Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

CS7CS4 Machine Learning
Final Assignment

Boris Flesch
20300025

January 25, 2021

MSc Computer Science, Intelligent Systems

Contents

1 Part 1 – Reviews predictions

1.1 Introduction

Based on the provided video games reviews from Steam platform, our aim is to evaluate whether the review text can be used to (i) predict the review polarity and (ii) predict whether the review is for an "early access" version of a game or not.

We are first going to discuss data pre-processing. Then, we will mention the selected classification models and, finally, we will use these pre-processing steps and models to evaluate which predictions are possible based on the review text.

1.2 Data pre-processing

1.2.1 Parsing

To parse the JSON collection provided, we can use *jsonlines* that allows to parse seamlessly every item of the collection and store it in a Python list as following:

```
for item in jsonlines.open(path, 'r'):
    self.reviews.append(item['text'])
    self.voted_up.append(item['voted_up'])
    self.early_access.append(item['early_access'])
```

Reviews will systematically be used as our input feature. Depending on the prediction to make, either *voted_up* or *early_access* can be used as the output.

1.2.2 Transforming reviews text to features

To convert a review text to an input feature X, we could use a bag of words with *one-hot encoding*: every single word would act as a feature vector and a 1 would be added in the bag of words (i.e. matrix) every time the word is used in a review.

A quite similar but more sophisticated approach is to use TF-IDF. That approach considers the importance of words using Term Frequency (the frequency of a term in a document) and Inverse Document Frequency (the frequency of a term in the overall collection of documents). *Tfidf(t, d)* of token t in document d "is large for a token that occurs a lot in document d but only rarely in overall collection of documents" (Text Features lecture PDF, slide 12). Sklearn's TfidfVectorizer implements that TF-IDF approach. It can be used as following:

```
self.tfidf = TfidfVectorizer()
self.X = self.tfidf.fit_transform(self.reviews).toarray()
```

In this process of vectorization, we are not considering any language difference: all tokens are generated independently of the review language.

1.2.3 Tuning TfidfVectorizer

TfidfVectorizer provides parameters that can be tuned to improve the relevance of the generated features (i.e. vectors, tokens). These parameters are *min_df* and *max_df*, respectively ignoring "terms that have a document frequency strictly lower than the given threshold" and "terms that have a document frequency strictly higher than the given threshold" (Sklearn documentation for TfidfVectorizer). These parameters can be either integers representing an absolute threshold or floats representing a portion of the documents. By default, *min_df* is set to 1 (i.e. not any term is excluded) and *max_df* is set to 1.0 (i.e. again, not any term is excluded by default). The approach chosen to tune these parameters is the following:

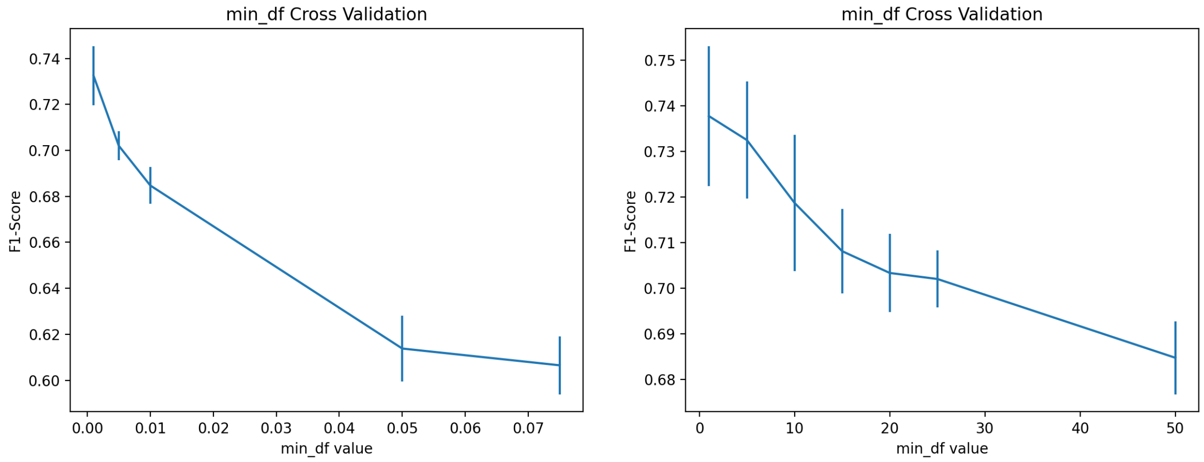
1. Use TfidfVectorizer with its default parameters
2. Use a test classification model such as Logistic Regression without any particular tuning (just for reference, i.e. to evaluate the relevance of changes made in data pre-processing)
3. Test a range of values for parameter *min_df* with a fixed *max_df* value and inversely
4. Cross-validate results (e.g. with F1-score) to choose appropriate *min_df* and *max_df*

That approach has been implemented with the following code (and a similar approach has been used for *cross_validate_max_df* method):

```
def cross_validate_min_df(self, min_df_range, max_df):
    mean_scores = []; std_scores = []
    for min_df in min_df_range:
        self.preprocess_data(predict="voted_up", min_df=min_df,
        ↪  max_df=max_df)
        X, y = self.get_data()
        tmp_model = LogisticRegression(C=10, max_iter=1000)
        scores = cross_val_score(tmp_model, X, y, cv=5, scoring='f1')
        mean_scores.append(np.array(scores).mean())
        std_scores.append(np.array(scores).std())
```

A first test has been done with a wide range of floats for *min_df*. However, 0.1 did already leave no term in the features. After focusing on a range of values < 0.1, it appeared that very small values for *min_df* maximises the F1-score (Figure 1a). It is therefore interesting to test a range of integers representing only a very small amount of documents (Figure 1b):

```
    # min_df_range = [0.001, 0.005, 0.01, 0.05, 0.075]
    min_df_range = [1, 5, 10, 15, 20, 25, 50]
    preprocessor.cross_validate_min_df(min_df_range, max_df=1.0)
```



(a) Floats range

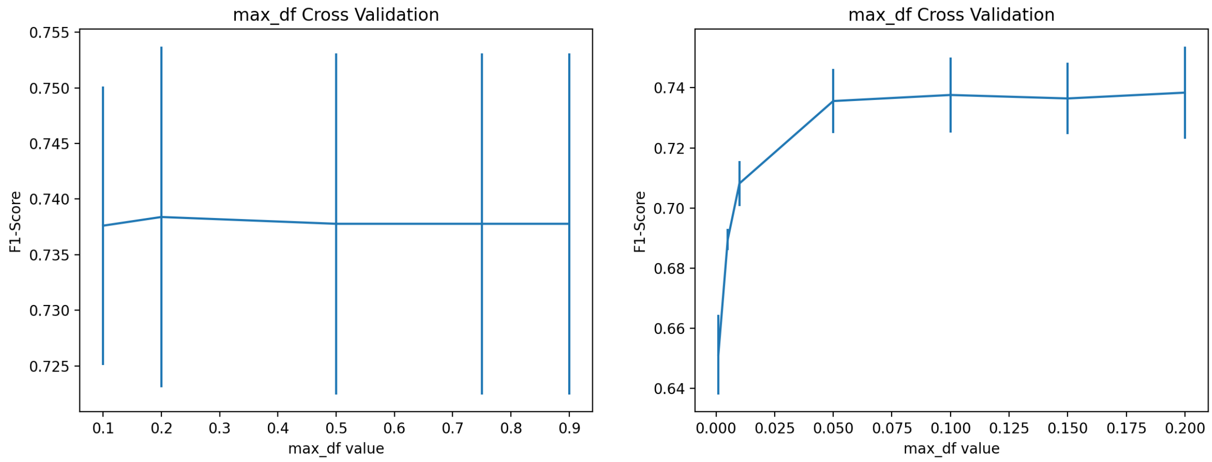
(b) Integers range

Figure 1: Cross-validation of *min_df*

It is worth noting that the default value of 1 which does not exclude any terms based on their low frequency actually maximises the F1-score. We can therefore argue that keeping *min_df* $= 1$ is a suitable choice.

A similar reasoning can be applied to *max_df*, for which choosing a value of 0.05 seems appropriate to maximise the F1-score while minimising its standard deviation.

```
# max_df_range = [0.1, 0.2, 0.5, 0.75, 0.9]
max_df_range = [0.001, 0.005, 0.01, 0.05, 0.1, 0.15, 0.2]
preprocessor.cross_validate_max_df(min_df=1, max_df_range=max_df_range)
```



(a) Wide range

(b) Small range (≤ 0.2)

Figure 2: Cross-validation of *max_df*

1.2.4 Improving the tokenizer

Stopwords

NLTK library provides lists of common stop words in several languages that can be used as a parameter of Sklearn's TfidfVectorizer. However, our collection is composed of reviews in many languages and no flag indicates that language. Two approaches have been tested:

(i) Focusing on stop words only in a single language, for example English

(ii) Including stop words from several or even all languages available in NLTK

The following code has been used to test approach (i):

```
nltk.download('stopwords')
stop = list(stopwords.words('english'))
self.tfidf = TfidfVectorizer(stop_words=set(stop), min_df=min_df,
    max_df=max_df)
self.X = self.tfidf.fit_transform(self.reviews)
self.X = self.X.toarray()
```

Tested on a similar Logistic Regression classifier as for TfidfVectorizer tuning, approach (i) provided a F1-Score of 0.73, approach (ii) a score of 0.725 and not using any stop words list provided a F1-Score of 0.73. As their impact is negligible, NLTK's stop words have not been used in this project.

