MULTILINGUAL CODE RETRIEVAL WITHOUT PAIRED DATA: A NEW BENCHMARK AND EXPERIMENTS

João Monteiro, Torsten Scholak, Veeru Mehta, David Vázquez, Chris Pal
ServiceNow Research, {FirstName.LastName}@servicenow.com

ABSTRACT

We seek to overcome limitations to code retrieval quality posed by the scarcity of data containing pairs of code snippets and natural language queries in languages other than English. We correspondingly test the following hypothesis: if a model can map from English to code, and from other natural languages to English, then how well can the model directly map from those non-English languages into representations of code? To do so, we introduce two new datasets. For training models, we build a corpus corresponding to paired English/Code data and combine it with existing translation datasets given by pairs of English and other natural languages. For evaluation, we make a new benchmark available, dubbed M^{2}CRB, containing pairs of text and code, for multiple natural and programming language pairs – namely: Spanish, Portuguese, German, and French, each paired with code snippets for: Python, Java, and JavaScript. Evaluation on both our new benchmark tasks as well as on an existing code-to-code search task confirms our hypothesis: models are able to generalize to unseen source/target language pairs they indirectly observed during training. We examine models which both generate and retrieve natural and programming languages and through ablations, we further verify the influence of different design choices and training tasks in terms of whether or not they contribute to generalization with unseen language pairs.

1 INTRODUCTION

Recent work has observed significant progress in settings where one seeks to obtain code snippets conditional on natural language queries. In the generative setting, for instance, cases such as AlphaCode (Li et al., 2022) obtained human-level performance in generating code from competitive programming problem statements in plain English. In the retrieval/search setting on the other hand, `cpt-code` (Neelakantan et al., 2022) showed that contrastive training of encoders using pairs of docstring and code results in a semantic embedding where search can be efficiently performed. Evaluation of `cpt-code` is made on the CodeSearchNet benchmark (Husain et al., 2019) where, given a query in English, the model retrieves a code block deemed relevant among 1000 candidates. CodeSearchNet encompasses 6 programming languages, while queries are only in English. As evidenced by the examples above, the quality of code retrieval from natural language queries is rapidly increasing. However, there's a focus on using English as the underlying language of the source query, and multi-source-language models are still scarce. This is mostly due to the lack of large scale parallel data between different natural languages and code. An attempt towards defining multi-programming-language translation systems, or transcompilers, was carried out in Transcoder (Roziere et al., 2020; 2021) where multiple unparalleled data sources in different programming languages are considered. Beyond that, in this work, we address the question of whether one can improve performance under settings where models map multiple natural languages to multiple programming target languages.

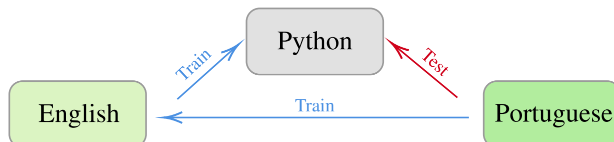

Figure 1: Illustration of the scheme we consider. *Hypothesis*: if a model is able to map from English (the anchor) to Python and from Portuguese to English, then it should be able to directly map from Portuguese to Python.

In particular, we study the *code search from text* setting and train models under multiple combinations of source and target languages. Naive supervised training under such a setting would require $|\mathcal{S}| \times |\mathcal{T}|$ parallel datasets, where $\mathcal{S}$ and $\mathcal{T}$ correspond to the sets of source natural and target programming languages, respectively. To work around that, we employ a strategy aiming at multi-modal language-invariant semantic embedding, where encoded data depends only on the underlying implementation represented by programming languages or their descriptions represented in text. We can then elect one of the source datasets as the anchor, denoted $S^* \in \mathcal{S}$, and instead train models using parallel datasets between sources and the anchor, and between the anchor and the targets. In other words, we replace the need for parallel data between all elements of $\mathcal{S}$ and $\mathcal{T}$ for parallel data between elements of $\mathcal{S}$, which enables the use of available natural language translation datasets. At testing time, we put models to test under source/target combinations that are unseen during training. An illustration of the described training scheme is provided in Figure 1. In this example, $\mathcal{S} = \{\text{English}, \text{Portuguese}\}$, $\mathcal{T} = \{\text{Python}\}$, and $S^* = \text{English}$. At training time, models observe pairs of sentences in English and Python snippets as well as Portuguese and English pairs. At testing time, we then search Python directly from queries in Portuguese. Put simply, we test whether models generalize to new language combinations, only indirectly observed during training.

