

# Cross-lingual Transfer in Programming Languages: An Extensive Empirical Study

**Razan Baltaji**

*Department of Electrical and Computer Engineering  
University of Illinois at Urbana-Champaign*

*baltaji@illinois.edu*

**Saurabh Pujar**

**Louis Mandel**

**Martin Hirzel**

**Luca Buratti**

*IBM Research*

*saurabh.pujar@ibm.com*

*lmandel@us.ibm.com*

*hirzel@us.ibm.com*

*luca.buratti2@ibm.com*

**Lav R. Varshney**

*Department of Electrical and Computer Engineering  
University of Illinois at Urbana-Champaign*

*varshney@illinois.edu*

**Reviewed on OpenReview:** <https://openreview.net/forum?id=1PRBHKgQVM>

## Abstract

Large language models (LLMs) have achieved state-of-the-art performance in various software engineering tasks, including error detection, clone detection, and code translation, primarily leveraging high-resource programming languages like Python and Java. However, many critical languages, such as COBOL, as well as emerging languages, such as Rust and Swift, remain low-resource due to limited openly available code. This scarcity hampers the training and effectiveness of LLMs for these languages, increasing software maintenance costs and stifling innovation. Addressing this gap, we investigate the potential of transfer learning to enhance LLM performance on low-resource programming languages by leveraging data from high-resource counterparts. Our extensive empirical study evaluates transferability across 10 to 41 programming languages and five key tasks: code generation, clone detection, code repair, solution domain classification, and error detection. Additionally, we develop a performance prediction model to guess the best source languages for a given target and task, and analyze the features that influence transfer performance. We further replicate a representative subset of experiments with a larger model to test the generalizability of our conclusions to contemporary large-scale LLMs. Our findings demonstrate that cross-lingual transfer significantly outperforms zero-shot learning, with effectiveness varying based on both source and target languages. Languages such as Java and Go emerge as the best targets, while Kotlin and JavaScript are excellent sources. Furthermore, our model reliably predicts successful transfer sources by considering linguistic and dataset-specific features, offering practical guidance for data acquisition and model training. This work contributes to the development of LLM-driven tools for low-resource programming languages and provides insights into the characteristics that facilitate transfer across language pairs.

## 1 Introduction

Large language models (LLMs) leverage the fact that software code is often readable and repetitive (much like natural language (Hindle et al., 2016)) to achieve state-of-the-art performance on many software engineering tasks such as error detection, clone detection, and code translation (Lu et al., 2021). However, LLMs have thus far been trained and evaluated mainly on *high-resource* programming languages such as Python and

Java with vast amounts of openly available code. Yet there are many *low-resource* programming languages that lack enough openly available code to train LLMs.

COBOL, which supports many critical financial applications and may have over 775 billion lines of code overall (MicroFocus, 2022), is low-resource due to lack of open availability. Notwithstanding their popularity (Zhou et al., 2022; Coursera, 2024), new languages such as Rust and Swift start out low-resource, by definition. Other languages such as R are low-resource because most code in that language has a license that inhibits use for LLM training (Li et al., 2023). Besides being underrepresented in training of LLMs, organizations often spend significant resources on training software developers to work with unfamiliar low-resource languages. This adds to the cost of software maintenance and may also stifle innovation (Edwards & King, 2021). There is a need for state-of-the-art AI tools to enhance developer productivity for low-resource languages.

Recent works have shown the possibility of transfer learning: leveraging data from one programming language to compensate for the lack of data in another (Ahmed & Devanbu, 2022; Yuan et al., 2022; Pian et al., 2023; Cassano et al., 2024). Another recent work has developed several similarity metrics to decide which high-resource programming language dataset can be used to augment fine-tuning data for a target language (Chen et al., 2022). Prior work, however, has used at most 6 programming languages, none of which are truly low-resource. Further, the notion of similarity among programming languages is underexamined: more research is needed to make reliable claims.

To overcome these shortcomings, we examine transferability from source to target language combinations for 5 tasks and for many more languages than prior work (10–41 depending on the task). We start by using training data for a task in a source language to fine-tune a small LLM (for 4 tasks) or to perform in-context learning using a larger LLM (for the 5th task). Then, we feed the respective LLM disjoint test data for the same task on a target language. Using this methodology, we evaluate the results across all source languages, target languages, and tasks. The source languages are, by definition, high-resource languages since we have enough data to train a model on them. Target languages include both high-resource and low-resource ones. The finetuning experiments focus on the four tasks clone detection, code repair, solution domain classification, and error detection. The in-context learning experiments use a fifth task, code generation.

Figure 1 summarizes the methodology for the finetuning part of our empirical study, which builds upon the work of de Vries et al. (2022) by exploring programming languages and multiple tasks. On the left, we start with a pretrained LLM that has seen unlabeled code in several programming languages. Next, for each of four tasks, for each source language with a suitable amount of labeled training data for that task, we finetune the LLM. For each target language with a suitable amount of labeled test data for the task, we evaluate the finetuned LLM. For example, Figure 1 illustrates this with a solution domain classification task, using C as the source language and Rust as the target language, which yields an F1 score of 0.48. Doing this for four different tasks and all their source and target language combinations yields four different heat maps. The methodology using in-context learning on the fifth task is analogous and yields a fifth heat map.

This paper addresses three research questions (RQ1–RQ3).

**RQ1:** *How well does cross-lingual transfer work for a given task across different language pairs?* We answer this question directly by exploring the heat maps.

**RQ2:** *Given a task and target language, how should we pick the source language for best performance?* We answer this question by training a performance prediction model from the language pair features and the ground-truth target labels from the heat maps, which predicts the ranking of source languages. For example, in Figure 1, the performance prediction model for the solution domain classification task might rank C behind Kotlin as a source language for transfer to Rust. Given the cost of training an LLM, the performance prediction model is useful in directing data acquisition efforts and deciding how to spend compute resources.

**RQ3:** *Which characteristics of a language pair are predictive of transfer performance, and how does that depend on the given task?* We answer this question by measuring the importance of language pair features in the prediction model from the previous experiment.

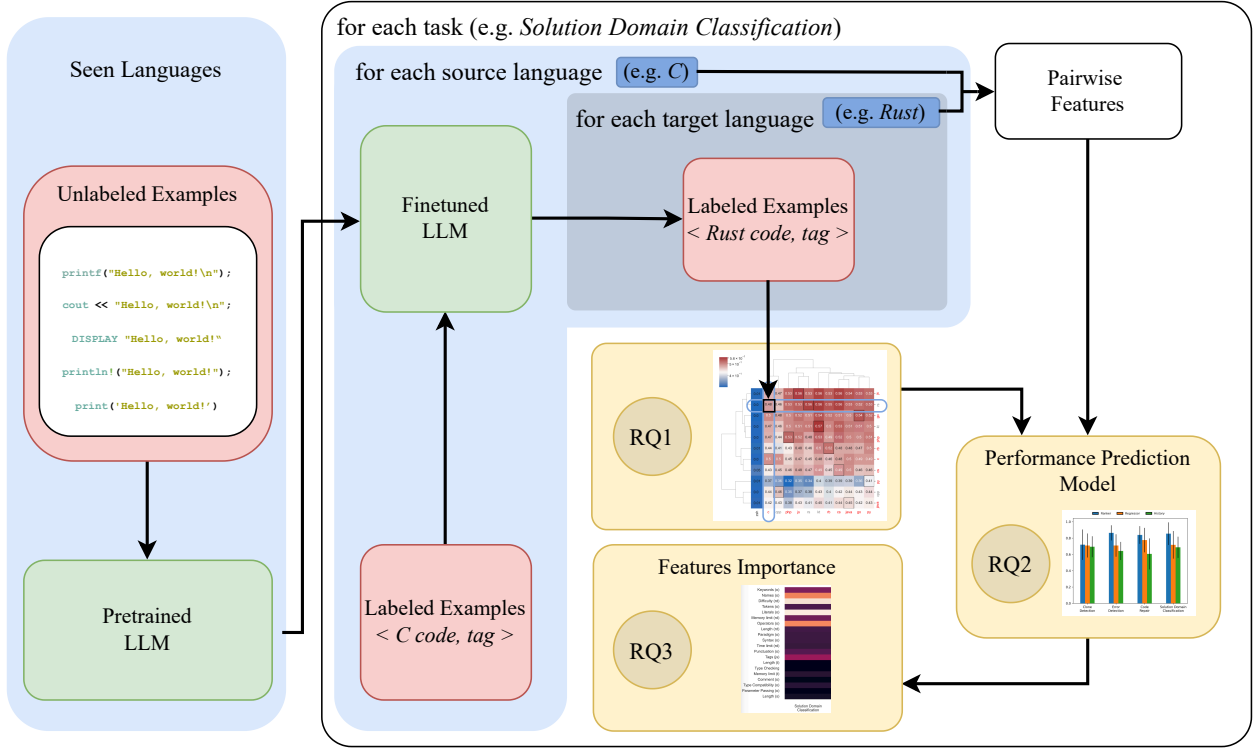


Figure 1: Overview. Consider an LLM pretrained on unlabeled code in multiple seen languages. Finetune on task-specific labeled samples from a source language. For RQ1, test performance on a target language. For RQ2, train a model that ranks transfer performance given features of a language pair. For RQ3, measure how important the features of language pairs are. Repeat for several languages and four tasks.

This paper makes the following contributions:

- Evaluating the pairwise transferability for several programming languages (including low-resource ones) and five tasks using language models.
- Developing a method to predict the best language to transfer from, for different target languages and tasks.
- Characterizing the features of programming language pairs that are predictive of transferability for given tasks.

