FLAME: A Small Language Model for Spreadsheet Formulas

Harshit Joshi[1*], Abishai Ebenezer[4*], José Cambronero Sanchez[2†], Sumit Gulwani[2†], Aditya Kanade[3†], Vu Le[2†], Ivan Radiček[5†*], Gust Verbruggen[2†]

[1] Stanford University
[2] Microsoft, USA
[3] Microsoft Research, India
[4] University of Washington
[5] Demiurg
harshitj@cs.stanford.edu, {abishai.ebenezer.m, ivan.radicek}@gmail.com, {jcambronero, sumitg, kanadeaditya, levu, gverbruggen}@microsoft.com

Abstract

Spreadsheets are a vital tool for end-user data management. Using large language models for formula authoring assistance in these environments can be difficult, as these models are expensive to train and challenging to deploy due to their size (up to billions of parameters). We present FLAME, a transformer-based model trained exclusively on Excel formulas that leverages domain insights to achieve competitive performance while being substantially smaller (60M parameters) and training on two orders of magnitude less data. We curate a training dataset using sketch deduplication, introduce an Excel-specific formula tokenizer, and use domain-specific versions of masked span prediction and noisy auto-encoding as pre-training objectives. We evaluate FLAME on formula repair, formula completion, and similarity-based formula retrieval. FLAME can outperform much larger models, such as the Davinci (175B) and Cushman (12B) variants of Codex and CodeT5 (220M), in 10 of 14 evaluation settings for the repair and completion tasks. For formula retrieval, FLAME outperforms CodeT5, CodeBERT, and GraphCodeBERT.

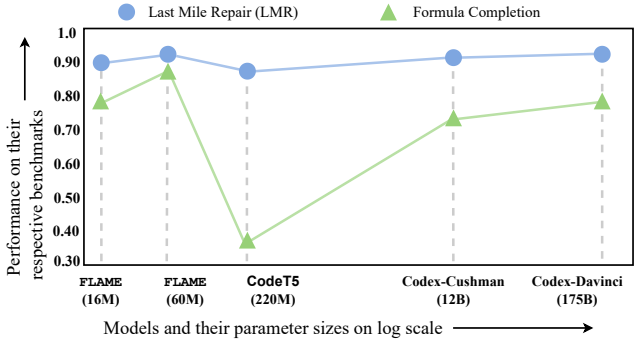

Figure 1: Model performance (top-5 candidate cutoff) for last-mile repair and completion on the benchmark introduced by (Joshi et al. 2022) and our new benchmark for completion. Note that Codex-Davinci results are few-shot, and completion is zero-shot for all systems except CodeT5. Completion results are the fraction of benchmarks successfully completed (based on sketch match metric) given a 90% prefix.

Introduction

Spreadsheets remain an important data management and processing tool for end-users (Rahman et al. 2020) and estimates of the number of spreadsheet users are in the billions (Morgan Stanley). However, despite a large user base, spreadsheet environments still do not have access to nearly the same range of productivity tools available for general programming environments. The latter typically have code completion, refactoring, linting, and a wide range of extensions for additional functionality, like generating tests, inserting code snippets, and summarizing code. Many of these recent advancements in programming assistance tools are driven by large language models (LLMs) trained on code (Chen et al. 2021a; Xu et al. 2022; Fried et al. 2022; Nijkamp et al. 2022), and include features for code completion (GitHub 2021), repair (Joshi et al. 2022), and automated reviews (Li et al. 2022).

To capture the complexity and variety of code and comments in different languages, these models need billions of parameters (Codex has 12B and 175B variants), are trained for long periods of time on millions of programs (Incoder was trained on 159GB of code for 24 days), and result in expensive inference due to hardware requirements. For tasks constrained to a spreadsheet formula language without natural language interaction, such as repairing a broken spreadsheet formula, a substantial amount of such models' expressive power is left unused. This raises the question: *can we substantially reduce the model size by focusing on spreadsheets and exclusively on the formula-language used to carry out computations in such an environment?*

In this paper, we present FLAME, a Formula LAnguage Model for Excel trained exclusively on Excel formulas. FLAME is based on T5-small (Raffel et al. 2020) and has only 60 million parameters, yet it can compete with much larger models (up to 175B parameters) on last-mile formula repair and formula completion. Additionally, we compare FLAME to CodeT5, CodeBERT, and GraphCodeBERT on

*Work done while at Microsoft
†Listed in alphabetical order

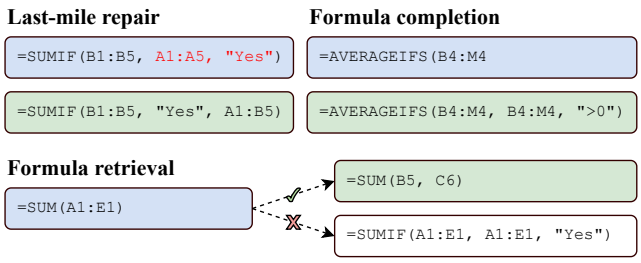

Figure 2: We experiment on three downstream tasks: last-mile repair, formula completion, and retrieving similar formulas. Blue indicates the input and green indicates the expected output. Red text denotes the buggy part of the formula in the repair task, where the user has swapped the correct order of arguments resulting in a type error. FLAME succeeds in all of these cases.

retrieval of similar formulas.

Figure 1 shows a summary of results as a function of model on a public dataset, where FLAME can outperform or compete in both formula completion and repair tasks. Figure 2 shows some examples, solved by FLAME, for these tasks.