Stemming

TfidfVectorizer accepts custom tokenizer. It is therefore possible to implement NLTK's stemmer which aim is to remove all affixes from words, keeping only their stem. NLTK provides two stemmers: PorterStemmer and SnowballStemmer. Two approaches have been tested to implement stemming:

(i) Detecting the language of each review and applying only the appropriate stemmer

(ii) Focusing on stemming only in a single language, for example English

Approach (i) seems to be the most appropriate approach. Some Python libraries such as *langdetect* are meant to detect the language of sentences:

```python
def tokenize(self, text):
    try:
        lang = self.isoToLanguage(langdetect.detect(text))
    except langdetect.lang_detect_exception.LangDetectException:
        lang = None
    tokens = nltk.word_tokenize(text)
    stems = []
    for item in tokens:
        if lang:
            stems.append(SnowballStemmer(lang).stem(item))
        else:
            stems.append(item)
    return stems
```

After testing, *langdetect*'s approach often decreased F1-score (down to approximately 0.715) and precision, depending on the test sample. An hypothesis is that it could be due to classification errors in language prediction. That approach also increased processing time.

While SnowballStemmer is said to be "better" than the original PorterStemmer in NLTK's documentation, it appeared that applying a generic PorterStemmer to every token of the review text increased the F1-score in this case. Therefore, approach (ii) seemed to be more efficient, leading to a slightly increased F1-score of 0.745:

```python
def tokenize(self, text):
    tokens = nltk.word_tokenize(text)
    stemmer = nltk.PorterStemmer()
    stems = [stemmer.stem(item) for item in tokens]
    return stems
```

In conclusion, approach (ii) seems to provide a more stable behaviour over the tests and increases our F1-score. We can therefore argue that keeping a generic PorterStemmer is an appropriate solution.

N-grams

The TfidfVectorizer approach provides feature vectors for each token which are, by default, 1-grams (i.e. words). However, that approach does not consider any relation between many words (i.e. sentences, paragraphs, 2 words, etc.).

It is possible to specify to TfidfVectorizer the range of n-gram to use. For instance, ranges of $(1, 2)$, $(2, 2)$ and $(1, 3)$ have been tested. However, adding n-grams ranges drastically increases the number of feature vectors, so as the overall processing time.

Ranges of $(1, 2)$ and $(2, 2)$ did not improve either F1-score, accuracy or any other metric of the test Logistic Regression model. $(1, 3)$ range slightly improved the F1-score but not enough to be relevant, especially considering the large increase of processing time (i.e. 477913 features for range $(1, 3)$ against only 43625 for range $(1, 1)$). For instance, the default $(1, 1)$ range is arguably sufficient.

1.3 Selected classification models

Different machine learning approaches have been tested to predict the review polarity and whether or not the review is for an "early access" version of a game based on the review text. For each model, we can first tune its hyperparameters using cross-validation and then evaluate its performance with different metrics (e.g. F1-score, confusion matrix, ROC/AUC, accuracy). As F1-score combines both precision and recall, it is a great metric to evaluate the performance of a classifier.

For convenience, a class *MyModel* has been defined with the following methods:

- *split(self, test_size=0.20)*: splits the data using Sklearn's *train_test_split* function (only training data is used in model's cross-validation and splitted in training/validation sets; so that test data remains unseen from the model during hyperparameters tuning)
- *train(self)*: trains the model using Sklearn's *model.fit()* function
- *save_model(self, path="")* and *load_model(self, path="")*

Abstract methods *confusion_matrix*, *print_report*, *plot_roc_curve* and *cross_validate_penalty* have also been defined. The idea of such an implementation is to provide a similar interface for every model to ease the usage of different models and the implementation of new ones.

1.3.1 Baseline

The "most frequent" strategy has been chosen as a baseline model. This baseline will act as a reference to evaluate the relevance and performance of actual models.

1.3.2 Logistic regression

Logistic regression classifier aim is to establish a decision boundary $\theta Tx = 0$ that allows it to predict the class of a given input with a certain confidence (i.e. probability). It relies on a cost function $J(\theta)$ that parameters θ need to minimise. It also takes advantage of a L2 penalty that encourages small (but non-zero) parameters values. That penalty can be tuned using C parameter.

The Logistic regression is implemented as following:

```
class MyLogisticRegression(MyModel):
    def __init__(self, X, y):
        # ...
    def init_model(self, C=10, max_iterations=10000):
        self.model = LogisticRegression(C=C, max_iter=max_iterations)

    def cross_validate_penalty(self, C_range, k_fold_nb, max_iterations=10000):
        mean_scores = []; std_scores = []
        for C in C_range:
            tmp_model = LogisticRegression(C=C, max_iter=max_iterations)
            scores = cross_val_score(tmp_model, self.X_train, self.y_train, cv=k_fold_nb, scoring='f1')
            mean_scores.append(np.array(scores).mean())
            std_scores.append(np.array(scores).std())
        plt.errorbar(C_range, mean_scores, yerr=std_scores)
        # ...
    def confusion_matrix(self):
        disp = plot_confusion_matrix(self.model, self.X_test, self.y_test)
        # ...
    def print_report(self):
        print(classification_report(self.y_test, y_pred))
        # ...
    def plot_roc_curve(self, display_plot=True):
        fpr, tpr, _ = roc_curve(self.y_test, decision_fct)
        plt.plot(fpr, tpr)
        # ...
```

Note: some methods displayed above have been shortened to keep them legible; full code is available in appendix. Also, as methods used to print models' metrics are very similar from one model another, they will not be displayed for each further model.

1.3.3 k-Nearest Neighbours (kNN)

kNN is an instance-based model: it bases its predictions directly on the training data. To predict the label of a point, kNN relies on the majority vote of its k closest training points. The distance is commonly measured as an Euclidian distance. Sklearn's KNeighborsClassifier provides a kNN classifier which is implemented in the following class:

```
class MyKNeighborsClassifier(MyModel):
    def __init__(self, X, y):
        # ...
    def init_model(self, n=5):
        self.model = KNeighborsClassifier(n_neighbors=n, weights='uniform')

    def cross_validate_n(self, n_range=[1, 10, 100, 1000], k_fold_nb=5):
```

```
        mean_scores = []; std_scores = []
        for n in n_range:
            tmp_model = KNeighborsClassifier(n_neighbors=n,
             ↪  weights='uniform')
            scores = cross_val_score(tmp_model, self.X_train,
             ↪  self.y_train, cv=k_fold_nb, scoring='f1')
            mean_scores.append(np.array(scores).mean())
            std_scores.append(np.array(scores).std())
        plt.errorbar(n_range, mean_scores, yerr=std_scores)
        # ...
```

1.3.4 Multi-Layer Perceptron (MLP)

A MLP is a neural network composed of an input layer, hidden layer(s) and an output layer. The number and size of hidden layers can be determined by cross-validation. In our case, we can use Sklearn's MLPClassifier with its standard solver *adam* implementing a "stochastic gradient-based optimizer" (Sklearn documentation for MLPClassifier) and the commonly used *ReLu* activation function:

```
class MyMLPClassifier(MyModel):
    def __init__(self, X, y):
        # ...
    def init_model(self, n=(25,), max_iterations=10000):
        self.model = MLPClassifier(hidden_layer_sizes=n,
         ↪  max_iter=max_iterations)

    def cross_validate_n(self, prev_hidden_layers=(), n_range=[1, 10,
     ↪  100, 1000], k_fold_nb=5, max_iterations=10000):
        mean_scores = []; std_scores = []
        for n in n_range:
            hls = prev_hidden_layers + (n,)
            tmp_model = MLPClassifier(hidden_layer_sizes=hls,
             ↪  max_iter=max_iterations)
            scores = cross_val_score(tmp_model, self.X_train,
             ↪  self.y_train, cv=k_fold_nb, scoring='f1')
            mean_scores.append(np.array(scores).mean())
            std_scores.append(np.array(scores).std())
        plt.errorbar(n_range, mean_scores, yerr=std_scores)
        # ...
```

The cross-validation function takes both the range of new hidden layer sizes and the sizes of previous hidden layers (if there are any). The aim of that implementation is to be able to add and cross-validate easily new layers one after another.

1.4 Predicting the review polarity

1.4.1 Experiments and results

Pre-processing

By using parameter *predict="voted_up"*, *y* (i.e. our output variable) will contain the field *voted_up* of each review and *X* will contain the pre-processed reviews texts.

```
preprocessor = MyPreprocessor()
preprocessor.read_data('dataset/reviews_17.jl.json')
preprocessor.preprocess_data(predict="voted_up", min_df=1, max_df=0.05)
X, y = preprocessor.get_data(); preprocessor.print_report()
```

The short data report indicates that 2500 reviews have been *voted_up* over a total of 5000 reviews (i.e. 50%): the data is very well balanced, which is helpful to train classifiers.

Baseline

The baseline model can be used through *MyBaseline* class as following:

```
baseline = MyBaseline(X, y)
baseline.split(test_size=0.20)
baseline.init_model()
baseline.train()
baseline.confusion_matrix()
baseline.print_report()
```

This baseline provides the following metrics and the confusion matrix displayed in Figure 3: Precision = 0.25, Recall = 0.5, Accuracy = 0.5 and F1-score = 0.33.

It is interesting to note that the accuracy obtained reflects well our 50/50 balanced data.

