Natural Language Processing and Machine Learning for Stylometry
Problem (9)

Mohamed Shawky Zaky AbdelAal Sabae
Section:2,BN:15
mohamed.sabae99@eng-st.cu.edu.eg

***Abstract*— Stylometry is the application of the study of linguistic style, usually to written language. In this work, we discuss the proper methods of using *natural language processing* and *machine learning* for binary classification of authors' writings.**

I. INTRODUCTION

The report contains the discussion of stylometry problem. We discuss the means of dataset collection, feature extraction and language modeling. Moreover, we mention the advantage of the submitted solutions over the other possible approaches. Finally, we review the code structure and usage. The provided code mainly contains two solutions. **First,** *Naive Bayes* classifier with *n-grams* features. **Second,** a simple *neural network* with *word embeddings*.

II. APPROACH

This section contains the discussion of *dataset collection*, *feature extraction* and *language modeling*.

A. Dataset Collection

Dataset is collected from *Kaggle's* **Spooky author identification problem**, as it contains authors of the same era and genre. Other authors are not considered mainly because no available data for multiple authors of the same era. The dataset contains three authors :

- **Edgar Allan Poe** *(EAP)* : 7900 phrases.
- **HP Lovecraft** *(HPL)* : 5635 phrases.
- **Mary Wollstonecraft Shelley** *(MWS)* : 6044 phrases.

B. Feature Extraction

Before feature extraction stage, some text processing is performed on the input phrases. Basically, **tokenization**, **stemming** and **lemmatization** are performed. **stemming** performs better than **lemmatization** in our case.
For feature extraction, five methods are considered :

- **Bag of Words** *(BoW)* : yields the worst performance.
- **n-grams** : offers moderate to decent performance based on the classifier.
- **TF-IDF vectorization** : *n-grams + term frequency - inverse document frequency*, offers decent performance with most classifiers.
- **PCA / Truncated SVD on previous features** : using *principal component analysis* or *truncated singular value decomposition* on our features does not seem to perform well.
- **Word Embeddings** : basically using *GloVe* pretrained embeddings for *neural networks* training.

C. Language Modeling

The submitted solution only offers two methods *Naive Bayes* classifier as a classical language model and a *deep neural network* as a deep learning language model.

- **Classical language modeling : Naive Bayes** is used with *n-grams (count vectorization)* features. This is basically because it yields better results than all the other considered classical approaches. The other classical approaches, considered in this work, are **support vector machines** *(SVM)*, **logistic regression** and **gradient boosting**.
- **Deep learning-based language modeling :** a simple *neural network* is used with *word embeddings*, in order to test the ability of neural networks on our dataset. The network consists of one **bidirectional GRU** layer followed by two **fully-connected** layers with **dropouts**.

III. CODE STRUCTURE

A. Code Files

The submission contains 5 main code files :

- **text_dataset.py :** contains the dataset class that contains the code for building and processing the text dataset. *NLTK* and *sklearn* libraries are used in the implementation.
- **model.py :** contains one class for *Naive Bayes* as linear classifier (*sklearn MultinomialBN* is used) and one class for *the simple neural network* (*PyTorch* is used for network implementation).
- **train.py :** contains the main code for training both *Naive Bayes* and *neural network* and uses both *text_dataset.py* and *model.py*.
- **evaluate.py :** contains the main code for evaluating both *Naive Bayes* and *neural network* and uses both *text_dataset.py* and *model.py*.
- **naive_bayes.py :** contains a custom implementation of *Naive Bayes* classifier. However, it's not used in the main code, because it yields lower performance than *sklearn MultinomialBN*.

B. Dataset

The folder named *data* has 6 files for train and test data :

- **EAP_train.txt :** *Edgar Allan Poe* train data.

- **EAP_test.txt :** *Edgar Allan Poe* test data.
- **HPL_train.txt :** *HP Lovecraft* train data.
- **HPL_test.txt :** *HP Lovecraft* test data.
- **MWS_train.txt :** *Mary Wollstonecraft Shelley* train data.
- **MWS_test.txt :** *Mary Wollstonecraft Shelley* test data.

C. Miscellaneous

- **config** folder : contains *JSON* config files.
- **models** folder : contains neural network model file *(deep_model.pt)* and linear classifier model file *(nb_model.sav)*.
- **report** folder : contains submission report source and *PDF*.
- **README.md** : contains code installation and usage.
- **requirements.txt** : contains required dependencies.
- **w2v_models** folder : *[REQUIRED]* needs to be created *(follow code usage details)*.

IV. CODE USAGE

The details, mentioned in this section, are also mentioned in *README.md*

A. Dependencies

The following *python* packages and libraries are required for code usage :

- *pytorch*
- *sklearn*
- *nltk*
- *numpy*
- *tqdm*
- *argparse*
- *pickle*

B. Usage

Follow the instructions to run code functionalities :

- **For training linear classifier model :**
  - Create *models* folder.
  - Edit *configs/lc_config.json*.
  - Run *train.py* :
    - *python train.py train-lc*
- **For training neural network model :**
  - Create *models* and *w2v_models* folders.
  - Download *GloVe* embeddings into *w2v_models* folder.
  - Edit *configs/nn_config.json*.
  - Run *train.py* :
    - *python train.py train-nn*
- **For inference on linear classifier model :**
  - *python evaluate.py eval-lc –author1 /path/to/author1/text –author2 /path/to/author2/text –model /path/to/model/file*
- **For inference on neural network model :**
  - *python evaluate.py eval-nn –author1 /path/to/author1/text –author2 /path/to/author2/text –model /path/to/model/file –w2v_path /path/to/w2v/model*

V. EXPERIMENTAL RESULTS

This section mainly addresses *points* 3 and 4 in the requirements, by discussing which **language model** is better and the results of altering **features**.

