TeamXBC at BEA 2026 Shared Task 1: How AI (and I) won the shared task: Vibe and agentic coding solutions for practical machine learning problems

Xiaobin Chen
Universität Tübingen
xiaobin.chen@uni-tuebingen.de

Abstract

The paper describes how the author used AI coding agents and a technique called *vibe coding* to successfully tackle the BEA 2026 shared task on vocabulary difficulty prediction. Three sets of predictions (runs) were submitted to the competition, corresponding to three experiments the author ran by giving the coding agent different levels of agency: (1) a one-off solution fully planned and implemented by the AI, (2) an AI self-determined iterative process that ran for 24 hours, and (3) a collaborative human-in-the-loop process where solutions were discussed between the author and the AI. Competition results showed that the collaborative mode delivered the best performance, demonstrating that at the current stage domain expert input and decision making are important and necessary for vibe coding solutions to practical machine learning problems.

1 Introduction

First and foremost, it should be acknowledged that we (AI and the author) have not "won" the top places for both tracks and all the three languages. However, our best performing models for the open track did reach the third place consistently. Table 1 summarizes the ranks and relative improvement over the baseline of our best runs based on the official results summary[1]. On average, our best runs across the tracks achieved 20.3% RMSE improvement and 11.2% Pearson correlation gains over the baseline. AI agents, especially coding agents were heavily used in the process of the experiments through a technique called *vibe coding*, first coined and made popular by the AI researcher and influencer Andrej Karpathy.

In this paper, three different ways (corresponding to the three runs submitted for competition) coding agents were used to tackle the shared task will be described. The study shows that vibe coding with AI agents can democratize machine learning (ML) and greatly increase the efficiency of ML experiments. Coding agents can also function as collaborators and instructors that help with brainstorming, discussion, and human learning.

2 The Shared Task

The first language (L1) aware vocabulary difficulty prediction shared task concerns the prediction of the difficulty levels of second language (L2) English words to learners of three different L1s: Spanish, German, and Chinese. Word difficulty is defined as the degree to which L2 learners of specific L1s are able to correctly spell out the L2 word given its L1 translation and an example sentence of its use in the L1 (Laufer and Goldstein, 2004). An example test item for Spanish learners of English is as follows:

> Vivo en una casa grande que tiene tres dormitorios.
> casa
> h _ _ _ _

The dataset was adapted from earlier studies Schmitt et al. (2021, 2024) in which more than 100,000 crowd workers submitted over 3.3 million responses to 7,516 vocabulary test items. Each test item corresponds to one English word and contains three L1 variations. Therefore, three difficulty values can be calculated for the three L1s, estimating how difficult an English word is for learners of the corresponding L1s. The difficulty values in the dataset were calculated by fitting random-item-random-person (RPRI) Rasch models (De Boeck, 2008) with the generalised linear mixed model (GLMM) framework (Dunn, 2024). Data for 748 items were withheld as a test set for evaluation of the submissions. The training and development sets contained 6,091 and 677 items respectively.

[1]The full results are available at https://github.com/britishcouncil/bea2026st/blob/main/results/results_summary_test.md

Proceedings of the 21st Workshop on Innovative Use of NLP for Building Educational Applications, pages 997–1009

				RMSE			r		
Track	L1	Run	Ranking	TeamXBC	Baseline	Δ RMSE	TeamXBC	Baseline	Δr
Closed	ES	2	13/22	1.146	1.257	+8.8%	0.823	0.765	+7.6%
Closed	DE	1	11/21	1.114	1.258	+11.4%	0.837	0.773	+8.3%
Closed	CN	1	10/22	0.968	1.140	+15.1%	0.851	0.753	+13.0%
Open	ES	3	3/9	0.876	1.198	+26.9%	0.885	0.783	+13.0%
Open	DE	3	3/9	0.826	1.166	+29.2%	0.888	0.786	+13.0%
Open	CN	3	3/10	0.722	1.034	+30.2%	0.904	0.804	+12.4%

Table 1: Performance of TeamXBC's best-performing runs across the two shared-task subtracks and three languages. Languages: ES = Spanish, DE = German, CN = Chinese. Δ columns show relative improvement over the baseline for RMSE (reduction) and Pearson r (increase).

The shared task is essentially a regression problem with the test items as predictors and the continuous GLMM scores as outcomes. Participants could submit their predictions to the *closed* and/or *open* subtracks, the former restricting the use of training data to the corresponding L1 subset and publicly available Transformer models or linguistics databases, while the latter allows maximum flexibility on the use of datasets and external resources, including large language models (LLMs). A previous study by Skidmore et al. (2025) attempted the task by fine-tuning Transformer-based models, whose performance was used as baselines of the shared task (see Table 1).

It is not difficult to envision the value of vocabulary difficulty prediction models on L2 learning and teaching research and practice. If successful, such models will be useful for custom content creation, computerized adaptive testing, and personalized learning. They may also play important roles in applications such as essay scoring, proficiency assessment, readability assessment, test design, and psycholinguistic experiment on vocabulary processing (Skidmore et al., 2025).