There are three main challenges for training a model on Excel formulas: obtaining diverse training data, tokenizing formulas' unique structure, and designing pre-training objectives that teach the model about this structure.

To create a small but varied pre-training corpus, we reduced a corpus of 927M formulas down to 6.1M by comparing formulas based on syntax, removing near-duplicate formulas, which only vary in terms of cell inputs (a common occurrence due to the grid nature of the environment). To account for formula structure, we combine formula-specific tokenization with byte-pair encoding (BPE) to train a tokenizer. To incorporate formula structure into pre-training, we introduce two new domain-specific pre-training objectives: language-aware masked span prediction and user-inspired denoising, which complement two generic objectives (tail-masking and denoising auto-encoding).

We evaluate FLAME on formula completion and repair, showing that FLAME can significantly improve over general code models and can compete with much larger models. Specifically, FLAME outperforms other models in 10 of 14 settings in our repair and completion evaluation. In addition, we also show that FLAME embeddings can be used to retrieve similar formulas more efficiently and effectively than CodeT5, CodeBERT, and GraphCodeBERT.

We make the following contributions:

- We present FLAME, the first language model designed exclusively for Excel formulas. To this end, we introduce domain-specific dataset curation, tokenization, and pre-training objectives.
- We extensively evaluate FLAME and other larger language models on three formula assistance tasks: last-mile repair, formula completion, and formula retrieval.
- We analyze the impact of deduplication, tokenization, and training objectives on FLAME.

Related Work

Multiple language model architectures have been successfully adapted to code to produce popular models such as CodeBERT (Feng et al. 2020), CodeT5 (Wang et al. 2021), Codex (Chen et al. 2021a), PolyCoder (Xu et al. 2022) and others. These models are trained on multiple programming languages and use language-agnostic pre-training objectives. In contrast, FLAME focuses on a single domain and uses domain-specific objectives. BART (Lewis et al. 2020) and UL2 (Tay et al. 2022) introduced similar objectives in the domain of natural language.

To train FLAME, we perform syntax-aware deduplication within workbooks to avoid repeating similar formula structures typical in spreadsheet environments. Allamanis (2019) highlighted the related problem of duplication in traditional sources used for training machine learning models for code.

We evaluate FLAME's ability to perform last-mile formula repair (Bavishi et al. 2022), completion, and similar formula retrieval. Prior work in the repair domain includes DeepFix (Gupta et al. 2017), BIFI (Yasunaga and Liang 2021), Dr.Repair (Yasunaga and Liang 2020), TFix (Berabi et al. 2021), and RING (Joshi et al. 2022), which use deep learning to perform syntax, compilation, or diagnostics repair in general-purpose programming languages. Popular general autocompletion systems include GitHub Copilot in VS Code (GitHub 2021). There is a long history of language models being used for retrieval tasks (Tao et al. 2006; Zhao and Yun 2009; Song and Croft 1999; Gao et al. 2004). This work is complementary as FLAME is exclusively designed for spreadsheet environments and is only trained on formulas (and not the associated data context).

In the spreadsheet domain, SpreadsheetCoder (Chen et al. 2021b) predicts simple formulas from the data context. HermEs (He et al. 2023) leverages a table encoder (TuTA) to generate a formula using hierarchical decoding. Previous work has also shown that pre-training over table and code data can improve table task performance (Dong et al. 2022; Singh et al. 2023). This body of work trains on spreadsheet table contents for tasks such as in-context formula prediction or general spreadsheet intelligence. In contrast, FLAME exclusively uses formulas and pre-trains a general formula model that we fine-tune for tasks like repair.

Approach

We now describe the FLAME architecture and pre-training process (training data, tokenization and objectives).

Architecture

Encoder models like CodeBERT (Feng et al. 2020) show remarkable code understanding capabilities. Decoder models like CodeGen (Nijkamp et al. 2022) and Codex (Chen et al. 2021a) perform well on code generation. Encoder-decoder models seek to blend these strengths. To facilitate both formula understanding and generation, FLAME uses the T5 (Raffel et al. 2020) encoder-decoder architecture.

