Ashkan Zarkhah
610399196

- **Documentation**
  Here we describe our code line by line and summarize what we have done in our report section.
  - **Document Preprocessing**
    First, we load the data from sorted CSV files that were available in the course's group.
    After that, we remove extra columns from both train and test data and only keep their text column.
    After that, we merge all our train data into one train dataset and we add a label column to it with a 1 value for positive sentiments and a -1 value for negative sentiments. We do so for the test data too and, at the end, we remove our old data to save some memory.

	text	sentiment
0	Bromwell High is a cartoon comedy. It ran at t...	1
1	Homelessness (or Houselessness as George Carli...	1
2	Brilliant over-acting by Lesley Ann Warren. Be...	1
3	This is easily the most underrated film inn th...	1
4	This is not the typical Mel Brooks film. It wa...	1
...	...	...
24995	Towards the end of the movie, I felt it was to...	-1
24996	This is the kind of movie that my enemies cont...	-1
24997	I saw 'Descent' last night at the Stockholm Fi...	-1
24998	Some films that you pick up for a pound turn o...	-1
24999	This is one of the dumbest films, I've ever se...	-1

    Now for tokenizing our documents, we need a unique tokenizer, so we develop a tokenizer, which gets a document and returns a list of tokens. To tokenize our documents, we first needed to replace all our characters with their lowercase form. After that, we needed to replace shortened forms with their original form so that our tokenizer wouldn't mistake them for punctuation marks. We used our list of shortened forms and added "i.e." and "e.g." to them. After that, it was time to use our nltk word tokenizer. After tokenizing our documents, we needed to fix our library's mistakes. So first, we removed all punctuation marks and all tokens that were built of only punctuation marks. After that, we removed punctuation marks that were left at the end of our tokens and then we split tokens that had punctuation marks in between. After that, we removed empty tokens which were made from splitting tokens, and also removed our stop word list from them. In the end, we used our "nltk.stemer" library and stemmed our tokens.
    Now that we have our tokenizer, we tokenize all our train documents and achieve a general token list or a dictionary. But as we can see, our dictionary is very big and it is going to bother us in our classification, so we decided to find the most repeated terms and only keep them.
    Now as we can see only the last 2000 terms are enough and the ones before that are very rare and won't affect our accuracy.

After that, it is time to build a hash function so that we can build our tf-idf matrix. In the hash function, we take each token and convert it to its 256 base equal number and module it by the number of different tokens, and, if it is unique, give it as its ID, and if it isn't, we will loop until we find a new unique ID for it.
Now that we have our hash function, it's time to tokenize all our documents again and build our train and test tf-idf matrix.
Now that we have our tf-idf matrixes it is time to build a data frame from them so that we can feed them to our models and learn and test our models. Also just to make sure we don't lose our data, because it is large and hard to calculate again, we save it and also remove our matrices to clear a bit of our memory.
Now everything is ready to learn our Naive-Bayes model.

- **Naive Bayes Classification**
  Now that everything is ready, we first define a Gaussian Naive-Bayes model for our classification. We use a Gaussian Naive-Bayes model because our tf-idf space is very sparse and normal Naive-Bayes models are for a small set of values for their features. After that, we first split our data into X and y and then we fit our model with our train data and test it on the test data and, we get very convincing evaluation values.

```
Accuracy train: 0.76
Precision train: 0.84
Recall train: 0.64
F1-score train: 0.73
Accuracy test: 0.72
Precision test: 0.81
Recall test: 0.58
F1-score test: 0.68
```