One goal of this paper is to offer practical guidance to engineers who build LLM-driven software engineering tools towards more informed choices for data acquisition and modeling. A second goal of this paper is to use the lens of transfer learning to shed light on programming language characteristics. Some highlights of our results are as follows:

- Learning transfers well for all tasks and cross-lingual learning transfers better than zero-shot.
- Transfer learning depends on the target language. Java and Go are the best target languages whereas C++ and Python are the worst.
- Transfer learning also depends on source language used for finetuning. Kotlin and JavaScript are the best source languages, C++ is the worst.

- We can reliably predict which source languages will perform well on a given task by considering linguistic, synthetic, dataset-specific, and model-specific features.
- Different tasks rely on different features; for some, keywords and names are most important.

Assuming software engineers increasingly benefit from LLM support, we believe this paper will help democratize such support into low-resource settings and help inspire more research in this area.

## 2 Related Work

**Cross-lingual transfer for natural languages.** Lin et al. (2019) use 4 tasks to study transfer among up to 60 natural languages and explore how important features of language pairs are for predicting how well learning transfers between them, finding that feature importances vary a lot across tasks. Lauscher et al. (2020) use 5 tasks to study transfer among up to 15 languages and argue that while zero-shot performance can work for low-level tasks, higher-level tasks benefit from at least a few target-language shots. de Vries et al. (2022) use 1 task (POS tagging) with 65 source and 105 target languages and study the effects of language families and writing systems, with Romanian emerging as a particularly good source language. Ahuja et al. (2022) use 11 tasks with 1 source and varying numbers of target languages, confirming earlier findings that feature importances vary a lot across tasks. Our paper takes inspiration from these works but differs by studying programming languages instead of natural languages, which have different tasks and different features affecting transferability.

**Cross-lingual transfer for programming languages.** Zhou et al. (2022) call for studying transfer, motivating with 1 task (code completion) and 2 languages (from Hack to Rust). Chen et al. (2022) use 2 tasks (code summarization and search) and 6 languages to study transfer to Ruby, and propose picking the source language based on language similarity. Ahmed & Devanbu (2022) use 3 tasks to study transfer among 6 languages, demonstrating that due to the nature of their tasks, signals from identifiers are highly important for transferability. Yuan et al. (2022) use 1 task (automated program repair) to study transfer among 5 languages, sequentially fine-tuning on multiple languages with innovative tricks to prevent catastrophic forgetting. Pian et al. (2023) use 2 tasks (code summarization and completion) to study transfer among 4 languages, using meta-learning to improve a base learner. Cassano et al. (2024) introduce MultiPL-T, a new semi-synthetic training dataset, and show that it improves transfer for 1 task (NL to code completion) across 19 languages. Our paper also focuses on programming languages, but considers more tasks and more languages to explore conditions for effective transfer.

There is also work on learning across multiple programming languages that has been instrumental in making studies like ours possible, but it differs in that it does not focus on cross-lingual transfer. Unsupervised pretraining on multiple languages has become common, but transferability of supervised tasks has not yet been thoroughly studied; our paper addresses that gap. One could also imagine exploring transfer from multiple source languages to a low-resource target; we leave this to future work.

## 3 Experimental Setup

Most of our experiments are based on finetuning a small model for four tasks as illustrated in Figure 1; in addition, there are also some experiments based on few-shot prompting a larger model without finetuning. For most experiments, given a task, our experimental approach first finetunes the CodeT5 (Wang et al., 2021) model for each source language individually and then tests each finetuned model on all target languages. We applied this approach to four tasks where the number of source languages varies from 6 to 22, leading to 58 finetuned models in total. For each source language, we finetune the model using one A100 GPU for 6 to 30 hours depending on the task. Each model was then evaluated on 11 to 43 target languages producing 1,808 experiments. All results are presented in Figures 2(a)–(d) and analyzed in Section 4. For the few-shot prompting experiments, we generated responses with the LLAMA 3.3 70B-Instruct model via TogetherAI, 2025 using a temperature of 0.8. Those results are presented in Figure 3.

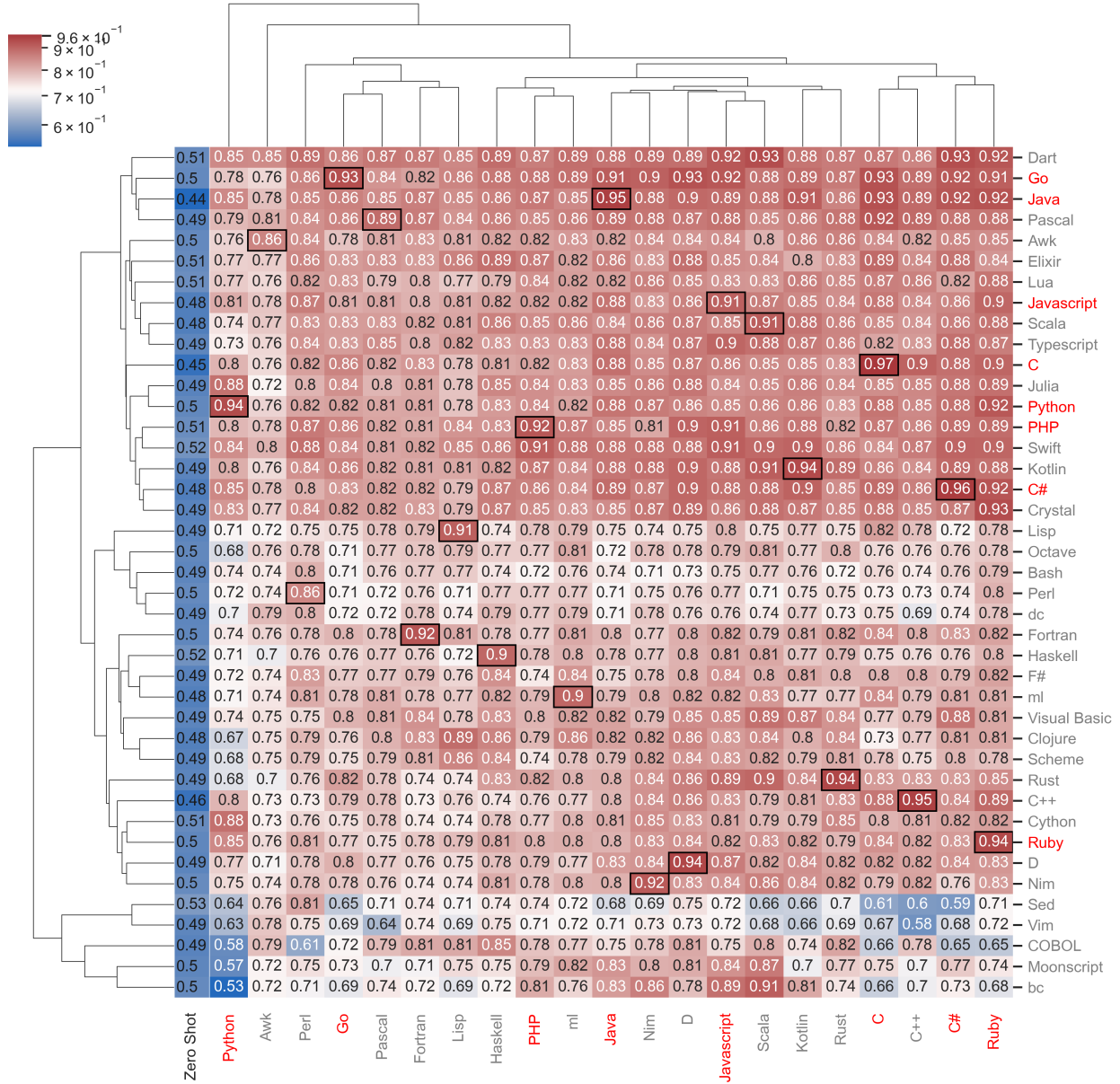
(a) Clone Detection: 21 source languages  $\times$  41 target languages. Metric: F1 Score.

Figure 2: Transfer scores heat map. The figure shows scores for every combination of source and target language. Each column corresponds to a source language, with “Zero Shot” showing zero shot performance, i.e., without fine-tuning on any source language. Each row corresponds to a target language. The languages whose language-name label uses red font were seen during pre-training. Framed black boxes highlight the performance of a source language (column) on itself as the target language (row). The dendrograms show results of hierarchical clustering based on similarity of the performance vectors. The row and column order is also determined by the same clustering.

**Base datasets.** The five tasks studied are derived from three multilingual code datasets: CodeNet, XCodeEval, and MBXP. CodeNet (Puri et al., 2021) consists of about 14M code samples in 55 different programming languages, derived from submissions to online coding judge websites. The dataset comes with benchmarks for code classification and code similarity. XCodeEval (Khan et al., 2023) contains 25M samples

