Semantic Classification of Math Problems Using NLP

Deepika Reddygari – Individual Final Project Report
Course: NLP Final Project (Amir Jafari)

1. Introduction

Automatic categorization of word-based math problems into eight domains (Algebra, Arithmetic, Calculus, Geometry, Number Theory, Probability, Statistics, Trigonometry) enables adaptive learning systems to deliver targeted practice and assessment. In our group proposal, we outlined a two-phase evaluation: (1) classical-model baselines I implemented, and (2) advanced transformer-based and ensemble approaches developed by teammates . This report focuses on my baseline pipelines (Multinomial Naive Bayes, Random Forest, XGBoost) and situates their performance within the full spectrum of models we evaluated.

---

2. Dataset & Task

- **Dataset**: KAUST Math Problem Classification (25 000 train, 5 000 test)
- **Classes**: 8 balanced domains (~3 125 examples each)
- **Preprocessing challenges**:
    - LaTeX spans for equations (~40% of samples)
    - Varied phrasing and question lengths

---

3. Description of My Individual Contributions

I led the design, implementation, and validation of the classical‑model baselines. My work can be broken down into six major components:

3.1 Text Preprocessing Pipeline

- **Math-span normalization**
  - Detected all LaTeX spans (`\$…\$`) and replaced them with the single token `MATH` to ensure consistency across examples (e.g., `$x^2 + y^2 = 1$` → `MATH`).
- **Noise removal and normalization**
  - Lowercase all text to eliminate case variance.
  - Stripped out non-alphanumeric characters (except spaces) to remove extraneous punctuation.
  - Collapsed multiple whitespace characters into a single space.

Implementation details:
‘’’
```
import re
def clean_text(s: str) -> str:

   # 1) Normalize math spans
   s = re.sub(r'\$.*?\$', ' MATH ', s)

   # 2) Remove punctuation / special chars
   s = re.sub(r'[^a-z0-9 ]', ' ', s.lower())

   # 3) Collapse whitespace
   return re.sub(r'\s+', ' ', s).strip()
```
‘’’

- This function was applied in a vectorized fashion using `pandas.Series.apply` over the 25 000 training and 5 000 test examples .

3.2 Feature Engineering

- **Word-level TF-IDF**
  - Built a `TfidfVectorizer` with 1–2 gram tokens, sublinear TF scaling, and `max_features=7 000` to capture common word sequences for Naive Bayes and Random Forest.

- **Character-level TF-IDF**
  - Configured a separate `TfidfVectorizer` with `analyzer='char_wb'`, n-grams of length 3–5, and `max_features=10 000` to capture subword patterns useful for XGBoost.
- **Rationale**
  - Word n-grams capture semantic phrases (e.g., “find the derivative”), while char n-grams capture morphological cues (e.g., “sin”, “tan”) that are particularly salient in math text .

3.3 Model Development

- **Multinomial Naive Bayes**
  - Utilized `sklearn.naive_bayes.MultinomialNB(alpha=1.0)`.
  - Trained on word TF-IDF features for rapid baseline performance.
- **Random Forest Classifier**
  - Employed `sklearn.ensemble.RandomForestClassifier(n_estimators=100, class_weight='balanced')`.
  - Balanced class weights to counteract any minor class-distribution noise.
- **XGBoost**
  - Used `xgboost.XGBClassifier(max_depth=6, learning_rate=0.1, subsample=0.8, colsample_bytree=0.8)`.
  - Leveraged character TF-IDF inputs to obtain the strongest classical baseline.

3.4 Evaluation & Reporting

- **Metrics computed**:
  - Macro-averaged F1, precision, recall on the held-out validation split.
  - Final private‑test (Kaggle) F1-Micro for comparison across models.
- **Error analysis**:
  - Generated confusion matrices to identify frequent misclassifications (e.g., Geometry ↔ Trigonometry).
  - Produced per‑class classification reports to spot low-recall classes and guide future feature enhancements.

3.5 Code Organization & Reproducibility

- **Modular script structure**:
  - preprocessing.py: cleaning utilities.
  - features.py: TF-IDF vectorizer setup.
  - train_baselines.py: end-to-end training loops for NB, RF, XGB.
  - evaluate.py: metric computation and plotting.
- **Version control & experiment tracking**:
  - Logged all hyperparameter combinations and results to a CSV via Python's `logging` and `itertools`.
  - Ensured that random seeds were fixed (`numpy`, `sklearn`, `xgboost`) for full run reproducibility.

4. Extended Model Comparisons

Below are the training times and F1-Micro scores on both the training split and the private Kaggle test set. Baseline models that I implemented appear at the top; subsequent rows show teammates' advanced approaches.

Model	Train Time	F1-Micro (Train)	F1-Micro (Private Test)
Multinomial Naive Bayes	1.04 m	0.72	0.704
Random Forest Classifier	6.51 m	0.74	0.7388
XGBoost	10.02 m	0.79	0.7678
LightGBM	6.24 m	0.789	0.7862

Table 1: Comparison of baseline models on both training and private test splits.

Discussion:

- My three baselines show a clear progression: Naive Bayes (0.7040) → Random Forest (0.7388) → XGBoost (0.7678) on the private test set.
- LightGBM (0.7862) offers a modest boost over XGBoost.
- Transformer-based models (MathBERT, Llama-1b, T5, DeBERTa) further improve performance, with DeBERTa achieving the highest single-model F1 of 0.8510.
- A hard-voting ensemble of the top three transformers reaches 0.8588, demonstrating complementary strengths across architectures.

5. Analysis & Insights

1. **Feature impact**: Character n-grams (XGBoost) outperform word n-grams (RF), indicating the value of subword patterns in math language.

2. **Model complexity vs. gain**: Transformer-based methods require far more compute (hours vs. min) but yield up to ~15 points of F1 improvement over baselines.

3. **Ensembling benefit**: Hard voting among top transformers adds ≈0.0078 F1, justifying ensemble integration for production pipelines.

---

6. Summary & Future Directions

- **Key takeaways**:
    - Well-tuned classical baselines achieve respectable F1 scores (>0.76) with minimal compute.
    - Advanced transformers and ensembling push F1 beyond 0.85, at the cost of substantially longer training times.

- **Future work**:
    - Explore distilled transformer variants to reduce training and inference cost.
    - Implement dynamic ensembling or stacking to further boost performance.
    - Incorporate data augmentation (paraphrasing, equation rewriting) to alleviate short-text sparsity.

---

7. Code Attribution

Across my three Python scripts (≈340 LOC):

- ~160 LOC adapted from online examples (47%)
- ~180 LOC original and modified by me

____________________________END OF THE REPORT_______________________________