# Report | Coursework 1: Sentiment analysis from tweets

## Solution 1)

- Started by implementing **parse_data_line()** and **pre_process()** function.
- The **parse_data_line()** is taking line from the 'sentiment-dataset.tsv' file and returns the tuple with label and text. The line will have three elements but we are just interested in element at '1' index and element at '2' index. I have added a check to see in the data is not empty and has three elements to avoid any **IndexError**. Moreover, I could also have transformed the label and text to lower case, but I thought pre_process() function would be the best place to do so.
- The **pre_process()** takes a text and return a list of tokens. This might look a simple function but indeed gives a lot of opportunity to add optimization to our tokens. In second NLP lab has similar function implemented that I am going to refer that as a starting point.

## Solution 2)

- Implement **to_feature_vector()** function for feature extraction.
- Given a list of tokens it will return a python dictionary with tokens as keys and corresponding weights as values. I am keeping weight of each word as 1 using the binary feature approach.
- The **global_feature_dict** variable represents bag of words though.
- These are the algorithm steps: -
    - Use Counter from collections library to create a frequency count for words.
    - Update **global_feature_dict** with key count
    - Override the value of the keys to 1 since following binary feature approach.

NOTE: - This method is optimised later in NLP_Assignment_2.ipynb

## Solution 3)

Cross-validation on training data: -

- Implemented **cross_validate()** function to do a 10-fold cross validation on the training data.
- Storing **precision**, **recall**, **f1 score**, and **accuracy** of your classifier in a global variable **cv_results** contain average scores for all folds.
- Using **precision_recall_fscore_support** function from **sklearn.metrics** to calculate precision, recall, f1 score
- The **cross_validate()** function returns **cv_results** as a response.

## Solution 4)

- Using the already implemented function **confusion_matrix_heatmap()** for measuring performance of the classes using a confusion matrix.
- The balance of false positives and false negatives are as follows: -
    - Total dataset – 2684 | False Positives – 228 | False Negative – 158
- Carried out the error analysis on first fold data from cross-validation function.
- Writing all the FP and TN into a csv file with column headers being "Expected Values", "Predicted Values", "Text". Note:- Files will be generated automatically in relative path.

FP observation note for 'negative': – The text predicted as 'positive' but were actually 'negative' is termed as false positive.

- The Recall of positive class is high 0.91 as compare to precision value of 0.88, which suggests that the model would be better at capturing a larger proportion of actual positive cases at the expense of precision, ie this could lead to more false positives.
- The training data is also imbalance with positive label values and negative being in ratio 2:1 approx. We can use the parameters like class_weight, or C to tweak the accuracy.
- When analysing the bag of words for the positive world I found that there are words like 'tomorrow', 'may', 'like' etc which have high term-frequency for positive but also present for the negative texts hence confusing model.

FN observation notes for 'positive': - The text predicted as 'negative' but were actually 'positive' is termed as false negative.

- The high term frequency words like 'not', 'nothing' is present due to which model is predicting those text as negative. Moreover, I see a lot of noise in dataset which might be causing abnormal behaviour. I will be optimizing these tokens to see if I can increase the precision for positive.

## Solution 5) – Ref NLP_Assignment_2.ipynb

These are the optimization I did after doing the error analysis: -

**Optimisation in pre_process() function: -**

- Filtered out words which are URL or web link. frequency of these were quite high and were degrading accuracy.

- I am not removing numbers because I think the numbers can also be useful feature for analysing sentiment. Eg- "There market closed at -10 pt." is negative.
- I have removed punctuations because I think they are not useful features for sentiment analysis where the labels are binary. Had it been more labels, I would have kept punctuations.
- Furthermore, added stop word removal to remove words like 'the', 'and', 'is'; normalization to cast into lowercase and lemmatising to transform words in their base form.

**Optimisation in to_feature_vector() function: -**

- Using bigram along with unigram for extracting combination of words as features.
- Started using '**CountVectorizer**' from **sklearn.feature_extraction.text**
  - CountVectorizer(binary=True, ngram_range=(1,2))
  - CountVectorizer(binary=True, ngram_range=(1,2), max_features=200000)
- Implemented **to_feature_vector_weighted** to return weighted feature dictionary.