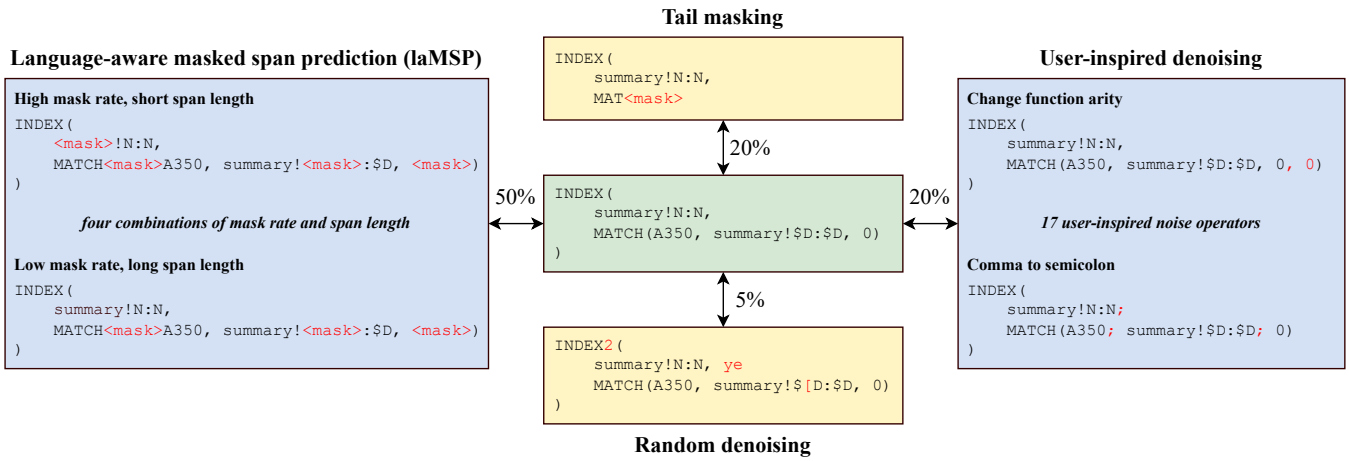

Figure 3: Four pre-training objectives used by FLAME. For each batch, we randomly (with weighted probability) sample one of the four objectives. Generic objectives (tail masking and random noise) are shown in yellow, formula-specific variants (language-aware span masking and user-inspired noise) are shown in blue. For all pre-training objectives, FLAME needs to generate the complete original formula (green).

Training Data

We start with a dataset of 927M formulas drawn from a corpus of 1.8M public Excel workbooks collected from the web.[1] Each workbook contains one or more worksheets, and each worksheet contains zero or more formulas. Formulas are often repeated with only cell reference changes.

We compute formula sketches to preserve a single instance of each unique formula per workbook. In a formula sketch, numeric constants, string constants and cell references are replaced by their token type. For example, the sketch of `=SUM(A1:A10)` is `=SUM(cell:cell)`. After applying sketch deduplication, we are left with 6.1M formulas. Note that applying this globally to the corpus, rather than per workbook, results in only 591K formulas. We found this globally de-duplicated corpus to be insufficient for training as it skews the distribution of formulas (see Evaluation).

Tokenizing Formulas

A popular method for tokenization of code for neural models is byte pair encoding (BPE) (Sennrich, Haddow, and Birch 2016), which iteratively joins consecutive tokens that appear together most frequently until a target vocabulary size is reached. BPE is appealing for the formula domain as it can mitigate the variety in natural language content, such as string literals, named tables, column names, and sheet names. However, applying this procedure without modification can have adverse effects on formulas. For example, in our corpus, `SUM` and `(` are combined to get `SUM(`, which can reduce expressiveness and hurt performance for tasks like repair.

Our tokenizer considers punctuation, whitespace, built-in function names, and digits as individual tokens (Chowdhery et al. 2022). These are often the main lexical marker between separate Excel tokens. We then apply BPE to the remaining parts of formulas like string constants, table names, column names, and sheet names. Because Excel is case insensitive (with the exception of string constants) we convert all input tokens to lowercase to map differently capitalized tokens to a single token, otherwise the same function (like `SUM` and `sum`) will map to different tokens. The final vocabulary size is 16,000 tokens. As a full example, `=SUMIF(B1:B5, "Not available", A1:A5)` is tokenized as `= sumif ( b 1 : b 5 , ␣ " not ␣ available " , ␣ a 1 : a 5 )` with space tokens denoted by ␣.

Pre-training

We use a combination of two existing (tail masking and noisy auto-encoding) and two adapted pre-training objectives (language-aware masked span prediction and user-inspired denoising). An overview is shown in Figure 3.

Tail masking (TM) We perform tail masking (Lewis et al. 2020) on the character level, where the model has to predict the trailing $\{30\%, 40\%, \cdots, 70\%\}$ of characters given the prefix.

Language-aware masked span prediction (laMSP) We apply masked span prediction but force each span to respect Excel lexer boundaries. For example, when an Excel cell reference `BC18` is divided into four tokens `B C 1 8`, we ensure that either all or none of its constituent tokens is masked. Consecutive masked tokens are represented with a single `<mask>` token. Inspired by Mixture-of-Denoisers (Tay et al. 2022), we mask spans of tokens using combinations of high (35%) and low (15%) masking rates, and long (6 tokens) and short (2 tokens) average span lengths.

Random noise (RN) We randomly corrupt (insert, delete, or update) 10% of tokens in the input sequence.

[1] Publicly available sources of spreadsheets that can be used as alternatives include Enron (Hermans and Murphy-Hill 2015), EUSES (Abraham and Erwig 2007), and FUSE (Barik et al. 2015).

User-inspired noise (UN) We introduce 17 noise operators that mirror mistakes that real users might make when writing Excel formulas. These operators are based on a combination of questions on help forums, our analysis of a user-study and our domain expertise. For example, users often write formulas with the incorrect function arity for built-in functions such as `SUMIF`. While pre-training, we randomly choose one of these operators to introduce noise into the input.

Following Tay et al. (2022) and Chang et al. (2020), rather than apply all pre-training objectives on every batch and then combine losses, we pick a single objective for each batch. We use a higher probability for laMSP, as it contains four more combinations of high and low mask rates and long and short span lengths. Empirically, we found these probabilities for objective sampling to perform well: 50% laMSP, 20% TM, 20% UN, and 5% RN. We leave the sequence intact with a 5% probability to mitigate the risk that the model learns to change all sequences (ignoring correctness/completeness). [2]

Downstream Tasks

We consider three different downstream tasks: last-mile repair, formula completion and formula retrieval.

