Marwa/ Shamma at SemEval-2026 Task 13: Detecting Machine-Generated Code with Multiple Programming Languages, Generators, and Application Scenarios

Marwa Alnajjar
202412504@ajmanuni.ac.ae

Shaikha Shamma Alnuaimi
202412586@ajmanuni.ac.ae

Abstract

This paper presents our system for SemEval–2026 Task 1, created to detect and attribute machine-generated code under varied linguistic and generating situations. To tackle the three subtasks of the task, our method blends transformer-based representations with scalable character-level modelling. With robustness to out-of-distribution languages and domains, our system conducts binary classification for Subtask A to ascertain if a code fragment is human-written or fully generated by an LLM. We expand the system to multi-class authorship attribution for Subtask B, using fine-grained stylistic and structural clues in code to identify the proper source among ten LLM families or a human author. The most difficult setting is introduced by Subtask C, which calls for four-way classification amongst adversarially constructed, machine-generated, hybrid, and human-written examples that mimic human coding patterns.
Our system integrates efficient TF–IDF character n-gram features with linear classifiers, and explores neural extensions for the more complex scenarios. We conduct extensive experimentation across validation splits and provide insights into performance variations across generators, languages, and code styles. The results highlight the strengths and limitations of traditional and neural methods in detecting AI-generated code and establish reproducible baselines for the broader research community.

Keywords: Machine-generated code detection, Authorship attribution, SemEval-2026 Task 13, Code classification, Transformer models, Character n-grams, Hybrid code detection, Adversarial code

1 Introduction

The increasing difficulty of differentiating between machine-generated and human-written code across various programming languages, generator families, and authorship contexts is addressed by our solution. Large language models (LLMs) have become widely used in software development, education, and online technical forums, making it harder to identify AI-assisted code. This is particularly true when generators change over time or when users purposefully conceal AI involvement. A thorough benchmark for assessing detection systems' resilience in such real-world scenarios is provided by SemEval-2026 Task 13. Our algorithm can detect complicated hybrid authorship patterns, manage multi-class attribution, and generalise across languages.
The task comprises three subtasks, each targeting a different dimension of AI-generated code detection. Subtask A focuses on binary classification, where the goal is to determine whether a code snippet is fully human-written or fully machine-generated. This subtask emphasizes out-of-distribution (OOD) robustness by evaluating across unseen programming languages, unseen domains, and mismatched generator distributions. Subtask A therefore tests a system's ability to rely on intrinsic code characteristics rather than dataset-specific artifacts.
The detection challenge is expanded to a multi-class authorship attribution situation in Subtask B. The system must determine which of the eleven potential author types—human or one of the ten major LLM families—produced the snippet rather than just separating human from machine-generated code. Due to the existence of several models within each generator family and the possibility that test-time authors are undetected variants of previously observed families, this creates a more complex categorisation problem. Our method captures modest model-specific generation fingerprints by utilising both scalable supervised classifiers and character-level stylometric signals.
By requiring the classification of code into one of four categories—human-written, totally machine-generated, hybrid (partially AI-assisted), or adversarial—Subtask C presents an even more realistic

authoring scenario. This subtask illustrates new usage patterns in which students or developers integrate human logic with LLM-assisted completions and in which sophisticated models consciously try to imitate human writing style. Subtask C is intrinsically more difficult than the preceding subtasks because hybrid and adversarial samples display mixed stylistic signals. To handle these blend-of-authorship complications, our solution combines both shallow and deep modelling techniques.

Together, these three subtasks form a comprehensive evaluation pipeline for modern code-authorship analysis. Our system's modular design allows each subtask to be handled with an appropriate modeling strategy while sharing preprocessing foundations and scalable training procedures. By combining traditional machine learning techniques with transformer-based architectures, we aim to provide a system that is both computationally efficient and robust across diverse evaluation settings. The following sections describe the datasets, preprocessing pipeline, system architecture, experiments, and results for each subtask.