3 Vibe Coding and Agentic Machine Learning

As an intelligent computer assisted language learning (ICALL) researcher, the author has been working with various natural language processing and ML tools for many years. However, the accelerating emergence of large number of new technologies, tools, and research methods still feels overwhelming, although they often offer exciting opportunities for new research. Because of the same reason, collaborating with computational and ML experts is often seen among researchers of other domains that require the technological expertise (Lee and Huh, 2025). The recent emergence of a technique called "vibe coding" supported by advanced coding agents such as Claude Code, OpenAI Codex, and Github Copilot has made it much easier for researchers with less computational and ML backgrounds to make use of the new technologies.

Vibe coding is a term coined by the AI researcher Andrej Karpathy[2] to describe a way to create computer programs by specifying the requirements in natural language to an AI coder and allowing it to generate computer code automatically. It essentially adds a semiotic layer to traditional coding (Yuan et al., 2025; Yin and Xiao, 2025) where the programmer had to write structured instructions following rigid syntax of the programming language. A more advanced version of vibe coding is called agentic coding which features AI agents capable of planning, executing, and verifying coding tasks by decomposing the requirements and automatically using available tools in the environment (Kumar et al., 2025; Roychoudhury et al., 2025). Although agentic coding can be viewed as giving the coding agent higher levels of agency, in reality, the distinction between vibe coding and agentic coding is blurry. Therefore in the study, we will stick with the former term. Vibe coding lowers the barrier into programming and accelerates software creation. Using it to create machine learning programs could therefore address the issue of lack of technological expertise from domain experts. To date, mentions of vibe coding in research, especially in

[2]He is also an OpenAI co-founder and former AI director at Tesla. The original Tweet where the term was first used can be viewed at https://x.com/karpathy/status/1886192184808149383

ML or deep learning studies have been rare (Lee and Huh, 2025).

4 The Experiments

In the present study, vibe coding was employed to tackle the BEA 2026 shared task on vocabulary difficulty prediction. It focuses on the following research questions (RQs).

1. Can vibe coding be used to solve real-life ML problems?

2. Does giving the coding agent a higher level of agency help with solving practical ML problems?

3. What is the researcher's perception of working with coding agents as research collaborators?

It is interesting to see how the AI coded solutions perform compared to the other submissions to the shared task (RQ1). For vibe coding, there could be more or less human involvement. On one extreme, the coding agent can be instructed to run indefinitely or until a certain performance goal is achieved: design a solution, run it, look at the result, improve the solution and iterate. On the other extreme, the researcher may suggest a solution, ask the agent to code it, run the solution, look at the result, think of an improvement and ask the agent to update the code and iterate until a satisfactory result is achieved. On the middle ground, the researcher and the agent may work together to discuss the solution and results to come up with possible improvements together, mimicking how human collaboration occurs. In other words, more or less human involvement would correspond to more or less AI agency as asked in the second RQ.

Given that three runs of predictions can be submitted, in light of the research questions, three corresponding experiments were devised with vibe coding in Claude Code (Opus 4.6) as described below. The coded experiments were run on a university High-Performance Computing (HPC) cluster with NVIDIA H100 GPUs.

Run 1: One shot fully agentic solution. The agent was given the URL of the shared task home page and instructed to understand the task and create the experiment code and SLURM scripts for submitting the training tasks on the HPC. Appendix A lists the detailed prompt.

Run 2: 24-hour iterative agentic solution. This run follows Karpathy's idea of auto research[3] by providing the coding agent with a research program in which the task, the environment setup, experimentation constraints, evaluation criteria, and looping requirements are specified. The experiment was kicked off with one prompt: "Take a look at *program.md* and let's start the experiment." The full research program used for this run can be found in Appendix B. The agent was instructed to run the iterations indefinitely, but for the current experiment the process was stopped after about 24 hours.

Run 3: hybrid human-in-the-loop collaborative solution. In this experiment, the coding agent was treated as a collaborator and ML expert with whom the author could discuss the potential solutions. With domain expertise, the author suggested some solution plans and asked for feedback and suggestions from the agent. When the agent's suggestions were taken, the plans were updated and agreed upon between the author and the agent. They were then implemented and the results were discussed again for improvement. The experiment encompasses multiple sessions over a few days until the final models had given satisfactory performance on the development set.

5 Results

5.1 Run 1

Given the prompt in Appendix A, the coding agent designed a six-stage experiment to progressively explore how to achieve the best models.

Stage 0: Feature-based baseline. Gradient boosting models trained on hand-crafted features extracted via *wordfreq* and NLTK. They served as a non-neural lower bound and ran locally without GPUs. One joint model for the open track and three separate models for the closed track were trained.

Stage 1: Architecture search. Question asked: Is *XLM-RoBERTa-large* better than *mDeBERTa-v3-base* for this task? The winner of this stage was intended to be carried forward into Stage 2. The models were trained with learning rate LR=3e-5, 5 epochs, fp16, effective batch size 32 (batch=16 x grad-accum=2).

Stage 2: Hyperparameter search + closed track. Questions asked: What learning rate and epoch

[3] see https://github.com/karpathy/autoresearch

count work best for the winning architecture? Does it transfer to the closed track? The best hyperparameters found (LR=2e-5, 8 epochs) were applied for subsequent training.

