

# Kotlin ML Pack: Technical Report

Sergey Titov\*  
JetBrains Research  
Paphos, Cyprus  
sergey.titov@jetbrains.com

Mikhail Evtikhiev\*  
JetBrains Research  
Paphos, Cyprus  
mikhail.evtikhiev@jetbrains.com

Anton Shapkin\*  
JetBrains Research  
Paphos, Cyprus  
anton.shapkin@jetbrains.com

Oleg Smirnov  
JetBrains Research  
Amsterdam, the Netherlands  
oleg.smirnov@jetbrains.com

Sergei Boytsov  
JetBrains Research  
Berlin, Germany  
sergey.boytsov@jetbrains.com

Dariia Karaeva  
JetBrains Research  
Limassol, Cyprus  
dariia.karaeva@jetbrains.com

Maksim Sheptyakov  
JetBrains Research  
London, UK  
maksim.sheptyakov@jetbrains.com

Mikhail Arkhipov  
JetBrains Research  
Amsterdam, the Netherlands  
mikhail.arkhipov@jetbrains.com

Timofey Bryksin  
JetBrains Research  
Limassol, Cyprus  
timofey.bryksin@jetbrains.com

Egor Bogomolov  
JetBrains Research  
Amsterdam, the Netherlands  
egor.bogomolov@jetbrains.com

**Abstract**—In this technical report, we present three novel datasets of Kotlin code: **KStack**, **KStack-clean**, and **KExercises**. We also describe the results of fine-tuning CodeLlama and DeepSeek models on this data. Additionally, we present a version of the HumanEval benchmark rewritten by human experts into Kotlin — both the solutions and the tests. Our results demonstrate that small, high-quality datasets (KStack-clean and KExercises) can significantly improve model performance on code generation tasks, achieving up to a 16-point increase in pass rate on the HumanEval benchmark. **Lastly, we discuss potential future work in the field of improving language modeling for Kotlin, including the use of static analysis tools in the learning process and the introduction of more intricate and realistic benchmarks.**

## I. INTRODUCTION

To stay relevant in the world of AI revolution, the language needs to be well-represented in the machine learning community. The less a language is represented, the fewer works are done around it. This leads to a lower quality of generated code, which in turn results in the decreased usage of the language and even poorer representation. To avoid such a situation for Kotlin, we propose the *Kotlin ML Pack*. The goal of the project is to provide all the necessary tools, data, and models to promote code modeling tasks for Kotlin.

In this report, we present three Kotlin datasets:

- **KStack**<sup>1</sup> — the biggest collection of permissively licensed Kotlin files;
- **KStack-clean**<sup>2</sup> — a highly filtered version of KStack containing **twenty-five thousand high-quality examples**;

\*The first three authors contributed equally to this work.

<sup>1</sup>KStack: <https://huggingface.co/datasets/JetBrains/KStack>

<sup>2</sup>KStack-clean: <https://huggingface.co/datasets/JetBrains/KStack-clean>

- **KExercises**<sup>3</sup> — translated and improved version of the **Python exercises dataset** [1].

More information about dataset sizes, filtration techniques, and comparison with other datasets can be found in Section III.

We also provide a version of **HumanEval** [2] **translated into Kotlin by human experts**. In Section IV, we demonstrate problems with the current evaluations of code generation for Kotlin and elaborate on the translation process.

Lastly, we provide multiple checkpoints for fine-tuned models. We took two base models: CodeLlama [3] and Deepseek-coder [4], and for each of them, **performed fine-tuning on variations of the KStack dataset and KExercises**. More information on the fine-tuning process, techniques used during the training, and results can be found in Section V.

We believe that this work provides the bedrock for further research on Kotlin code generation. In Section VII, we describe possible directions for future research, including such promising ideas as using a compiler during the training process and designing a more realistic benchmark for Kotlin.

## II. CURRENT STATE OF THE ART OF KOTLIN CODE GENERATION

First of all, we needed to determine how well the current generation of models can generate Kotlin code. To benchmark Kotlin code generation, we used our human-translated version of the HumanEval benchmark, specifically adapted for Kotlin. More details on the setup of the Kotlin HumanEval are provided in Section IV. Additionally, we test each model on the Python HumanEval to control the used methodology

