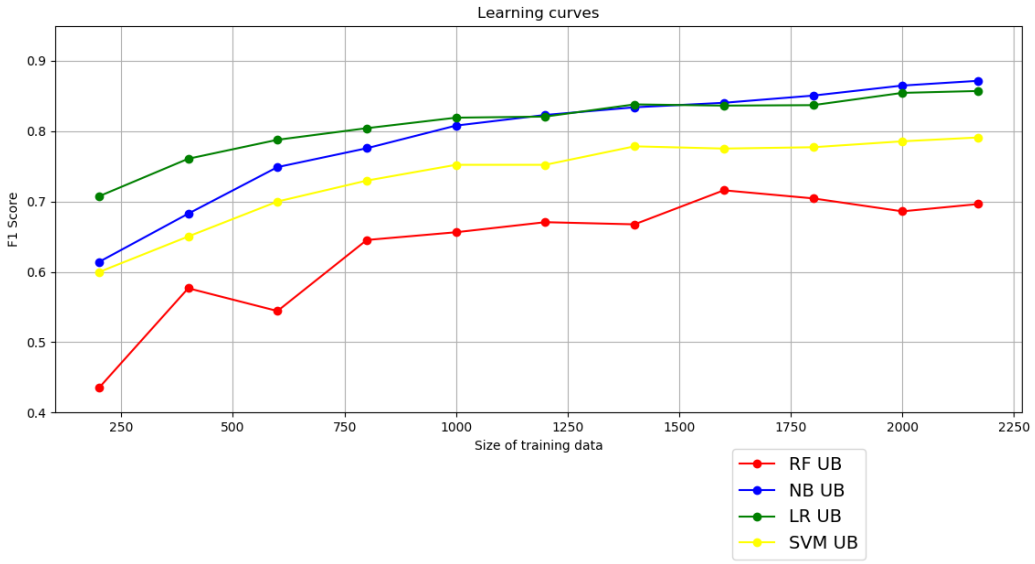
Basic Comparison with Baselines:

Result:

Algorithm + Config	Precision	Recall	F1
NB UB	0.8936558945161623	0.8641090299984772	0.8716180776182416
NB BB	0.8990388753140941,	0.8177641744073905	0.8293348067709161
LR UB	0.8623100725797657	0.8545274351555759	0.8571381125990172
LR BB	0.8570170690382728	0.8472711537485408	0.850245238119066
SVM UB	0.7901458992281062	0.794786152987158	0.7910517156602809
SVM BB	0.8054208233357942	0.8056557535150499	0.8050312126198642
RF UB	0.7685652086747324	0.7129749758895487	0.7055208180119485
RF BB	0.7606633562515915	0.6947232120196943	0.6964121601666926

(Precision, Recall and F1 are calculated with marco average)

Plot:



Findings:

Overall, unigrams have the better performance (higher f1 score and precision) than bigrams. It could be the result that the data size or the frequency of any word pair are lower than that of a single word. We might get a better result if we do combine training with both unigrams and bigrams.

Also, performance of the 4 algorithms is NB > LR > SVM > RF. This could be the result of the frequency of the words/word pairs in our dataset.

My Best Configuration:

MBC_final.py result:

0.8807711403768458

Explanation:

Among my combinations, “NB(unigram), CountVectorizer+Stemmer+Lower case (with stopwords filter)” has the best performance. Both filters feature makes the training data more informative and accurate, so using them improves the performance. Also CountVectorizer is doing better than TFIDFVectorizer since CountVectorizer is better in gathering similarity, which is what text classification needs.