Stage 3: Input format experiments. Question asked: Does the content or ordering of the input text affect predictions? Fixed architecture and hyperparameters at the Stage 2 best (`XLM-RoBERTa-large`, LR=2e-5, 8 epochs). Experimented on shifting instance component order, e.g. putting *en-target-pos* at the end, or putting L1 learner fields first compared to using an English-first (*en-target-word; en-target-pos; en-target-clue; L1-source-word; L1-context*) ordering.

Stage 4: Advanced training. Question asked: Can combining the best techniques, or training longer, squeeze out additional gains? Changes over Stage 3 included having stronger decay and more epochs.

Stage 5: Ensemble. After running the models trained in previous stages, ensemble models were trained with two methods: (1) weighted averaging: top 5 models by Pearson, weighted by their performance on the development set, and (2) stacking: a meta-learner trained on the development predictions of the top 5 models.

In all, 15 models were designed and predictions for 14 models were successfully obtained after running the generated code on the HPC. The best overall performing model with the lowest average RMSE across the three L1s trained on the training set and evaluated on the development set was a Transformer model fine-tuned for 8 epochs on *XLM-RoBERTa-large* with a batch size of 16, a learning rate of 2e-5, a warmup ratio of 0.1, and weight decay of 0.1, using the AdamW optimizer and an English first component order *en-target-word, en-target-pos, en-target-clue, L1-source-word, L1-context*. For different tracks and L1s, slightly different hyperparameters were used in the best performing models. These models were retrained with both the training and development sets to predict on the test set for submission. Their performance on the test set is listed in Table 2.

Through the experiment, it was learned that:

- Model scale was the biggest driver of improvement. Switching from *XLM-RoBERTa-base* (baseline) to *XLM-RoBERTa-large* delivered the largest single gain, roughly +10% RMSE improvement (decrease) across all tracks and L1s.

- Open track models outperformed closed track models. The joint cross-lingual open model consistently achieved better Pearson scores than L1-specific closed models. Training on all three L1s simultaneously appears to provide useful cross-lingual signal, echoing the findings of Skidmore et al. (2025).

- Moderate learning rate and epochs generalized best. Among the hyperparameter experiments, LR=2e-5 with 8 epochs performed best or near-best across all L1s. Lower learning rate with more epochs was competitive but slightly weaker. Extended training to 12 epochs hurt performance, suggesting the model overfit beyond 8 epochs.

- Part-of-speech (POS) tags did not help. Adding POS information showed no improvement on any L1—Pearson r was slightly lower in all three cases.

- Advanced training techniques were largely ineffective. Layerwise LR decay helped for DE but hurt ES. Combining it with a higher LR degraded performance across the board, performing worse even than the basic large model. These techniques did not reliably improve over careful standard fine-tuning.

- L1 Chinese was consistently the easiest to predict and Spanish the hardest, again corroborating Skidmore and colleagues' (2025) findings.

5.2 Run 2

For Run 2, the coding agent was instructed to run the auto research procedure in Appendix B on the HPC. It ran for about 24 hours before being manually stopped. It made 72 commits and evaluated 405 unique model configurations in three phases.

Phase 1: Neural base models. Starting from the model used for the baseline *XLM-RoBERTa-base*, the agent systematically explored:

- Backbone upgrades: by using *XLM-RoBERTa-large*, the performance immediate jumped to 0.83–0.84 on Pearson correlation

		RMSE			r			Ranking	
Track	L1	Baseline	Run 1	Δ RMSE	Baseline	Run 1	Δr	Team	Submission
Closed	ES	1.257	1.192	+5.2%	0.765	0.816	+6.7%	16/21	36/56
Closed	DE	1.258	1.114	+11.4%	0.773	0.837	+8.3%	11/20	25/53
Closed	CN	1.140	0.968	+15.1%	0.753	0.851	+13.0%	10/21	23/53
Open	ES	1.198	1.114	+7.0%	0.783	0.832	+6.3%	7/8	20/24
Open	DE	1.166	1.042	+10.6%	0.786	0.848	+7.9%	7/8	20/24
Open	CN	1.034	0.943	+8.8%	0.804	0.864	+7.5%	7/9	20/26

Table 2: Performance of TeamXBC Run 1 compared to the baseline, with competition rankings

- Hyperparameter tuning: using different learning rates (1e-5, 2e-5, 5e-6), epochs (5–15), weight decay, warmup ratio, batch sizes (32, 64)
- Alternative architectures: tried *RemBERT*, *Multilingual-E5-large*, and *mDeBERTa-v3-base*.
- Training tricks: cosine learning rate schedule, Pearson correlation loss, gradient accumulation, layerwise LR decay, model soups, multi-seed averaging, English-first feature ordering, L1-code as a feature, etc.

The best single neural models in this phase achieved Pearson correlations of 0.855 and 0.838 on the open and closed tracks respectively.