<sup>3</sup>KExercises: <https://huggingface.co/datasets/JetBrains/KExercises>

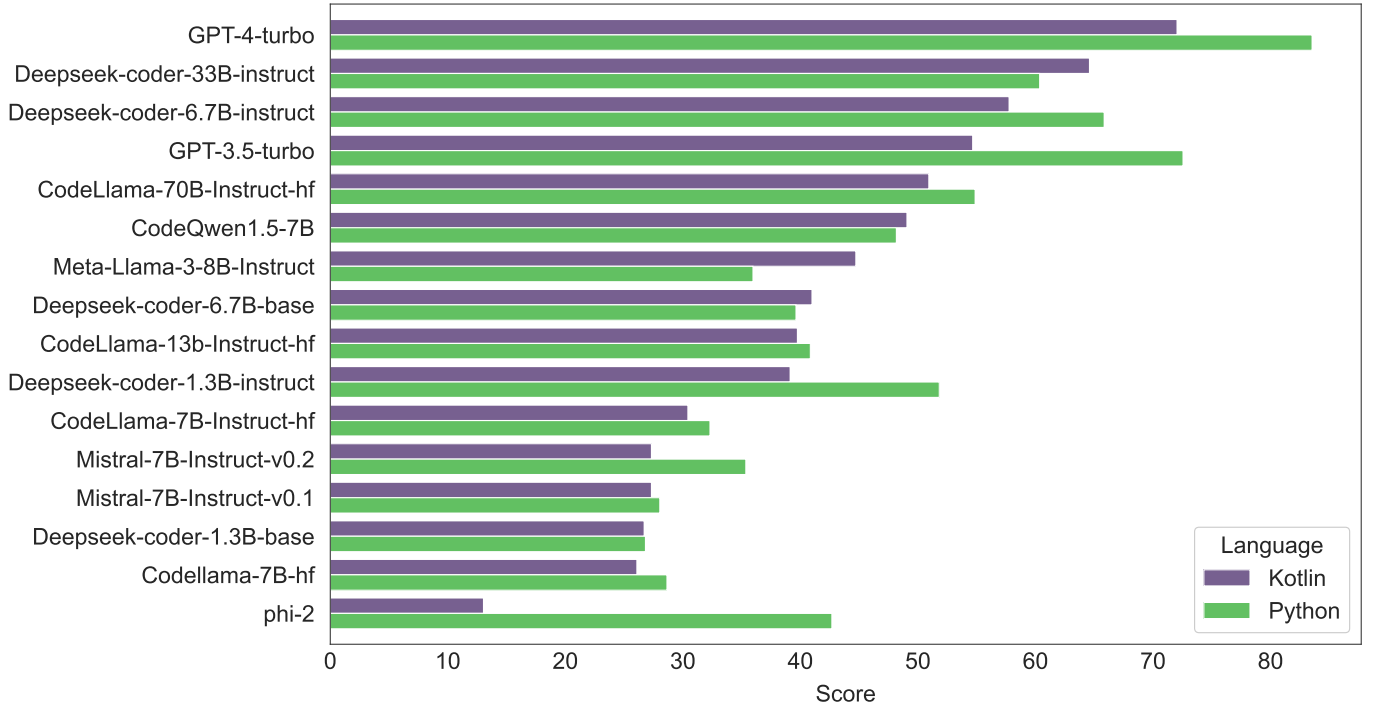


Fig. 1. Kotlin and Python HumanEval Scores for various models.

and to compare the model performance between different programming languages. The findings from these benchmarks are shown in Figure 1.

Interestingly, the performance gap between larger models, such as CodeLlama-70B, DeepSeek-coder-33B, and the GPT family, is not as significant as the difference in their number of parameters. This is in contrast to smaller models, like the 7B models, where the performance gap is more pronounced. Improvements in performance thus do not appear to depend linearly on the parameters number.

Importantly, in most cases, the performance on Kotlin is worse than the performance of these models on Python. One of the reasons for this is the popularity of Python in the research community, which impacts both model training and evaluation. Many models are evaluated on tasks involving Python code, and their performance on Kotlin is not considered. To achieve better quality in Python tasks, models are tuned on Python-specific data. Furthermore, we observe that models fine-tuned on instruction data for Python exhibit improved performance on the HumanEval benchmark for both languages.

Despite the development of specialized coding models, OpenAI’s models continue to set the state-of-the-art standard in coding tasks. This could be attributed not only to the size of the models but also to the fact that natural language comprehension can play a crucial role in code generation from natural language. Training on natural language texts provides contextual understanding that enhances the model’s ability to interpret and generate code more effectively.

As a result, we can conclude that the current generation of models underperforms when solving coding tasks in Kotlin.

While large proprietary models provide good scores on the tasks, there is significant space for improvement for open source models.

### III. KOTLIN DATA

Data is the cornerstone of machine learning in every domain. Python appears to be one of the most represented programming languages in machine learning — not only is Python the third language by popularity in the biggest code dataset — the Stack [5], — there are also multiple high-quality and domain-specific datasets available for it. For example, there are datasets like APPS [6] and Python exercises [1], which contain high-quality Python tasks with solutions, as well as the DS-1000 dataset [7], which focuses on data science-specific tasks for Python.

These datasets were created to teach and measure various aspects of Python language modelling. Following this, we focus on two main datasets for Kotlin: the language corpus and the instruction dataset.