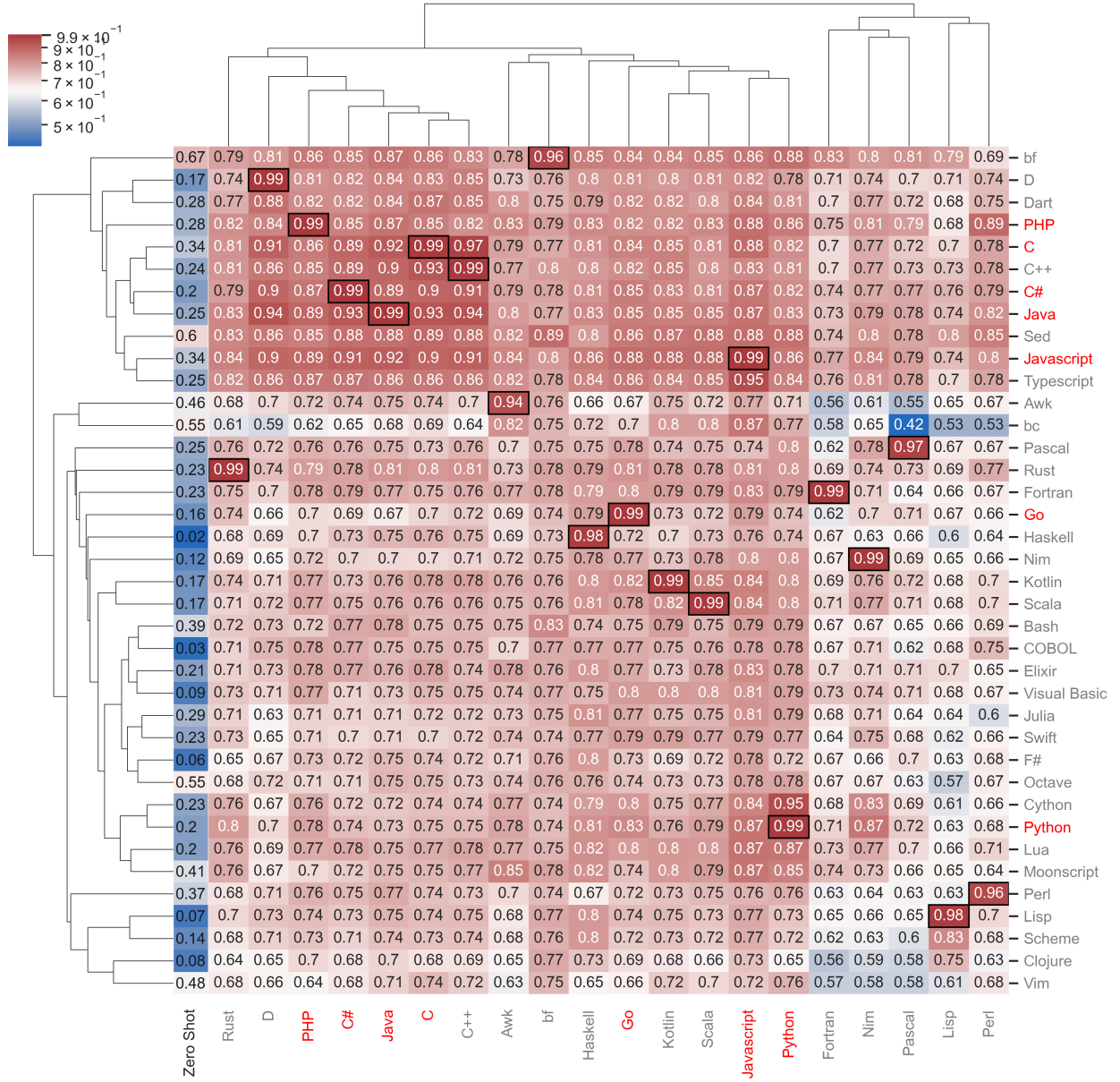
(b) Code Repair: 20 source languages  $\times$  38 target languages. Metric: BLEU score.

Figure 2: (continued; see caption from 2a)

in 11 programming languages from another different online coding judge website. This dataset comes with an execution-based evaluation framework and several different benchmarks: from classification (solution domain classification and error detection) to generation (program synthesis, automatic program repair, and code translation). MBXP (Athiwaratkun et al., 2023) is a multilingual extension of MBPP (Austin et al., 2021) to Java, JavaScript, TypeScript, Go, Ruby, Kotlin, PHP, C#, Scala, C++, Swift, and Perl. A companion code package is provided to perform execution in all supported languages. The dataset is designed for investigating the effectiveness of few-shot prompting, zero-shot translation, as well as the robustness to prompt perturbation. In our experiment, we utilize the few shot prompting setup.

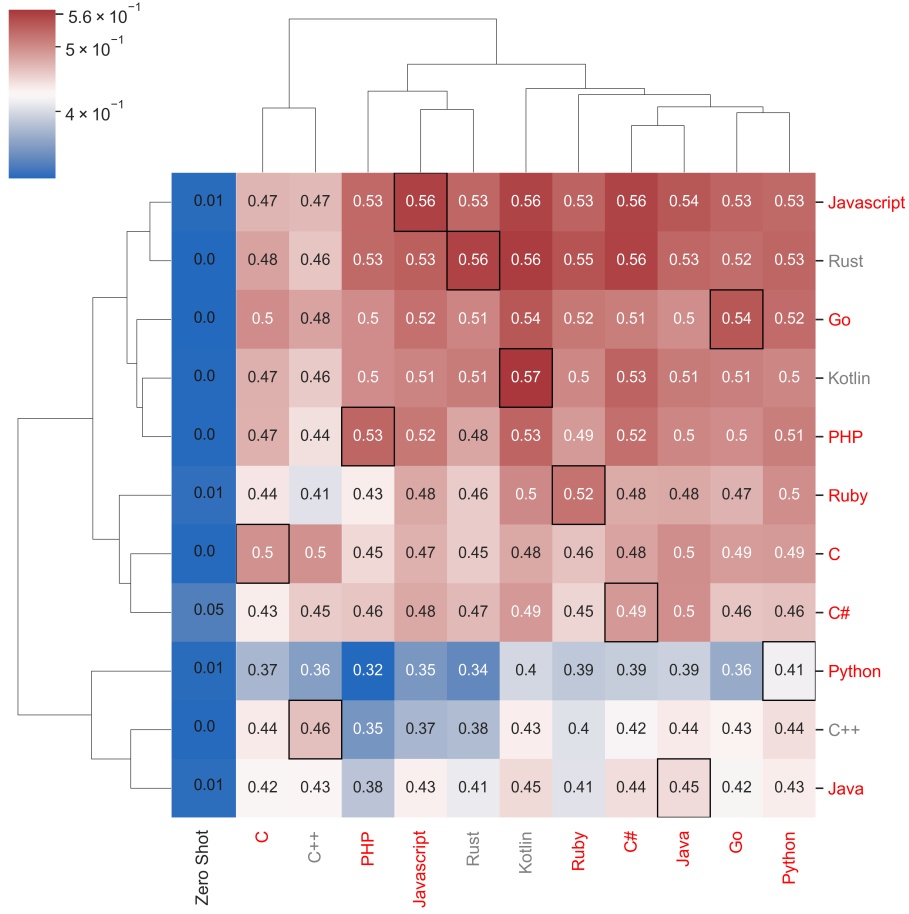
(c) Solution Domain Classification: 11 source languages  $\times$  11 target languages. Metric: F1 score.

Figure 2: (continued; see caption from 2a)

**Tasks.** To study how learning transfers between programming languages, we explore two type of tasks: classification (error detection, solution domain classification, clone detection) and generation (code repair, code generation). For the classification tasks, we follow Lewis et al. (2020) by predicting class labels based on the final decoder hidden state. We use F1 score for evaluating the classification tasks, which is a widely accepted practice since it is good at dealing with class imbalance. We use BLEU score for evaluating the code repair task; while an execution-based metric would provide value, it was infeasible due to the large number of languages in CodeNet (Puri et al., 2021). (Other prominent benchmarks for the code repair task also adopt BLEU score (Lu et al., 2021).) Finally, we use execution-based Pass@1 score for evaluating the code generation task. The execution-based evaluation checks the functional correctness of the generated code by running it against the provided tests.

*Error Detection:* Given a code  $C$  in language  $L$ , do binary classification of whether  $C$  compiles (or can be loaded by an interpreter) without error.

*Solution Domain Classification:* Given a code  $C$  in language  $L$ , do multi-label classification into a set of tags corresponding to algorithmic techniques required to write the solution (e.g., sorting, graphs).

*Clone Detection:* Given two code samples  $C_1$  and  $C_2$  in language  $L$ , do binary classification of whether they are type-IV clones (semantically similar) (Roy & Cordy, 2007). We derived the dataset from CodeNet as follows. Given all combinations of solutions to all problems in language  $L$ , we identify positive samples (clones) as pairs of accepted solutions for the same problem and the others as negative examples. To balance the data across problems, we ensure a ratio of 0.15 of positive samples across different languages.

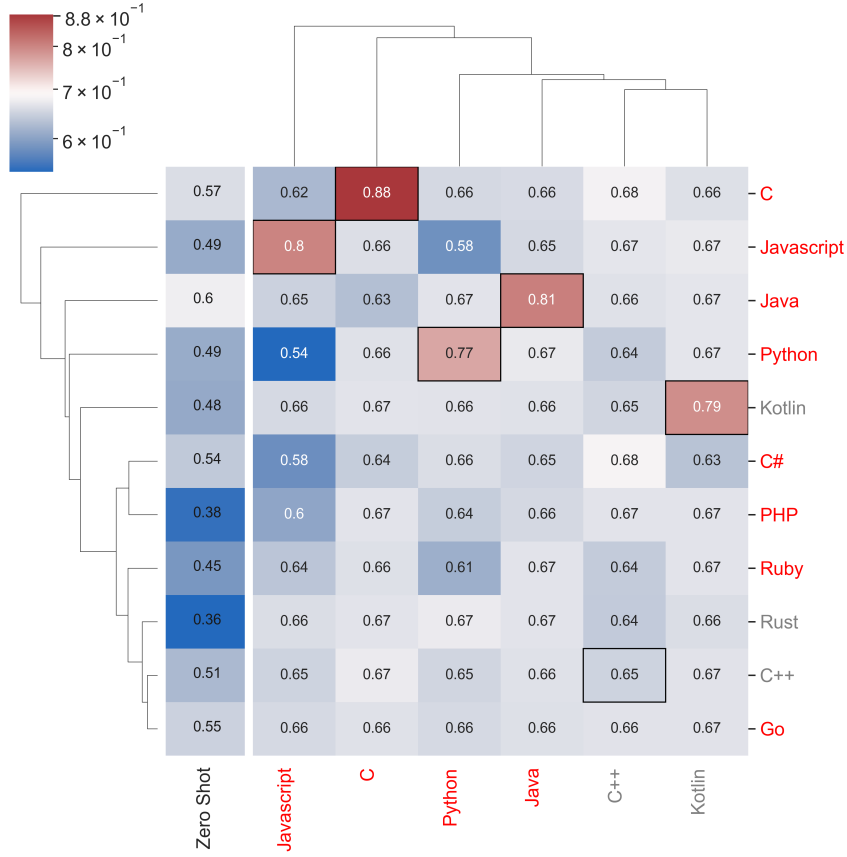
(d) Error Detection: 6 source languages  $\times$  11 target languages. Metric: F1 Score.