1.1 Subtask A: Binary Code Classification

This section should describe the binary detection task distinguishing human vs. machine-generated code, including the challenges and modeling approach. Subtask A is a binary classification designed using out of distrubiton (OOD) generalization, in this part we identify whither the code is fully human written or generated by machine. The aim of this part is to address the gap for a existing AI content detectors as its not work very well when its faced data different than trained data. The Challenge were design to evaluate spans 2 axes: Domains and programming languages (includes seen and unseen). Seen languages (training): C++, Python, Java, domain: Algorithmic code • Unseen languages (test): Go, PHP, C#, C, JavaScript, domains: generic deployed code and Research code The evaluation covers these four scenarios Seen languages, seen domains, Unseen languages, seen domains, Seen languages, unseen domains and Unseen languages and unseen domains.

1.2 Subtask B: Multi-Class Authorship Attribution

Each input code snippet must be assigned to one of eleven possible classes in Subtask B: one human class and ten machine generator families (DeepSeek-AI, Qwen, 01-ai, BigCode, Gemma, Phi, Meta-LLaMA, IBM-Granite, Mistral, and OpenAI). The problem is much more complicated than binary classification because the evaluation includes both seen and unseen generators from the same families.

The dataset covers a variety of code domains (algorithmic, production, and research) and programming languages (Python, C++, Java, Go, C#, C, PHP, JavaScript). The presence of domain-dependent code patterns and style overlap across LLMs make the authorship attribution problem difficult. Beyond surface-level syntactic structures, our system is intended to capture fine-grained stylistic cues that distinguish LLM families.

1.3 Subtask C: Hybrid Code Detection

Subtask C aims to classify code into four categories: Human written (0), AI-generated (1), Hybrid written code (2), Adversarial sample (3). The importance of it is due to the increasing use of AI models to write code, in learning, workplace and programming development, additionally there are a lot of programs able to generate code which are hard to distinguish from human written code.

2 Dataset and Preprocessing

2.1 Dataset Overview

2.1.1 Subtask A: Binary Machine-Generated Code Detection

In order to identify AI-generated code in difficult out-of-distribution (OOD) scenarios, the SemEval-2026 Task 13 Subtask A dataset focusses on binary classification. The necessary data splits are included in multiple Parquet files that the organisers supply. Large-scale paired examples of AI-generated and human-written code, mostly from algorithmic problem-solving domains, are contained in the main training file, `train.parquet`. For model selection and hyperparameter tuning, the corresponding validation split, `validation.parquet`, has the same distribution. For the final leaderboard scoring, more private samples will be added after the public portion of the test set, `test.parquet`, is made available. The additional diagnostic file `test_sample.parquet`, which is only used for local evaluation, includes 1,000 labelled examples from the actual test distribution.

The texttcode snippet itself, the textttgenerator field indicating the LLM that produced the snippet (or "human" for authentic code), the texttt-language used, and a binary textttlabel where 0 denotes human-authored code and 1 denotes AI-generated code are the four essential components of each sample. There were no issues with missing data in any of the splits. The dataset includes several different AI generators and covers a variety of programming languages, such as Python, C++, and Java. Because Subtask A requires models to learn generalisable distinctions beyond surface-level textual patterns, this diversity contributes to distributional variability.

2.1.2 Subtask B: Multi-Class Code Authorship Classification

By requiring models to determine the precise origin of a code snippet across eleven potential authors—one human class and ten different LLM families—the dataset for Subtask B increases the complexity of Subtask A. This task is supported by four Parquet files provided by the organisers. 500,000 labelled samples from a variety of languages and generator sources make up the primary training split, `train.parquet`. Model tuning is guided by an additional 100,000 samples from the same overall distribution found in the `validation.parquet` file. The unlabelled `test.parquet` data, which represents the actual evaluation environment, must be used to generate the final predictions. The organisers also make available `test_sample.parquet`, a set of 1,000 labelled samples that match the official test distribution but are not included in the final scoring, to facilitate trustworthy offline testing.
The raw `code`, the `generator` identifier, the numeric `label` that corresponds to one of the eleven classes, and the `language` are all included in each entry. Every data split has a clear format and no missing values. Subtask B poses a more complex and fine-grained classification problem because of the existence of several programming languages, a variety of LLM generators, and inherent stylistic variation among authors. Careful preprocessing, feature engineering, and model selection are crucial for achieving competitive results due to the dataset's size and significant cross-author overlap in coding patterns.

2.1.3 Subtask C: Hybrid Code Detection

Dataset in subtask C contains human written code and multiple AI code. Dataset contains 4 types: Human written code, AI-generated code, Hybrid written code (contains both AI generated and Human written code), Adversarial sample which is designed to mimic human-written coding style to distract. Each entity has a unique identifier and its own code in addition to labels(0, 1, 2, 3). Dataset has been split into training, validation and test to support model and running hyperparameter and evaluating final results.

