Feature Extraction and Embeddings

Data preparation

We load and preprocess the dataset that we got from the previous tp, encoding the *author* column using *One-hot Encoding* and get new column named *author_encoded*

```
label_encoder = LabelEncoder()
df['author_encoded'] = label_encoder.fit_transform(df['author'])
```

Construction of training/test set

We split the dataset into training and test sets (30% for the test) using *train_test_split* and since the dataset is imbalanced, we use the *stratify* parameter to ensure that the distribution of classes in training/test is similar

```
x_train, x_test, y_train, y_test = train_test_split(df['text'], df['author_encoded'], test_size=0.3, random_state=0, stratify = df['author_encoded'].values)
```

we get

	train set	**test set**
0	5529	2369
2	4230	1813
1	3944	1691

train_set = test_set * 2.33

Vectorization methods

we use two vectorization methods: *frequency-based* from *CountVectorizer* and *TF-IDF (Term Frequency-Inverse Document Frequency)* in order to convert raw text data into numerical resresentation that machine learning models can process, *frequency-based* will split text into individual words/tokens, count the frequency of each token in the document and generate a matrix where each row refers to a document, each column refers to unique token in the corpus and the values represented are the frequency of each token in that document, for *TF-IDF* used to evaluate the importance of token is the document relative to the curpos with

TF(token, document)=(how many time this token appears in the document)/(total tokens in the deocument)

```
count_vectorizer = CountVectorizer(binary=False,analyzer= 'word', stop_words='english')
x_train_cv = count_vectorizer.fit_transform(x_train)
x_test_cv = count_vectorizer.transform(x_test)
```

```
tfidf_vectorizer = TfidfVectorizer()
x_train_tfidf = tfidf_vectorizer.fit_transform(x_train)
x_test_tfidf = tfidf_vectorizer.transform(x_test)
```

Train the models and predict

We train 3 *MLP classifier* models *(multilayer perception models)* with only one hidden layer which has 100 neurons and 100 max iterations, each model will be trained on the count vectorized training set, tf-idf training set and one-hot encoding training set in order to compare the results we gill get, after creating and training the three models, we will make prediction on the same training set and after displaying the classification report for the three models we get

```
mlp_count = MLPClassifier(hidden_layer_sizes=(100,), max_iter=100, solver='adam',
random_state=1)
mlp_count.fit(x_train_cv, y_train)
y_train_pred_count = mlp_count.predict(x_train_cv)
```

```
mlp_tfidf = MLPClassifier(hidden_layer_sizes=(100,), max_iter=100, solver='adam',
random_state=1)
mlp_tfidf.fit(x_train_tfidf, y_train)
y_train_pred_tfidf = mlp_tfidf.predict(x_train_tfidf)
```

```
mlp_onehot = MLPClassifier(hidden_layer_sizes=(100,), max_iter=100, solver='adam',
random_state=1)
mlp_onehot.fit(x_train_cv, y_train)
y_train_pred_onehot = mlp_onehot.predict(x_train_cv)
```

after make prediction on test set we get that the *accuracy* of *TF-IDF* is the highest with 0.8 while the two other models give 0.74 for each one, for the execution time that each models need to make prediction on the test set we get

```
count vectorizer: 1.506387710571289 seconds

tfidf vectorizer: 1.01865553855896 seconds
```

One-Hot Encoding: 1.5098521709442139 seconds

```

so, we get that tf-idf is the faster

### Embedding vectorization result

Using accuracy as evaluation metrics where we get

| model | train set | test set |
|---|---|---|
| **glove** | **0.67** | **0.65** |
| **FastText with skip-gram** | **0.42** | **0.41** |
| **word2vec (skip-gram)** | **0.27** | **0.40** |

we get that the glove model has the best performance because it captures the co-occurrence, it learns from the co-occurrence statistics across the entire corpus, then fastText model, it's worse than glove because we have small dataset where the OOV words are rare so its subword information might not add much value in our case, the worst was skip-gram, it's from scratch, and it focuses on rare words while we have small dataset means not a lot of rare words, it relies on predecting context words from a given word, which doesn't perform well on the limited dataset as ours

when it comes to execution time we have

skip-gram: 0.00046s with only 100 sample from our dataset

fastText with CBOW: 0.00361s with the entire dataset

fastText with skip-gram: 0.00486s with the entire dataset

glove: 0.00474s with the entire dataset

the faster is fastText with CBOW but we can consider glove as the best model, it doesn't take lot of time (0.00474s) with good performance (accuracy more than 60 on both train and test datasets)
```