Figure 2: (continued; see caption from 2a)

*Code Repair:* Given a buggy code  $C$  in language  $L$ , generate the corresponding repaired code. We derived the dataset from CodeNet, modifying samples by sequentially inserting, removing, or replacing tokens of different types for fixed ratios for different token types.

*Code Generation:* Given a prompt  $P$  consisting of a function signature and a docstring in language  $L$ , the task is to generate the function body completion. In the few shot prompting setup, few-shot prompts consisting of three correct functions from the respective MBXP dataset are provided. Few shot examples precede the function completion prompt for each evaluation.

**Data sampling.** Our objective is to facilitate an extensive study on transfer across the maximum number of language pairs, which can be helpful for those working with low-resource languages. Sources are selected based on the relative performance of fine-tuned checkpoints compared to a random baseline, in order to determine if there is enough signal in the data to facilitate learning. Languages that demonstrate higher performance than the baseline are included as sources. Given variation in the size of datasets for different languages, we follow the sampling procedure of de Vries et al. (2022) to select the number of training samples and identify potential source or high-resource languages. We first finetune the model with datasets of different number of samples  $N=\{10K, 30K, 50K, 70K, 100K\}$ . Depending on the number of training examples  $N_L$  for a language  $L$ , we randomly upsample languages with We filter the set of source languages for a given task based on the relative performance compared to a baseline model. We select sample sizes of 50K for solution domain classification, 70K for error detection, and 100K for clone detection and code repair.

**Models.** Our finetuning experiments are based on CodeT5-base (220M parameters) using the Hugging Face transformers library (Wolf et al., 2020). Code-T5 is an open-source model, pre-trained on eight programming

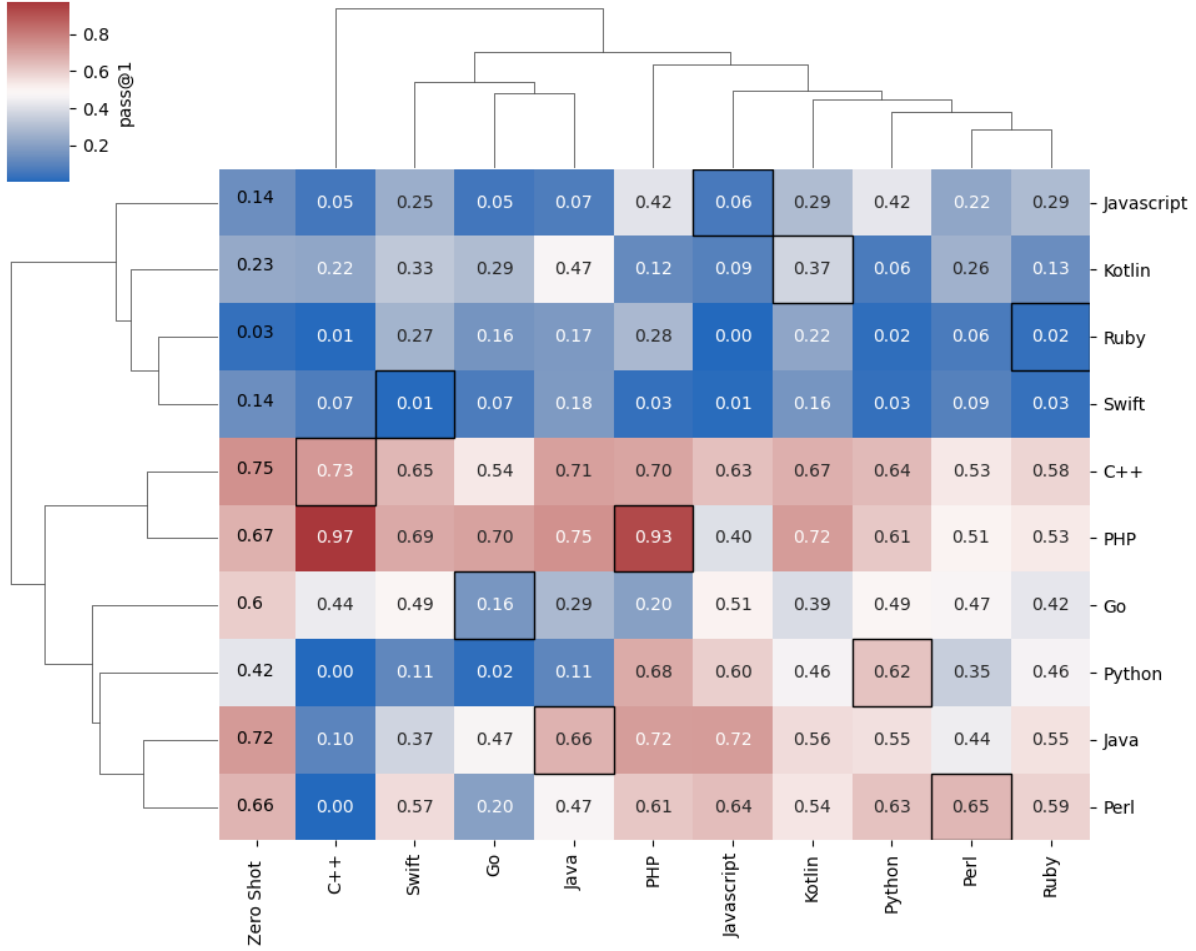
Code Generation: 10 source languages  $\times$  10 target languages. Metric: Pass@1 Score.

Figure 3: Few shot prompting transfer scores heat map. The figure shows scores for every combination of source and target language. Each column corresponds to a source language, with “Zero Shot” showing zero shot performance. Each row corresponds to a target language. Similarly, dendrograms show results of hierarchical clustering and framed black boxes highlight the performance of a source language on itself as the target language.

languages (Ruby, JavaScript, Go, Python, Java, PHP, C, and C#). We consider these languages as *seen languages* and Figure 2 highlights the names of these pre-training languages in red font. Due to its encoder-decoder nature, the model performs well on both code generation and code understanding tasks (Wang et al., 2021) like code repair, error detection, and clone detection, among others. This, along with its relatively small size, makes it a good fit for our empirical study, which requires 58 finetuned models with numerous inference runs each. We keep the same hyperparameters for all the experiments: learning rate of  $2e-5$ , batch size of 8, and the number of epochs set to 20. The few-shot prompting experiment use LLAMA 3.3 70B-Instruct,<sup>1</sup> an instruction-tuned multilingual LLM (70B parameters) (Grattafiori et al., 2024).

**Data Splits** Train and test splits for Error Detection and Solution Domain Classification are provided from the original dataset as described in the dataset statistics from xCodeEval (Khan et al., 2023). For Code Repair, 50,000 training examples and 1,000 test examples are synthetically generated for each language. For Clone Detection, a distinct set of problems is used for train and test splits, including languages with a

<sup>1</sup><https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct>

Language	Train	Test	Language	Train	Test	Language	Train	Test
php	914,924	1,975	ml	175,056	2,008	sed	21,584	1,349
rb	912,159	3,963	f	172,046	1,253	lua	20,688	1,253
hs	843,011	2,045	cpp	150,492	1,792	pyx	18,184	931
py	785,760	4,235	pas	136,961	1,370	dc	17,322	1,706
kt	731,575	1,552	rs	128,271	478	vb	7,003	1,132
js	713,854	1,600	awk	95,312	1,035	octave	5,370	1,476
c	555,924	2,363	jl	88,700	1,737	vim	4,176	745
scala	473,594	1,705	sh	64,899	1,070	cob	3,437	1,038
go	352,181	867	ts	47,414	799	clj	1,058	910
java	266,248	1,338	l	45,986	585	dart	506	480
d	247,954	1,355	swift	38,531	737	moon	242	453
cs	233,059	1,151	fs	31,001	962	ex	330	549
pl	185,685	1,762	scm	22,787	826			
nim	175,839	765	cr	21,733	1,356			

Table 1: Number of train and test samples for Clone Detection for each language.

minimum of 450 test examples. Table 1 shows the number of examples for Clone Detection for each language. In the few-shot prompting experiment, we conduct a cross-lingual evaluation across all language pairs, using nearly 1,000 examples for each. We additionally use the provided synthetic canonical solutions to measure dataset features across overlapping problem solutions.

## 4 Results and Discussion

This section addresses the research questions RQ1–RQ3 stated in the introduction.

### 4.1 Transfer analysis

We explore RQ1 via extensive experiments on every combination of source and target programming language, for each task. The four heat maps of Figure 2 show the results. A *source* programming language is the language used for fine-tuning a model, with samples in the training data. A *target* programming language is the language used to evaluate a model, with samples in the test data. Figure 2 shows source languages on the horizontal axis and target languages on the vertical axis. Monolingual scores (where the source and the target language are the same) are highlighted with bold boxes. The first column from the left indicates zero-shot performance of the base Code-T5 model on different target languages. For RQ1 we analyze model performance trends with respect to task, source language, and target language.

**Task dependency.** The heat maps in Figure 2 show that transferability of source languages varies depending on the task. Visually this can be seen by how the color gradient varies for each task. To quantify this observation, we calculate the mean scores across all the language combinations. The mean cross-lingual scores are 0.78 for clone detection, 0.75 for code repair, 0.65 for error detection, and 0.47 for solution domain classification. The monolingual scores (in bold boxes) are higher (more red) than the cross-lingual scores, averaging 0.91, 0.98, 0.78, and 0.51 for the same four tasks. The zero-shot columns, where the model is not finetuned on any source language, have lower scores (more blue) for all heat maps, averaging 0.49, 0.28, 0.49, and 0.01. Variations in the mean monolingual, cross-lingual, and zero-shot scores across tasks confirm that model performance is task dependent. The difference in overall zero-shot and cross-lingual scores indicates that in the absence of fine-tuning data for any task in any programming language, it is better to finetune with some other language in that task than to use a zero-shot setting. This cross-lingual advantage holds for every task.