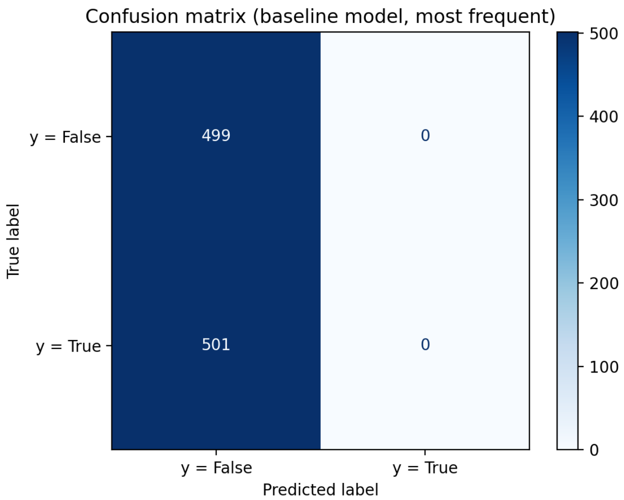


Figure 3: Baseline's confusion matrix

Logistic regression

We can use the following code to cross-validate hyperparameter *C* (see Figure 4 below):

```
logreg = MyLogisticRegression(X, y); logreg.split(test_size=0.20)
logreg.cross_validate_penalty(C_range=[0.1, 1, 10, 100, 1000],
 ↪  k_fold_nb=5)
```

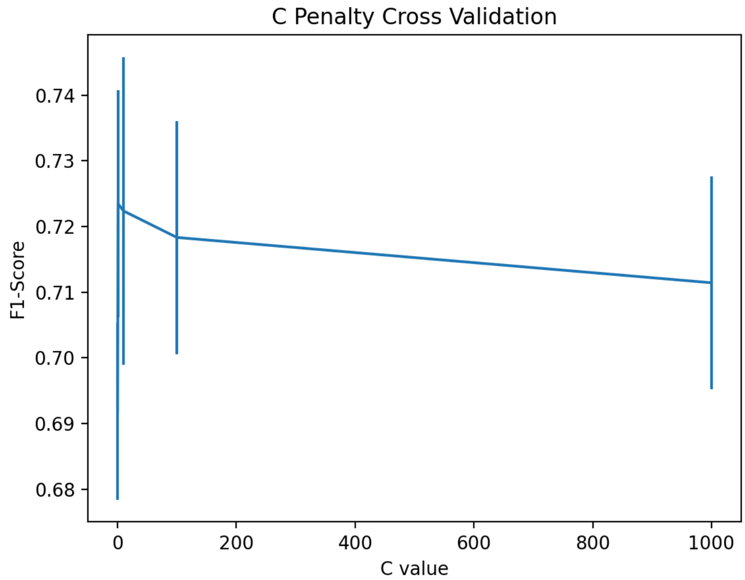
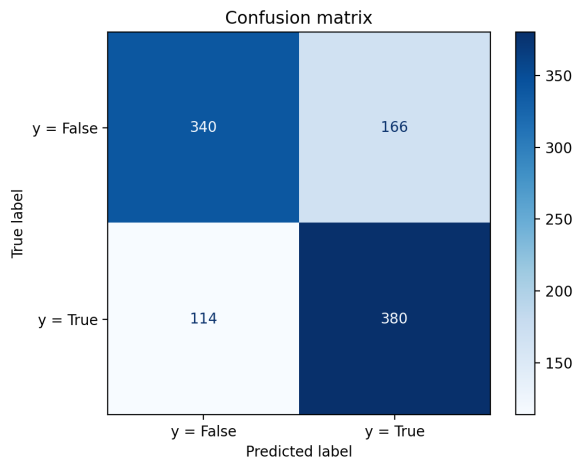


Figure 4: Logistic regression - C cross-validation

As $C = 10$ seems to maximise F1-score while preventing an important spread of the error (i.e. standard deviation), we can train and test the model with that hyperparameter value:

```
logreg.init_model(C=10, max_iterations=10000)
logreg.train(); logreg.confusion_matrix(); logreg.print_report()
```



(a) Confusion matrix

```
> Classification report:
                      precision    recall  f1-score   support

Not voted up (y=-1)        0.78      0.64      0.70       508
     Voted up (y=1)        0.68      0.82      0.74       492

           accuracy                            0.72      1000
          macro avg        0.73      0.73      0.72      1000
       weighted avg        0.73      0.72      0.72      1000
```

(b) Classification report

Figure 5: Logistic regression results

kNN

When cross-validating the kNN, it appears that a number of neighbours between 1 and 10 provides the highest F1-score as shown in Figure 6 below:

```
knn = MyKNeighborsClassifier(X, y); knn.split(test_size=0.20)
```

```
# knn.cross_validate_n(n_range=[1, 5, 10, 50, 100, 500, 1000],
↪  k_fold_nb=5)
knn.cross_validate_n(n_range=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
↪  k_fold_nb=5)
```

Therefore $n = 3$ neighbours is the most appropriate value as it maximises the F1-score while minimising the standard deviation. Our kNN can be trained using the code below (results Figure 7):

```
knn.init_model(n=3)
knn.train()
knn.confusion_matrix()
knn.print_report()
```

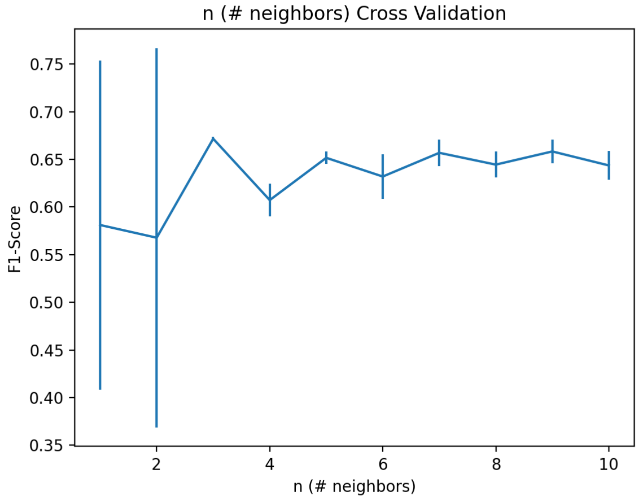
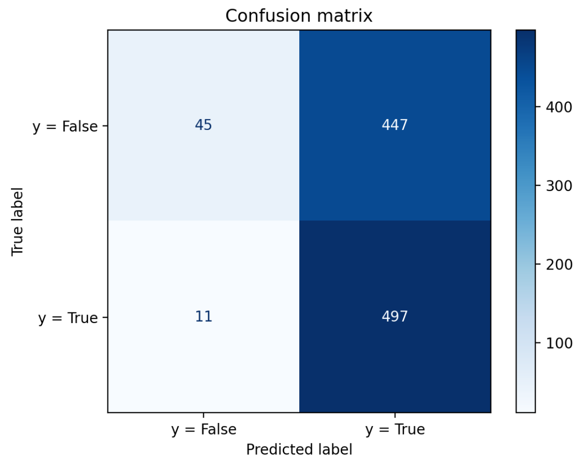


Figure 6: kNN Cross-validation



(a) Confusion matrix

	precision	recall	f1-score	support
Not voted up (y=-1)	0.81	0.11	0.20	496
Voted up (y=1)	0.53	0.97	0.68	504
accuracy			0.55	1000
macro avg	0.67	0.54	0.44	1000
weighted avg	0.67	0.55	0.44	1000

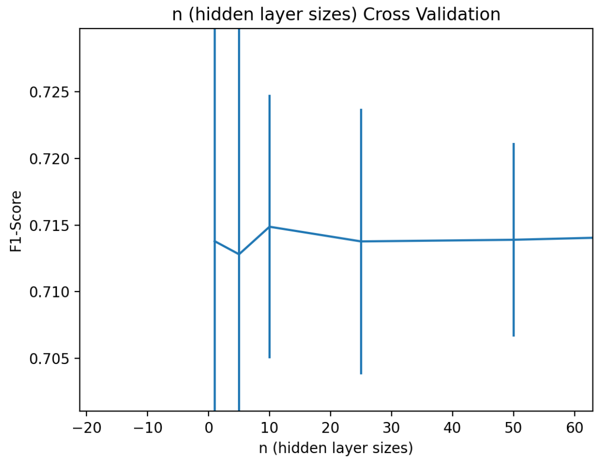
(b) Classification report

Figure 7: kNN results

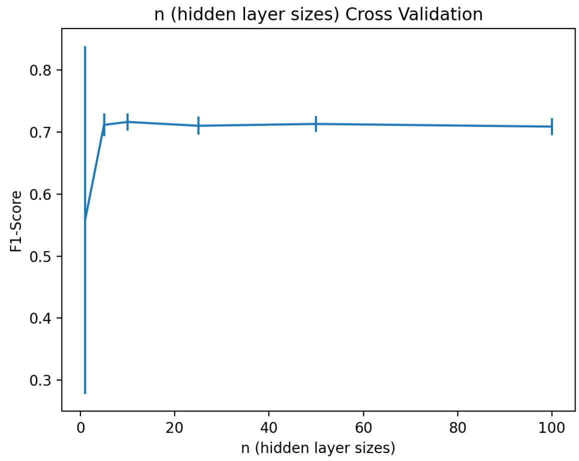
MLP

MLP layers can be cross-validated using the following code. The idea is to begin by cross-validating the size of a single hidden layer. Then, we can iterate to cross-validate the size of a potential second layer using that first cross-validated layer size.

```
mlp = MyMLPClassifier(X, y); mlp.split(test_size=0.20)
# mlp.cross_validate_n(prev_hidden_layers=(), n_range=[1, 5, 10, 25, 50,
↪  100, 500, 1000], max_iterations=10000)
mlp.cross_validate_n(prev_hidden_layers=(50,), n_range=[1, 5, 10, 25, 50,
↪  100], max_iterations=10000)
```