  After that, we remove the tf-idf data frames because we don't need them any longer and we can save some space for our next models.
- **Word Embeddings with SVM Classification**
  Now it is time for our word embeddings and using the SVM model to classify them. For our first embedding, we choose the Word2Vec embedding. First, we get all our documents tokenized again, and then we define our model, and, we feed it with our train-tokenized documents and we train it for 10 epochs.
  After that, we go through all our train and test data again, and we get a vector for each token of each document and choose the average function as our document embedding function.
  After having a vector for each of our train and test documents, we use an SVM model for classifying our documents. First, we define our model and then we train it on our vectorized train data and then we try to use it to predict our test data. After having all the predictions, we can evaluate our model and see how well it has worked.

```
Accuracy train: 0.83
Precision train: 0.83
Recall train: 0.84
F1-score train: 0.83
Accuracy test: 0.83
Precision test: 0.83
Recall test: 0.82
F1-score test: 0.83
```

- **LSA with SVM Classification**
  Now it is time to use LSA to reduce our dimension and train another SVM model on our dimension-reduced data again. First, we build an SVD model with 30 components. After that, we train it on our train data and we transform both our train and test data with it. Also, we remove the original vectors because we don't need them any longer and we need their space for our next embeddings.
  After having a reduced vector for each of our train and test documents, we use an SVM model for classifying our documents. First, we define our model and then we train it on our vectorized train data and then we try to use it to predict our test data. After having all the predictions, we can evaluate our model and see how well it has worked.

```
Accuracy train: 0.81
Precision train: 0.81
Recall train: 0.82
F1-score train: 0.81
Accuracy test: 0.81
Precision test: 0.82
Recall test: 0.81
F1-score test: 0.81
```

- **Using all Word Embeddings(Optional)**
  Now we try to use GloVe embedding. First, we download a learned GloVe model and use it exactly like the W2V method to vectorize our train and test documents. After having a vector for each of our train and test documents, we use an SVM model for classifying our documents. First, we define our model and then we train it on our vectorized train data and then we try to use it to predict our test data. After having all the predictions, we can evaluate our model and see how well it has worked.

```
Accuracy train: 0.81
Precision train: 0.80
Recall train: 0.81
F1-score train: 0.81
Accuracy test: 0.80
Precision test: 0.80
Recall test: 0.80
F1-score test: 0.80
```

Now we try to use FastText embedding. First, we train our FastText model on the train data and use it exactly like the W2V method to vectorize our train and test documents.
After having a vector for each of our train and test documents, we use an SVM model for classifying our documents. First, we define our model and then we train it on our vectorized train data and then we try to use it to predict our test data. After having all the predictions, we can evaluate our model and see how well it has worked.

```
Accuracy train: 0.81
Precision train: 0.80
Recall train: 0.81
F1-score train: 0.81
Accuracy test: 0.80
Precision test: 0.81
Recall test: 0.80
F1-score test: 0.80
```

- **Evaluation of the Test Dataset**
  We evaluated each method both on our train and test data and here we try to compare them together.

	Accuracy of Train data	Accuracy of Test data	Precision of Train data	Precision of Test data	Recall of Train data	Recall of Test data	F1-score of Train data	F1-score of Test data
Naive Bayes	76%	72%	84%	81%	64%	58%	73%	68%
W2V	83%	83%	83%	83%	84%	82%	83%	83%
W2V & LSA	81%	81%	81%	82%	82%	81%	81%	81%
GloVe	81%	80%	80%	80%	81%	80%	81%	80%
FastText	81%	80%	80%	81%	81%	80%	81%	80%

As we can see Naive Bayes model has worked a bit weaker than the other models and we can guess that words happenings are not very independent. Also in the embedding models merged with an SVM model, we can see that the Word2Vec model has worked a bit better than the others, even though for the GloVe model we have used a pre-trained model with a bigger dictionary. Also if we compare using LSA and not using it, we can see that with an LSA model, our learning process is much cheaper and faster, but we are losing some data and it has affected our models predictions.