Looking at these scores in absolute terms reveals that the choice of tasks also affects transferability. For solution domain classification, the difference in mean cross-lingual and monolingual scores is only 0.04,

Task	Mean Score			
	Mono	Cross	Overall	Zero-Shot
Clone Detection	0.91	0.78	0.79	0.49
Code Repair	0.98	0.75	0.76	0.28
Error Detection	0.78	0.65	0.66	0.49
Solution Domain Classification	0.51	0.47	0.47	0.01

Table 2: Mean Scores by Task. Monolingual implies finetuning where the train and test data language is the same. Cross-lingual implies finetuning where the train and test data languages differ. Overall scores include both monolingual and cross-lingual scenarios. Zero-shot means the performance of the base pre-trained model (CodeT5-220M) on the test set without finetuning.

Target	Clone Detection		Code Repair		Error Detection		Solution Domain		Mean Rank
	cross	0-shot	cross	0-shot	cross	0-shot	cross	0-shot	
Java	0.86	0.44	0.84	0.25	0.66	0.60	0.42	0.009	4.25
Go	0.86	0.50	0.71	0.16	0.66	0.55	0.51	0.003	4.25
Rust	0.80	0.49	0.77	0.23	0.66	0.36	0.53	0.001	4.50
JavaScript	0.83	0.48	0.85	0.34	0.65	0.49	0.52	0.008	4.75
Kotlin	0.84	0.49	0.76	0.17	0.66	0.48	0.50	0.005	5.00
PHP	0.84	0.51	0.82	0.28	0.65	0.38	0.50	0.004	5.25
C#	0.84	0.48	0.82	0.2	0.64	0.54	0.47	0.046	5.50
C	0.83	0.45	0.82	0.34	0.66	0.57	0.48	0.004	5.75
C++	0.79	0.46	0.81	0.24	0.66	0.51	0.41	0.004	7.00
Python	0.83	0.50	0.76	0.20	0.64	0.49	0.37	0.006	8.75

Table 3: Score distribution by target languages common for all tasks. Scores are the mean of the score of every source for a given target language. Mean rank is used to rank the languages based on their ranking for each task. All the languages shown are high-resource target languages for Clone Detection, Code Repair and Solution Domain Classification, which means they are also among the source languages for those tasks. C#, Go, PHP, Ruby, and Rust are low-resource languages for Error Detection. The table shows that all languages benefit from transfer learning. Java and Go seem to benefit the most and C++ and Python seem to benefit the least.

whereas for code repair, it is 0.23. The difference in mean cross-lingual and zero-shot scores for the same two tasks is 0.46 and 0.48. Although overall trends show learning transfers well in all tasks, cross-lingual performance in relation to monolingual performance is highly task dependent. Within each task, model performance depends on the choice of source and target programming language, which we explore next. Table 2 shows the mean score by task.

Learning transfers well for all tasks. Cross-lingual learning works better than zero-shot.

**Target language dependency.** In Figure 2, high-resource target programming languages are represented on both axes. Low-resource target programming languages are represented only on the vertical axis. Models are finetuned only on high-resource programming languages and tested on both high- and low-resource programming languages.

To understand overall trends across all four tasks, we consider ten target programming languages that are common to all four tasks. Six (Kotlin, JavaScript, Java, Python, C, and C++) are high-resource in all four tasks. Four (Go, Rust, PHP, and C#) are low-resource in error detection. For each of these ten languages, for each of the four tasks, we calculate the mean cross-lingual score using all scores in the row of Figure 2 for each language, excluding the monolingual score in the bold box and the zero-shot score in the extreme left. Based on this mean we rank each of the 10 languages for each task, and then calculate the mean rank for

Target Language	Clone Detection		Code Repair		Mean Rank
	cross	0-shot	cross	0-shot	
Dart	0.87	0.51	0.79	0.28	2.0
TypeScript	0.83	0.49	0.83	0.25	3.5
Elixir	0.83	0.51	0.75	0.21	5.0
Lua	0.82	0.51	0.77	0.20	5.0
Swift	0.85	0.52	0.72	0.23	6.5
Visual Basic	0.80	0.49	0.75	0.09	7.5
Cython	0.79	0.51	0.75	0.23	7.5
Julia	0.83	0.49	0.72	0.29	8.0
Moonscript	0.74	0.50	0.75	0.41	9.0
Sed	0.69	0.53	0.85	0.60	9.5
Scheme	0.78	0.49	0.71	0.14	11.5
Shell	0.74	0.49	0.74	0.39	12.0
Clojure	0.80	0.48	0.67	0.08	12.5
COBOL	0.73	0.49	0.74	0.03	12.5
F#	0.78	0.49	0.71	0.06	13.0
Octave	0.76	0.50	0.71	0.55	13.0
bc	0.73	0.50	0.67	0.55	16.5
Vim	0.69	0.49	0.67	0.48	16.5

Table 4: Score distribution for low resource target languages common for Clone Detection and Code Repair. Scores are the mean of the score of every source language for the given target language. Mean rank is used to rank the languages based on their ranking across each task. All languages benefit from transfer learning. Dart, TypeScript benefit the most and bc, Vim benefit the least.

each language. The most transferrable target languages from best to worst are Java, Go, Rust, JavaScript, Kotlin, PHP, C#, C, C++, and Python. Some target programming languages are good at monolingual performance, but much worse at cross-lingual performance. For example, in-language performance for C++ is 0.95, 0.99, and 0.46 for clone detection, code repair, and solution domain classification, whereas its mean cross-language performance is 0.79, 0.81, and 0.41 for the same tasks. This might be explained by the fact that C++ has a reputation as being one of the hardest programming languages to learn for human software developers. All numbers are in Table 3.

We do a similar analysis for low-resource target languages in clone detection and code repair, since these two tasks have a wider and more varied set of low-resource languages. Here, the most transferrable target languages from best to worst are Dart, TypeScript, Elixir, Lua, Swift, COBOL, F#, Octave, bc, and Vim. See Table 4 for details.

For error detection, only a smaller set of languages can be considered low resource. Average scores vary less across different sources with highest being Rust with 0.66 and lowest being C# with 0.64. For solution domain classification there are no low-resource target languages.

While all target languages benefit from cross-lingual training, some benefit much more than others. Among low-resource target languages, Dart and TypeScript benefit the most.

**Most Transferable Source Language.** The number of source languages vary for each of the four tasks. To understand the source language dependency and to identify the most transferrable source language, we start from languages all four tasks have in common (C, C++, Java, JavaScript, Kotlin, and Python). For each task, for each language, we calculate the mean cross-lingual score. Once we have the cross-lingual mean for each language, we rank the source languages based on their mean score for each task. Based on these rankings, we calculate the mean rank, for overall cross-lingual performance of source languages, across all target languages, across all tasks.

Source	Clone Detection		Code Repair		Solution Domain		Error Detection		Mean Rank
	score	rank	score	rank	score	rank	score	rank	
Kotlin	0.8	3	0.78	3	0.5	1	0.67	1	2.0
JavaScript	0.82	1	0.81	1	0.47	4	0.62	6	3.0
Java	0.8	4	0.78	5	0.49	3	0.66	2	3.5
Python	0.73	6	0.79	2	0.49	2	0.66	5	3.75
C	0.8	2	0.78	4	0.45	5	0.66	4	3.75
C++	0.79	5	0.78	6	0.44	6	0.66	3	5

Table 5: Source languages common for all tasks ranked by mean score across target languages and tasks. Score is the mean of the scores of all target languages for a given source language. We see that Kotlin is relatively the best source language and C++ the worst. This result is surprising since Kotlin is not a seen language during pre-training.

Source	Clone Detection		Code Repair		Error Detection		Mean Rank
	score	rank	score	rank	score	rank	
JavaScript	0.81	1	0.82	1	0.63	6	2.67
Java	0.79	2	0.75	5	0.66	1	2.67
Kotlin	0.78	3	0.77	3	0.66	4	3.33
C	0.77	4	0.76	4	0.66	2	3.33
Python	0.71	6	0.79	2	0.65	5	4.33
C++	0.76	5	0.75	6	0.66	3	4.67

Table 6: Source languages common for all tasks ranked by mean score across low-resource target languages and three tasks. Solution Domain Classification is missing because all the target languages for this task are high-resource. Score is the mean of the scores of all target languages for a given source language. We see that JavaScript, Java are relatively the best source languages and C++ the worst.

The most transferrable source languages from best to worst are Kotlin, JavaScript, Java, Python, C, and C++. It may be surprising that Kotlin is the best-performing source language, since it is rarely discussed in the context of training LLMs and is not part of the pre-training corpus for CodeT5 (Wang et al., 2021). Prior work on transfer learning for natural languages (de Vries et al., 2022) yielded a similar surprise, with Romanian as the best-performing source language. A possible explanation could be that Romanian and Kotlin both have strong roots in dominant languages of the past (Latin and Java), while sitting at a cross-roads that helped them absorb many influences. Detailed scores and rankings are in Table 5.

The above analysis considers all target languages, including high-resource target languages. In a more realistic setting, we can ignore high-resource target languages because the monolingual option is available for every high-resource language in every task. If we repeat the above analysis with only low-resource languages, we are left with a smaller set of target languages and only three tasks (all target languages for solution domain classification are high-resource). For the remaining three tasks and six common source languages, the most transferrable source languages from best to worst are JavaScript, Java, Kotlin, C, Python, and C++. JavaScript and Java contribute the most data to CodeT5 pretraining; Kotlin and C++ are absent (Wang et al., 2021). For details see Table 6.