(a) Cross-validation of hidden layer 1 (zoomed)

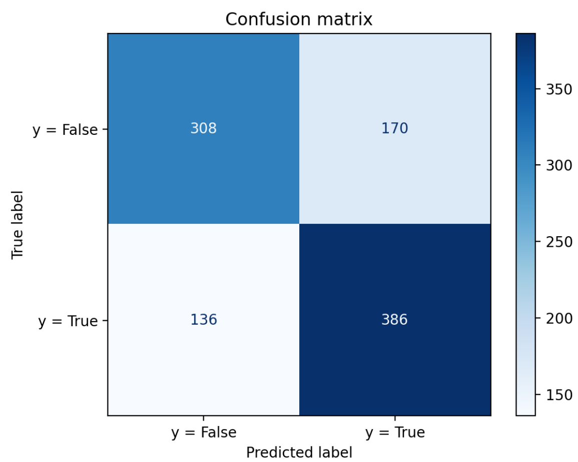


(b) Cross-validation of hidden layer 2

Figure 8: MLP Cross-validation

We can see from the cross-validation of the first layer (Figure 8a) that size ≥ 10 gives the highest F1-scores. To minimise standard deviation while keeping the model simple (i.e. which helps preventing over-fitting), we could choose $n = 50$ for the first layer. Adding new layers and/or nodes can allow our MLP to capture more behaviours/patterns in our training data (although too much nodes/layers can lead to over-fitting). As in this cas adding a second hidden layer (Figure 8b) does not provide any improvement of the F1-score, we can keep a single hidden layer of size 50.

```
mlp.init_model(n=(50,))
mlp.train(); mlp.confusion_matrix(); mlp.print_report()
```



(a) Confusion matrix

	precision	recall	f1-score	support
y = False	0.69	0.64	0.67	478
y = True	0.69	0.74	0.72	522
accuracy			0.69	1000
macro avg	0.69	0.69	0.69	1000
weighted avg	0.69	0.69	0.69	1000

(b) Classification report

Figure 9: MLP results

1.4.2 Discussion of the results

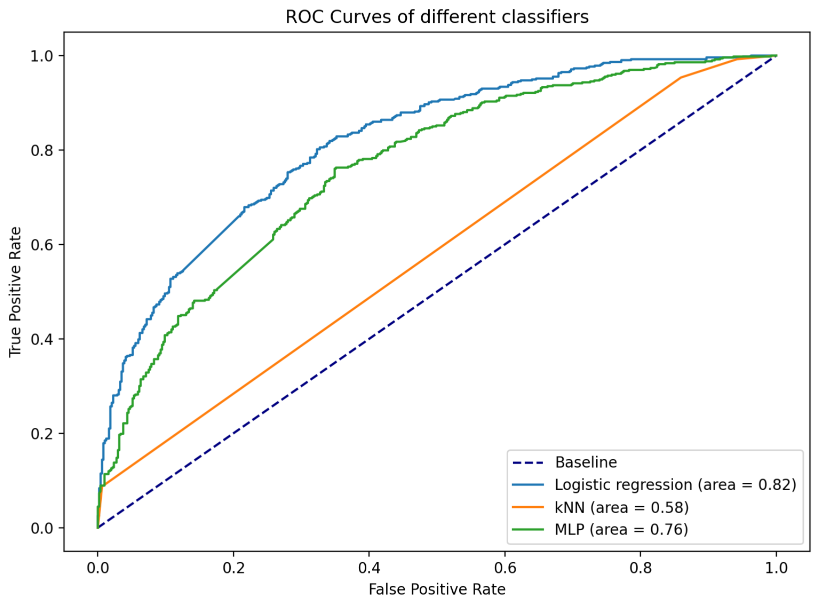


Figure 10: ROC Curves of different classifiers

ROC curves and AUC shown in Figure 10 beside are great metrics to evaluate the performance of our different models. The ROC curve is given by True Positive Rate vs False Positive Rate. As an ideal classifier would give 100% of true positive rate and 0% of false positive rate, we want a classifier which ROC curve comes as close as possible to the upper-left corner of the plot.

We can see that the kNN provides only slightly better classification performances than our baseline model. Therefore, we can argue that kNN is not suitable to predict the polarity of a review.

MLP and Logistic Regression are much closer to each other while providing a largely better ROC curve with AUCs of respectively 0.76 and 0.82. We can establish a small comparison table from the previous results (Figures 5 and 9) and our baseline model:

	Baseline	**MLP**	**Logistic Regression**
Precision	0.25	0.69	0.73
Recall	0.5	0.69	0.73
F1-score	0.33	0.69	0.72
Accuracy	0.5	0.69	0.72

Table 1: Comparison of Baseline, MLP and Logistic Regression (review polarity prediction)

We can see that both MLP and Logistic Regression provide great improvements over the baseline model: they both offer an increase of at least 175% of the precision, 38% of the recall, 109% of the F1-score and 38% of the accuracy.

Logistic Regression seems to provide overall better classification performance than MLP. In addition to that, Logistic Regression provides a few advantages over MLP:

1. Logistic Regression is easier and faster to train
2. Logistic Regression is sort of interpretable given the values of θ (i.e. to evaluate which features/tokens of the text have more importance to predict the polarity of the review), whereas MLP acts more as a "black box" which cannot be interpreted

In conclusion, we can argue that the Logistic Regression model is a suitable classifier to predict the polarity of a review based on its text.

1.5 Predicting if the review is for an "early access"

The same approach as for previous section 1.4 will be used to determine whether or not a review is for an "early access" version of a game. Therefore, code snippets will not be displayed again for readability. The results of tested models will still be given and discussed in the last part of this section.

1.5.1 Experiments and results

Pre-processing

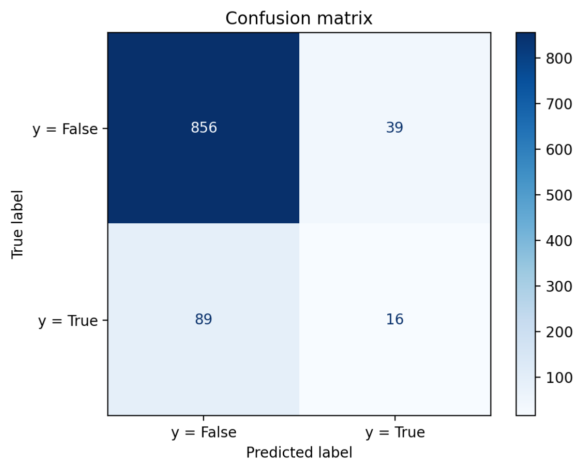
After pre-processing the data, it is interesting to note that only 518 reviews have $early\_access = True$ on a total of 5000 reviews (i.e. only 10%): the data is highly imbalanced. This bias may make the process of learning more difficult to classification models as they could easily tend toward the majority class.

Baseline

The "most frequent" baseline model gives the following metrics for *early_access* predictions: Precision = 0.43, Recall = 0.5, Accuracy = 0.86 and F1-score = 0.46.
The accuracy of that baseline model also reflects well how imbalanced is our data. Indeed, we have only 318/5000 "early access" reviews. In other terms, around 90% of our reviews are not "early access" and can be easily classified by that baseline's "most frequent" strategy.

Logistic regression

Logistic regression classifier's penalty has been cross-validated on a wide range of values from 0 to 5000. $C \leq 100$ provide a F1-score close to 0, while $C \geq 1000$ is required to "maximise" F1-score which peak (averaged) is around 0.12 (i.e. we can expect poor predictions from such a low score). Once trained with $C = 1000$, we obtain the following results:



(a) Confusion matrix

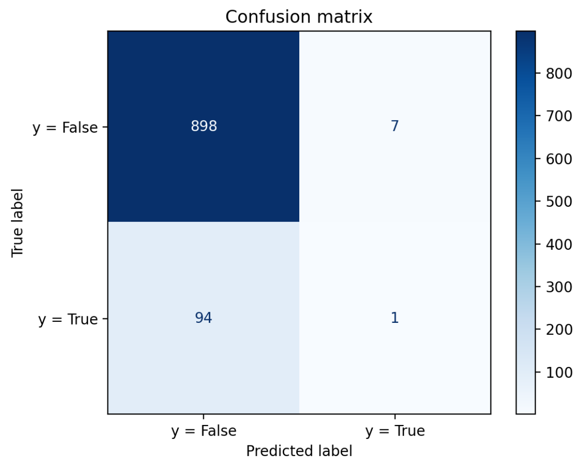
	precision	recall	f1-score	support
y = False	0.91	0.96	0.93	895
y = True	0.29	0.15	0.20	105
accuracy			0.87	1000
macro avg	0.60	0.55	0.57	1000
weighted avg	0.84	0.87	0.85	1000

(b) Classification report

Figure 11: Logistic Regression results

kNN

When cross-validating number of neighbours to consider in kNN classifier on a wide range of values (i.e. from 1 to 1000), the F1-score quickly drops to 0 when $C > 5$. When focusing on a shorter range of values, it seems that $n = 1$ provides the highest F1-score and gives the following classification results:



