max_df` & `ngram_range` parameters and a Classifier with just `max_iter = 1000` and `mutliclass = "multinomial"`parameters.

Resources & Development

- The chapters 4 & 5 of the given book, along with the `sklearn` documentation were eye-opening.
- This is not a literature resource, but it was the first ML tutorial I ever followed and is pretty straightforward.
- The notebook has been developed in WSL Ubuntu 20.04, using Visual Studio Code & Python 3.8.10. It has been tested successfully in Google Colab environment as well.