#### A. KStack: Kotlin language corpus

A large language corpus is needed to fine-tune various models on Kotlin. Even though some popular datasets, such as the Stack [5], already have Kotlin code, we present the most complete permissively licensed up-to-date collection of open-source Kotlin code.

We collect repositories from GitHub where the main language is Kotlin, as well as repositories containing Kotlin files that have ten or more stars (as of February 2024). Additionally, we gather repositories with Kotlin files from Stack v1.2 [5].

We identify Kotlin files using *go-enry* [8], and include files with extensions such as *.kt*, *.kts*, and *.gradle.kts*. Next, we use the file content hash to conduct full deduplication, as well as near deduplication using the same method as in Stack v1.2 [5]. We then leave a single representative from each near-deduplicated cluster, picking the file from the repository with the most stars. We also filter the dataset, leaving only repositories with permissive licenses. We use the license information provided by GitHub, and fall back to *go-license-detector* [9] if license information is not available on GitHub. The list of permissive licenses used in KStack can be found at our HuggingFace page.<sup>4</sup> Finally, we filter personal and sensitive information using the *star-pii* model [10].

### B. KStack-clean: Learning the code quality

After collecting the almost full corpus of Kotlin code in KStack, we continue with creating a dataset of specifically high-quality Kotlin code. Using curated datasets to fine-tune a model can provide larger improvements than fine-tuning it on a bigger corpus of non-curated data [11].

To find high-quality data in our dataset, we build a classifier that predicts the quality of the code. We start by labeling a small portion of KStack documents (128K examples) with LLMs (Mistral-7B-Instruct-v0.2 and GPT-3.5-Turbo [12]) in a zero-shot setting. We used pairwise strategy for labeling. In this strategy, we score a pair of files by asking which file has greater “educational value for learning algorithms in Kotlin”. We prepend tokens A and B before each file. To get the score for the file, we calculate the difference between log probabilities of A and B tokens. To avoid the effects of ordering in the prompt, we use the following formula:

$$s(f) = \frac{(s(f, c)_A - s(f, c)_B) + (s(c, f)_B - s(c, f)_A)}{2}$$

where  $f$  is the file, for which we are estimating quality,  $c$  is the highest rated sample,  $s(f, c)_A$  is the probability of choosing the code labeled as A (first argument  $f$ ) over the code labeled as B (second argument  $c$ ).

While scoring each pair is a computationally expensive process, we designed an approximation in a form of three-pass procedure. We begin by scoring a random sample against the full dataset to get a relative quality score for each example. Next, we selected the example with the highest score and repeated the process, scoring it against the full dataset to get a second set of scores. Finally, we repeated the procedure with the highest score sample from the second pass. To obtain the final scores, we averaged the scores from second and third passes (omitting the random sample scores) for the 128 thousand examples.

After we get the scores for the initial sample of 128K examples, we train a binary classifier based on 220M-CodeT5+ [13]. Using the obtained scores, we mark the top 5% of samples as positive examples (high-quality examples), and the rest we mark as negatives. The classifier is trained to

<sup>4</sup>List of licenses in KStack: <https://huggingface.co/datasets/JetBrains/KStack/blob/main/licenses.json>

	Files	Repositories	Lines	Tokens
<b>The Stack v2</b>	2M	109,547	162M	1.7B
<b>KStack</b>	4M	163,310	293M	3.1B
<b>KStack-clean</b>	25,000	3,366	2M	22M

TABLE I  
THE DESCRIPTIVE STATISTICS FOR THE COLLECTED DATASETS. TO CALCULATE TOKEN NUMBERS WE USE CODELLAMA TOKENIZER.

distinguish these examples for three epochs and then applied to the entire KStack dataset. For each sample in the dataset, the log-probability of the positive class is used as a quality estimate. Finally, to obtain KStack-clean dataset, we select 25,000 samples from KStack with the highest probabilities of the positive class.

### C. Comparison with other datasets

As a result of our data collection procedure, we end up with two datasets: *KStack* and *KStack-clean*. In Table I, we compare the descriptive statistics for these two datasets and the Stack v2 [10].

### D. KExercises: Kotlin instructions dataset

Our next goal with regards to the dataset development is creation of the Kotlin instructions dataset. Typically, such datasets consist of sets of instructions or tasks along with their solutions. Training on such data aids models in better comprehending the relationship between natural and programming languages.

There is a number of such datasets available for the Python programming language, like APPS [6] and Python Code Exercises [1]. Moreover, there are also multi-language datasets, such as CodeAlpaca [14] and CommitPackFT [15]. However, these datasets only have a relatively modest representation of Kotlin code or do not contain Kotlin at all.