Last-mile Repair

Last-mile repair refers to repairs that require few edits and fix syntax and simple semantic errors, such as wrong function call arity (Bavishi et al. 2022). In this setting, `FLAME` is given the buggy formula as the input sequence, and the task is to generate the intended (valid) formula.

Example 1. *A wrong arity mistake in* `ISERROR`.
Buggy: `=IF(ISERROR(G6 *1.2, "") )`
Repaired: `=IF(ISERROR(G6 *1.2 ), "")`

Fine-tuning We create a fine-tuning dataset for all systems by taking 200K well-formed formulas from Excel help forums[3] and randomly applying our user-inspired noise operators to generate broken formulas.

Evaluation We evaluate all systems on two benchmarks. We use the 273 labeled Excel formulas used in recent last-mile repair literature (Joshi et al. 2022), which was sourced from Excel help forums, and refer to this benchmark set as **Forum**. In addition, we reserve a split of randomly sampled 500 formulas derived using the same procedure as our fine-tuning dataset to create a **Synthetic** benchmark set. We report the fraction of exact matches in top-1 and top-5 candidates for each system. Before comparing, we normalize formulas by capitalizing cell references and identifiers, and removing whitespace tokens.

[2] A technical appendix with operator details and evaluation dataset information is available at https://github.com/microsoft/prose-benchmarks/tree/main/FLAME

[3] We use mrexcel.com and excelforum.com for curating fine-tuning dataset for last-mile repair task.

Formula Completion

Code completion (given a prefix) is a popular code task for language models. We perform this task on formulas.

Example 2. *Formula completion.*
Prefix: `=B2<=EDATE(`
Completion: `=B2<=EDATE(TODAY(),-33)`

Fine-tuning We curated a fine-tuning dataset for completion by tokenizing 189k formulas and sampling a prefix length of $\{0.2, \cdots, 0.7, 0.8\}$ fraction of tokens.

Evaluation We evaluate completion on a single benchmark, consisting of the 273 ground truth formulas from the last-mile repair Forum benchmark. We predict completions given a $\{0.5, 0.75, 0.90\}$% prefix. Some parts of formulas are hard to predict without more context (Guo et al. 2021), such as cell references, sheet names, string literals, and numbers. Therefore, in addition to **exact match**, we also consider **sketch match** for completion with respect to the ground truth. For example, in Example 2, the number `-33` is highly contextual, so in a sketch we match with its token type (`number`).

Similar Formula Retrieval

A key task in data management is to perform efficient information retrieval based on a user query. In this downstream task, we embed a formula to retrieve similar formulas. We use mean pooling to convert token embeddings from `FLAME`'s encoder to a fixed-size embedding (Reimers and Gurevych 2019).

Example 3. *Given three formulas*

```
VLOOKUP($A1, $A$1:$AY$132, 42, FALSE)
VLOOKUP(P6, 'Other'!$A$3:$C$6, 3, FALSE)
C1-VLOOKUP(A1, 'F'!$A$3:$D$16, 4, FALSE)
```

the edit similarity is higher between the first two formulas compared to the first and third. The embedding similarity should reflect this.

Fine-tuning We sample 10K formulas from the public Enron spreadsheet corpus (Hermans and Murphy-Hill 2015) and mask constants. We then compute all pairwise similarities using the Levenshtein edit similarity over lexer tokens. Instead of fine-tuning the whole encoder, we train two fully connected layers that transform the embeddings of a pair of formulas to make their cosine similarity match the edit similarities (Houlsby et al. 2019).

Evaluation We evaluate formula retrieval on a collection of 1000 formulas from the Enron corpus—also pre-processed as in the previous section. We compare the Pearson's correlation between the pairwise cosine similarity over embeddings and the (desired, but expensive) target token edit similarity.

Evaluation

We perform experiments to answer the following research questions:

- **RQ1**: How does `FLAME` perform on formula repair, completion, and similarity formula retrieval compared to substantially larger language models?

System	Architecture	# parameters
Codex-Cushman	Decoder	12 billion
Codex-Davinci	Decoder	175 billion
CodeT5 (base)	Encoder-Decoder	220 million
FLAME (ours)	Encoder-Decoder	60 million

Table 1: Architecture and size comparisons for models

- **RQ2**: How do design decisions (data curation, model size, objectives, and tokenizer) affect FLAME's downstream performance?

Baselines and Configurations We compare FLAME to the (much larger) models summarized in Table 1. For Codex baselines, we use nucleus sampling (Holtzman et al. 2019) (temperature = 0.7) and sample 50 sequences per task. We sort these sequences based on their average token log probabilities (Joshi et al. 2022). For CodeT5, we use beam search (width = 50), and we consider the top 50 sequences.

Training Details

We pre-train FLAME for 10 epochs and fine-tune CodeT5 and FLAME on a cluster with 16 AMD MI200s, 96 cores and 900 GB RAM. We use an AdaFactor optimizer with 1e-4 learning rate, clip factor of 1.0, and a linear learning rate schedule with 100 warm-up steps. We fine-tune FLAME for 2 epochs for repair and completion and fine-tune CodeT5 for 25 epochs with a patience of 5 epochs. We fine-tune FLAME and others for 10 epochs for the formula retrieval experiments. For fine-tuning, we use a weight decay of 0.1. We carry out all Codex experiments on a cluster with 8 V100s, 40 cores, and 672 GB RAM. For Codex fine-tuning, we use LoRA (Hu et al. 2021).

RQ1: Downstream Performance

We now compare FLAME to other language models, including substantially larger ones, on three formula intelligence tasks.