If we consider only clone detection and code repair, we get a much broader set of about 20 common source languages. Repeating the analysis for these two tasks for low-resource target languages only, we find once again that JavaScript is the most transferable and C++ is among the least. More details are in Table 7.

Kotlin is the best source language over all target languages and tasks. JavaScript is the best source language for low-resource target languages. C++ is a poor source language.

Source	Clone Detection		Code Repair		Mean Rank
	score	rank	score	rank	
JavaScript	0.81	2	0.82	1	1.5
Scala	0.81	1	0.77	6	3.5
Haskell	0.81	4	0.78	3	3.5
Kotlin	0.78	11	0.77	5	8
D	0.81	3	0.71	15	9
Java	0.79	9	0.75	9	9
PHP	0.79	7	0.74	13	10
Nim	0.80	5	0.71	16	10.5
Python	0.71	19	0.79	2	10.5
Rust	0.79	8	0.72	14	11
Perl	0.79	6	0.68	17	11.5
C	0.76	15	0.76	8	11.5
Go	0.76	16	0.77	7	11.5
C#	0.77	12	0.74	12	12
bf	0.55	20	0.77	4	12
Fortran	0.78	10	0.67	18	14
Awk	0.76	17	0.75	11	14
C++	0.76	18	0.75	10	14
Lisp	0.77	13	0.67	19	16
Pascal	0.77	14	0.66	20	17

Table 7: Source languages common for Clone Detection and Code Repair ranked by mean score across low-resource target languages. Score is the mean of the scores of all target languages for a given source language. We see that JavaScript, Scala, and Haskell are relatively the best source languages and C++, Lisp, and Pascal the worst.

**Language Pair Performance** The best model performance for each task is almost always in the monolingual setting. However, some cross-lingual pairs do stand out and show very good model performance, comparable to monolingual. Another reason to consider individual language pairs is that from the perspective of developers working with low-resource languages, it would be helpful to know which programming languages act as the best source language.

For clone detection in Figure 2(a), a model finetuned on Ruby does almost as well on Crystal, a language whose syntax is inspired by Ruby. Figure 2(a-b) also shows that surprisingly, Haskell does very well on COBOL, much better than FORTRAN does on COBOL. Lisp does very well on Scheme, which can be expected as Scheme is a dialect of Lisp. Similarly Python does very well on Cython. Figure 2(a-d) also show that C-category languages (C, C++, and C#) do well on each other. Details are in Table 8.

Some language pairs show particularly good results. We need to analyze their features to understand why, motivating Section 4.3.

**Dendrograms.** Dendrograms for each task cluster programming languages with similar transfer learning performances. In the code repair task, syntactically and semantically related languages are next to each other. For example, C is paired with C++, JavaScript with TypeScript, Python with Cython, Lisp with Scheme. On the other hand, relatedness of languages plays a smaller role in the dendrograms for other tasks. This might be because while code repair requires deep language understanding, other tasks depend more upon other features. For instance, Ahmed & Devanbu (2022) show that some tasks depend heavily on identifiers.

Relatedness of languages plays a big role for code repair but less so for other tasks.

Target	Clone Detection		Code Repair		Error Detection	
	Source	score	Source	score	Source	score
Crystal	Ruby	0.93	–	–	–	–
Dart	Scala	0.93	D	0.88	–	–
Swift	JavaScript	0.91	Kotlin	0.79	–	–
bc	Scala	0.91	JavaScript	0.87	–	–
TypeScript	JavaScript	0.90	JavaScript	0.95	–	–
Elixir	C	0.89	JavaScript	0.83	–	–
Clojure	L	0.89	bf	0.77	–	–
Visual Basic	Scala	0.89	JavaScript	0.81	–	–
Julia	Ruby	0.89	JavaScript	0.81	–	–
Cython	Python	0.88	Python	0.95	–	–
Lua	Ruby	0.88	JavaScript	0.87	–	–
Moonscript	Scala	0.87	JavaScript	0.87	–	–
Scheme	Lisp	0.86	L	0.83	–	–
COBOL	Haskell	0.85	Python	0.78	–	–
F#	Haskell	0.84	Haskell	0.80	–	–
Sed	perl	0.81	C	0.89	–	–
Octave	ml	0.81	Python	0.78	–	–
Bash	perl	0.80	bf	0.83	–	–
dc	perl	0.80	–	–	–	–
Awk	Vim	0.78	–	–	–	–
C#	–	–	–	–	C++	0.68
Rust	–	–	–	–	Python	0.67
Ruby	–	–	–	–	Kotlin, Java	0.67
PHP	–	–	–	–	Kotlin, C++, C	0.67
Go	–	–	–	–	Kotlin	0.67

Table 8: The best performing source language for each low resource target language, in each task. For high resource languages, the best performing source language is usually the same language. Ruby is high-resource for Clone Detection and not a target for Code Repair, hence its results are not show for these two tasks. Similarly C#, Rust, PHP, and Go are high-resource for Clone Detection and Code Repair.

## 4.2 Performance Prediction

Given a task and target language, how should we pick the source language for best performance? Knowing this without having to train models on different source languages can save significant resources. It also provides a basis to study important features that characterize a successful transfer from a language to another. To predict the performance of the models on the different tasks for different source languages, we train a ranking model based on gradient-boosted decision trees (Ke et al., 2017).

We consider the features defined in Table 9, grouped into four categories. (1) *Linguistic features*: general properties characterizing a language pair like whether they support the same paradigms, the same style of memory management (garbage collector or not), or the same kind of type-system. (2) *Syntactic features*: properties of the syntax of a language pair measured by overlap in the counts of certain token types, as determined using the Pygments library on the corpus of code in the given languages. (3) *Dataset-specific features*: properties of the problems associated with code samples for the language pair in the online coding judge website from which the dataset came. (4) *Model-specific features*: whether the model saw the source, the target, or both languages during pretraining. The linguistic features of each language are given Table 10.

To predict the top source languages for a given task and target language  $L_t$  in the set of target languages  $T$ , we train a ranker using the LightGBM (Ke et al., 2017) implementation of the LambdaRank algorithm (Burgess, 2010). The model takes features in Table 9 as inputs and scores source languages  $L_s$  in terms of their

Feature	Measure	Description
Linguistic Features		
Object oriented	e	Is the language object oriented?
Type strength	e	Is the language strongly or weakly typed?
Type checking	e	Is the language typed statically or dynamically?
Type safety	e	Is the language type safe?
Garbage collection	e	Does the language use garbage collector?
Standardized	e	Is the language standardized by a committee?
Expression of types	e	Are the types written explicitly?
Paradigm	o	Paradigms supported (e.g., functional, imperative)
Type compatibility	o	Is language nominally-typed or structurally-typed?
Parameter passing	o	How are params. passed (e.g. by value, by name)?
Syntactic Features		
Name	o	Number of names which overlap
Text	o	Number of text data which overlap
Keyword	o	Number of keywords which overlap
Literal	o	Number of literals which overlap
Punctuation	o	Number of punctuation signs which overlap
Operator	o	Number of operators which overlap
Comment	o	Number of comment tokens which overlap
Syntax	o	Number of AST nodes which overlap
Tokens	o	Number of tokens which overlap
Dataset-Specific Features		
Difficulty	$x$	Average difficulty of dataset problems
Length	$x$	Average number of tokens
Time limit	$x$	Average time limit of dataset problems
Memory limit	$x$	Average memory limit of dataset problems
Model-Specific Features		
Pretrained	s	Source language is included during pretraining
Pretrained	t	Target language is included during pretraining
Pretrained	b	Both source and target languages in pretraining

Table 9: Features of language pairs. Column “Measure” indicates how the feature is computed: (e)quivalence, (o)verlap, (s)ource, (t)arget, (b)oth source and target, (rd) for relative difference. For dataset specific features, ( $x$ ) can be s, t, or rd. Details of the linguistic features are in Table 10.














































































































































Language	Paradigm	Object Oriented	Standardized	Type Strength	Type Safety	Expression of Types	Type Compatibility	Type Checking	Parameter Passing	Garbage Collection
Awk	  			strong		implicit		dynamic	by value, by reference	
Bf	  			weak		implicit		dynamic		
C	 			weak		implicit		dynamic	by value, by reference	
Clojure	     			strong		implicit	nominal	dynamic	by value	
COBOL	   			strong		explicit	nominal	static	by value, by reference	
C++	     			strong		explicit	nominal, structural	static	by value, by reference	
C#	        			strong		implicit	nominal	static	by value, by reference	
D	  			strong		explicit	nominal, structural	static	by value, by reference	
Dart	   			strong		inferred	nominal	static	by value	
Elixir	   			strong		implicit	structural	dynamic	by value	
Fortran	     			strong		explicit	nominal	static	by reference	
F#	      			strong		implicit	nominal	static	by value	
Go	 			strong		explicit	nominal	static	by value	
Haskell				strong		inferred	structural	static	by name	
Java				strong		explicit	nominal	static	by value	
Julia				strong		inferred	nominal	dynamic	by value, by reference	
JavaScript				weak		implicit	structural	dynamic	by value	
Kotlin				strong		explicit	nominal	static	by value	
Lisp				strong		implicit	nominal, structural	dynamic	by value	
Lisp				strong		implicit	nominal, structural	dynamic	by value	
Lua				strong		implicit	structural	dynamic	by value	
ocaml				strong		inferred	structural	static	by value	
Moonscript				weak		implicit	structural	dynamic	by value	
Nim				strong		explicit	nominal, structural	static	by value, by reference	
Octave				weak		implicit	nominal, structural	dynamic	by value	
Pascal				strong		explicit	nominal	static	by reference, by value	
PHP				weak		implicit	nominal	dynamic	by value	
Python				strong		implicit	structural	dynamic	by value, by reference	
Cython				strong		implicit	nominal, structural	dynamic	by value	
Ruby				strong		implicit	structural	dynamic	by value	
Rust				strong		explicit	nominal	static	by value, by reference	
Scala				strong		implicit	nominal, structural	static	by value, by name	
Scheme				strong		implicit	structural	dynamic	by value	
Bash				weak		implicit	nominal, structural	dynamic	by value, by reference	
Swift				strong		inferred	nominal	static	by value	
Typescript				strong		explicit	structural	static	by value	
Visual Basic				strong		implicit	nominal	static	by reference, by value	
Vim				weak		implicit	structural	dynamic	by value	