Kotlin could be considered a low-resource language due to the scarcity of publicly available data and the limited opportunities for improvement using data collected from open-source projects. This motivated us to use synthetic data for instruction tuning. Generating synthetic data can present many challenges, and one of the biggest of them is a low sample diversity. According to Gunasekar et al. [11], generated data should encompass a broad spectrum of coding concepts, skills, and scenarios, varying in difficulty, complexity, and style. Since there are many synthetic Python datasets that have addressed these potential issues, we decided to adapt one of these datasets by translating it from Python to Kotlin, rather than creating an entire dataset from scratch.

For this purpose, we used the CodeExercises dataset [1] that replicates the steps outlined in the work of Gunasekar et al. [11], demonstrating its high quality and effectiveness. We used GPT-3.5-turbo to translate the data from Python to Kotlin (see the translation prompt in Figure 2). We iteratively translated segments of data and monitored the downstream Kotlin generation quality during validation. Additionally, after the translation, we manually reviewed a sample of the

data to ensure the accuracy of the translations. Finally, we compiled an instruction dataset comprising 15K Kotlin tasks (approximately 3.5 million tokens, 335K lines of code). A more detailed investigation of the impact of the size of the synthetic sample on the final quality of the fine-tuned models is left for future work.

System: You are a helpful assistant.  
 User: Rewrite to Kotlin (do not forget about docstring):\n\nPYTHON\_CODE

Fig. 2. Prompt for Python to Kotlin translation.

#### IV. KOTLIN EVALUATION

A vital aspect of machine learning is accurate and efficient evaluation procedures, with HumanEval being a standard for code LLMs evaluations [16], [17], [18], [10]. Though initially designed for the Python programming language, since its debut in 2020, HumanEval has been translated into multiple programming languages. It has also been adapted for use with compiled languages and has been expanded with new tasks.

##### A. HumanEval for Kotlin

While there is a version of HumanEval for Kotlin [19], it required significant improvement before using. Most of the issues we discovered were related to faulty prompts. The most common prompt issue was too generic variable type in the Kotlin function signature. In such cases, it is impossible to apply many built-in Kotlin methods (e.g., one cannot use *sort* method on Kotlin array of *Any* type). The other source of issues were tests. The difference in rounding procedures between Kotlin and Python caused tests to fail on a perfectly reasonable Kotlin program. For example, for *HumanEval task #2*, the task was:

```
Given a positive floating point number,
it can be decomposed into integer part
(largest integer smaller than given number)
and decimals (leftover part always
smaller than 1).
Return the decimal part of the number.
```

For this case, the following solution

```
fun truncate(number : Double) : Double {
    return number - Math.floor(number)
}
```

fails due to the rounding error of less than  $1e-8$ .

To solve the problems with the existing Kotlin HumanEval, we asked human experts to rewrite the HumanEval from scratch. All HumanEval solutions and tests in Kotlin were thus written by an expert competitive programmer with six years of experience in Kotlin, and independently reviewed by a programmer with four years of experience in Kotlin. The tests we implemented are equivalent to the original HumanEval tests for Python, and we fixed the prompt signatures to address the generic variable signature we described above.

##### B. Evaluation setup for code generation

To evaluate the code generation quality for our fine-tuned models, we use our version of HumanEval and the greedy generation strategy. The generation worked as follows.

To prompt the model, we use the prompt from our version of Kotlin HumanEval (*You are an expert Kotlin programmer...*). We use the *generate* method from HuggingFace transformers, greedy generation, and force the model to generate from 128 to 256 new tokens. To speed up evaluation and simplify answer parsing, we use the early stopping token sequence “\n\n”, which corresponds to the end of the Kotlin method according to the Kotlin coding conventions. Additionally, we remove all comments from the answer as these do not affect the performance and can clutter the outputs when one wants to read them. Since some models repeat part of the prompt in the answer (which we do not deem to be a model failure), we find the first definition of the function in the answer (the line starts with “*fun*”), and delete this line and all the lines before from the answer before concatenating it to the prompt.

We then pass the resulting code to the MXEval [19] to score compilation and pass rates. We report the following metrics for each model:

- **Pass@1** — percentage of generations that successfully passed the tests;
- **Test Error Rate** — percentage of generations that failed one or more tests;
- **Compilation Error Rate** — percentage of generations failed on the compilation step;
- **Out of Time Error Rate** — percentage of methods that failed to pass one or more tests in 15 seconds.
- **Runtime Error rate** — percentage of generations failed during runtime. In addition to the usual runtime errors, these include methods which have *TODO()* in the body.