Last-mile repair We fine-tune FLAME, CodeT5, and Cushman and use few-shot prompts with three shots for Davinci. Even though noisy auto-encoding closely resembles last-mile repair, we find that fine-tuning FLAME helps direct it towards a particular task.

We summarize the results in Table 2. Overall, the Forum benchmark is more difficult, with all models scoring lower at top-1 compared to the Synthetic benchmark (with the exception of Davinci, which may have observed more forum content in its training). Because FLAME's pre-training incorporates noise operations that are inspired by real-user mistakes, we find that FLAME achieves similar performance to the much larger Davinci model at both top-1 and top-5 on the Forum benchmark. Cushman, which we fine-tune on similar noise operations, shows the most consistently strong performance. We also find that a general-purpose code model, like CodeT5, despite being fine-tuned on spreadsheet formula errors substantially lags on real user mistakes.

The bottom section of Table 2 shows performance without fine-tuning of FLAME, and without examples for Codex. Because of the denoising objectives, FLAME performs better than Codex baselines. Especially at top-1, fine-tuning helps FLAME perform the right task.

	Forum		Synthetic	
	T@1	T@5	T@1	T@5
Cushman	**0.79**	0.88	**0.87**	**0.93**
Davinci (FS)	0.76	**0.89**	0.54	0.77
CodeT5-base	0.70	0.84	0.84	0.90
CodeT5-small	0.72	0.83	0.82	0.89
FLAME	0.76	**0.89**	0.83	0.91
Cushman (ZS)	0.55	0.85	0.41	0.63
Davinci (ZS)	0.60	0.82	0.51	0.75
FLAME	**0.71**	**0.88**	**0.74**	**0.85**

Table 2: Performance for last-mile repair. The top section reports fine-tuned and few-shot (FS) results, the bottom section reports zero-shot (ZS) and un-fine-tuned results. With fine-tuning FLAME outperforms larger models in the Forum benchmark at top-5, and comes in second at top-1. Without fine-tuning or examples, FLAME outperforms all other models. (**best** and second best)

In Figure 4, we show examples where FLAME gets the correct fix and other models do not, and vice versa. We note that in some cases, FLAME's fixes appear to be more natural, but fail to match the user's ground truth repair.

FLAME's substantially smaller model size can facilitate model training and deployment. To demonstrate this empirically, we consider the average inference latency for the last-mile repair task. We focus on this task as it can be directly translated to a user task (repairing formulas) and in contrast to completion, where model performance may vary based on the length of prefix, all models here have the same input. We carry out all measurements on a CPU to emulate client-side deployment, where a GPU may not be available. Figure 5 shows each model's last-mile repair rate (with a top-5 cutoff) and the average latency (seconds) to generate repair candidates for the Forum dataset. Cushman and Davinci results include network time as they are only available through an API. Davinci is used in a few-shot setting, and all other models are fine-tuned. CodeT5 (60M) and FLAME have the lowest, and comparable, latency. However, FLAME achieves a substantially higher repair rate compared to CodeT5.

Formula completion The auto-regressive nature of Codex models and FLAME's pre-training allows us to evaluate their zero-shot performance[4] for formula completion. Note that we fine-tune CodeT5 for this task as it is pre-trained on smaller span lengths (1 to 5 tokens) and generates special mask tokens (<MASK1>) in a zero-shot setting. We compute exact match and sketch match with top-5 results.

Table 3 summarizes our results. FLAME completions reflect correctly chosen cell references and other constants, and function names, and thus obtains a higher exact match rate across all prefix cutoffs. When we consider sketch match, which ignores cell reference and constant differences, we

[4]We fine-tuned Codex-Cushman and FLAME but observed worse performance, possibly from over-fitting.

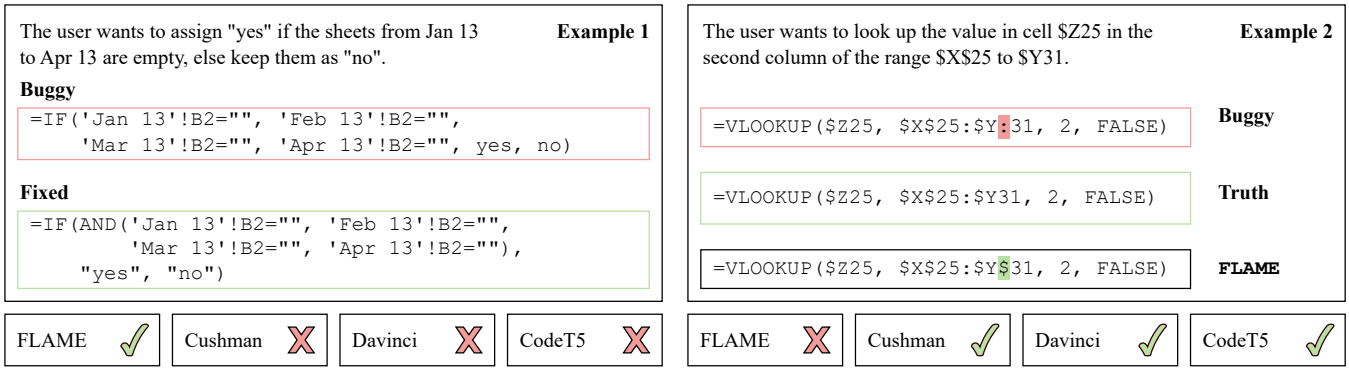

Figure 4: Repair tasks with diverging performance. In Example 1, the user did not use the AND function and missed double quotes around string literals yes and no. FLAME fixes this (in top-5), while other models fail. In Example 2, FLAME's top candidate is syntactically valid but does not match the user's fix, while other models' predictions do.