Table 10: Linguistic features. Paradigms: : scripting : imperative : functional : object-oriented : structured : aspect-oriented : concurrent : array : data-driven : esoteric : reflective : generic : agent-oriented : functional : object-oriented : procedural : logic : compiled : event-driven : metaprogramming : distributed : modular : block structured : pipeline : macro : multistaged : process-oriented : task-driven : declarative : process-driven

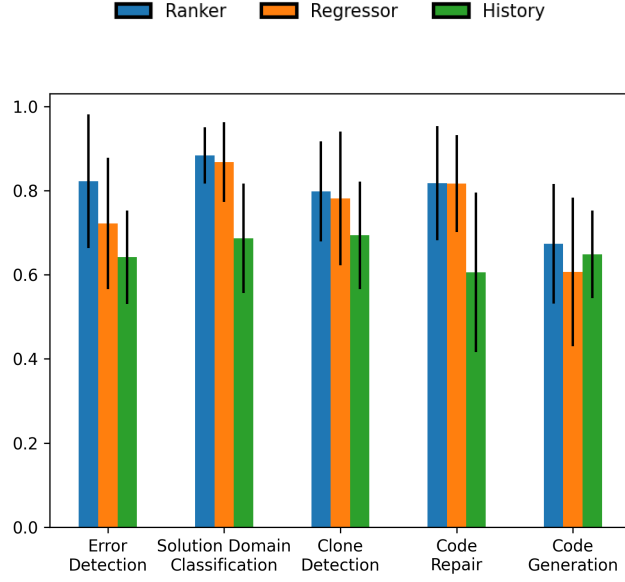


Figure 4: NDCG@3 scores for different rankers and tasks corresponding to LOO evaluation over the set of target languages. The first four tasks are evaluated on ranking predictions on CodeT5 transfer experiments, while code generation is evaluated over ranking predictions on LLAMA 3.3 70B-Instruct.

relevance to the target  $L_t$ . Our ranker uses a boosting ensemble of 100 decision trees with 16 leaves each. We consider the normalized discounted cumulative gain score (NDCG@K) at  $K = 3$  as our evaluation metric. We evaluate the model using leave-one-out (LOO) cross-validation on the set  $T$ . For each target language  $L_t$ , we train a ranker to predict rankings of different sources for each language in  $T$  leaving out the source ranking for  $L_t$  as a test set. For each fold, we compute the NDCG@3 score on the test set.

We compare the performance of our ranker with a regression-based ranker and a history ranker. The regression-based ranker uses a LightGBM regressor with the same hyperparameter settings as our ranker and selects the top  $K$  source languages with the highest predicted scores. The history ranker selects the top  $K$  source languages with the shortest path to the given target language in a programming languages history graph<sup>2</sup> (contracted to merge different versions of the same language). Figure 4 shows the mean and standard deviation of NDCG@3 using LOO on  $T$ . While the error bars are overlapping, both regression and rankers outperform the history ranker on all tasks, indicating the importance of considering a variety of features for predicting top-performing sources rather than simplified heuristics. Our model outperforms baseline rankers on all tasks with the exception of clone detection, for which it shows slightly lower performance than the regression model. This indicates that ranking is the best suited method for predicting top performing sources. Based on these results, we use our ranker for the feature analysis in RQ3.

Rankers outperform regressors for predicting top-performing sources. Using historical relationships among programming languages alone is insufficient.

### 4.3 Feature analysis

While Section 4.1 explored how well transfer learning performs for different languages on different tasks, we would like to dig deeper into *why* they perform that way. That is, how do the features of language pairs affect transfer? Which characteristics of the source and target language best explain the variation of transfer across tasks, the dependence of transfer on source and target pairing, as well as the superior transfer supported by specific sources? These questions can be answered by measuring feature importance.

<sup>2</sup><http://rigaux.org/language-study/diagram.html>

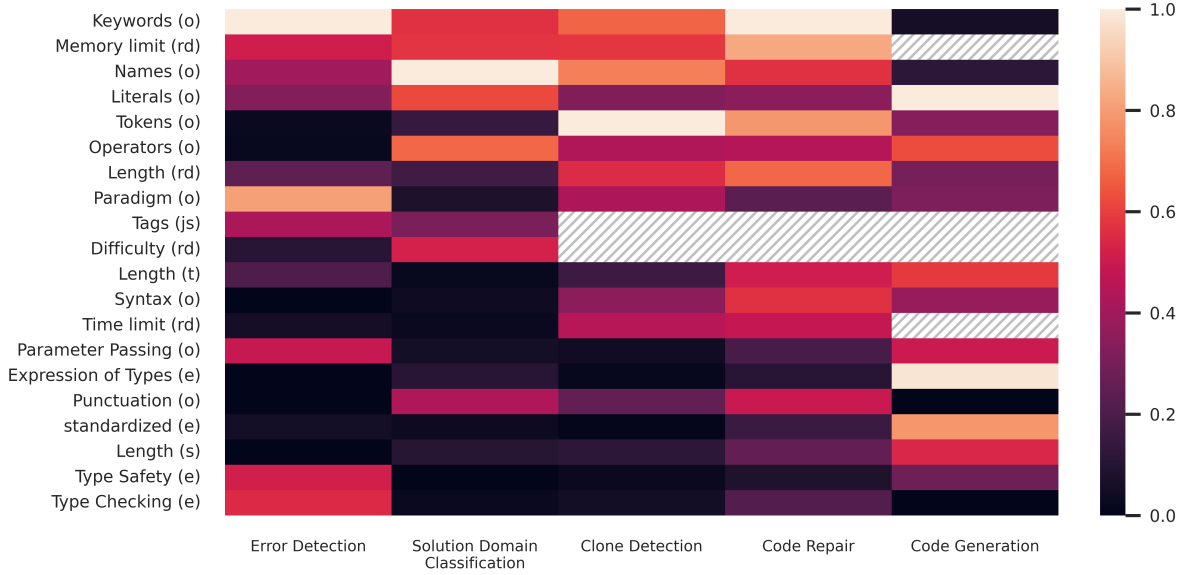


Figure 5: Normalized SHAP values aggregated by tasks for the features defined in Table 9. The features are sorted by mean rank. Grayed-out features were unavailable for the given task. The first four tasks are evaluated on ranking predictions on CodeT5 transfer experiments, while code generation is evaluated over ranking predictions on LLAMA 3.3 70B-Instruct.

The state-of-the-art technique for measuring feature importance, SHAP values (Lundberg & Lee, 2017), considers all possible subsets  $S \subseteq F$  of features. Then, the importance of a given feature is the difference it makes to model performance in combination with all possible sets of other features. In the context of ranking different source languages for cross-lingual transfer, SHAP values measure how different features contribute to transfer. Figure 5 shows SHAP values computed based on the ranking models evaluated in Section 4.2. For presentation purposes, it rescales the values so the most important feature for each task has importance 1.

**Task-dependent feature importance.** While previous works on transfer learning across programming languages emphasize the importance of specific features for cross-lingual transfer, the lack of comparison on different tasks limits their conclusions. The supported cross-task analysis demonstrates a task-dependent feature importance whereby different features contribute differently to cross-lingual transfer depending on the task. For example, the top features for the solution domain classification task are *Difficulty*, *Literals*, *Names*, and *Operators*. Being able to predict the solution domain for a code solution requires some knowledge of the underlying problem for which the tags are an attribute. The difficulty score is another attribute of the problem. Problems with similar difficulty scores require similar algorithms. The literals, names, and operators are other indicators of the algorithms used in a code sample. In comparison, the most significant features for the clone detection task are *Token*, *Names*, and *Keywords*. Detecting a clone requires different skills than classifying domains. A deeper understanding of the code semantics irrespective of the similar algorithms used is needed. The overlaps in names, keywords, and more generally tokens is key for understanding the semantics of code (Ahmed & Devanbu, 2022).

Prior work on transfer learning in natural languages (Lin et al., 2019) showed feature importances to vary greatly across tasks. We confirm this for programming languages.

**Range of important features.** Different tasks seem to not only focus on different features, but also on a different number of features. While the feature importance for solution domain classification and error detection is concentrated on a small number of features, it is more spread out for clone detection and code repair. For example, the code repair task requires a transfer of knowledge from overlapping keywords, names,

and more generally different tokens from a source language for fixing a code in a target language. On the other hand, solution domain classification focuses on fewer features, related to the problems attributes.

Higher-level tasks seem to draw upon more features to best understand the code.

**Comparison across categories.** Comparing the importance of different feature categories across tasks, we observe a relatively higher significance of syntactic features as demonstrated by the top two features *Keywords* and *Names*. Compared to transfer learning across natural languages, model-specific and linguistics features seem to carry less significance for transfer in programming languages. While de Vries et al. (2022) find the potential of cross-lingual transfer dropping significantly for sources and targets that are not seen by the model, our finding brings hope to not only transferring learning to low-resource target languages, but even finetuning the model on unseen languages that could serve as good sources, such as Kotlin.

Overall, the most important features across the four tasks are keywords and names.

**Within-category differences.** While different categories seem to have a higher impact on the model predictions, the impact of single features within a category varies across task. Our results are consistent with prior work that found *Names* and *Keywords* to be the most important features for cross-lingual transfer in certain tasks (Ahmed & Devanbu, 2022). However, half of the tasks show keywords to be more important than names, while the other half has the opposite order of importance.