Phase 2: Ensemble stacking. After some models have been trained in Phase 1 the agent pivoted to using trained neural model predictions as features for stacking and explored the following options:

- Ridge regression over base model outputs achieving 0.856 on Pearson correlation
- Polynomial features (quadratic, cubic, quartic) resulting in progressively higher performance but tend to overfit on development set
- Support Vector Regression stacking (various C values) achieving 0.858 Pearson correlation
- Gradient Boosted Trees as meta-learner achieving 0.865 Pearson correlation
- Honest out-of-fold stacking to avoid leakage with confirmed honest ceiling at 0.966

Phase 3: Deep cascade meta-stacking. The deep stacking approach experimented in this phase achieved major breakthroughs on the development set. A three-level stacking of base, ensemble, and ridge models delivered an average Pearson's correlation of 0.96 (likely overfitting). It was also found that adding *RemBERT* predictions improved the ensemble ceiling, but *mDeBERTa-v3-base* did not help.

For the final submission, again the best performing models were retrained with both the training and development sets and predicted on the test set. The final open model was a 3-level stacking cascade model integrating 36 *XLM-RoBERTa-large* and 3 *RemBERT* base models. The close track models were single *XLM-RoBERTa-large* based models with different hyperparameter configurations. Table 3 shows the test performance of the final models and their rankings in the competition. It can be observed that Run 2 had significant improvement over Run 1, showing that coding agents for ML is capable of self-iterative improvement when given more agency.

5.3 Run 3

This is a hybrid human-in-the-loop process in which the solutions were discussed between the author and the coding agent. With limited expertise in ML and mostly traditional feature engineering based ML, the author started by suggesting to train some feature-engineered regression models. The coding agent agreed that such models could be competitive, but it also predicted that they would not outperform the Transformer baselines. Therefore, Ridge and XGBoost models were trained with linguistic features such as word frequency, word length, POS tags, L1 context length, Levenshstein string edit distance ratios between the L1 and L2 target words, etc. Table 4 compares the performance of the traditional regression models against the baseline on the development set, confirming the coding agent's predictions.

Building on the these results, the author proposed integrating the engineered features into the

		RMSE			r			Ranking	
Track	L1	Baseline	Run 2	ΔRMSE	Baseline	Run 2	Δr	Team	Submission
Closed	ES	1.257	1.146	+8.8%	0.765	0.823	+7.6%	13/21	28/56
Closed	DE	1.258	1.121	+10.9%	0.773	0.835	+8.0%	14/20	30/53
Closed	CN	1.140	1.002	+12.1%	0.753	0.859	+14.1%	11/21	25/53
Open	ES	1.198	1.003	+16.3%	0.783	0.846	+8.0%	4/8	10/24
Open	DE	1.166	0.947	+18.8%	0.786	0.856	+8.9%	5/8	11/24
Open	CN	1.034	0.830	+19.7%	0.804	0.880	+9.5%	4/9	9/26

Table 3: Performance of TeamXBC Run 2 compared to the baseline, with competition rankings

		r			RMSE		
Model	Track	CN	DE	ES	CN	DE	ES
Baseline	closed	**0.736**	**0.753**	**0.748**	**1.175**	**1.328**	**1.357**
XGB	closed	0.679	0.647	0.644	1.242	1.393	1.467
Ridge	closed	0.619	0.603	0.566	1.329	1.457	1.578
Baseline	open	**0.804**	**0.800**	**0.787**	**1.021**	**1.149**	**1.206**
XGB	open	0.670	0.639	0.629	1.256	1.404	1.490
Ridge	open	0.610	0.591	0.556	1.340	1.476	1.593

Table 4: Development set performance of traditional models vs. XLM-RoBERTa-base fine-tuned baseline.

baseline models to construct some hybrid models. The coding agent suggested and implemented 4 strategies to combine feature engineering with neural models:

- Ensemble: using weighted average of the predictions from both the XGBoost and the *XLM-RoBERTa-base* fine-tuned models
- Feature injection: concatenating the feature values to the Transformer's vector before attaching the regression head
- Stacking: training a meta learner with the output of the traditional and neural models
- Features as extra tokens: prepending feature values as text to the transformer input

The ensemble models turned out to have the best performance as shown in Table 5. Feature injection and features as extra tokens gave marginal gains only. The stacking strategy resulted in second best RMSE but lower Pearson's correlation, meaning that the strategy is better at predicting the magnitude of scores but worse at identifying the pattern of variation.

Looking at the results from the previous steps, a clear direction to further explore and improve the models' performance was to try to boost the performance of the neural models because they contributed more than the traditional models in the ensembles. Therefore, the coding agent was asked to suggest some bigger base models for fine-tuning. It did, also suggested a few other base models such as the decoder only Qwen models and different fine-tuning methods like QLoRA. Six models were successfully trained and their performance is listed in Table 6.

The coding agent was also instructed to try using the embeddings from OpenAI's embedding API, which is a commercial service to be compared with the open source models used so far. But the performance of the resulting models was rather unimpressive—both the large and small embeddings performed significantly worse than the baseline. The coding agent was then prompted to further improve on the trained models. It suggested new ensembles, larger Qwen models, and using labeled input format as shown below for fine-tuning:

> **Default format:** `casa </s> Vivo en una casa grande que tiene tres dormitorios. </s> h _ _ _ _ </s> house`
>
> **Labeled format:** `source: casa context: Vivo en una casa grande que tiene tres dormitorios. clue:`

	r		RMSE	
L1	Closed	Open	Closed	Open
CN	0.749 (+1.8%)	0.808 (+0.5%)	1.125 (+4.3%)	1.000 (+2.1%)
DE	0.770 (+2.3%)	0.802 (+0.3%)	1.213 (+8.7%)	1.117 (+2.8%)
ES	0.769 (+2.8%)	0.795 (+1.0%)	1.242 (+8.5%)	1.171 (+2.9%)