**Pass and error rates as a measure.** Typical usage of a test-based benchmark involves reporting only the pass rate as a measure of the model’s success. The pass rate is a reasonable proxy for assessing code generation from description on simple tasks. In addition to the pass rate, we suggest using **syntax error rate** as a complementary metric that tracks how the model comprehends Kotlin. The syntax error rate is a sum of runtime error rate and compilation error rate. Together with the pass rate, this metric helps to distinguish between the model’s language knowledge and its level of task understanding. On a more granular level, one can improve understanding of the model performance by examining the errors generated by the code. Since the size of the HumanEval benchmark allows one to go over model outputs one by one, we believe that manual checks can help to find the weak spots of the model. This approach also enabled us to find the issues with the original HumanEval benchmark.

##### C. Evaluation setup for code completion

In addition to the evaluation of code generation, we also provide the evaluation of code completion based on a small holdout set extracted from the KStack, consisting of 630



examples. To score completion, we use all models in FIM setup if possible and then calculate the percentage of exact matches for the first generated line.

We believe that completion metrics help us not only measure the performance of a model for a completion task, but also control model for overfitting on the HumanEval tasks. As the base models we use have some Kotlin capabilities from pre-training, fine-tuning may decrease their performance on the code completion tasks.

## V. LEARNING KOTLIN

To showcase our dataset, we train several models using different setups. The section below presents the experimental framework and the main results.

### A. Base models

We use Code Llama 7B [20], Deepseek-coder-6.7B [21] and Deepseek-coder-1.3B [22] as our base models.

Code Llama 7B [20] is an autoregressive language model that uses optimized transformer architectures. It supports infilling text generation, fine-tuning with up to 16K tokens context sizes, and supports up to 100K tokens at inference time.

Deepseek-coder-6.7B base model, implemented by Deepseek [4], is a 6.7B-parameter model with Multi-Head Attention trained on two trillion tokens of natural language texts in English and Chinese. It is also pre-trained on project-level code corpus by employing a window size of 16K tokens and an extra fill-in-the-blank task to support project-level code completion and infilling. Deepseek-coder-1.3B shares the same architecture and training procedure, but with fewer parameters.

### B. Datasets

As described above, we used several of our datasets as part of the training setup.

1) *KStack*: To improve the dataset even more, we developed a number of filtration techniques. First of all, we started with filtering out files that we deem to be low quality:

- We filter out files that belong to low-popular repos (the sum of stars and forks is less than 6).
- Next, we filter out files that belong to the repos with less than five Kotlin files.
- Finally, we remove files that have less than 20 SLOC.

We then clean the remaining dataset entries by removing all non-ASCII entries, removing all package lines such as `package kotlinx.coroutines.channels`, and removing half of the import lines in a probabilistic way (the line is removed with a probability of 0.5). This was done because we found out empirically that many Kotlin files contain many import lines. When one trains the model on them, the model can lock in generating import after import, when given the beginning of the file as the context. As for the package information, it is specific to each project and, outside of the project adaptation domain, will only add noise to the training data.

2) *KStack-clean*: As this dataset already presents a highly preprocessed version of KStack, no further preprocessing was done. However, we study an alternative version of filtering which uses OpenAI GPT-3.5 as a base classifier.

3) *KExercises*: Because of the synthetic nature of this dataset, its size is a hyperparameter and we ran multiple experiments with datasets of varying size, resulting in the optimal size of fifteen thousand examples.

### C. Training setup

For all cases, we perform all fine-tuning on an NVidia A100 GPU in bf16 precision, using AdamW optimizer [23]. We perform validation on the small holdout part of KStack consisting of 630 handpicked examples.

All other hyperparameters are varied between different setups. For precise data on the learning rate, batch size, and learning schedule, please refer to the corresponding model card on Hugging Face.

Additionally, we use several techniques to stabilise the training process:

- **Z-loss**: we use z-loss [16] to control possible divergence of logit outputs. Let  $y_j$  denote the output logits, so the class probabilities are computed as  $p_j = y_j/Z$ , where  $Z = \sum \exp(y_j)$ . The instability may happen closer to the end of the training, when the logits become very negative [24]. The auxiliary z-loss term is given by  $\log^2 Z$ , and it forces  $\log Z$  to remain close to zero.
- **Weight decay**: similarly to the z-loss, weight decay can be used to control output logits divergence, as it prevents weights from growing too large. Since we use standard weight decay PyTorch optimization, we set the weight decay coefficient to be quite large, as it then multiplies a relatively small maximal learning rate [24].
- **Dynamic  $\beta$** : following the Palm technical report [16], we try changing  $\beta_2$  AdamW learning parameter as  $\beta_2 = 1 - k^{-0.8}$ , where  $k$  is step number. While Chowdhery et al. [16] claim this technique helps because rare embedding tokens can have poorly estimated second moments over shorter windows, in our setup it fails to improve model fine-tuning results.
- **Decreasing  $\epsilon$  for AdamW**: we follow the suggestion of Wortsman et al. [24] to decrease the parameter  $\epsilon$  of AdamW, since the default value of  $\epsilon = 10^{-8}$  may dampen the model updates for larger models. We find that in our case, the setting of  $\epsilon = 10^{-16}$  slightly improves both train loss and downstream benchmark scores at no extra costs.
- **Gradient norm clipping**: to limit the impact of data outliers on the training process, we use gradient norm clipping for training on the KStack dataset. We choose gradient clipping so that very few gradients are clipped, avoiding the effective decrease of the learning rate caused by the aggressive gradient clipping.
- **Warm-up**: We use longer warm-up period to reduce the model's learning rate sensitivity. While using warm-up for training Transformer models with an adaptive