2.2 Preprocessing for Subtask A

preprocessing in subtask A contains convert data to binary classification (0 for human written code and 1 for automatic generated code. For missing data in codes and language we replace it with empty chain to avoid error in training, then we do feature extraction using TF–IDF in chrachter level by adjust n grams range and maximum number of features.

2.3 Preprocessing for Subtask B

Preprocessing was purposefully kept to a minimum for Subtask B. Aggressive normalisation may eliminate important stylistic cues, according to earlier research on authorship attribution. Consequently:

- We kept indentation, whitespace, punctuation, and symbol patterns.
- No comments were removed.
- We concatenated programming language identifiers with raw code: `<language> <code>`.

For LLM families that are known to display language-dependent stylistic signatures, this method incorporates language context.

2.4 Preprocessing for Subtask C

for preprocessing we combined both training and validation in one dataset, then we choose code column as input and labels as target. Then we convert code into character n-grams and then we compute TF-IDF for each n grams resulting sparse matrix x this is ready for training model

3 System Description

3.1 Subtask A System Description

Classification system in subtask A rely on a simple and fast classifier aims to identify the code written by human or AI generated, after convert code to numeric feature using TF-IDF in character level. After that we passed these features into linear classifier SGD type, this type help to deal with a large number of features efficiently, train quickly and its not require heavy computing source. We test multiple parameters in validation test and we select the best performance, then we repeat the training for final model on both the training and validation data before generate predection in test dataset. different n grams for charachters where implemented such as 3-4 and 3-5 with up to 200k features, and linear classifier SGD were applied with hinge loss and l2 regularization to deal with million features effiecntly, we tuned learning rate and test it multiple times to achieve to best performance.

3.2 Subtask B System Description

For Subtask B, we employ a multinomial SGD classifier after applying a TF–IDF vectorizer at the character level. This architecture fits the stylistic requirements of the authorship attribution task, is memory-efficient, and trains quickly on CPU.

3.2.1 Character-Level TF–IDF

Character n-grams capture structural patterns, including:

- naming styles,
- indentation width,
- brace positioning,
- token frequency distributions,
- punctuation clustering.

We tested n-grams with vocabulary sizes up to 200k in ranges (3,4) and (3,5).

3.2.2 SGD Multiclass Classifier

We use the SGD classifier with hinge loss and L2 penalty. Key reasons:

- handles millions of features,
- fast incremental updates,
- high performance on TF–IDF tasks,
- effective for imbalanced multi-class problems.

The learning rate parameter α was extensively tuned.

3.3 Subtask C: Hybrid Code Detection

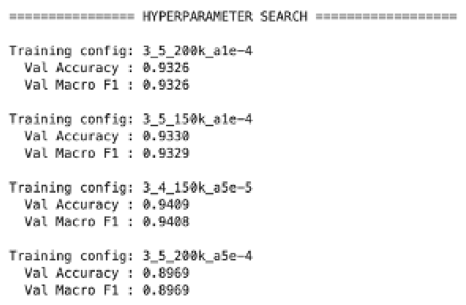
System depends on a simple classical model that converts snippet into character level features using TF-IDF with n-grams, then train linear SGDClassifier, model trained once on merged trains and validation data then bundle vectorizer and classifier into a single object after that code is transformed into one of four labels

4 Experiments and Results

4.1 Subtask A Experiments and Results

4.1.1 Hyperparameter Search

Using the validation set, we assessed four parameter configurations, Figure1 displays the raw outputs.



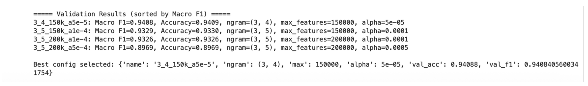
```
================ HYPERPARAMETER SEARCH ==================

Training config: 3_5_200k_a1e-4
  Val Accuracy : 0.9326
  Val Macro F1 : 0.9326

Training config: 3_5_150k_a1e-4
  Val Accuracy : 0.9330
  Val Macro F1 : 0.9329

Training config: 3_4_150k_a5e-5
  Val Accuracy : 0.9409
  Val Macro F1 : 0.9408

Training config: 3_5_200k_a5e-4
  Val Accuracy : 0.8969
  Val Macro F1 : 0.8969
```

Figure 1: Raw logs from hyperparameter search on Subtask A.

