Information Retrieval and Search Engines

Text classification with Rapidminer & Python

ONOUR IMPRACHIM

A.Abstract

B.Introduction

C. Document Classification with RapidMiner v10.3

1. Design Process of the Naive Bayes Classifier
2. Model Results with Different Vector Creation Methods
3. Comparison of Vector Creation Methods

D. Document Classification with Python Code

1. Introduction
2. Results of Bernoulli & Multinomial Naive Bayes Classifiers
3. Comparison between Bernoulli & Multinomial Naive Bayes Classifiers

E. Comparison between Python and RapidMiner

F. Conclusion

A) Abstract

This report examines text classification using the Naive Bayes algorithm in two different environments: RapidMiner and Python with the scikit-learn library. Both **Multinomial Naive Bayes** and **Bernoulli Naive Bayes** classifiers were tested for sentiment classification. The preprocessing steps included tokenization, stopword removal, case conversion, and percentual pruning. The models were evaluated with accuracy, precision, recall, F1-score, and execution time. The results show that the Bernoulli Naive Bayes classifier performed better overall, while Python provided faster execution compared to RapidMiner.

B) Introduction

Text classification is an important task in **Information Retrieval** and is widely used in applications such as sentiment analysis, spam filtering, and document organization. The **Naive Bayes algorithm** is often applied because it is simple, efficient, and works well with text data.

In this project, Naive Bayes was implemented and tested in two environments: **RapidMiner**, a graphical data mining tool, and **Python**, using the scikit-learn library. Two versions of Naive Bayes were used: **Multinomial Naive Bayes**, which considers the frequency of terms, and **Bernoulli Naive Bayes**, which uses binary term presence.

The purpose of this work is to compare the two classifiers and evaluate them in both environments using the same preprocessing methods and testing procedure. The evaluation focuses on accuracy, precision, recall, F1-score, and execution time, in order to show the advantages and disadvantages of each method.

C) Document Classification with RapidMiner v10.3

A.1. Design process of Naive Bayes

We install the Text Processing extension (from Extensions > Marketplace > Top Rated > Text Processing 10.0.0), which is required for text analysis.

a. **Data Retrieval (documents)**

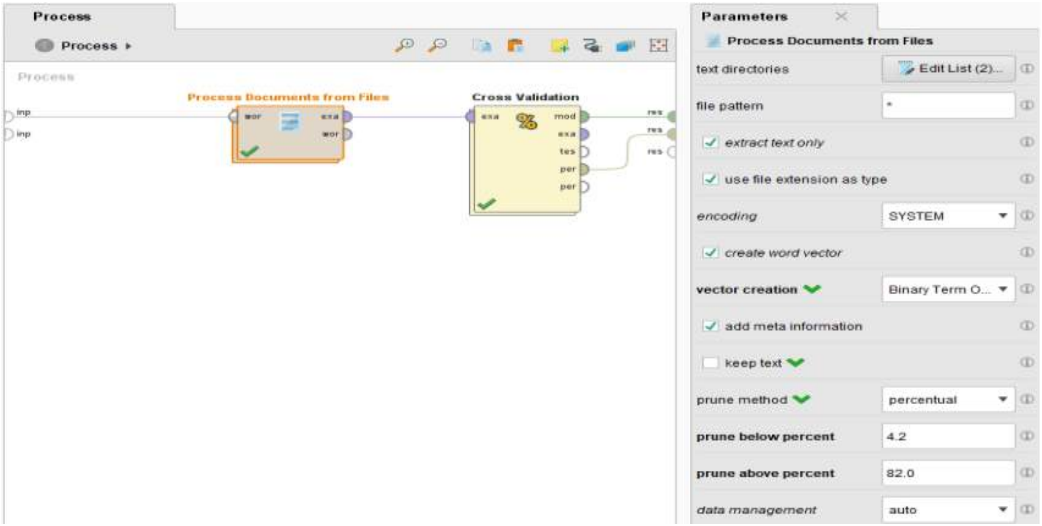
Add 'Process Documents from Files' in the Process (Design) tab and configure parameters:

i. **Edit list:** Add classes pos and neg with the corresponding directory folders containing the movie reviews as .txt.

ii. **Vector Creation:** Choose between binary term occurrences or term occurrences to transform documents into vectors.

iii. **Prune Method:** percentual with values below percent = 4 and above percent = 82, giving accuracy = 79.75%. These values reduce processing time by 20 seconds and improve accuracy by 10% compared to no pruning.

iv. **Enable parallel execution:** Enabled. Without the percentual prune method, the software crashed.



b. Data Preprocessing (documents)