- **Report**

  Here we summarize what we have done on our code.
  - **Document Preprocessing**
    - Loaded our test and train documents
    - Cleared extra columns
    - Merged all train data together
    - Merged all test data together
    - Defined a tokenizer function in which we:
      - Lowercased all characters
      - Replaced all shortened forms
      - Tokenized documents
      - Removed punctuation marks
      - Stemmed tokens
    - Build the dictionary
    - Filtered sparse terms
    - Defined a hash function for tf-idf matrixes
    - Build the tf-idf matrixes
  - **Naive Bayes Classification**
    - Used Gaussian Naive Bayes model
    - Trained Naive Bayes model
    - Predicted test data on the model
    - Evaluated the model both on train and test data based on:
      - Accuracy
      - Precision
      - Recall
      - F1-Score
  - **Word Embeddings with SVM Classification**
    - Used Word2Vec model for embedding terms
    - Trained the Word2Vec model on the train data
    - Calculated document embedding for both train and test data with an average document embedding function
    - Trained an SVM model on the embedded train documents
    - Predicted test data on the model
    - Evaluated the model both on train and test data based on:
      - Accuracy
      - Precision

            - Recall
            - F1-Score
    - **LSA with SVM Classification**
        - Trained an SVD model on the embedded train data
        - Reduced both embedded train and test data to 30 components with the SVD model
        - Trained an SVM model on the reduced embedded train documents
        - Predicted test data on the model
        - Evaluated the model both on train and test data based on:
            - Accuracy
            - Precision
            - Recall
            - F1-Score
    - **Using all Word Embeddings(Optional)**
        - Used GloVe model for embedding terms
        - Loaded the GloVe model
        - Calculated document embedding for both train and test data with an average document embedding function
        - Trained an SVM model on the embedded train documents
        - Predicted test data on the model
        - Evaluated the model both on train and test data based on:
            - Accuracy
            - Precision
            - Recall
            - F1-Score
        - Used FastText model for embedding terms
        - Trained the FastText model on the train data
        - Calculated document embedding for both train and test data with an average document embedding function
        - Trained an SVM model on the embedded train documents
        - Predicted test data on the model
        - Evaluated the model both on train and test data based on:
            - Accuracy
            - Precision
            - Recall
            - F1-Score
    - **Evaluation of the Test Dataset**
        - Evaluated all models in their learning time because of the memory limitations
        - Compared their results together and we achieved:
            - Words are not very independent. (Naive Bayes has worked weaker)
            - Word2Vec seems to be the best embedding.

            - Using the LSA model can reduce our computation cost but we can lose important data. (W2V with LSA works weaker than normal W2V)
    - **Key findings:**
        - We don't need to build all embeddings and evaluate them afterward, each embedding can be evaluated, deleted, and then go for the next one.
        - There can be shortened forms like "i.e." and "e.g." in the documents too.
        - There can be characters from different embeddings in a document set.
        - We can keep term frequencies and document frequencies separately and find them easily based on our terms and documents.
        - When we are using a Naive Bayes model on a continuous space, we must use the Gaussian Naive Bayes model.
        - With using dimension reduction, we are likely to lose important data and our model may work faster but it is likely to not be more accurate
    - **Challenges faced:**
        - Because we needed to tokenize train and test documents similarly we needed to implement a function to do so and without all tokens, finding special cases and stop words was hard and we were forced to use our last exercises data.
        - There were different embeddings in documents and we were forced to handle them manually.
        - The dictionary's size was too big and it was impossible to learn models on it.
        - The train data was small compared to the test data and the split could be much better.
        - The models sizes were too big and it was very hard to learn them on regular computers.
        - Embeddings were compared with an SVM model and because parameter finding was not in the exercise's purpose, the comparison is not very solid.
- **References**
    - **Pandas library (for loading and handling train and test data)**
    - **NLTK library (for tokenizing and stemming)**
    - **Sklearn library (for Gaussian Naive Bayes model, Evaluating models, SDM model, and SVD model)**
    - **Gensim library (for W2V, GloVe, and FastText model)**
    - **Numpy library (for building embedding matrixes)**