Table 5: Run 3 ensemble models' performance on the development set with improvements over the baseline in parentheses.

		r		
Model	Track	CN	DE	ES
`xlmr_large`	closed	0.813 (+10.5%)	0.826 (+9.7%)	0.822 (+9.9%)
`xlmr_large`	open	0.843 (+4.9%)	0.842 (+5.3%)	0.842 (+7.0%)
`qwen1.5b_lora`	open	**0.877 (+9.1%)**	**0.852 (+6.5%)**	**0.867 (+10.2%)**
`qwen3b_lora`	open	0.759 (−5.6%)	0.679 (−15.1%)	0.704 (−10.5%)
`bge` (frozen)	open	0.689 (−14.3%)	0.628 (−21.5%)	0.640 (−18.7%)
`e5` (frozen)	open	0.667 (−17.0%)	0.631 (−21.1%)	0.616 (−21.7%)
		RMSE		
`xlmr_large`	closed	1.026 (+12.7%)	1.138 (+14.3%)	1.181 (+13.0%)
`xlmr_large`	open	0.947 (+7.2%)	1.069 (+7.0%)	1.079 (+10.5%)
`qwen1.5b_lora`	open	**0.810 (+20.7%)**	**0.964 (+16.1%)**	**0.956 (+20.7%)**
`qwen3b_lora`	open	1.101 (−7.8%)	1.340 (−16.6%)	1.364 (−13.1%)
`bge` (frozen)	open	1.225 (−20.0%)	1.429 (−24.4%)	1.472 (−22.1%)
`e5` (frozen)	open	1.258 (−23.2%)	1.417 (−23.3%)	1.508 (−25.0%)

Table 6: Development set performance of larger and alternative architecture models with improvements over the baseline in parentheses. Bold indicates the best result per column.

```
h _ _ _ _ target: house
```

It turned out that the labeled input models fine-tuned on *Qwen-7b* had the best performance after the new ensemble models that require a sophisticated combination of other trained models. But the differences between them were only at the 1% level. Considering the resources and complexity required for training the ensemble models, as well as the practical applications, it was decided that the final submission for Run 3 should be based on the best performing individual models, namely the *XLM-RoBERTa-large* base model fine-tuned separately per language for the close track and *Qwen2.5-7B* with QLoRA and labeled input format trained jointly on all three L1s for the open track. Table 7 shows their performance on the test set.

In all, Table 8 summarizes the performance of all the three runs.

6 Discussion

To answer RQ 1, the experiments clearly showed that it is possible to vibe-code ML solutions to practical problems with the help of advanced coding agents (Claude Code in this case). The shared task already had a high baseline with Pearson's correlation coefficients ranging from 0.753 to 0.804 (See Table 1). However, the best agent-coded solutions in the study were able to push the boundaries by 7.6% to 13.0% in Pearson's r and 8.8% to 30.2% in RMSE. It is unclear if the other participating teams also used coding agents, but our AI coded solutions were ranked number 3 out of 10 participating teams in the open track. In the closed track, our submissions also had an average performance at around the fifty percentile. The results demonstrated that AI-supported vibe coding is able to deliver promising performance on real-life ML tasks.

RQ 2 concerns the level of agency AI agents should be given when solving practical ML problems. The three runs devised in the study gave the coding agent different levels of agency, from the first run's one-off full agency, via the second run's full agency and opportunities to learn from previous successes and failures in an iterative manner, to the human-in-the-loop half agency in the last run. Varying results have been obtained for the closed and open tracks (cf. Table 8).

The one-off agency run resulted in similar amount of improvement over the baseline for the both tracks. Giving the AI more agency and allowing it to iterate resulted in better performance in Run 2 compared to Run 1. For Run 3, with the human in the loop, the resulting models were able to achieve the best performance among the three runs in the open track. On average, the Run 3 models improved Pearson's r by 12.8% and RMSE by 28.8%, which were significantly more than the other two runs with full AI agency. However, the L1 German and Spanish models performed unusually bad, although the development models seemed to have performed well. A later code review (also conducted by the AI) revealed that the DE and ES models actually failed to train in the last step before the submission. The final training was supposed to be done with both the training and development sets of the data, but the training logs showed non-improving losses across the epochs, e.g., DE epoch 1: loss=3.04 $\rightarrow$ epoch 2: 3.19 $\rightarrow$ epoch 3: 3.18 $\rightarrow$ epoch 4: 3.18 $\rightarrow$ epoch 5: 3.17, indicating training failures. A possible reason for the training failure was the re-use of previously saved training checkpoints, a coding bug that was not caught before the submission.

The results clearly showed that giving the AI more agency without human intervention did not necessarily improve the ML solutions. However, with human in the loop, i.e., when a domain expert worked collaboratively with a coding agent to discuss and implement the solutions, the resulting performance tends to be an order of magnitude better than the only-AI setup. Domain knowledge and human decision making played important roles in human-AI collaborations. The AI coders were also able to greatly improve the efficiency of experiments. Reflectively thinking, it would have taken the author countless hours to learn about the related ML concepts, base models, practical coding skills and ML libraries, as well as the use of HPC resources before the Run 3 solutions could be implemented manually and it is unclear if the results could be on par with the current solutions.