The preprocessing steps applied to documents were:

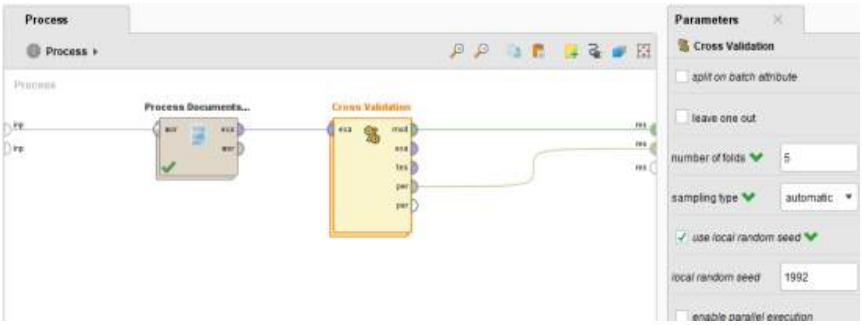
i. **Tokenize:** Splits text into words.
ii. **Transform Cases:** Converts text to lowercase so words like 'The'-'the' are treated as the same.
iii. **Filter Stopwords** (English): Removes common words (e.g., 'the', 'and') to reduce noise.
iv. **Filter Tokens** (by Length, min chars = 3): Removes tokens too short or too long as less useful.

The **Generate n-Grams (Word N-Grams)** operator was not added, although it reveals more word relations, because RapidMiner crashed with parallel execution. Without parallel execution, the process took 16 minutes with 76.50% accuracy. With percentual pruning, time dropped to 12 seconds with 79.90% accuracy. Finally, without n-Grams, only percentual prune, time dropped to 2 seconds with 79.75% accuracy. Thus, Generate n-Grams was not used as the difference was minimal.

After data retrieval and preprocessing, training was conducted using **5-fold cross validation** and classification with the **Naive Bayes Classifier** with **Laplace Correction** (to handle zero probabilities).

c. 5-Fold Cross Validation Training Method

The “Cross Validation” operator was added from the Operators panel in the Process design. The output example set (exa) from Process Documents from Files was connected to the input training set (tra) of the Cross Validation operator.

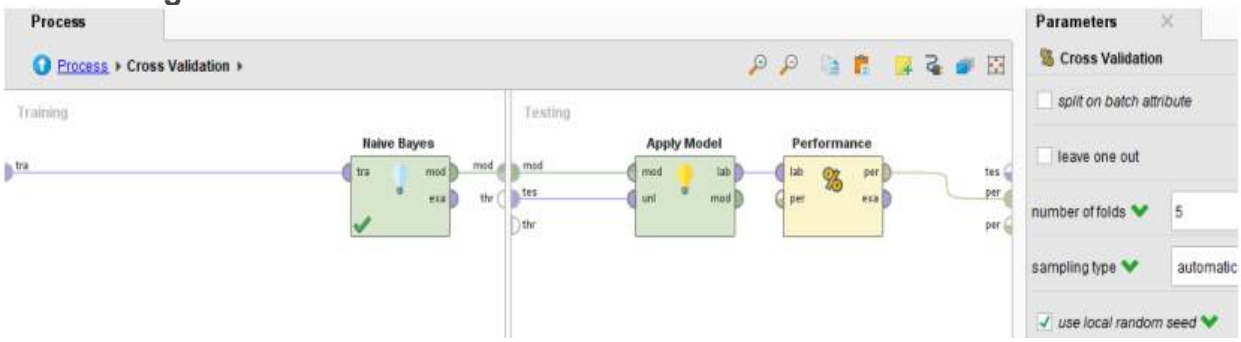


Parameter settings:

i. **Number of folds = 5**: The dataset was divided into five equal subsets (folds). Training was performed on four subsets, while the remaining subset was used as the test set.
ii. **Sampling type = “automatic”**: Since the dataset was balanced (equal number of positive and negative documents).
iii. **Use local random seed = enabled**: Ensures the same random seed is applied in each execution, resulting in identical sampling during every cross-validation run.

d. Classification with Naive Bayes Classifier

Within the Cross Validation operator, there are two stages: Training and Testing.



In the interface:

a. **Training**: Add the Naive Bayes classifier and enable in the Parameters tab.
b. **Testing**: Add the following operators:
   i. **Apply Model** – used to apply the trained model.
   ii. **Performance** – used to evaluate the performance of the model based on the predictions generated by .In the Parameters tab of , select the metrics: **accuracy**, **weighted mean recall**, and **weighted mean precision**.