#### 4.4 Few-shot prompting

We examine cross-lingual transfer learning for programming languages by pairing low- and high-resource languages across classification and code-repair tasks. Whereas our initial study employed CodeT5-base, we now replicate a representative subset of experiments with the larger LLAMA 3.3 70B-Instruct model to test the generalizability of our conclusions to contemporary large-scale LLMs. For evaluation we adopt the few-shot prompting scheme described in Section 3; this technique has been shown to boost out-of-domain code-generation quality for previously unseen languages and to markedly reduce the compilation and parsing failures that arise when the model lacks explicit exposure to a target language (Athiwaratkun et al., 2023).

We investigate whether rankers constructed from the features introduced earlier can accurately predict the best source language(s) for few-shot prompting. We compare different rankers explored in the previous section and test whether our ranker model can generalize to larger models. We train a ranker model on the same features described in Table 9 on Pass@1 scores. Figure 4 shows mean NDCG@3 scores. The learned ranker still surpasses other rankers and heuristic baselines, underscoring that considering multiple features is essential for accurately identifying the top source languages. SHAP analysis further reveals that expression of types and overlap in literals dominate the other features in predicting top performing sources. Languages whose type systems are richly expressed in code provide stronger semantic signals that transfer directly for code generation of other similarly typed languages solutions. Meanwhile a high lexical overlap in literals such as numeric and string constants offers immediate token-level anchors that reduce out-of-vocabulary risk during generation. Together, these two features jointly encode semantic structure and surface-level similarity, making them the most dependable indicators of best source languages for code generation.

## 5 Limitations

This section describes the limitations of our study, including the rationale for key design choices and the steps we took to address those limitations.

### 5.1 External validity

One limitation is that, to keep the large number of experiments feasible and reduce their carbon footprint, we constrained our fine-tuning experiments to a moderately sized model. We address this by using CodeT5 (Wang et al., 2021), a commonly adopted model in AI-for-Code research. Even if a larger model were used, insights from one model would not necessarily generalize to another. Accordingly, we included model-specific

features in our study to account for potential model dependence. These features are described in Table 9 and analyzed as part of Section 4.3. Despite model-specific features, we do acknowledge that experimenting with different models, not just of varying size but also architecture, would have made our study more robust. Because the source and target languages are usually different and come in many combinations, we do not necessarily run the risk of overfitting. However, using models of varying strength could have exposed a broader range of more interesting patterns which would have made our study richer and more general. To mitigate this limitation, we conduct separate experiments with a larger model and few-shot prompting.

Another limitation is that, to cover multiple languages, we limited ourselves to only five tasks. We address this by selecting tasks that span a broad range of difficulty and demonstrate diversity, as indicated by their differing performance and feature importance. Nonetheless, our study includes more tasks than any previous cross-lingual transfer study on programming languages.

A further limitation is that our datasets include synthetic elements—for instance, type-IV clones (Roy & Cordy, 2007) in the clone detection data or fault injection in the code repair data. However, using exclusively natural data imposes a demanding constraint that few academic studies meet, and it is also uncommon in industrial practice. To address this, we explicitly measure the importance of dataset-specific features for each language pair.

Another limitation is that we focused on the setting of zero data in the target language. This represents the lowest barrier to entry, as gathering even a small number of examples can be tricky. Therefore, we defer exploring scenarios that include some target-language data to future work.

## 5.2 Internal validity

We believe that our study avoids one common internal validity limitation, which revolves around insufficient runs. Across all experiments, each source-target pair is evaluated only once. But thanks to the sheer number of language pairs for each task, we get thousands of different experiments, whose results are consistent with each other. Moreover, each of these experiments involves fine-tuning with tens of thousands of samples, giving us confidence that the choice of samples is not a limitation either. However, having results based on single runs does limit our ability to assess variability in experiments. Since our study is the first of its kind at this scale, it lacks directly comparable baselines. We mitigate this limitation by noting that, wherever comparisons are possible, our results corroborate those of narrower studies—for instance, prior work on transfer for identifier-centric tasks (Ahmed & Devanbu, 2022).

## 6 Conclusion

We perform a systematic and extensive study of LLM transfer learning covering up to 41 programming languages, across 5 tasks including both classification and generation. Programming languages covered include several low-resource but often still widely-used languages. Cross-lingual transfer learning performs much better than zero-shot with an LLM that has been pre-trained on code. One interesting finding is that even a language not seen during pre-training, like Kotlin, can be good fine-tuning source language across several target languages and tasks. On the other hand, certain languages that are often used extensively to pre-train LLMs, like C++ and Python, are worse source or target languages relative to others.

To understand relative differences in cross-lingual performance between different programming languages, we define several linguistic, syntactic, dataset, and model-specific features of language pairs and then analyze the feature importance for a model that predicts transfer performance. We show how different features are needed for ranking source languages, compared to ad-hoc heuristics such as overlap on names. To explain how a language that was unseen during pre-training, like Kotlin, can be a good source language, we show that seen language features are less significant compared to dataset, linguistic, and syntactic source language features. Similar to previous work of Ahmed & Devanbu (2022), we show that keywords and names are top features on average for CodeT5. On the other hand, unlike previous work, we cover more languages and more diverse tasks, and find that feature importance vary strongly across tasks. Replicating a representative subset of experiments with a larger model under few-shot prompting confirms that these feature hierarchies and the performance prediction gains they enable generalize to state-of-the-art LLMs. Overall, we believe

that this paper sheds light on how learning transfers among programming languages, and that this ultimately leads to better models to assist users of those languages, particularly low-resource ones.

## 7 Data Availability

The experiments are based on the publicly available CodeT5-base (220M parameters) model (Wang et al., 2021) and the open-sourced datasets CodeNet (Puri et al., 2021) and XCodeEval (Khan et al., 2023). Our code is available at <https://github.com/baltaci-r/XPL>.

## References

- Toufique Ahmed and Premkumar Devanbu. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering, ICSE '22*, pp. 1443–1455, 2022. ISBN 9781450392211. doi: 10.1145/3510003.3510049.
- Kabir Ahuja, Shanu Kumar, Sandipan Dandapat, and Monojit Choudhury. Multi task learning for zero shot performance prediction of multilingual models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5454–5467, May 2022. doi: 10.18653/v1/2022.acl-long.374.
- Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations*, 2023. URL <https://openreview.net/forum?id=Bo7eeXm6An8>.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv:2108.07732*, 2021.
- Chris J.C. Burges. From RankNet to LambdaRank to LambdaMART: An overview. Technical Report MSR-TR-2010-82, Microsoft Research, 2010.
- Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. Knowledge transfer from high-resource to low-resource programming languages for code llms. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2024. doi: 10.1145/3689735.
- Fuxiang Chen, Fatemeh H. Fard, David Lo, and Timofey Bryksin. On the transferability of pre-trained language models for low-resource programming languages. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC '22*, pp. 401–412, 2022. doi: 10.1145/3524610.3527917.
- Coursera. Programming in Swift: Benefits of this popular coding language, 2024. <https://www.coursera.org/articles/programming-in-swift>.
- Wietse de Vries, Martijn Wieling, and Malvina Nissim. Make the best of cross-lingual transfer: Evidence from POS tagging with over 100 languages. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7676–7685, May 2022. doi: 10.18653/v1/2022.acl-long.529.
- Paul N. Edwards and John Leslie King. Institutions, infrastructures, and innovation. *Computer*, 54(1): 103–109, January 2021.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh

Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yearly, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shao-liang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vitor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenxin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuwei Wang, Yaelle Golschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpiere Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippou Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damla, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang,

- Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabza, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaoqian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The Llama 3 herd of models. *arXiv*, 2024.
- Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016. doi: 10.1145/2902362.
- Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in Neural Information Processing Systems*, 30, 2017.
- Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xCodeEval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval, 2023. arXiv:2303.03004v2.
- Anne Lauscher, Vinit Ravishankar, Ivan Vulić, and Goran Glavaš. From zero to hero: On the limitations of zero-shot language transfer with multilingual Transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 4483–4499, November 2020. doi: 10.18653/v1/2020.emnlp-main.363.
- Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7871–7880, July 2020. doi: 10.18653/v1/2020.acl-main.703.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel,

- Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder: may the source be with you! *arXiv*, 2023.
- Yu-Hsiang Lin, Chian-Yu Chen, Jean Lee, Zirui Li, Yuyan Zhang, Mengzhou Xia, Shruti Rijhwani, Junxian He, Zhisong Zhang, Xuezhe Ma, Antonios Anastasopoulos, Patrick Littell, and Graham Neubig. Choosing transfer languages for cross-lingual learning. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 3125–3135, 2019. doi: 10.18653/v1/P19-1301.
- Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv*, 2021.
- Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30, 2017.
- MicroFocus. Cobol market shown to be three times larger than previously estimated in new independent survey, 2022. URL <https://www.microfocus.com/en-us/press-room/press-releases/2022/cobol-market-shown-to-be-three-times-larger-than-previously-estimated-in-new-independent-survey>.
- Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. MetaTPTrans: A meta learning approach for multilingual code representation learning. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence*, AAAI’23, 2023. doi: 10.1609/aaai.v37i4.25654.
- Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian T Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, Virtual, 2021. Curran.
- Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. Technical Report 2007-541, Queen’s University, 2007.
- TogetherAI. Together ai, 2025. URL <https://together.ai>. Large Language Model Platform.
- Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, November 2021. doi: 10.18653/v1/2021.emnlp-main.685.
- Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface’s transformers: State-of-the-art natural language processing, 2020.
- Wei Yuan, Qunjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. CIRCLE: Continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, pp. 678–690, 2022. doi: 10.1145/3533767.3534219.

Wen Zhou, Seohyun Kim, Vijayaraghavan Murali, and Gareth Ari Aye. Improving code autocompletion with transfer learning. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 161–162, May 2022. doi: 10.1145/3510457.3513061.