A. Experimental Setup

The problem experiments are conducted in the following order :

- The **codebase** is developed, in order to try **all possible combinations** of features and classifiers.
- The **language models** are evaluated based on *accuracy score* and *binary logloss* using **only two of the three** authors.
- **Scores** are recorded and the **best combination** from both linear classifiers and neural networks are obtained.
- **Further experimentation** is conducted on the two solutions to see the effect of altering features and model parameters, as well as using different pairs of authors.

The dataset is divided into 80% **train**, 10% **validation** and 10% **test**. Also, the initial experiments are conducted using *HP Lovecraft* and *Mary Wollstonecraft Shelley* phrases.

B. Classifier Analysis (Language Model)

Model	TF-IDF	n-grams	embeddings
Logistic Regression	91.9%	88.97%	—
Support Vector Machines	92.5%	83.26%	—
Naive Bayes	92.5%	93.15%	—
Gradient Boosting	79.84%	81.17%	—
Neural Networks	—	—	92.7%

TABLE I: Accuracy Scores of different classifiers and features

Model	TF-IDF	n-grams	embeddings
Logistic Regression	2.8	3.9	—
Support Vector Machines	2.56	5.77	—
Naive Bayes	2.59	2.36	—
Gradient Boosting	6.96	6.5	—
Neural Networks	—	—	2.52

TABLE II: Binary Logloss of different classifiers and features

1) Linear Classifiers: The previous tables I and II show the results of using multiple linear classifiers with different features. It's obvious that **Naive Bayes** with **n-grams** features is the best combination resulting in 93.15% accuracy. On the other hand, **gradient boosting** results in very poor performance compared to other classifiers. Tuning the parameters of these models doesn't result in any significant performance improvement. Moreover, trying to reproduce the results of *sklearn* and *xgboost* libraries implementation results in **performance reduction**. For example, re-implementing **Naive Bayes** highly reduces its accuracy, however the implementation is **submitted**, as well as the usage of *sklearn*.

2) neural networks: The final network, used in this work, consists of *one* **bidirectional GRU** and *two* **fully-connected** layers with **dropouts**. The network is used with *GloVe* word embeddings, resulting in 92.7% accuracy. The model's performance is worse than *Naive Bayes* on this pair of authors, however it is submitted as a *Deep Learning* baseline. Due to the *dataset structure*, the network suffers from *overfitting*, which is reduced by *reducing the number of network layers* and *using dropouts between fully-connected layers*.

C. Features Analysis

As mentioned before, we considered *n-grams* and *TF-IDF* as classical features and *GloVe* word embedding model as deep learning features. So, let's discuss three main points regarding the used features :

- *TF-IDF* vectorization performs better than simple count *n-grams* for most classifiers, however simple count *n-grams* yield the best performance with *Naive Bayes* classifier, that's why it's chosen. *GloVe* word embedding model is specified for the neural network, though.
- Using *1-grams (single words)* yields very poor results. However, excessively increasing the number of *n-grams* results in overfitting. Consequently, using up to *2-grams* or *3-grams* results in the best performance *(around 93.2% accuracy)*.
- **Principal components analysis** *(PCA)* is considered to extract important components from features. However, since the features are *huge* and *sparse*, it's better to use **singular value decomposition** *(SVD)* truncation, which is the same as *PCA* but on non-zero centred data. Unfortunately, as the number of features components decrease, the accuracy decreases as well.

VI. Results and Conclusion

Authors	Naive Bayes	Neural Network
HPL / MWS	93.15%	92.7%
EAP / HPL	90%	91.6%
EAP / MWS	87.8%	92%

TABLE III: Accuracy of Naive Bayes and Neural Network classifier on all authors pairs

Table III shows the results of the two chosen models on validation/test data of different authors pairs. It's obvious that different pairs result in different accuracy. This is basically because the some authors might have **close style** to each other. We can, also, see that the two models are close to each other, however one model can be better than the other based on the authors pair. *Neural networks* don't perform much better than linear classifiers for this dataset. This can be due to the following reasons :

- Not all the vocabulary set of the provided phrases exists in the *GloVe* word embedding model, which can significantly affect the performance.
- The model overfits rapidly on the training data.
- The number of training samples isn't enough to train the network properly.
- Advanced *attention* mechanisms can be used for improving performance.

References

[1] Machine Learning Methods for Stylometry Book.

[2] *Kaggle's* **Spooky author identification problem** : https://www.kaggle.com/c/spooky-author-identification

[3] *GloVe* word embedding model : https://nlp.stanford.edu/projects/glove/