Connections:

- The training data ( ) is passed to the Naive Bayes operator for model training, and the trained model ( ) is connected to the input of the Testing stage.
- The trained model ( ) is connected to the input of Apply Model, which generates predictions ( ). The test dataset ( ) is connected to the input of Apply Model.
- The prediction output ( ) is connected to the input of Performance, where predictions are compared, and evaluation results are produced in the output of the Training stage.

A.2. Model Results with Different Vector Creation Methods

a. Results with Vector Creation: Binary Term Occurrences

Accuracy: 79.75% ± 2.35% (micro average: 79.75%)
Confusion Matrix:

	True Neg	True Pos
Pred. Neg	833	238
Pred. Pos	167	762

i. 833 negative examples were correctly classified as negative (TN).
ii. 238 positive examples were incorrectly classified as negative (FN).
iii. 762 positive examples were correctly classified as positive (TP).
iv. 167 negative examples were incorrectly classified as positive (FP).

Weighted Mean Recall: 79.75% ± 2.35%
Weighted Mean Precision: 79.94% ± 2.44%

b. Results with Vector Creation: Term Occurrences

Accuracy: 73.40% ± 1.59% (micro average: 73.40%)
Confusion Matrix:

	True Neg	True Pos
Pred. Neg	820	352
Pred. Pos	180	648

i. 820 negative examples were correctly classified as negative (TN).
ii. 352 positive examples were incorrectly classified as negative (FN).
iii. 648 positive examples were correctly classified as positive (TP).
iv. 180 negative examples were incorrectly classified as positive (FP).

Weighted Mean Recall: 73.40% ± 1.59%
Weighted Mean Precision: 74.13% ± 1.53%

A.3. Comparison of Vector Creation Methods

- **Accuracy**: The model using the Binary Term Occurrences method demonstrated a 6.35% higher accuracy, indicating superior performance compared to the Term Occurrences method.
- **Precision**: The Binary Term Occurrences method also achieved approximately 5.81% higher precision, reflecting greater accuracy in classifying positive examples.
- **Recall**: Recall was likewise higher for the Binary Term Occurrences method, confirming its effectiveness in identifying positive examples.
- **Confusion Matrix**:
    - With the **Binary Term Occurrences** method, the model correctly classified more positive and negative examples.
    - With the **Term Occurrences method**, the model produced more errors, with 358 negative examples incorrectly classified as positive and 189 positive examples incorrectly classified as negative.
    - The Binary Term Occurrences method yielded fewer total errors, with 238 false negatives (FN) and 167 false positives (FP), compared to 352 FN and 180 FP under the Term Occurrences method.

D) Document Classification with Python Code

B.1.Introduction

The Python implementation employs the same preprocessing techniques as those used in RapidMiner. Text classification is carried out using the Naive Bayes algorithm in two distinct variants: **Multinomial Naive Bayes**, based on term occurrences, and **Bernoulli Naive Bayes**, based on binary term occurrences.

The code first loads the data from directories, where positive and negative texts are stored in separate folders. The dataset is then split into training and testing sets (50% each, consistent with the RapidMiner setup). Feature vectors are generated using the **CountVectorizer** algorithm, applying the following steps:

1. Removal of common words ( ).
2. Filtering tokens by length (minimum of 3 characters, as in the RapidMiner implementation).
3. Case normalization (conversion to lowercase).
4. Application of percentual pruning by setting minimum and maximum document frequency thresholds
   (, as in the RapidMiner setup).

Additionally, the code provides flexibility for adjusting these parameter values.

Both models are trained and tested on the dataset, and their performance is evaluated in terms of **accuracy**, **confusion matrix**, and **classification report** (precision, recall, F1-score). Furthermore, the execution time of each model is recorded, providing a comprehensive evaluation of the efficiency of each approach.