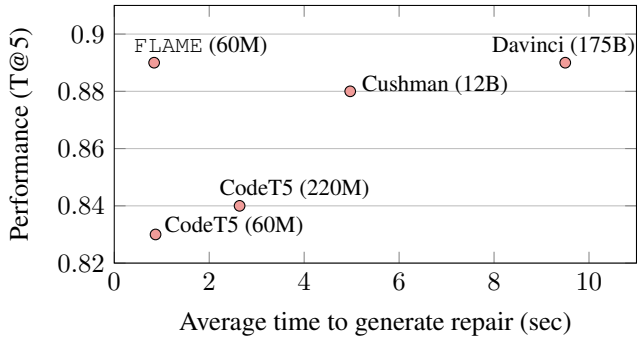

Figure 5: Last mile repair performance (top 5) compared to the average time (secs.) to generate repair candidates on the Forum benchmark.

find that larger models like Cushman and Davinci are competitive in a zero-shot setting. For longer prefixes, they fall behind FLAME. CodeT5 performs poorly as it does not have either benefit: it is not a model with substantial number of parameters nor has it been designed for spreadsheet formulas. Figure 6 shows an example where FLAME can complete with both exact and sketch match, while other systems fail on either or both criteria.

Formula retrieval For formula retrieval, we compare FLAME to CodeT5, CodeBERT (Feng et al. 2020), and GraphCodeBERT (GCB) (Guo et al. 2020). Table 5 shows the correlation between token-edit similarity and embeddings-based cosine similarity. FLAME's embeddings result in a higher Pearson's correlation with our desired (but expensive) token-edit similarity metric, both without fine-tuning and with fine-tuning. As a result of its small size, FLAME can also embed formulas up to 41% faster than the next best model (CodeT5). Surprisingly, CodeBERT and GraphCodeBERT, which were both trained on a code similarity objective, perform worse without fine-tuning. We hypothesize that the encoder-decoder architecture, shared by both CodeT5 and FLAME, leads to more effective formula embeddings.

Model	Exact Match			Sketch Match		
	0.50	**0.75**	**0.90**	**0.50**	**0.75**	**0.90**
Cushman	0.04	0.27	0.61	**0.26**	0.47	0.71
Davinci	0.03	0.31	0.64	0.25	0.53	0.76
CodeT5	0.02	0.10	0.27	0.09	0.20	0.39
FLAME	**0.06**	**0.34**	**0.70**	0.24	**0.55**	**0.84**

Table 3: Zero-shot completion for FLAME, Codex-Cushman and Codex-Davinci, and fine-tuned CodeT5. Given $\{0.50, 0.75, 0.90\}$ as formula prefix, we report the proportion of formulas completed in the top 5. FLAME outperforms all the large language models in the exact match setting and most (2/3) of the sketch match settings. (**best** and second best).

RQ2: Pre-training Design Decisions

We investigate the impact of FLAME's data curation, model size, the use of domain-specific pre-training objectives, and domain-specific tokenizer.

Training data curation Previous work (Lee et al. 2021; Kandpal, Wallace, and Raffel 2022) has shown that deduplication can improve language model performance. We curated our pre-training dataset by performing sketch-based formula deduplication within each workbook. Alternatively, one can perform global deduplication across all workbooks. This results in a pre-training set of 591K formulas. Table 4 shows that training on this smaller corpus results in a worse model (e.g., -8pp on fine-tuned repair, -19pp on exact match completion). While global deduplication allows for faster training, the resulting model observes less naturally occurring repetition across users' spreadsheets.

Model size Table 4 compares FLAME with 16M parameters to the original 60M parameters. We find that performance declines slightly across benchmarks when we reduce model size. However, we note that FLAME-16M still outperforms both Codex models in some tasks, such as zero-shot last-mile repair at top-1 on Forum benchmark, highlighting the potential of leveraging smaller models in specific domains. Note that FLAME-16M is pretrained using the same approach as

	Zeroshot						Fine-tuned	
	LMR		AC (EM)		AC (SM)		LMR	
	Forum	Synth	0.75	0.90	0.75	0.90	Forum	Synth
FLAME (60M)	**0.71**	**0.74**	**0.34**	**0.70**	**0.55**	**0.84**	**0.76**	**0.83**
FLAME (16M)	0.68	0.64	0.24	0.59	0.54	0.76	0.73	0.78
FLAME (60M w/ global deduplication)	0.57	0.56	0.15	0.45	0.41	0.59	0.68	0.76

Table 4: Impact of a smaller model size and global formula deduplication (versus our per-workbook deduplication) on top-1 Last-Mile Repair (LMR) and top-5 Autocompletion (AC) with Exact Match (EM) and Sketch Match (SM). Smaller model performs worse across the board. Global deduplication reduces performance by up to 30 points.

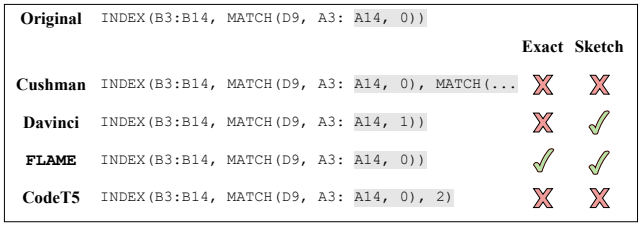

Figure 6: Completion for a 75% prefix. All models correctly predict that the MATCH range goes from A3 to A14 based on the INDEX range B3:B14 in the prefix but result in (otherwise) different completions. FLAME generates the exact ground truth while Codex-Davinci generates a formula with same sketch as the ground truth.