To RQ 3, the author's perception is that working with the AI in the shared task was a very enjoyable experience. It felt empowering because I would not have been able to do it within a period of weeks by myself. It also saved me the time and efforts to involve a potential ML expert in the project. In general, it offers a satisfactory solution without much complexity.

Track	L1	r	Δr	RMSE	Δ RMSE	Ranking	
						Team	Submission
Closed	CN	0.839	+11.4%	1.006	+11.8%	12/21	27/53
Closed	DE	0.299	−61.3%	1.796	−42.8%	20/20	53/53
Closed	ES	0.093	−87.8%	1.882	−49.7%	21/21	55/56
Open	CN	0.904	+12.4%	0.722	+30.2%	3/9	7/26
Open	DE	0.888	+13.0%	0.826	+29.2%	3/8	7/24
Open	ES	0.885	+13.0%	0.876	+26.9%	3/8	7/24

Table 7: Performance of TeamXBC Run 3 compared to the baseline, with competition rankings.

		CN		DE		ES		Avg	
Track	Run	r	RMSE	r	RMSE	r	RMSE	r	RMSE
Closed	run_1	+13.0%	+15.1%	+8.3%	+11.4%	+6.7%	+5.2%	+9.3%	+10.6%
Closed	run_2	+14.1%	+12.1%	+8.0%	+10.9%	+7.6%	+8.8%	+9.9%	+10.6%
Closed	run_3	+11.4%	+11.8%	−61.3%	−42.8%	−87.8%	−49.7%	−45.9%	−26.9%
Open	run_1	+7.5%	+8.8%	+7.9%	+10.6%	+6.3%	+7.0%	+7.2%	+8.8%
Open	run_2	+9.5%	+19.7%	+8.9%	+18.8%	+8.0%	+16.3%	+8.8%	+18.3%
Open	run_3	+12.4%	+30.2%	+13.0%	+29.2%	+13.0%	+26.9%	**+12.8%**	**+28.8%**

Table 8: Test set performance of all three TeamXBC submissions relative to the official baseline.

7 Conclusion

The study demonstrated the viability of vibe coding ML solutions to real-life education problems. The three experiments giving the coding agent different levels of agency showed that at the current stage, the human-in-the-loop approach in which a domain expert worked with an AI agent collaboratively could result in the best solution. In other words, human and AI agencies are both important contributors of success. By adding a semiotic layer enabling the use of natural language for programming, vibe coding is democratizing computational technologies for researchers from domains other than computer science and ML. Even for experienced computer experts, AI coders are also likely to greatly increase their work efficiency and productivity. Future research is encouraged to further explore the role of AI in scientific work and how to better make use of it to improve and accelerate scientific discoveries.

8 Limitations

The study was conducted with one coding agent and on one ML task. Therefore, the generalizability of the findings should be taken with a grain of salt. The shared task was clearly defined with structured datasets. Code for establishing the baseline was readily available, providing a good starting point and program structure for further experiments with the coding agent. Real-life ML tasks often come with messy and unstructured data, lack of gold-standard data for training and validation, as well as no previous successful implementations. Hence it is unclear whether the AI coding agent will be able to handle the tasks as well as with the BEA shared task. Another limitation concerns the approach of vibe coding, which depends on coding models trained on data up to certain time points. As a result, they might not be aware of the latest ML methods, library and API updates, or things that did not appear in the models' training data. The resulting solutions from the coding agents should therefore be carefully evaluated before they are put into production. Security and code quality should be validated by human experts to avoid problems such as the "failed to train" problem in Run 3. Last but not least, for the study, the potential leakage of solution information across the experiments could not be ruled out. The three experiments were conducted in separate Claude Code sessions with different local copies of the code repository. However, it is understood that Claude Code has a shared

memory across sessions for the same user account, which could result in the coding agent making use of information from the other experiment sessions. Therefore, it is unclear if the performance of the coding agent would remain the same when only one experimental approach was adopted.

9 Acknowledgments

The author would like to thank the anonymous reviewers for their valuable and detailed feedback on an earlier version of the paper.

References

Paul De Boeck. 2008. Random item IRT models. *Psychometrika*, 73(4):533–559.

Karen J. Dunn. 2024. Random-item Rasch models and explanatory extensions: A worked example using L2 vocabulary test item responses. *Research Methods in Applied Linguistics*, 3(3):100143.

Mayank Kumar, Jiaqi Xue, Mengxin Zheng, and Qian Lou. 2025. TFHE-Coder: Evaluating LLM-agentic fully homomorphic encryption code generation. *Preprint*, arXiv:2503.12217.

Batia Laufer and Zahava Goldstein. 2004. Testing vocabulary knowledge: Size, strength, and computer adaptiveness. *Language Learning*, 54(3):399–436.

Yoonhwan Lee and Sun Huh. 2025. How can clinicians leverage vibe coding for machine learning and deep learning research? *Endocrinology and Metabolism*, 40(5):659–667.