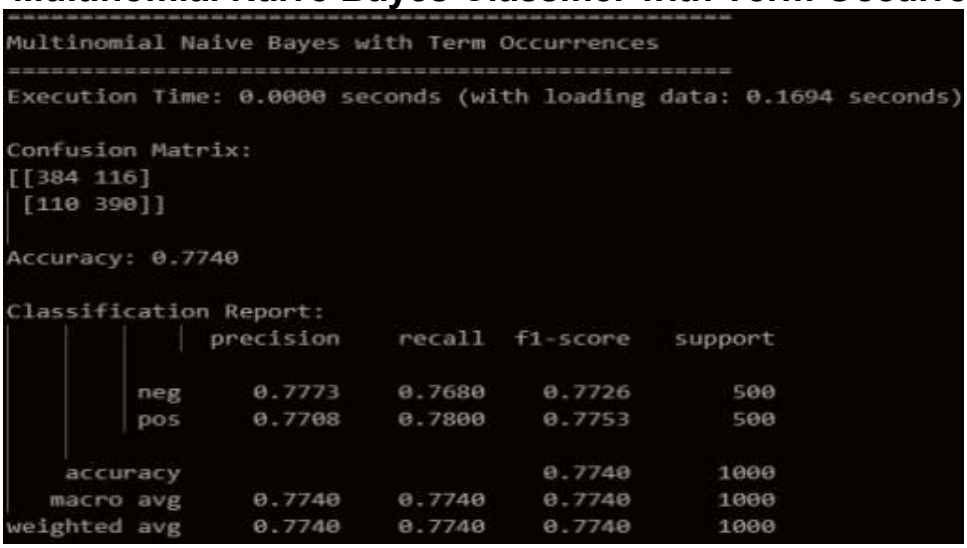
In summary, the process evaluates the models based on accuracy, confusion matrix, and classification report, while also documenting execution times, thereby offering a complete overview of the performance of each classification method.

For further details, refer to the **readme.md** file.

The results from the code implementation and the comparison between **Multinomial Naive Bayes** and **Bernoulli Naive Bayes** are presented below.

B.2. Results of Bernoulli & Multinomial Naive Bayes Classifiers

a. **Multinomial Naive Bayes Classifier with Term Occurrences**



```
==================================================
Multinomial Naive Bayes with Term Occurrences
==================================================
Execution Time: 0.0000 seconds (with loading data: 0.1694 seconds)

Confusion Matrix:
[[384 116]
 [110 390]]

Accuracy: 0.7740

Classification Report:
              precision    recall  f1-score   support

         neg     0.7773    0.7680    0.7726       500
         pos     0.7708    0.7800    0.7753       500

    accuracy                         0.7740      1000
   macro avg     0.7740    0.7740    0.7740      1000
weighted avg     0.7740    0.7740    0.7740      1000
```

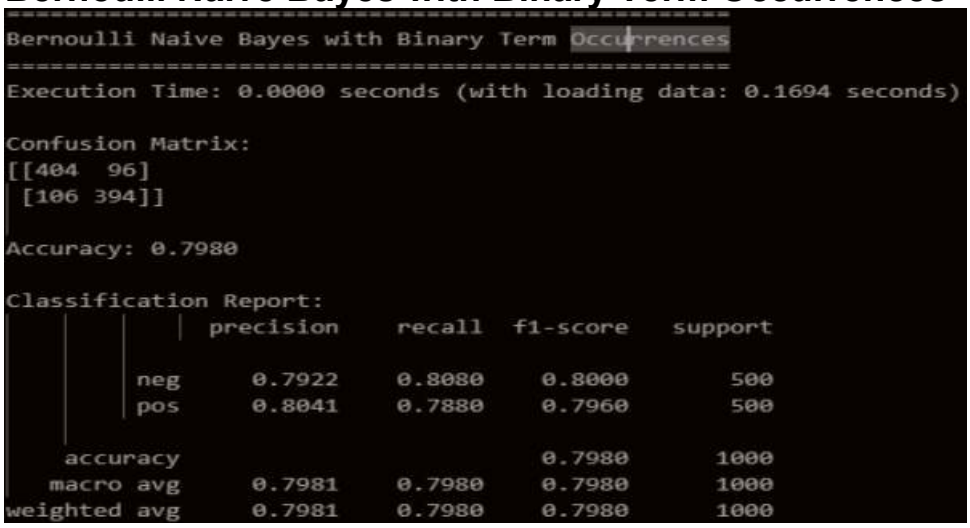
Accuracy: 77.40% , **Confusion Matrix**:

	True Neg	True Pos
Pred. Neg (neg)	384	110
Pred. Pos (pos)	116	390

i) 384 negative examples were correctly classified as negative (TN).
ii) 110 positive examples were incorrectly classified as negative (FN).
iii) 390 positive examples were correctly classified as positive (TP).
iv) 384 negative examples were incorrectly classified as positive (FP).