Model	Pre-trained	Fine-tuned	Time (ms)
FLAME	**0.707**	**0.904**	**13.2**
CodeT5	0.688	0.899	22.7
CodeBERT	0.493	0.868	24.7
GCB	0.551	0.881	26.4

Table 5: Correlation of cosine similarity over embeddings and token-edit similarity. FLAME's pre-trained embeddings are substantially more correlated with token-edit similarity. After fine-tuning, both FLAME (60M) and CodeT5 (220M) perform best, but FLAME embeds formulas up to 41% faster.

FLAME-60M with the same amount of data and then fine-tuned for respective downstream tasks.

Pre-training objectives and tokenizer To evaluate the impact of each pre-training objective and domain-specific tokenization, we incrementally add these to a T5 model. Table 6 summarizes these results.

Language-aware MSP (laMSP) causes the most notable improvements (up to 29 percentage points) across all downstream tasks. laMSP forces the noisy auto-encoding task to recover full Excel tokens without leveraging partial tokens, which can leak information, for example, in cell ranges. We then add user inspired denoising, which provides additional benefits for last-mile repair (which is a more difficult version of noisy auto-encoding). Finally, the Excel-specific tokenizer yields improvements across the board, but it specifically shines for completion. We attribute this improvement

Ablations	LMR		C (EM)		C (SM)	
	T@1	T@5	0.75	0.90	0.75	0.90
T5	0.67	0.81	0.07	0.22	0.25	0.37
+ M	0.73	0.85	0.15	0.51	0.35	0.54
+ M + N	0.75	0.87	0.16	0.51	0.34	0.59
+ M + N + T*	**0.76**	**0.89**	**0.34**	**0.70**	**0.55**	**0.84**

Table 6: We incrementally add laMSP (M), user-inspired denoising (N) and Excel-specific tokenizer (T) to T5. The final row represents FLAME. We evaluate on last-mile repair (LMR) and completion (C) with 75% and 90% prefixes for Forum benchmark and report exact (EM) and sketch match (SM). We observe improvements across settings. * is FLAME.

to the consistent tokenization of strings with different capitalization using the Excel-specific tokenizer. For example, we found base T5 to struggle with spreadsheet names and built-in function names, generating code such as Sum!C3 instead of SUM(C3), when given a non-upper-cased sum.

Future Work We provide three pointers for future work. First, fast inference of our smaller model provides an opportunity for more complex search strategies that incorporate symbolic constraints during decoding (Poesia et al. 2022). Second, FLAME embeddings, which we showed are effective for formula similarity, should be explored for embedding tables with formula content (Chen et al. 2021b; Liu et al. 2021). Third, further innovation in tokenization (e.g. separately encoding formula tokens and constants) may help drive model sizes even lower.

Conclusions

We present FLAME, a small (60M parameter) language model for spreadsheet formulas, designed with domain-specific data curation, tokenization, and pre-training objectives. We implemented FLAME for Excel formulas and evaluate on last-mile repair, auto-completion, and similar formula retrieval. We compare to the much larger CodeT5, Codex-Cushman, and Codex-Davinci for repair and completion, and to CodeT5, CodeBERT, and GraphCodeBERT for similar formula retrieval. FLAME obtains the best results in 11 out of 16 evaluation settings, competitively fixing, completing, and retrieving formulas.

References

Abraham, R.; and Erwig, M. 2007. GoalDebug: A spreadsheet debugger for end users. In *29th International Conference on Software Engineering (ICSE'07)*, 251–260. IEEE.

Allamanis, M. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 143–153.

Barik, T.; Lubick, K.; Smith, J.; Slankas, J.; and Murphy-Hill, E. 2015. Fuse: a reproducible, extendable, internet-scale corpus of spreadsheets. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 486–489. IEEE.

Bavishi, R.; Joshi, H.; Cambronero, J.; Fariha, A.; Gulwani, S.; Le, V.; Radiček, I.; and Tiwari, A. 2022. Neurosymbolic Repair for Low-Code Formula Languages. *Proc. ACM Program. Lang.*, 6(OOPSLA2).

Berabi, B.; He, J.; Raychev, V.; and Vechev, M. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, 780–791. PMLR.

Chang, W.-C.; Yu, F. X.; Chang, Y.-W.; Yang, Y.; and Kumar, S. 2020. Pre-training tasks for embedding-based large-scale retrieval. *arXiv preprint arXiv:2002.03932*.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021a. Evaluating Large Language Models Trained on Code.

Chen, X.; Maniatis, P.; Singh, R.; Sutton, C.; Dai, H.; Lin, M.; and Zhou, D. 2021b. Spreadsheetcoder: Formula prediction from semi-structured context. In *International Conference on Machine Learning*, 1661–1672. PMLR.

Chowdhery, A.; Narang, S.; Devlin, J.; Bosma, M.; Mishra, G.; Roberts, A.; Barham, P.; Chung, H. W.; Sutton, C.; Gehrmann, S.; et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Dong, H.; Cheng, Z.; He, X.; Zhou, M.; Zhou, A.; Zhou, F.; Liu, A.; Han, S.; and Zhang, D. 2022. Table pre-training: A survey on model architectures, pre-training objectives, and downstream tasks. *arXiv preprint arXiv:2201.09745*.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; and Zhou, M. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547. Online: Association for Computational Linguistics.

Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; Yih, W.-t.; Zettlemoyer, L.; and Lewis, M. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Gao, J.; Nie, J.-Y.; Wu, G.; and Cao, G. 2004. Dependence language model for information retrieval. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, 170–177.

GitHub. 2021. GitHub CoPilot. https://github.com/features/copilot/. [Online; accessed 09-January-2023].

Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Shujie, L.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

Guo, D.; Svyatkovskiy, A.; Yin, J.; Duan, N.; Brockschmidt, M.; and Allamanis, M. 2021. Learning to complete code with sketches. In *International Conference on Learning Representations*.

Gupta, R.; Pal, S.; Kanade, A.; and Shevade, S. 2017. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI conference on artificial intelligence*.

He, W.; Dong, H.; Gao, Y.; Fan, Z.; Guo, X.; Hou, Z.; Lv, X.; Jia, R.; Han, S.; and Zhang, D. 2023. HermEs: Interactive Spreadsheet Formula Prediction via Hierarchical Formulet Expansion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 8356–8372.

Hermans, F.; and Murphy-Hill, E. 2015. Enron's spreadsheets and related emails: A dataset and analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, 7–16. IEEE.

Holtzman, A.; Buys, J.; Du, L.; Forbes, M.; and Choi, Y. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*.

Houlsby, N.; Giurgiu, A.; Jastrzebski, S.; Morrone, B.; De Laroussilhe, Q.; Gesmundo, A.; Attariyan, M.; and Gelly, S. 2019. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*, 2790–2799. PMLR.

Hu, E. J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; and Chen, W. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Joshi, H.; Cambronero, J.; Gulwani, S.; Le, V.; Radicek, I.; and Verbruggen, G. 2022. Repair is nearly generation: Multilingual program repair with llms. *arXiv preprint arXiv:2208.11640*.

Kandpal, N.; Wallace, E.; and Raffel, C. 2022. Deduplicating training data mitigates privacy risks in language models. *arXiv preprint arXiv:2202.06539*.

Lee, K.; Ippolito, D.; Nystrom, A.; Zhang, C.; Eck, D.; Callison-Burch, C.; and Carlini, N. 2021. Deduplicating training data makes language models better. *arXiv preprint arXiv:2107.06499*.

Lewis, M.; Liu, Y.; Goyal, N.; Ghazvininejad, M.; Mohamed, A.; Levy, O.; Stoyanov, V.; and Zettlemoyer, L. 2020. BART:

Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In Jurafsky, D.; Chai, J.; Schluter, N.; and Tetreault, J. R., eds., *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, 7871–7880. Association for Computational Linguistics.

Li, Z.; Lu, S.; Guo, D.; Duan, N.; Jannu, S.; Jenks, G.; Majumder, D.; Green, J.; Svyatkovskiy, A.; Fu, S.; et al. 2022. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1035–1047.

Liu, Q.; Chen, B.; Guo, J.; Ziyadi, M.; Lin, Z.; Chen, W.; and Lou, J.-G. 2021. TAPEX: Table pre-training via learning a neural SQL executor. *arXiv preprint arXiv:2107.07653*.

Morgan Stanley. 2015. Morgan Stanley Technology, Media & Telecom Conference.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis.

Poesia, G.; Polozov, O.; Le, V.; Tiwari, A.; Soares, G.; Meek, C.; and Gulwani, S. 2022. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*.

Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; and Liu, P. J. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140): 1–67.

Rahman, S.; Mack, K.; Bendre, M.; Zhang, R.; Karahalios, K.; and Parameswaran, A. 2020. Benchmarking spreadsheet systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 1589–1599.

Reimers, N.; and Gurevych, I. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 3982–3992.

Sennrich, R.; Haddow, B.; and Birch, A. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 1715–1725. Berlin, Germany: Association for Computational Linguistics.

Singh, M.; Cambronero, J.; Gulwani, S.; Le, V.; Negreanu, C.; Nouri, E.; Raza, M.; and Verbruggen, G. 2023. FormaT5: Abstention and Examples for Conditional Table Formatting with Natural Language. *arXiv preprint arXiv:2310.17306*.

Song, F.; and Croft, W. B. 1999. A general language model for information retrieval. In *Proceedings of the eighth international conference on Information and knowledge management*, 316–321.

Tao, T.; Wang, X.; Mei, Q.; and Zhai, C. 2006. Language model information retrieval with document expansion. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, 407–414.

Tay, Y.; Dehghani, M.; Tran, V. Q.; Garcia, X.; Wei, J.; Wang, X.; Chung, H. W.; Bahri, D.; Schuster, T.; Zheng, H. S.; Zhou, D.; Houlsby, N.; and Metzler, D. 2022. UL2: Unifying Language Learning Paradigms.

Wang, Y.; Wang, W.; Joty, S.; and Hoi, S. C. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Xu, F. F.; Alon, U.; Neubig, G.; and Hellendoorn, V. J. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 1–10.

Yasunaga, M.; and Liang, P. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, 10799–10808. PMLR.

Yasunaga, M.; and Liang, P. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning*, 11941–11952. PMLR.

Zhao, J.; and Yun, Y. 2009. A proximity language model for information retrieval. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, 291–298.