optimizer is a de-facto standard by now, using warm-up period length of up to 10% of the train dataset allows training models at a higher learning rate without facing instabilities [24]. We found this technique to be especially useful for the smaller datasets we use.

#### D. Findings

Results for all models can be found in Table II. We observe improvements across all approaches that we used. **We achieve the most significant boost with a combination of Deepseek-coder-6.7B and the fine-tuning on the KExercises dataset, resulting in the pass rate of 55.28%.** Fine-tuning on instructions showed great results on other two base models as well. At the same time, fine-tuning on the full dataset shows weak results, increasing the pass rate for CodeLLama by only 3 percentage points. The clean version of the KStack shows much better results during fine-tuning, but the pass rate is still lower than the one that we achieved with the KExercises dataset. We suggest that this could be caused by a mismatch between the tuning dataset and the evaluation — the KExercises dataset allows the model to learn to follow instructions rather than learn Kotlin. This is supported by the completion rate data, where models tuned on KExercises dataset are outperformed by other models we fine-tune. However, the syntax error rate data for the KExercises-tuned model shows that the fine-tuning improved Kotlin comprehension, and only completion capabilities may have been affected. Thus, while we achieve our most significant score improvements with KExercises dataset, we believe that other datasets are still valuable for learning other tasks, such as completion or generation of documentation.

Additionally, **we performed an experiment with different base classifiers for the data cleaning procedure used in KStack-clean. We found that the same classifier built with OpenAI GPT-3.5 provides significantly lower quality of the sample ranking than Mistral.** This can be clearly seen from the difference in the pass rate dynamics presented in Figure 3. We attribute this to the noise in the log-probabilities of the completion distribution in the OpenAI API. We observe high variability of log-probabilities even when the query is exactly the same. This noise might be artificially added to the responses as a defence against distillation.

## VI. DISCUSSION

### A. Learning the language

1) *Base models:* We used CodeLlama-7B and Deepseek-coder-6.7B as the base models for fine-tuning. While these models are more suited for code generation in Python (see Table II), they nevertheless possess non-trivial Kotlin capabilities and were apparently trained on some amount of Kotlin code. Therefore, training these models in some random setup on a dataset of low-quality Kotlin code can worsen their performance. Correspondingly, the improvements in fine-tuned models caused by the dataset choice are not a low-hanging fruit of making the model learn at least something. We consider these improvements to be a utilization of some

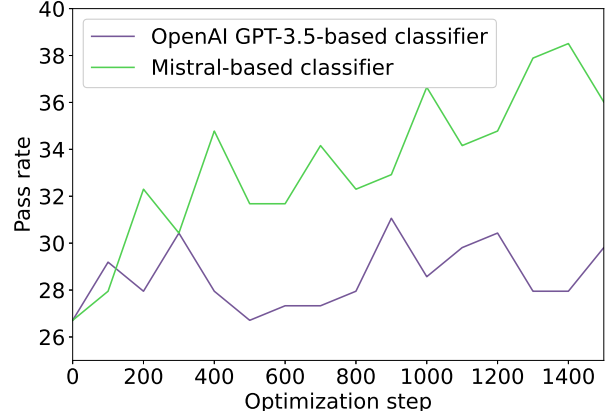


Fig. 3. Pass rate on HumanEval for Kotlin for different filtration strategies of the KStack-clean dataset, finetuning the CodeLlama-7B model.

of the unused model capabilities at a cost of one to 24 NVidia A100 GPU hours, depending on the model and dataset choice.

2) *Tracking model performance:* To assess the changes in model performance, we focus on the pass rate, and use syntax error rate and code completion as the auxiliary metrics. The reasons for that are twofold.

First, the pass rate is a standard measure for the test-based benchmarking. In addition to its benefits we describe in Section IV, using pass rate as a main metric allows a simple apples-to-apples comparison of the dataset impact to the other model improvement techniques used by researchers and practitioners. Second, HumanEval benchmark is a well-established, popular benchmark, which we further cleaned and improved. Our completion benchmark, however, was not verified with the same level of rigor.