(a) Confusion matrix

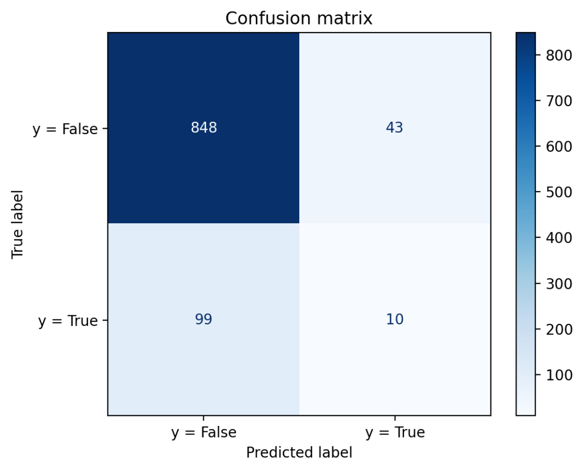
	precision	recall	f1-score	support
y = False	0.91	0.99	0.95	905
y = True	0.12	0.01	0.02	95
accuracy			0.90	1000
macro avg	0.52	0.50	0.48	1000
weighted avg	0.83	0.90	0.86	1000

(b) Classification report

Figure 12: kNN results

MLP

When determining the size of the first hidden layer of our MLP, using a size of $n = 50$ seems to maximise F1-score (approximately 0.125) while minimising the standard deviation. However, adding a new hidden layer does not improve the F1-score of our classifier: it is not able to capture further pattern/behaviour in our training data. Therefore, we can keep a single hidden layer of size $n = 50$ for this MLP classifier:



(a) Confusion matrix

	precision	recall	f1-score	support
y = False	0.90	0.95	0.92	891
y = True	0.19	0.09	0.12	109
accuracy			0.86	1000
macro avg	0.54	0.52	0.52	1000
weighted avg	0.82	0.86	0.84	1000

(b) Classification report

Figure 13: MLP results

1.5.2 Discussion of the results

Looking at the results of the tested models and the ROC curves displayed in Figure 14 beside, we can see that no model performs much better than the baseline.

The kNN's ROC curve and AUC are so close to the baseline that it is not relevant to consider this classifier; although it provides a F1-score and an accuracy which are slightly better than the baseline model (respectively 0.48 and 0.90). Actually, the training data is so imbalanced that the "most frequent" strategy of our baseline model already provides very high F1-score (0.46) and accuracy (0.86) compared to the tested models.

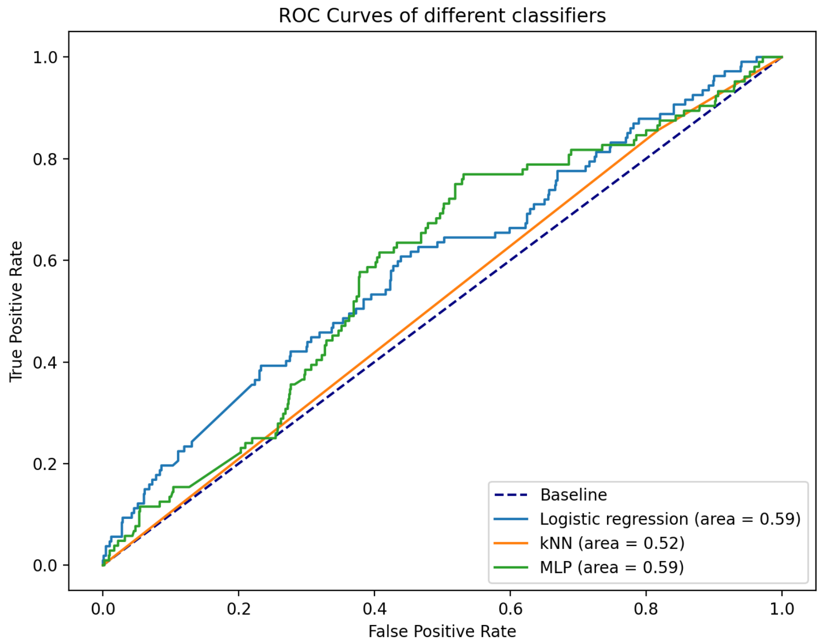


Figure 14: ROC Curves of different classifiers

MLP and Logistic Regression classifiers seem to perform better than the kNN. However, they do still not provide a big difference compared to the baseline. Again, we can compare MLP and Logistic Regression to our baseline model:

	Baseline	**MLP**	**Logistic Regression**
Precision	0.43	0.54	0.6
Recall	0.5	0.52	0.55
F1-score	0.46	0.52	0.57
Accuracy	0.86	0.86	0.87

Table 2: Comparison of Baseline, MLP and Logistic Regression (early access prediction)

As well as in section 1.4, Logistic Regression provides overall better classification performance than MLP. However, that model gives at most at increase of 24% of the baseline's F1-score and only 1% increase in accuracy.

In conclusion, none of the models seem to be suitable to predict whether a review has been given to an "early access" version of a game or not.

1.6 Conclusions and reflections

(i) We have shown that machine learning approaches such as MLP and Logistic Regression give significant classification performance to predict the review polarity (whether a game

has been “voted up” or not by the reviewer) based on the review text.
We can therefore conclude that the review text can be used to predict the review polarity.

However, these models could still be improved to offer even more robust classification performance. Further work would imply:

- Testing other approaches (e.g. SVM, Decision Tree Classifier, etc.).
- Considering the reviews languages (e.g. using different models depending on the language and/or tuning the model dependently of the language).
- Considering different pre-processing approaches such as Word2Vec or GloVe.

(ii) None of the models tested have been able to accurately predict whether the review is for an “early access” version of a game or not based on the review text. It would be almost as accurate to say that a review is *always* for a non-early access version of a game than using one of the tested machine learning approaches.
We can therefore conclude that the review text cannot be used to predict whether the review is for an "early access" version of a game or not.

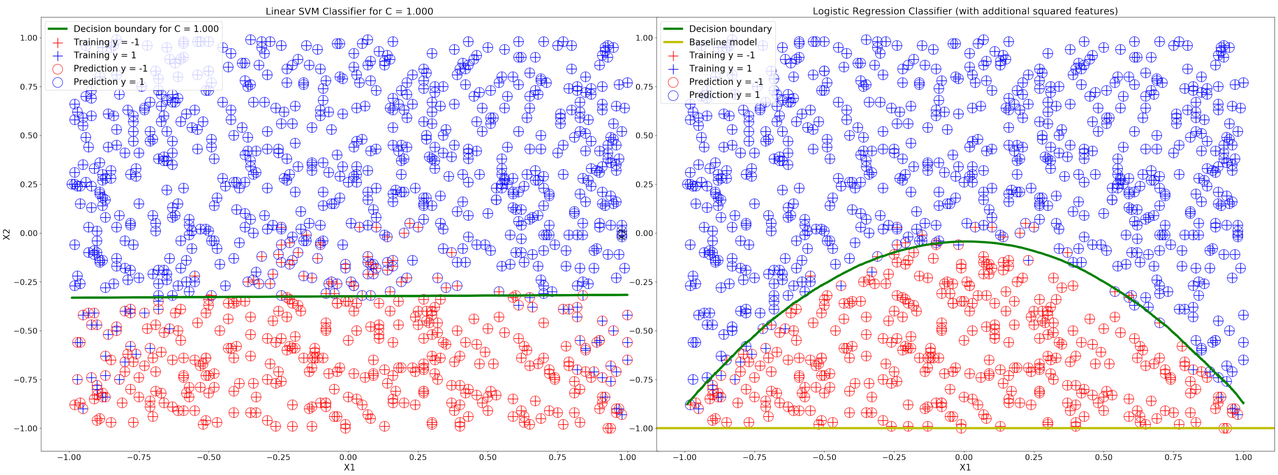
However, it is important to note that the given data was highly imbalanced and that could have impacted our classifiers. Therefore, it would be possible to make further work on that question to potentially obtain better classification performance:

- Undersampling could be used to decrease the number of reviews where $early\_access = False$ so that the data becomes more balanced. However, undersampling needs to be carefully done to prevent from adding bias into the data (e.g. not removing all reviews in a specific language). Also, balancing the data by undersampling would lead to a total of 1036 reviews, which is quite small compared to the original 5000 reviews.
- A better approach would be oversampling: instead of decreasing the number of reviews where $early\_access = False$, we could increase the number of reviews where $early\_access = True$ by including new reviews to the dataset.

2 Part 2.

2.1 Question (i)

We can speak of *under-fitting* when the model is too simple to catch the behaviour of our data (i.e. to fit to the shape of the data), thus leading to poor predictions. For example, trying to make prediction on quadratic-shaped data using a linear model would be a case of under-fitting. Using a more complex model, adding features (e.g. polynomial features) and tuning model's hyperparameters (e.g. penalty) can help preventing under-fitting.