Abhik Roychoudhury, Corina Pasareanu, Michael Pradel, and Baishakhi Ray. 2025. Agentic AI software engineers: Programming with trust. *Preprint*, arXiv:2502.13767.

Norbert Schmitt, Karen Dunn, Barry O'Sullivan, Laurence Anthony, and Benjamin Kremmel. 2021. Introducing knowledge-based vocabulary lists (KVL). *TESOL Journal*, 12(4):e622.

Norbert Schmitt, Karen Dunn, Barry O'Sullivan, Laurence Anthony, and Benjamin Kremmel. 2024. *Knowledge-Based Vocabulary Lists*. British Council Monographs on Modern Language Testing. University of Toronto Press.

Lucy Skidmore, Mariano Felice, and Karen Dunn. 2025. Transformer architectures for vocabulary test item difficulty prediction. In *Proceedings of the 20th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2025)*, pages 160–174, Vienna, Austria. Association for Computational Linguistics.

Michael Yin and Robert Xiao. 2025. VIBES: Exploring viewer spatial interactions as direct input for livestreamed content. In *Proceedings of the 2025 ACM International Conference on Interactive Media Experiences*, IMX '25, Niterói, Brazil. ACM.

Jun Yuan, Kevin Miao, Heyin Oh, Isaac Walker, Zhouyang Xue, Tigran Katolikyan, and Marco Cavallo. 2025. VibE: A visual analytics workflow for subgroup-based semantic error analysis of CVML models. In *Proceedings of the 30th International Conference on Intelligent User Interfaces*, IUI '25, pages 1529–1547, Cagliari, Italy. ACM.

A Agent prompt for one-shot experiment (Run 1)

```
You are to do a shared task with all information from the following webpage: https://www.britishcouncil.org/data-science-and-insights/bea2026st. Write a self-iterative script for experiments to achieve high performance on the shared task. The experimentation code may require fine-tuning large machine learning models. In that case, just write the code and make sure they can run correctly. I will run the model training on an HPC with GPUs. If the training of models does not require GPUs, you may run it locally. Try your best to get the best performance. Write a summary of what you have done when finished.
```

B Agent prompt for 24-hour iterative agentic solution (Run 2)

program.md

```
# autoresearch

This is an experiment to have the LLM do
     its own research.

## The task
You will be doing a Shared Task of
    Vocabulary Difficulty Prediction for
     English Learners. Details of the
    Shared Task can be found at https://
    www.britishcouncil.org/data-science-
    and-insights/bea2026st

The task is about predicting vocabulary
    difficulty values calculated from
    second language (L2) learners'
    responses to a vocabulary production
     test. The difficulty values are
    derived from a GLMM model and
    organized by respondents' native
    languages (L1s).
```

```
Essentially, you are given a dataset in the `data/` folder split into the `train` and `dev` sets. In each set, the data is further divided into subsets based on the L1 of the learners. The tested vocabulary items in each L1 subset is the same. The difference is only in the language the vocabulary test items is presented, i.e., a test item for a word is presented in Chinese if the respondent is an L1 Chinese speaker.

Your task is to predict the GLMM_score for each word in the list given information of the item used to test the production of the word. The information of the test item is specified in details in the `Data` section of the README.md file. You will design machine learning models to predict the GLMM_score.

Please note that for the shared task, there are two tracks:
- Closed Track: only data for specific L1 can be used for models targeting that L1. LLMs are not allowed in this track.
- Open Track: data from all three L1s can be used to fine-tune models. LLMs are also allowed for the task.

## Setup

To set up a new experiment, work with the user to:

1. **Agree on a run tag**: propose a tag based on today's date (e.g. `mar5`). The branch `autoresearch/<tag>` must not already exist --- this is a fresh run.
2. **Create the branch**: `git checkout -b autoresearch/<tag>` from the current main branch.
3. **Read the in-scope files**: Read these files for full context:
   - `README.md` --- repository context.
   - Access `https://www.britishcouncil.org/data-science-and-insights/bea2026st` for more information about the shared task.
4. **Verify conda environment**: Follow the instructions in the Quick Start section of README.md to create the python environment for running the code. Check that `python run_pipeline.py --evaluate` results in the expected output (the numbers do not need to be exactly the same as in README).
5. **Check the results/results_summary_dev.csv file**: Verify the baseline model performance is already included in the file. New model results will be appended here automatically during experimentation.
6. **Confirm and go**: Confirm setup looks good.

Once you get confirmation, enter the experiment loop below --- do not pause again.

## Experimentation

Each experiment runs within a **fixed time budget of 60 minutes** (wall clock training time, excluding startup/compilation). Some models can run on CPU; others require a GPU via slurm (see the experiment loop). You launch a run simply as: `python run_pipeline.py --finetune --predict --evaluate --models_to_run <experiment_model_name>`.

**What you CAN do:**
- Modify `finetune.py`, `models/model_parameters.csv` - these are the only files you edit. Everything is fair game: model architecture, optimizer, hyperparameters, training loop, batch size, model size, etc. Add new models to maximize the GLMM_score prediction performance.
- Add new scripts, resources, new packages, dependencies as needed, but don't change the other existing files.

**What you CANNOT do:**
- Modify `run_pipeline.py`. It is read-only. It contains the steps for finetuning, running, and evaluating models.
- Modify the evaluation harness. The `evaluate.py` is the approach for evaluation of different models. It calculates Pearson's correlation and Root Mean Square Error between the models' predictions and the ground truth GLMM score.

**The goal is simple: get the highest Pearson's correlation and lowest RMSE.** Since the time budget is fixed, you don't need to worry about training time --- it's always maximally 60 minutes. Everything is fair game: change the base model, ordering of context components, batch size, learning rate, weight decay, warmup ratio, epochs. Add any parameter as you see fit. The only constraint is that the code runs without crashing and finishes within the time budget.

**Simplicity criterion**: All else being equal, simpler is better. A small improvement that adds ugly complexity is not worth it. Conversely, removing something and getting equal or better results is a great outcome --- that's a simplification win. When evaluating
```