Contributions. Our contributions are summarized as follows:

1. We introduce a new evaluation dataset (*high quality* - *low volume*) dubbed $\texttt{M}^2\texttt{CRB}$, where multiple natural languages are used to search over a codebase containing multiple programming languages. In particular, the data contains docstrings in Spanish, Portuguese, German, and French, paired with code snippets in languages such as Python, Java, and JavaScript.

2. We supplement the training partition of CodeSearchNet with additional data (*high volume* - *low quality*) for a subset of the programming languages they consider (English to Python, Go, Java, and JavaScript). In particular, the natural/programming languages observed in this complementary dataset are chosen to be such that they are never seen at testing time.

3. We contribute a training recipe including data augmentation strategies that enables search over unseen language pairs by inducing a language-independent semantic embedding. We further report experiments showing the effects of different design choices and the performance of contrastive schemes and generative tasks, as well as their combination.

2 Problem setting

In the following, we formally define the setting briefly introduced on the above and exemplified in Figure 1. Given a set $\mathcal{S}$ of source data distributions along with a set $\mathcal{T}$ of target data, training will require paired data from one of the sources, which will be referred to as *anchor* and denoted S^*, and all the targets. In addition, we further require access to paired data between the anchor and the remaining source domains. Our training dataset thus corresponds to the union of finite samples observed from the following distributions:

$$(S^*, T) \; \forall \; T \in \mathcal{T}, (S^*, S) \; \forall \; S \in \mathcal{S} - S^*,$$

where the parenthesis notation $(\mathbb{P}, \mathbb{Q})$ indicates the joint over distributions $\mathbb{P}$ and $\mathbb{Q}$. In Section 3.1, we describe the approach we take in order to construct such a dataset in practice. While those two sets of paired data should suffice in order to enable mapping from any source to any target distribution, we can add data by considering non-anchor source/source combinations as well as target/target combinations whenever those are available. For testing on the other hand, we seek models able to map between any combination of source and target distributions. We thus build our test set by observing pairs from the union of the following set of joints:

$$(S, T) \; \forall \; S \in \mathcal{S}, T \in \mathcal{T}.$$

We finally remark that any data realization x observed from any distribution $x \sim \mathcal{D} \in \mathcal{S} \; \cup \; \mathcal{T}$ corresponds to a sequence with length L of symbols or tokens from a shared vocabulary $\mathbb{V}$: $x = [x_1, ..., x_L] : x_i \in \mathbb{V}$. Moreover, we highlight that we use the terms *map/mapping* in a rather general sense to mean either a generative case where one literally translates from source to target, or a discriminative case where one retrieves from a set of candidate targets given an instance from the source. As will be further discussed in Section 4, both types of mappings will be used when training models, but only the discriminative one is considered during testing since our ultimate goal is to define efficient code search from multi-natural-language source queries.

3 MULTI-LANGUAGE DATASETS

3.1 TRAINING SET - MULTIPLE SOURCE/TARGET COMBINATIONS

In Section 2, three types of paired data are discussed: anchor/target, anchor/source, and non-anchor source/target pairs. To build a dataset comprising all types of pairs, we combine a mix of both existing paired datasets of different kinds, along with new data we introduced. In what follows, we describe separately how we proceed in order to obtain or create each data component. Moreover, we highlight that we set the anchor S^* to English and the set $\mathcal{T}$ to the set of programming languages represented in CodeSearchNet, *i.e.*, $\mathcal{T} = \{\text{Python, Go, Java, JavaScript, PHP, Ruby}\}$.

3.1.1 ANCHOR-TARGET PAIRED DATA

The pairs represented by $(S^*, T)\ \forall\, T \in \mathcal{T}$ correspond to the training partition of CodeSearchNet (Husain et al., 2019) further augmented with supplementary data we prepared.