(a) Under-fitting (linear model)

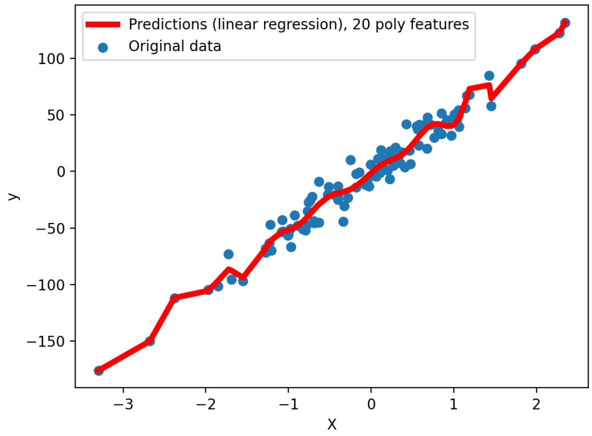
(b) Correct-fitting (quadratic model)

Figure 15: Example of under-fitting (*Source: Week 2 assignment*)

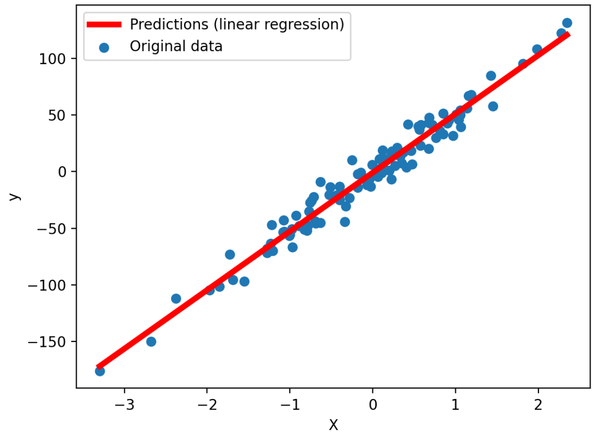
On the contrary, *over-fitting* occurs when the model is too complex and fits too much the noise of the training data. In other words, it means that the model is according too much importance to the noise of the training data and generalises poorly outside the scope of the training data. For example, using a model with too much polynomial features in a regression model can easily lead to over-fitting. Using a simpler model, using less features and tuning model's hyperparameters can help preventing over-fitting; a rule of thumb is to use the simplest model/parameters when many provide the same scores (e.g. F1-score, MSE, etc.).

The following code can be used to generate an example of over-fitting with too many polynomial features (see result Figure 16):

```
X, y = make_regression(n_samples=100, n_features=1, noise=10)
X, y = zip(*sorted(zip(X, y)))   # Sort data
reg = LinearRegression().fit(X, y)
plt.scatter(X, y)
plt.plot(X, reg.predict(X))
# ...
reg = make_pipeline(PolynomialFeatures(20), LinearRegression())
reg.fit(X, y)
plt.scatter(X, y)
plt.plot(X, reg.predict(X))
```



(a) Over-fitting (polynomial features up to degree 20)



(b) Correct-fitting (no polynomial features)

Figure 16: Example of over-fitting

2.2 Question (ii)

The following algorithm shows an implementation of k-fold cross-validation from scratch.

Notes:

- array[x : y] retrieves indexes x (included) to y (excluded) of the array.
- *hyperparameter* is the hyperparameter to cross-validate (e.g. a penalty, number of neighbours, etc.)
- *scoreFunction* represents a scoring function such as MSE, F1-score, etc.
- *meanScores* is used in this example but other metrics such as standard deviation of the scores or errors could also be considered

Algorithm 1 k-fold cross-validation

```
X, y ← Original data
nbDataPoints ← length(X)
k ← number of folds

// 1. Split data into k folds
for i = 0, ..., k − 1 do
    offset ← i ∗ nbDataPoints/k
    maxIndex ← offset + nbDataPoints/k
    folds[i][X] ← X[offset : maxIndex]
    folds[i][y] ← y[offset : maxIndex]
end for

// 2. Perform cross-validation
meanScores ← []
range ← [1, 10, 100, 1000]
for value in range do
    tmpScores ← []
    model ← TheModel(hyperparameter = value),
    for i = 0, ..., k − 1 do
        train ← concatenate(folds[0 : i], folds[i + 1 : k])
        test ← folds[i]
        model.train(train[X], train[y])
        predictions ← model.predict(test[X])
        tmpScores.append(scoreFunction(test[y], predictions))
    end for
    meanScores.append(mean(tmpScores))
end for

// 3. Evaluate
Plot(meanScores)
```

2.3 Question (iii)

Tuning model's hyperparameter(s) can help finding the right trade-off between over- and under-fitting, for example by adding penalty to some parameters of the model. A straightforward approach consists in splitting the data in a training set and a testing set (e.g. 80/20 split) to then train and evaluate the model over a range of hyperparameters values. However, if either the training data or the test data is noisy, hyperparameters' may be tuned inadequately, thus leading to poor predictions.

K-fold cross-validation provides a way to evaluate a model by equally considering the data provided. That is, training the model each time with a different slice (i.e. fold) of the original data. That process allows to smooth out the noise by averaging the scores (e.g. MSE, F1-score) of the trained model. Therefore, k-fold cross-validation is a great technique for tuning model's hyperparameters which are directly correlated with the balance between over- and under-fitting.

2.4 Question (iv)

An advantage of a kNN classifier over a logistic regression classifier is that kNNs are very easy to use, simple and effective. Indeed, a kNN only requires its number of neighbours k to be set properly and it then provides predictions based directly on the training data (i.e. kNNs are instance- or data-based models). Logistic regression classifiers are a bit more complex as they rely on the minimisation of a cost function to make predictions.

However, the fact that kNNs rely on the data itself is also a downside compared to logistic regressions. As a kNN requires to consider all data points available to make predictions, it is a good solution for small data but it can quickly become much slower than a logistic regression classifier because of its large computation cost at runtime if the amount of data is important.

Finally, another con of kNNs is that they are not good at making predictions when stepping away from training data (i.e. outliers). This is due to the fact that kNNs rely directly on the training data and "stepping away" from it implies that less data points are available in that area; that is, less relevant neighbours to consider. On the contrary, a logistic regression classifier can easily extrapolate on the training data to make predictions above that scope (i.e. by "extending" its decision boundary based on its parameters θ).

2.5 Question (v)

kNN classifiers are highly sensitive to imbalanced data. When making predictions on imbalanced data, a kNN would tend to predict the majority class.
That sensitivity to imbalanced data is directly correlated with the measurement of distances and the fact that kNNs are instance-based models: the more imbalanced is the original data, the more nearest neighbours of one class we can potentially find close to the data point to predict; thus leading to tend to predict the majority class.

A kNN would also give inaccurate predictions for high-dimensions data (i.e. data having an important number of features). For instance, if we consider two data points of dimension 2 $x_1 = (14, 13)$ and $x_2 = (19, 0)$, they are both at a distance of approximately $d = 19$ from the origin $(0, 0)$. However, we might want our kNN to find points which are closer on every axis instead of just a single one. The more features we add, the more difficult it becomes for our kNN to determine the closest (relevant) neighbours of a point. This challenge is also known as the "Curse of Dimensionality".
That may also explain why kNNs have not been able to provide accurate predictions for both parts (i) and (ii) of the first part of this assignment, considering the large number of features provided.

A Appendix

A.1 Python Code

A.1.1 Class MyPreprocessor

```python
import jsonlines
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
import nltk
from nltk.stem.snowball import SnowballStemmer
import langdetect


class MyPreprocessor:
    def __init__(self):
        self.reviews = []
        self.voted_up = []
        self.early_access = []
        self.X = None
        self.y = None
        self.tfidf = None

    def read_data(self, path):
        print("> Reading data")
        for item in jsonlines.open(path, 'r'):
            self.reviews.append(item['text'])
            self.voted_up.append(item['voted_up'])
            self.early_access.append(item['early_access'])

    def cross_validate_min_df(self, min_df_range, max_df):
        mean_scores = []; std_scores = []
        for min_df in min_df_range:
            self.preprocess_data(predict="voted_up", min_df=min_df,
             ↪ max_df=max_df)
            X, y = self.get_data()
            print("\t> For min_df = %.2f..." % min_df)
            tmp_model = LogisticRegression(C=10, max_iter=1000)
            scores = cross_val_score(tmp_model, X, y, cv=5, scoring='f1')
            mean_scores.append(np.array(scores).mean())
            std_scores.append(np.array(scores).std())

        print("> min_df cross validation done")
        plt.figure()
        plt.errorbar(min_df_range, mean_scores, yerr=std_scores)
```

```
        plt.title("min_df Cross Validation")
        plt.xlabel("min_df value")
        plt.ylabel("F1-Score")
        plt.show()

    def cross_validate_max_df(self, min_df, max_df_range):
        mean_scores = []; std_scores = []
        for max_df in max_df_range:
            self.preprocess_data(predict="voted_up", min_df=min_df,
            ↪  max_df=max_df)
            X, y = self.get_data()
            print("\t> For max_df = %.2f..." % max_df)
            tmp_model = LogisticRegression(C=10, max_iter=1000)
            scores = cross_val_score(tmp_model, X, y, cv=5, scoring='f1')
            mean_scores.append(np.array(scores).mean())
            std_scores.append(np.array(scores).std())

        print("> max_df cross validation done")
        plt.figure()
        plt.errorbar(max_df_range, mean_scores, yerr=std_scores)
        plt.title("max_df Cross Validation")
        plt.xlabel("max_df value")
        plt.ylabel("F1-Score")
        plt.show()

    def isoToLanguage(self, lang):
        if lang == "da":
            return "danish"
        elif lang == "nl":
            return "dutch"
        elif lang == "en":
            return "english"
        elif lang == "fi":
            return "finnish"
        elif lang == "fr":
            return "french"
        elif lang == "de":
            return "german"
        elif lang == "hu":
            return "hungarian"
        elif lang == "it":
            return "italian"
        elif lang == "pt":
            return "portuguese"
        elif lang == "ro":
            return "romanian"
        elif lang == "ru":
            return "russian"
```

```python
        elif lang == "es":
            return "spanish"
        elif lang == "sv":
            return "swedish"
        else:
            return None

    def tokenize(self, text, use_lang_detect=False):
        tokens = nltk.word_tokenize(text)
        stems = []

        if use_lang_detect:
            try:
                lang = langdetect.detect(text)
                lang = self.isoToLanguage(lang)
            except langdetect.lang_detect_exception.LangDetectException:
                lang = None

            for item in tokens:
                if lang:
                    stems.append(SnowballStemmer(lang).stem(item))
                else:
                    stems.append(item)
        else:
            stemmer = nltk.PorterStemmer()
            stems = [stemmer.stem(item) for item in tokens]

        return stems

    def preprocess_data(self, predict="voted_up", min_df=1, max_df=1.0):
        self.tfidf = TfidfVectorizer(tokenizer=self.tokenize,
        ↪  min_df=min_df, max_df=max_df, ngram_range=(1, 1))
        self.X = self.tfidf.fit_transform(self.reviews)
        self.X = self.X.toarray()

        if predict == "voted_up":
            self.y = self.voted_up
        else:
            self.y = self.early_access

    def get_data(self):
        return self.X, self.y

    def print_report(self, print_features_names=False):
        print("> Features:")
        print("\t> Number:", len(self.tfidf.get_feature_names()))
        if print_features_names:
            print("\t> Names:", self.tfidf.get_feature_names())
```

```python

        print("\n> Total number of reviews:", len(self.reviews))

        voted_up_reviews = 0
        for voted_up in self.voted_up:
            if voted_up:
                voted_up_reviews += 1

        print("> Total number of \"voted_up\" reviews:",
         ↪  voted_up_reviews)

        early_access_reviews = 0
        for early_access in self.early_access:
            if early_access:
                early_access_reviews += 1

        print("> Total number of \"early_access\" reviews:",
         ↪  early_access_reviews)
```