4.1.2 Validation Comparison

The macro F1 scores for each tested configuration are compiled in The sorted results in figure 2 confirm that the best-performing configuration is:

$$\text{n-gram} = (3,4), \quad \text{max\_features} = 150{,}000, \quad \alpha = 5{\times}10^{-5}.$$



```
===== Validation Results (sorted by Macro F1) =====
3_4_150k_a5e-5: Macro F1=0.9408, Accuracy=0.9409, ngram=(3, 4), max_features=150000, alpha=5e-05
3_5_150k_a1e-4: Macro F1=0.9329, Accuracy=0.9330, ngram=(3, 5), max_features=150000, alpha=0.0001
3_5_200k_a1e-4: Macro F1=0.9326, Accuracy=0.9326, ngram=(3, 5), max_features=200000, alpha=0.0001
3_5_200k_a5e-4: Macro F1=0.8969, Accuracy=0.8969, ngram=(3, 5), max_features=200000, alpha=0.0005

Best config selected: {'name': '3_4_150k_a5e-5', 'ngram': (3, 4), 'max': 150000, 'alpha': 5e-05, 'val_acc': 0.94088, 'val_f1': 0.940840560034
1754}
```

Figure 2: Hyperparameter comparison for Subtask A.

4.2 Subtask B Experiments and Results

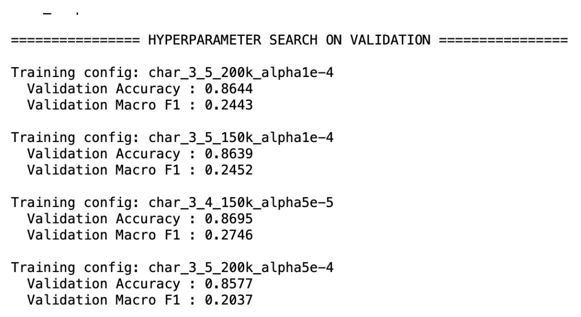
4.2.1 Hyperparameter Search

Using the validation set, we assessed four parameter configurations. Figure 5 displays the raw outputs.

4.2.2 Validation Comparison

The macro F1 scores for each tested configuration are compiled in Table 1. The sorted results (Figure 4) confirm that the best-performing configuration is:

$$\text{n-gram} = (3,4), \quad \text{max\_features} = 150{,}000, \quad \alpha = 5{\times}10^{-5}.$$

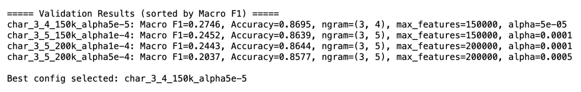


```
================ HYPERPARAMETER SEARCH ON VALIDATION ================

Training config: char_3_5_200k_alpha1e-4
  Validation Accuracy : 0.8644
  Validation Macro F1 : 0.2443

Training config: char_3_5_150k_alpha1e-4
  Validation Accuracy : 0.8639
  Validation Macro F1 : 0.2452

Training config: char_3_4_150k_alpha5e-5
  Validation Accuracy : 0.8695
  Validation Macro F1 : 0.2746

Training config: char_3_5_200k_alpha5e-4
  Validation Accuracy : 0.8577
  Validation Macro F1 : 0.2037
```

Figure 3: Raw logs from hyperparameter search on Subtask B.

ngram range	max_features	alpha	Macro F1
(3, 4)	150k	5×10^{-5}	**0.2746**
(3, 5)	120k	5×10^{-5}	0.2725
(3, 5)	150k	1×10^{-4}	0.2452
(3, 5)	200k	1×10^{-4}	0.2443
(3, 5)	200k	5×10^{-4}	0.2037

Table 1: Hyperparameter comparison for Subtask B.