Nevertheless, we strongly believe that using just the pass rate to assess the model is not sufficient to gauge its performance. For example, one can accidentally overfit the model on the benchmark test dataset [25] (which is all the more possible once the benchmarks become public), artificially inflating the pass rate. In such a case, a completion dataset will allow to check if there was such an overfit, as the code completion capabilities of the model will be likely strongly affected. The syntax error rate, on the other hand, allows to estimate the model’s language capabilities, and thus augments the code completion score information.

### B. Impact of datasets on model performance

1) *KStack:* **Despite KStack being the largest dataset we used, fine-tuning models on it provides only modest improvements in the pass rate, as well as inconclusive changes in completion rate and syntax error rate.** We surmise that these little improvements are due to the relatively low quality of the dataset, which was not amended by the filtering. This hypothesis is supported by larger improvements on KStack-clean (which we discuss below).

2) *KStack-clean:* Even though KStack-clean dataset is circa 100 times smaller than the full KStack dataset, we observe

Model	Modification	Pass rate	Test fail rate	Compilation error rate	Runtime error rate	Out of time rate	Syntax error rate	Completion (Exact match, first line)
CodeLlama-7B	Base	26.09	50.31	19.25	3.73	0.62	22.98	0.388
	KStack	29.19	47.2	18.63	4.35	0.62	22.98	0.396
	KStack-clean	37.89	42.86	<b>15.53</b>	3.11	0.62	<b>18.64</b>	<b>0.403</b>
	Kexer	<b>42.24</b>	<b>37.89</b>	17.39	<b>1.86</b>	0.62	19.25	0.344
Deepseek-7B	Base	40.99	37.27	11.8	9.32	0.62	21.12	0.403
	Kexer	<b>55.28</b>	<b>29.19</b>	<b>11.18</b>	<b>4.35</b>	<b>0</b>	<b>15.53</b>	<b>0.411</b>
Deepseek-coder-1.3B	Base	26.71	54.04	<b>14.91</b>	4.35	<b>0</b>	19.26	0.403
	KStack	27.95	51.55	16.77	3.11	0.62	19.88	<b>0.404</b>
	Kexer	<b>36.65</b>	<b>44.72</b>	16.15	<b>2.48</b>	<b>0</b>	<b>18.63</b>	0.388

TABLE II  
FINE-TUNING RESULTS

significant pass rate improvements of over 8 percentage points for the models fine-tuned on it. This performance improvement is accompanied by a significant decrease in syntax error rate (15%, or more than 3 points), and a smaller improvement of the code completion score. This success proves the viability of using a classifier to find the best subsample of the dataset for such widely-applicable area as algorithms, and overall shows the model improvement in Kotlin comprehension, completion and code generation. However, it would be interesting to further explore the capabilities of the classifier-based approach.

For instance, it is interesting how the prompt choice affects the model performance. We hypothesize that different prompt choice can yield datasets which are more suitable for improving particular areas of model performance (*e.g.*, knowledge of different packages, ability to generate high-quality tests, etc.).

3) *KExercises*: The results on the KExercises dataset show a large pass rate increase, from 35% to 60% (or ten to 16 points) on the models we fine-tuned. This enormous improvement comes with an improvement in syntax error rate similar to the one for KStack-clean. However, the changes in the completion scores are inconclusive with an improvement for Deepseek-7B and deterioration for the other models. This could signify that fine-tuning on KExercises fits the model for the code generation from prompt, and it would be interesting to perform an exhaustive ablation study to find the limitations of KExercises and other datasets of this kind. Our anecdotal experience with different prompts (such as the one we showcase on the HuggingFace page)<sup>5</sup> shows that the fine-tuning yields a model that understands various kinds of prompts, *e.g.*, raw docstring.

## VII. FUTURE WORK

This work covers only the essential basics for the Kotlin learning pipeline, including data and evaluation. However, we believe that the Kotlin ecosystem can offer much more to the language modeling community — additional data, tools, and benchmarks.

We see the following directions for future research:

- **Learning from tools.** Several papers demonstrate the potential to improve models using feedback from various tools, such as compilers [26]. The Kotlin ecosystem offers

a vast array of tools that can be used in the learning process, including the compiler, linters, and translators. Additionally, modern IDEs possess a robust suite of inspections to check for code correctness, which could be directly integrated into the learning process.

- **Synthetic data.** In this paper, we presented KStack, which is a comprehensive representation of open-source Kotlin code. Despite our plans to keep it up-to-date, we will still miss many cases of Kotlin usage due to its production-oriented nature — we believe that a significant part of Kotlin code is stored in private repositories. As a workaround, we want to focus on generating more synthetic and high-quality code to cover not only coding exercises but also more realistic production tasks.
- **More benchmarks.** While in this work we suggest the most essential benchmark — HumanEval, it can hardly represent real-life Kotlin tasks. In recent years, there has been a new generation of issue-based benchmarks like SWE-Bench [27]. Producing a similar dataset consisting of real-world Kotlin projects and issues would greatly benefit the assessment of whether the current generation of models can be useful in the day-to-day development process.