A.1.2 Class MyModel

```python
import abc
from joblib import dump, load
from sklearn.model_selection import train_test_split


class MyModel:
    def save_model(self, path=""):
        if not path:
            path = "saved-models/" + self.__class__.__name__ + ".joblib"

        dump(self.model, path)

    def load_model(self, path=""):
        if not path:
            path = "saved-models/" + self.__class__.__name__ + ".joblib"

        self.model = load(path)

    def split(self, test_size=0.20):
        self.X_train, self.X_test, self.y_train, self.y_test =
         ↪  train_test_split(self.X, self.y, test_size=test_size)   #
         ↪  random_state=42

    def train(self):
        self.model.fit(self.X_train, self.y_train)

    @abc.abstractmethod
```

```
    def init_model(self):
        pass

    @abc.abstractmethod
    def confusion_matrix(self):
        pass

    @abc.abstractmethod
    def print_report(self):
        pass

    @abc.abstractmethod
    def plot_roc_curve(self):
        pass

    @abc.abstractmethod
    def cross_validate_penalty(self):
        pass
```

A.1.3 Class MyBaseline

```
from models.MyModel import MyModel
from sklearn.dummy import DummyClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_recall_fscore_support
import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix


class MyBaseline(MyModel):
    def __init__(self, X, y):
        self.X = X
        self.y = y
        self.X_train = None
        self.X_test = None
        self.y_train = None
        self.y_test = None
        self.model = None

    def init_model(self):
        self.model = DummyClassifier(strategy="most_frequent")

    def cross_validate_penalty(self):
        print("No cross-validation available for Baseline")

    def confusion_matrix(self):
        print("> Confusion matrix & scores")
```

```python
        disp = plot_confusion_matrix(self.model, self.X_test,
        ↪ self.y_test, display_labels=["y = False", "y = True"],
        ↪ cmap=plt.cm.Blues, values_format='d')
        disp.ax_.set_title("Confusion matrix (baseline model, most
        ↪ frequent)")
        plt.show()

    def print_report(self):
        y_pred = self.model.predict(self.X_test)
        precision, recall, fscore, _ =
        ↪ precision_recall_fscore_support(self.y_test, y_pred,
        ↪ average="macro")
        print("> Baseline scores (strategy: most frequent):")
        print("Precision (avg):", precision, ", Recall (avg):", recall,
        ↪ ", F1-Score (avg):", fscore)
        print("Accuracy:", accuracy_score(self.y_test, y_pred))

    def plot_roc_curve(self, display_plot=True):
        """Plot ROC curve and ROC area of the model"""
        plt.figure(123)
        plt.plot([0, 1], [0, 1], color='navy', linestyle='--',
        ↪ label="Baseline")
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('ROC Curve - Baseline')
        plt.legend(loc="lower right")
        if display_plot:
            plt.show()
```

A.1.4 Class MyLogisticRegression

```python
from models.MyModel import MyModel
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import plot_confusion_matrix
import numpy as np
from sklearn.metrics import classification_report
from sklearn.metrics import roc_curve, auc


class MyLogisticRegression(MyModel):
    def __init__(self, X, y):
        self.X = X
        self.y = y
        self.X_train = None
        self.X_test = None
        self.y_train = None
```

```python
        self.y_test = None
        self.model = None

    def init_model(self, C=10, max_iterations=10000):
        self.model = LogisticRegression(C=C, max_iter=max_iterations)

    def cross_validate_penalty(self, C_range=[1, 10, 100, 1000], k_fold_nb=5, max_iterations=10000):
        print("> C Cross validation (range:", C_range, ")")
        mean_scores = []; std_scores = []

        for C in C_range:
            print("\t> For C = %.2f..." % C)
            tmp_model = LogisticRegression(C=C, max_iter=max_iterations)
            scores = cross_val_score(tmp_model, self.X_train, self.y_train, cv=k_fold_nb, scoring='f1')
            mean_scores.append(np.array(scores).mean())
            std_scores.append(np.array(scores).std())

        print("> C Cross validation done")
        plt.figure()
        plt.errorbar(C_range, mean_scores, yerr=std_scores)
        plt.title("C Penalty Cross Validation")
        plt.xlabel("C value")
        plt.ylabel("F1-Score")
        plt.show()

    def confusion_matrix(self):
        print("> Confusion matrix:")
        plt.figure()
        disp = plot_confusion_matrix(self.model, self.X_test, self.y_test, display_labels=["y = False", "y = True"], cmap=plt.cm.Blues,
                                     values_format='d')
        disp.ax_.set_title("Confusion matrix")
        plt.show()

    def print_report(self):
        y_pred = self.model.predict(self.X_test)
        print("> Classification report:")
        print(classification_report(self.y_test, y_pred, target_names=["y = False", "y = True"]))

        print("> Intercept:")
        print(self.model.intercept_)
        print("> Coefficients:")
        print(self.model.coef_)

```

```
    def plot_roc_curve(self, display_plot=True):
        """Plot ROC curve and ROC area of the model"""
        decision_fct = self.model.decision_function(self.X_test)
        fpr, tpr, _ = roc_curve(self.y_test, decision_fct)
        plt.figure(123)
        plt.plot(fpr, tpr, label='Logistic regression (area = %0.2f)' %
        ↪  auc(fpr, tpr))
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('ROC Curve - Logistic Regression')
        plt.legend(loc="lower right")
        if display_plot:
            plt.show()
```

A.1.5 Class MyKNeighborsClassifier

```
from models.MyModel import MyModel
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import plot_confusion_matrix
import numpy as np
from sklearn.metrics import classification_report
from sklearn.metrics import roc_curve, auc


class MyKNeighborsClassifier(MyModel):
    def __init__(self, X, y):
        self.X = X
        self.y = y
        self.X_train = None
        self.X_test = None
        self.y_train = None
        self.y_test = None
        self.model = None

    def init_model(self, n=5):
        self.model = KNeighborsClassifier(n_neighbors=n,
        ↪  weights='uniform')

    def cross_validate_penalty(self):
        print("No cross-validation available for kNN")

    def cross_validate_n(self, n_range=[1, 10, 100, 1000], k_fold_nb=5):
        print("> n cross validation (range:", n_range, ")")
        mean_scores = []; std_scores = []

        for n in n_range:
```

```
            print("\t> For n = %d..." % n)
            tmp_model = KNeighborsClassifier(n_neighbors=n, weights='uniform')
            scores = cross_val_score(tmp_model, self.X_train, self.y_train, cv=k_fold_nb, scoring='f1')
            mean_scores.append(np.array(scores).mean())
            std_scores.append(np.array(scores).std())

        print("> n cross validation done")
        plt.figure()
        plt.errorbar(n_range, mean_scores, yerr=std_scores)
        plt.title("n (# neighbors) Cross Validation")
        plt.xlabel("n (# neighbors)")
        plt.ylabel("F1-Score")
        plt.show()

    def confusion_matrix(self):
        print("> Confusion matrix:")
        plt.figure()
        disp = plot_confusion_matrix(self.model, self.X_test, self.y_test, display_labels=["y = False", "y = True"], cmap=plt.cm.Blues,
                                     values_format='d')
        disp.ax_.set_title("Confusion matrix")
        plt.show()

    def print_report(self):
        y_pred = self.model.predict(self.X_test)
        print("> Scores:")
        print(classification_report(self.y_test, y_pred, target_names=["y = False", "y = True"]))

    def plot_roc_curve(self, display_plot=True):
        """Plot ROC curve and ROC area of the model"""
        y_pred_proba = self.model.predict_proba(self.X_test)
        fpr, tpr, _ = roc_curve(self.y_test, y_pred_proba[:, 1])
        plt.figure(123)
        plt.plot(fpr, tpr, label='kNN (area = %0.2f)' % auc(fpr, tpr))
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('ROC Curve - kNN')
        plt.legend(loc="lower right")
        if display_plot:
            plt.show()
```