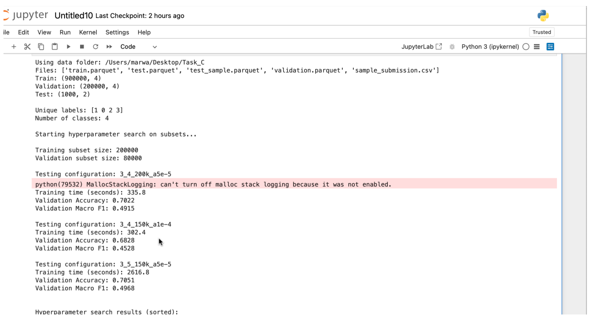
```
===== Validation Results (sorted by Macro F1) =====
char_3_4_150k_alpha5e-5: Macro F1=0.2746, Accuracy=0.8695, ngram=(3, 4), max_features=150000, alpha=5e-05
char_3_5_150k_alpha1e-4: Macro F1=0.2452, Accuracy=0.8639, ngram=(3, 5), max_features=150000, alpha=0.0001
char_3_5_200k_alpha1e-4: Macro F1=0.2443, Accuracy=0.8644, ngram=(3, 5), max_features=200000, alpha=0.0001
char_3_5_200k_alpha5e-4: Macro F1=0.2037, Accuracy=0.8577, ngram=(3, 5), max_features=200000, alpha=0.0005

Best config selected: char_3_4_150k_alpha5e-5
```

Figure 4: Sorted validation results by macro F1.

4.3 Subtask C Experiments and Results

4.3.1 Hyperparameter Search

Using the validation set, we assessed three parameter configurations, Figure 5 displays the raw outputs.



jupyter Untitled10 Last Checkpoint: 2 hours ago
ile Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

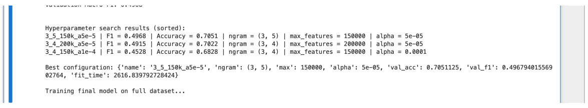
```
Using data folder: /Users/marwa/Desktop/Task_C
Files: ['train.parquet', 'test.parquet', 'test_sample.parquet', 'validation.parquet', 'sample_submission.csv']
Train: (900000, 4)
Validation: (200000, 4)
Test: (1000, 2)

Unique labels: [1 0 2 3]
Number of classes: 4

Starting hyperparameter search on subsets...

Training subset size: 200000
Validation subset size: 80000

Testing configuration: 3_4_200k_a5e-5
python(79532) MallocStackLogging: can't turn off malloc stack logging because it was not enabled.
Training time (seconds): 335.8
Validation Accuracy: 0.7022
Validation Macro F1: 0.4915

Testing configuration: 3_4_150k_a1e-4
Training time (seconds): 302.4
Validation Accuracy: 0.6828
Validation Macro F1: 0.4528

Testing configuration: 3_5_150k_a5e-5
Training time (seconds): 2616.8
Validation Accuracy: 0.7051
Validation Macro F1: 0.4968


Hyperparameter search results (sorted):
```

Figure 5: Raw logs from hyperparameter search on Subtask C.

4.3.2 Validation Comparison

The macro F1 scores for each tested configuration are compiled in The sorted results in figure 6 confirm that the best-performing configuration is:

$$\text{n-gram} = (3, 5), \quad \text{max\_features} = 150{,}000, \quad \alpha = 5{\times}10^{-5}.$$



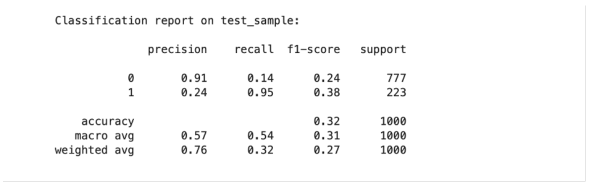
```
Hyperparameter search results (sorted):
3_5_150k_a5e-5 | F1 = 0.4968 | Accuracy = 0.7051 | ngram = (3, 5) | max_features = 150000 | alpha = 5e-05
3_4_200k_a5e-5 | F1 = 0.4915 | Accuracy = 0.7022 | ngram = (3, 4) | max_features = 200000 | alpha = 5e-05
3_4_150k_a1e-4 | F1 = 0.4528 | Accuracy = 0.6828 | ngram = (3, 4) | max_features = 150000 | alpha = 0.0001

Best configuration: {'name': '3_5_150k_a5e-5', 'ngram': (3, 5), 'max': 150000, 'alpha': 5e-05, 'val_acc': 0.7051125, 'val_f1': 0.496794015569
02764, 'fit_time': 2616.839792728424}

Training final model on full dataset...
```

Figure 6: Hyperparameter comparison for Subtask C.