Recall: Negative (neg): 77.73% , Positive (pos): 77.08%
Precision: Negative (neg): 76.80%, Positive (pos): 78.00%
F1-Score: Negative (neg): 77.26%, Positive (pos): 77.53%

b. **Bernoulli Naive Bayes with Binary Term Occurrences**



```
==================================================
Bernoulli Naive Bayes with Binary Term Occurrences
==================================================
Execution Time: 0.0000 seconds (with loading data: 0.1694 seconds)

Confusion Matrix:
[[404  96]
 [106 394]]

Accuracy: 0.7980

Classification Report:
              precision    recall  f1-score   support

         neg     0.7922    0.8080    0.8000       500
         pos     0.8041    0.7880    0.7960       500

    accuracy                         0.7980      1000
   macro avg     0.7981    0.7980    0.7980      1000
weighted avg     0.7981    0.7980    0.7980      1000
```

Accuracy: 79.80%, **Confusion Matrix**:

	True Neg	True Pos
Pred. Neg (neg)	404	106
Pred. Pos (pos)	96	394

i) 404 negative examples were correctly classified as negative (TN).
ii) 106 positive examples were incorrectly classified as negative (FN).
iii) 394 positive examples were correctly classified as positive (TP).
iv) 96 negative examples were incorrectly classified as positive (FP).
Recall:Negative (neg): 80.80%, Positive (pos): 78.80%
Precision:Negative (neg): 79.22%,Positive (pos): 80.41%
F1-Score:Negative (neg): 80.00%, Positive (pos): 79.60%

B.3. Comparison between Bernoulli & Multinomial Naive Bayes Classifiers

The Bernoulli Naive Bayes Classifier outperforms the Multinomial Naive Bayes Classifier in nearly all categories:

1. **Accuracy**: Bernoulli Naive Bayes achieves 2.40% higher accuracy (79.80%), indicating fewer misclassifications compared to Multinomial Naive Bayes (77.40%).

2. **Confusion Matrix**: Bernoulli Naive Bayes yields 5.21% more True Negatives (404 vs. 384) and 17.24% fewer False Positives (96 vs. 116), demonstrating superior recognition of negative examples. Additionally, it produces 0.77% more True Positives (394 vs. 390), making it slightly more accurate in positive predictions.

3. **Precision**: Bernoulli Naive Bayes achieves 1.49% higher precision for the negative class (79.22% vs. 77.73%) and 3.33% higher precision for the positive class (80.41% vs. 77.08%). This indicates that Bernoulli makes fewer errors in positive predictions.

4. **Recall**: Bernoulli Naive Bayes exhibits 4.00% higher recall for the negative class (80.80% vs. 76.80% for Multinomial). Conversely, Multinomial Naive Bayes shows slightly better recall (by 0.80%) for the positive class (78.00% vs. 78.80% for Bernoulli).
5. **F1-Score**: Bernoulli Naive Bayes achieves a 2.74% higher F1-score for the negative class (80.00% vs. 77.26%) and a 2.07% higher F1-score for the positive class (79.60% vs. 77.53%). This highlights a better balance between precision and recall.

	TN	FP	FN	TP	Accuracy	Precision	Recal
Multinomial Naive Bayes	384	116	110	390	77.40%	77.73%	76.80
Bernoulli Naive Bayes	404	96	106	394	79.80%	79.22%	80.80

Conclusion: The Bernoulli Naive Bayes Classifier demonstrates overall superior performance compared to the Multinomial Naive Bayes Classifier.

E) Comparison between Python and RapidMiner

The comparative evaluation highlights several differences between the two frameworks:

- **Bernoulli Naive Bayes with Binary Term Occurrences:**
    - Accuracy reached **79.80% in Python** and **79.75% in RapidMiner**, with Python slightly outperforming RapidMiner.
    - In terms of **precision, recall, and F1-score**, Python demonstrated consistently higher values.
    - Both implementations achieved balanced classification of positive and negative examples.
- **Multinomial Naive Bayes with Term Occurrences:**
    - Accuracy was **77.40% in Python** compared to **73.40% in RapidMiner**.
    - This model exhibited weaker performance overall, particularly in the classification of positive examples, where more errors were observed.

In conclusion, the **Bernoulli Naive Bayes classifier** demonstrates greater efficiency and reliability across both platforms. Importantly, Python exhibits a significant **computational advantage**, executing both Naive Bayes models in approximately **2 seconds**, compared to RapidMiner, which required **4 seconds** for the same tasks. This confirms Python as the more scalable and time-efficient environment for text classification experiments.

F) Conclusion

In this work, text classification was carried out using Naive Bayes in both RapidMiner and Python. The results showed that the **Bernoulli Naive Bayes classifier** gave better overall performance than the **Multinomial Naive Bayes**, with higher accuracy, precision, recall, and F1-score.

The comparison between the two environments also highlighted that Python, with the scikit-learn library, is more efficient than RapidMiner, since it produces similar or slightly better results but in less execution time.

Overall, the study confirms that Bernoulli Naive Bayes is a more effective choice for sentiment classification tasks, and that Python is a faster and more practical environment for such experiments.