A.1.6 Class MyMLPClassifier

```
from models.MyModel import MyModel
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import plot_confusion_matrix
import numpy as np
from sklearn.metrics import classification_report
from sklearn.metrics import roc_curve, auc


class MyMLPClassifier(MyModel):
    def __init__(self, X, y):
        self.X = X
        self.y = y
        self.X_train = None
        self.X_test = None
        self.y_train = None
        self.y_test = None
        self.model = None

    def init_model(self, n=(25,), max_iterations=10000):
        self.model = MLPClassifier(hidden_layer_sizes=n, max_iter=max_iterations)

    def cross_validate_penalty(self):
        print("No cross-validation available for Baseline")

    def cross_validate_n(self, prev_hidden_layers=(), n_range=[1, 10, 100, 1000], k_fold_nb=5, max_iterations=10000):
        # Add parameters "prev_hidden_layers" (i.e. to cross-validate a new layer's size)
        print("> n cross validation (range:", n_range, ")")
        mean_scores = []; std_scores = []

        for n in n_range:
            hls = prev_hidden_layers + (n,)
            print("\t> For hidden_layer_sizes =", hls, "...")
            tmp_model = MLPClassifier(hidden_layer_sizes=hls, max_iter=max_iterations)
            scores = cross_val_score(tmp_model, self.X_train, self.y_train, cv=k_fold_nb, scoring='f1')
            mean_scores.append(np.array(scores).mean())
            std_scores.append(np.array(scores).std())

        print("> n cross validation done")
        plt.figure()
```

```
        plt.errorbar(n_range, mean_scores, yerr=std_scores)
        plt.title("n (hidden layer sizes) Cross Validation")
        plt.xlabel("n (hidden layer sizes)")
        plt.ylabel("F1-Score")
        plt.show()

    def confusion_matrix(self):
        print("> Confusion matrix:")
        plt.figure()
        disp = plot_confusion_matrix(self.model, self.X_test, self.y_test, display_labels=["y = False", "y = True"], cmap=plt.cm.Blues,
                                     values_format='d')
        disp.ax_.set_title("Confusion matrix")
        plt.show()

    def print_report(self):
        y_pred = self.model.predict(self.X_test)
        print("> Scores:")
        print(classification_report(self.y_test, y_pred, target_names=["y = False", "y = True"]))

    def plot_roc_curve(self, display_plot=True):
        """Plot ROC curve and ROC area of the model"""
        y_pred_proba = self.model.predict_proba(self.X_test)
        fpr, tpr, _ = roc_curve(self.y_test, y_pred_proba[:, 1])
        plt.figure(123)
        plt.plot(fpr, tpr, label='MLP (area = %0.2f)' % auc(fpr, tpr))
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('ROC Curve - MLP Classifier')
        plt.legend(loc="lower right")
        if display_plot:
            plt.show()
```

A.1.7 Main script

```
from models.MyBaseline import MyBaseline
from models.MyLogisticRegression import MyLogisticRegression
from models.MyPreprocessor import MyPreprocessor
from models.MyKNeighborsClassifier import MyKNeighborsClassifier
from models.MyMLPClassifier import MyMLPClassifier
import matplotlib.pyplot as plt

part_i = True
part_ii = True

if part_i:
```

```
    # (i) predict the review polarity (where a game has been “voted up" or not by the reviewer)

    # (i)(i) Preprocessing data
    print("> Preprocessing data")
    preprocessor = MyPreprocessor()
    preprocessor.read_data('dataset/reviews_17.jl.json')
    preprocessor.preprocess_data(predict="voted_up", min_df=1, max_df=0.05)
    X, y = preprocessor.get_data()

    preprocessor.print_report()

    # print("\n=== min_df cross-validation ===")
    # min_df_range = [1, 5, 10, 15, 20, 25, 50]
    # print("> min_df cross validation (range:", min_df_range, ")")
    # preprocessor.cross_validate_min_df(min_df_range=min_df_range, max_df=1.0)  # max_df=0.150

    # print("\n=== max_df cross-validation ===")
    # max_df_range = [0.001, 0.005, 0.01, 0.05, 0.1, 0.15, 0.2]
    # print("> max_df cross validation (range:", max_df_range, ")")
    # preprocessor.cross_validate_max_df(min_df=1, max_df_range=max_df_range)

    # (i)(ii) Baseline model
    print("\n=== BASELINE MODEL ===")
    baseline = MyBaseline(X, y)
    baseline.split(test_size=0.20)
    baseline.init_model()
    baseline.train()
    # baseline.confusion_matrix()
    baseline.print_report()
    baseline.plot_roc_curve(display_plot=False)

    # (i)(iii) Logistic regression
    print("\n=== LOGISTIC REGRESSION ===")
    logreg = MyLogisticRegression(X, y)
    logreg.split(test_size=0.20)
    # logreg.cross_validate_penalty(C_range=[0.1, 1, 10, 100, 1000], k_fold_nb=5)
    logreg.init_model(C=10, max_iterations=10000)
    logreg.train()
    # logreg.confusion_matrix()
    logreg.print_report()
    logreg.plot_roc_curve(display_plot=False)

    # (i)(iv) kNN
```

```
    print("\n=== K NEAREST NEIGHBORS CLASSIFIER ===")
    knn = MyKNeighborsClassifier(X, y)
    knn.split(test_size=0.20)
    # knn.cross_validate_n(n_range=[1, 5, 10, 50, 100, 500, 1000],
    ↪  k_fold_nb=5)
    # knn.cross_validate_n(n_range=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    ↪  k_fold_nb=5)
    knn.init_model(n=3)
    knn.train()
    knn.confusion_matrix()
    knn.print_report()
    knn.plot_roc_curve(display_plot=False)

    # (i)(v) MLP
    print("\n=== MLP CLASSIFIER ===")
    mlp = MyMLPClassifier(X, y)
    mlp.split(test_size=0.20)
    # mlp.cross_validate_n(prev_hidden_layers=(), n_range=[1, 5, 10, 25,
    ↪  50, 100, 500, 1000], max_iterations=10000)
    mlp.cross_validate_n(prev_hidden_layers=(50,), n_range=[1, 5, 10, 25,
    ↪  50, 100], max_iterations=10000)
    # mlp.cross_validate_n(prev_hidden_layers=(50, 5), n_range=[1, 5, 10,
    ↪  25, 50, 100], max_iterations=10000)
    mlp.init_model(n=(50,))
    mlp.train()
    # mlp.save_model("saved-models/MyMLPClassifier-50.joblib")
    # mlp.load_model()
    mlp.confusion_matrix()
    mlp.print_report()
    mlp.plot_roc_curve(display_plot=False)

    # Print all ROC Curves in a single figure
    plt.figure(123) # ROC Curves are added by each model on figure #123
    plt.title("ROC Curves of different classifiers")
    plt.show()


if part_ii:
    # (ii)(i) Preprocessing data
    print("> Preprocessing data")
    preprocessor = MyPreprocessor()
    preprocessor.read_data('dataset/reviews_17.jl.json')
    preprocessor.preprocess_data(predict="early_access", min_df=1,
    ↪  max_df=0.05)
    X, y = preprocessor.get_data()

    preprocessor.print_report()

```

```
    # (ii)(ii) Baseline model
    print("\n=== BASELINE MODEL ===")
    baseline = MyBaseline(X, y)
    baseline.split(test_size=0.20)
    baseline.init_model()
    baseline.train()
    # baseline.confusion_matrix()
    baseline.print_report()
    baseline.plot_roc_curve(display_plot=False)

    # (ii)(iii) Logistic regression
    print("\n=== LOGISTIC REGRESSION ===")
    logreg = MyLogisticRegression(X, y)
    logreg.split(test_size=0.20)
    # logreg.cross_validate_penalty(C_range=[0.1, 1, 10, 100, 1000, 5000], k_fold_nb=5)
    logreg.init_model(C=1000, max_iterations=10000)
    logreg.train()
    # logreg.confusion_matrix()
    logreg.print_report()
    logreg.plot_roc_curve(display_plot=False)

    # (ii)(iv) kNN
    print("\n=== K NEAREST NEIGHBORS CLASSIFIER ===")
    knn = MyKNeighborsClassifier(X, y)
    knn.split(test_size=0.20)
    # knn.cross_validate_n(n_range=[1, 5, 10, 50, 100, 500, 1000], k_fold_nb=5)
    # knn.cross_validate_n(n_range=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], k_fold_nb=5)
    knn.init_model(n=1)
    knn.train()
    # knn.confusion_matrix()
    knn.print_report()
    knn.plot_roc_curve(display_plot=False)

    # (ii)(v) MLP
    print("\n=== MLP CLASSIFIER ===")
    mlp = MyMLPClassifier(X, y)
    mlp.split(test_size=0.20)
    # mlp.cross_validate_n(prev_hidden_layers=(), n_range=[1, 5, 10, 25, 50, 100, 500, 1000], max_iterations=10000)
    mlp.init_model(n=(50,))
    mlp.train()
    # mlp.save_model("saved-models/MyMLPClassifier-ii-50.joblib")
    # mlp.load_model()
    # mlp.confusion_matrix()
    mlp.print_report()
```

```
    mlp.plot_roc_curve(display_plot=False)

    # Print all ROC Curves in a single figure
    plt.figure(123)  # ROC Curves are added by each model on figure #123
    plt.title("ROC Curves of different classifiers")
    plt.show()
```