4.3.3 Evaluation on test_sample

4.3.4 Subtask A

We evaluated performance of test sample after retrain model on both training andvalidation sets. Figure 5 shows model have very recall results in class 1 (Machine generated code) but it have struggles to identify class 0 correctly (human written code), which means that there is a small bias to predicting AI generated code.

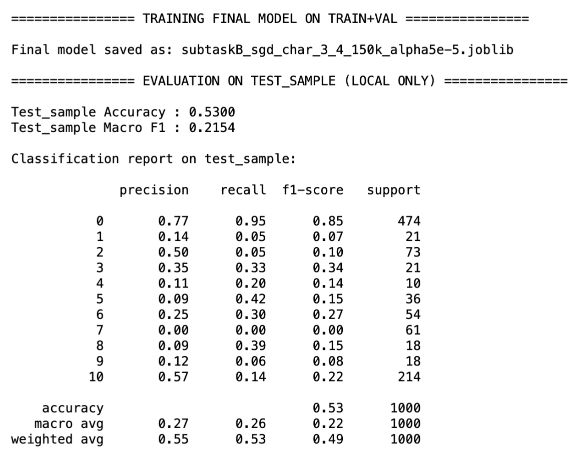


```
Classification report on test_sample:

              precision    recall  f1-score   support

           0       0.91      0.14      0.24       777
           1       0.24      0.95      0.38       223

    accuracy                           0.32      1000
   macro avg       0.57      0.54      0.31      1000
weighted avg       0.76      0.32      0.27      1000
```

Figure 7: Classification report for the best Subtask A model on test_sample.

4.3.5 Subtask B

The chosen model was retrained using the combined training and validation sets, and it was assessed using `test_sample`. Strong performance on the human class is demonstrated by the results, but minority LLM families present challenges (Figure 8).



```
================ TRAINING FINAL MODEL ON TRAIN+VAL ================

Final model saved as: subtaskB_sgd_char_3_4_150k_alpha5e-5.joblib

================ EVALUATION ON TEST_SAMPLE (LOCAL ONLY) ================

Test_sample Accuracy : 0.5300
Test_sample Macro F1 : 0.2154

Classification report on test_sample:

              precision    recall  f1-score   support

           0       0.77      0.95      0.85       474
           1       0.14      0.05      0.07        21
           2       0.50      0.05      0.10        73
           3       0.35      0.33      0.34        21
           4       0.11      0.20      0.14        10
           5       0.09      0.42      0.15        36
           6       0.25      0.30      0.27        54
           7       0.00      0.00      0.00        61
           8       0.09      0.39      0.15        18
           9       0.12      0.06      0.08        18
          10       0.57      0.14      0.22       214

    accuracy                           0.53      1000
   macro avg       0.27      0.26      0.22      1000
weighted avg       0.55      0.53      0.49      1000
```

Figure 8: Classification report for the best Subtask B model on test_sample.

4.3.6 Subtask C

Code stopped running because kernel got stuck even after many attempts, there is no error in implementation but because of heavy computational load because there are a large number of samples with high dimensional features.

5 Conclusion

In subtask A we used binary classification using TF-IDF in character level with SGD linear classifier, model shows high performance in detecting machine generated code (strong recall) , but it struggles to identify human written codes correctly , which cause a low Micro F1 value and have bias for AI generated code. In future to reduce bias for AI generated code we might use class-balancing methods, to combine a different feature type we use ensemble models, and replace single unified pipeline with language specific models.

For Subtask B of SemEval-2026 Task 13, we demonstrated a simple yet efficient system. We discovered an ideal character-level TF–IDF and SGD pipeline through thorough hyperparameter exploration, which produced a macro F1 of 0.2746 on validation and 0.2154 on the official test_sample. The system shows good discriminative ability for major families, but performance on minority generator classes is still difficult. Future research will focus on three areas:
(1) focal loss variants and class rebalancing;
(2) ensemble architectures that combine lexical and semantic features; and
(3) language-specific modelling as opposed to a single multilingual vectorizer.

As Subtask-C of SemEval-2026 Task 13 stucked in kernal, so in future we will do the following
(1) Reduce number of dataset (smaller part of data).
(2)Reduce features (use 80 k instead of 150 k)
(3) Use computer with more RAM
(4) Reduce n grams