NOTE: - The average feature extracted per sentence is 11 after optimizing.

**Other optimizations include: -**

- I saw that the datasets are imbalance so I shuffled dataset. Below are the values before and after using shuffled dataset.

| Before shuffling dataset | After shuffling dataset |
|---|---|
| {'precision': 0.8539605466539374,<br>'recall': 0.8558809999576742,<br>'f1': 0.8537997900365687,<br>'accuracy': 0.8558809999576742} | {'precision': 0.8592266422753401,<br>'recall': 0.8605019726040825,<br>'f1': 0.8587634679162669,<br>'accuracy': 0.8605019726040825} |

- Implemented average number of words per sentence for a dataset. Added those changes as part of split_and_preprocess_data() function.
- Tried playing around with the parameters of SVM. Here is the observation: -
  - Added dual='auto' for classifier to choose dual or primal optimization.
  - There was no major change observed for change is `C`. (Ref: - Results Table)
  - Tried '*class_weights = {0: 10.0, 1: 1.0}* ' with C=1.0, the recall of positive decreased a little. (Ref: - Results Table). I was trying to balance the recall of positive with weight but looks like there was some convergence warning saying that ` *ConvergenceWarning: Liblinear failed to converge, increase the number of iterations*`.
  - To resolve the above warning, I used '*max_iter int, default=1000'.*
  - Reducing the value of C now was affecting the precision and recall of the label.
- Tried **max_feature**=200000 parameter in **CountVectorizer** to only consider most frequent words out of 300k features.

| | **Baseline Value – Without Optimization (I)** | **Intermediate Optimized (II)** | **Intermediate Optimized (III)** | **Intermediate Optimized (IV)** | **Final Optimized (V)** |
|---|---|---|---|---|---|
| **Evaluation Metrics** | {'precision': 0.84704,<br>'recall': 0.84831,<br>'f1': 0.84730,<br>'accuracy': 0.84831} | {'precision': 0.85908,<br>'recall': 0.86024,<br>'f1':<br>0.8580755176810435,<br>'accuracy': 0.86024} | {'precision': 0.85743,<br>'recall': 0.85871,<br>'f1': 0.85698,<br>'accuracy': 0.85871} | {'precision': 0.85877,<br>'recall': 0.86009,<br>'f1': 0.85833,<br>'accuracy': 0.86009} | {'precision': 0.85922,<br>'recall': 0.86050,<br>'f1': 0.85876,<br>'accuracy': 0.86050} |
| **Pre-processing** | filtering start and end punctuations | normalisation,<br>lemmatising,<br>stop word,<br>filtering start and end punctuations | normalisation,<br>lemmatising,<br>stop word,<br>filtering start and end punctuations, | normalisation<br>lemmatising<br>stop word<br>filtering start and end punctuations | normalisation,<br>lemmatising,<br>stop word,<br>filtering start and end punctuations, |
| **Feature extraction** | unigram, binary weighted | binary=True,<br>ngram_range=(1,2),<br>binary weighted | binary=True,<br>ngram_range=(1,2),<br>max_features=200000,<br>binary weighted | binary=True,<br>ngram_range=(1,2),<br>max_features=200000,<br>binary weighted | binary=True,<br>ngram_range=(1,2),<br>max_features=200000,<br>frequency weighted |
| **SVM parameter** | default | C=1.0 | C=1.0,<br>class_weights = {0:<br>10.0, 1: 1.0} | C=1.0, class_weights =<br>{0: 100.0, 1:<br>1.0},max_iter= 100000 | C=1.0, class_weights = {0:<br>100.0, 1: 1.0},max_iter=<br>100000 |

- The **best accuracy** was achieved when using weighted features with cost parameter and class weight. Class weight normalised the sensitivity of the positive label. **Ref- V**
- When using C with class weight, there was a little improvement in overall precision and accuracy. This is because I have given weight to negative label to normalised high sensitivity of positive label. **Ref- III**
- The weighted feature extraction improved the accuracy. **Ref - V**