We hope that producing more artefacts for the Kotlin learning will not only help improve the quality of Kotlin code generation, but also provide researchers more tools and ideas for other languages.

## VIII. CONCLUSION

In this work, we aimed to achieve two primary goals: first, to produce a set of artifacts for the further development of Kotlin language models, and second, to abstract a set of techniques that aid in the development of language models for low-resource languages. We believe that the approach we used in developing the Kotlin ML Pack can be replicated for any other programming language, thereby enhancing current

<sup>5</sup>CodeLlama-7B-Kexer: <https://huggingface.co/JetBrains/CodeLlama-7B-Kexer> open-source language models for them.

## REFERENCES

- [1] “Python exercises,” URL: [https://huggingface.co/datasets/jinaai/code\\_exercises](https://huggingface.co/datasets/jinaai/code_exercises), 2023.
- [2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating large language models trained on code,” 2021.
- [3] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.
- [4] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, “Deepseek-coder: When the large language model meets programming—the rise of code intelligence,” *arXiv preprint arXiv:2401.14196*, 2024.
- [5] D. Kocetkov, R. Li, L. Ben Allal, J. Li, C. Mou, C. Muñoz Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, “The stack: 3 tb of permissively licensed source code,” *Preprint*, 2022.
- [6] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, “Measuring coding challenge competence with apps,” *NeurIPS*, 2021.
- [7] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, “Ds-1000: A natural and reliable benchmark for data science code generation,” in *International Conference on Machine Learning*. PMLR, 2023, pp. 18 319–18 345.
- [8] “go-enry,” URL: <https://github.com/go-enry/go-enry>, 2024.
- [9] “go-license-detector,” URL: <https://github.com/go-enry/go-license-detector>, 2021.
- [10] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, “Starcode 2 and the stack v2: The next generation,” *arXiv preprint arXiv:2402.19173*, 2024.
- [11] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. D. Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi, A. Salim, S. Shah, H. S. Behl, X. Wang, S. Bubeck, R. Eldan, A. T. Kalai, Y. T. Lee, and Y. Li, “Textbooks are all you need,” 2023.
- [12] “Openai. introducing chatgpt,” <https://openai.com/index/chatgpt/>, accessed: 2024-04-01.
- [13] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. H. Hoi, “Codet5+: Open code large language models for code understanding and generation,” *arXiv preprint*, 2023.
- [14] S. Chaudhary, “Code alpaca: An instruction-following llama model for code generation,” <https://github.com/sahil280114/codealpaca>, 2023.
- [15] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. von Werra, and S. Longpre, “Octopack: Instruction tuning code large language models,” *arXiv preprint arXiv:2308.07124*, 2023.
- [16] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, “Palm: Scaling language modeling with pathways,” *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.
- [17] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.
- [18] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “Starcode: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.
- [19] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang, S. K. Gonugondla, H. Ding, V. Kumar, N. Fulton, A. Farahani, S. Jain, R. Giaquinto, H. Qian, M. K. Ramanathan, R. Nallapati, B. Ray, P. Bhatia, S. Sengupta, D. Roth, and B. Xiang, “Multi-lingual evaluation of code generation models,” 2022. [Online]. Available: <https://arxiv.org/abs/2210.14868>
- [20] “Code llama model card,” URL: [https://github.com/meta-llama/codellama/blob/main/MODEL\\_CARD.md](https://github.com/meta-llama/codellama/blob/main/MODEL_CARD.md), 2024.
- [21] “Deepseek-coder-6.7b model card,” URL: <https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-base>, 2024.
- [22] “Deepseek-coder-1.3b model card,” URL: <https://huggingface.co/deepseek-ai/deepseek-coder-1.3b-base>, 2024.
- [23] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” in *International Conference on Learning Representations*, 2017. [Online]. Available: <https://api.semanticscholar.org/CorpusID:53592270>
- [24] M. Wortsman, P. J. Liu, L. Xiao, K. Everett, A. Alemi, B. Adlam, J. D. Co-Reyes, I. Gur, A. Kumar, R. Novak *et al.*, “Small-scale proxies for large-scale transformer training instabilities,” *arXiv preprint arXiv:2309.14322*, 2023.
- [25] R. Schaeffer, “Pretraining on the test set is all you need,” *arXiv preprint arXiv:2309.08632*, 2023.
- [26] D. Zan, A. Yu, W. Liu, D. Chen, B. Shen, W. Li, Y. Yao, Y. Gong, X. Chen, B. Guan *et al.*, “Codes: Natural language to code repository via multi-layer sketch,” *arXiv preprint arXiv:2403.16443*, 2024.
- [27] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, “SWE-bench: Can language models resolve real-world github issues?” in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: <https://openreview.net/forum?id=VTF8yNQM66>