whether to keep a change, weigh the complexity cost against the improvement magnitude. A 0.001 pearson improvement that adds 20 lines of hacky code? Probably not worth it. A 0.001 pearson improvement from deleting code? Definitely keep. An improvement of ~0 but much simpler code? Keep.

The first run: Your very first loop iteration should establish a trained baseline. Run `python run_pipeline.py --finetune --predict --evaluate` with the existing configuration as-is, before making any changes. (The `--evaluate` only run in Setup was just a smoke test to verify the environment; this first loop iteration performs actual training.)

Fine-tuning and evaluating a model

The `run_pipeline.py` script is the entry point for running models and evaluating their results. After designing a new model by adding model information in `models/model_parameters.csv` and adding the necessary training code, you may start to train the model by running `python run_pipeline.py --models_to_run <new_model_name> --finetune --predict --evaluate`.

You are on a HPC environment, so if GPU is needed for training a model, you may create a slurm script to run the training script with a GPU. In the slurm job specification, put in `#SBATCH --partition=h100-ferranti` to gain access to the GPU. For the other requirements, such as CPUs, memory, or nodes, specify as needed. Make sure to check running status and errors periodically and act accordingly.

Write the training script to always stop after 60 minutes if it has not finished. Do not train for too long.

The experiment loop

The experiment runs on the dedicated branch created during setup (e.g. `autoresearch/mar5`).

LOOP FOREVER:

1. Look at the git state: the current branch/commit we're on.
2. Tune `finetune.py` with an experimental idea by directly hacking the code. Use other base models if necessary. Try different model sizes, resources, and training parameters. Choose CPU or GPU as needed: use CPU (`python run_pipeline.py ...`) for lightweight/fast models; use a GPU via slurm for large neural models that require a GPU to train within the time budget.
3. git commit.
4. Run the experiment (redirect everything --- do NOT use tee or let output flood your context):
   - CPU: `python run_pipeline.py --models_to_run <new_model_name> --finetune --predict --evaluate`
   - GPU: create a slurm script (with `#SBATCH --partition=h100-ferranti`) that calls the same command, then `sbatch slurm_train.sh`
5. Read out the results: `cat results/results_summary_dev.csv` and see if the new model's performance is listed.
6. If the new model's performance is not listed, the run crashed. Check the python output or slurm `.out` or `.err` to read the Python stack trace and attempt a fix. If you can't get things to work after more than a few attempts, give up and git reset back to where you started.
7. If the run succeeded, store the newly trained model under `models/` alongside the baseline models.
8. If Pearson's correlation improved (higher), commit the results and think of new ways to improve further. If you have exhausted all options of that model, try a different model that can potentially work better.
9. If Pearson's correlation is equal or worse, git commit the trial (with details of what was tried and the result), then git reset back to where you started.

The idea is that you are a completely autonomous researcher trying things out. If they work, keep. If they don't, try something different while keeping a record of the failure. You are advancing the branch so that you can iterate. If you feel like you're getting stuck, you can rewind, but do this very sparingly (if ever).

Timeout: Each experiment should take no more than 60 minutes total (+ a few seconds for startup and eval overhead). If a run exceeds 60 minutes, kill it and treat it as a failure (after committing with the failure reason, discard and revert).

Crashes: If a run crashes (OOM, or a bug, or etc.), use your judgment: If it's something dumb and easy to fix (e.g. a typo, a missing import), fix it and re-run. If the idea itself is fundamentally broken, just skip it, log "crash" as the status in the commit message, and move on.

```
**NEVER STOP**: Once the experiment loop
     has begun (after the initial setup)
    , do NOT pause to ask the human if
    you should continue. Do NOT ask "
    should I keep going?" or "is this a
    good stopping point?". The human
    might be asleep, or gone from a
    computer and expects you to continue
     working *indefinitely* until you
    are manually stopped. You are
    autonomous. If you run out of ideas,
     think harder --- read papers
    referenced in the task specification
     information pages and the code repo
    , re-read the in-scope files for new
     angles, try combining previous near
    -misses, try more radical
    architectural changes. The loop runs
     until the human interrupts you,
    period.

As an example use case, a user might
    leave you running while they sleep.
    If each experiment takes you ~60
    minutes then you can run approx a
    total of about 10 experiments over
    the duration of the average human
    sleep. The user then wakes up to
    experimental results, all completed
    by you while they slept!
```