New dataset: additional CodeSearchNet-style training data. CodeSearchNet was built by leveraging the fact that function docstrings and implementations define easy to obtain text/code pairs, and then pre-processing of GitHub data was performed. In particular, `Tree-sitter`[1] parsers are used to create docstring/code pairs for all languages in $\mathcal{T}$. We introduce additional data by repeating that procedure on the dump of GitHub in The Stack Kocetkov et al. (2022), focusing only on repositories not covered by CodeSearchNet and on a subset of $\mathcal{T}$, substantially increasing the total amount of English/Code training pairs.

3.1.2 ANCHOR-SOURCE PAIRED DATA

For anchor/source pairs, we leveraged datasets introduced for machine translation tasks. Namely, a subset of WMT-19 (Wikimedia-Foundation, 2019) was considered given by the English/German and English/Finnish partitions. We further considered non-anchor source/source combinations to increase the amount of training data. As such, we considered the French/German subset of WMT-19 as well as the Spanish/Portuguese and Spanish/Galician partitions of Tatoeba (Tiedemann, 2020). Finally, a single target/target case is considered and is given by the code translation data within CodeXGLUE (Lu et al., 2021). Summary statistics of the complete training data is shown in Table 1.

Dataset	**Row count**	**Sampling proportion** ($\times 10^{-5}$)	**Epochs**
CodeSearchNet	1880853	0.0532	3.5
GH-Python (*Ours*)	15000002	0.0067	0.5
GH-Java (*Ours*)	15000014	0.0067	0.5
GH-GO (*Ours*)	15000078	0.0067	0.5
GH-JavaScript (*Ours*)	2000040	0.0500	3.3
WMT-19 (DE-EN)	1995208	0.0501	3.3
WMT-19 (FR-DE)	1999990	0.0500	3.3
WMT-19 (FI-EN)	2000000	0.0500	3.3
Tatoeba (ES-PT)	67777	1.4754	29.1
Tatoeba (ES-GL)	3132	31.9285	30.0
CodeXGLUE	10300	9.7087	30.0

Table 1: Statistics of each of the datasets used to compose the full training corpus we use. Given the variability in size of each dataset as indicated by the row counts, we adjust sampling proportions during training so that datasets are uniformly represented in the actual training sample. The number of epochs represented in the rightmost column indicates the approximate number of times we iterate over the entire dataset each time we feed approximately 30 billion tokens to a model. The *GH* prefix indicates the supplementary data we processed.

[1] `https://tree-sitter.github.io/tree-sitter/`

3.2 M^{2}CRB - UNSEEN PAIRS OF LANGUAGES

For testing, we build a new evaluation task considering source/target pairs that do not appear in the training partition. To do so, we use data collected from The Stack Kocetkov et al. (2022), perform parsing on files corresponding to languages of interest from repositories that are not included in CodeSearchNet to get docstring/code pairs, and finally perform language identification on docstrings to end up with data containing non-English docstrings. To perform language identification, we ensemble three different off-the-shelf open-source language classifiers from text.[2,3,4] To build the test set, we then consider only the instances for which the three classifiers agreed upon a language. Finally, we further perform human evaluation to filter out misclassifications, *i.e.*, native speakers of each of the considered natural languages verified each docstring to ensure they correspond to the correct language. The resulting M^{2}CRB's row counts per natural/programming language pair are reported in Table 2 and further details on data filtering and processing can be found in Appendix A.

4 MODELS AND TRAINING

Our main goal is to define a semantic embedding so as to enable efficient cross-modality search. To do that, we leverage a multi-modal contrastive training strategy and encode input sequences into vectors denoted z_{EMB}, and such vectors are encouraged to match for corresponding source and target instances. For example, docstrings and corresponding implementations should map to similar vectors in terms of some distance measure. Similarly, matching sentences in two different natural languages should embed into neighboring points, as should two code snippets implementing the same functionality but written in different programming languages. In addition to training objectives for which properties just described will hold, we also consider multi-task models where generative components are added to enable translation from source to target assuming that such a functionality is often required. We then evaluate the impact in search/retrieval performance given by giving up some of the capacity of the semantic encoder to instantiate a decoder. That is, we consider the standard sequence-to-sequence setup and introduce tasks such as translation and denoising, both performed alongside the main contrastive loss. Additionally, we further consider a decoder-only model where we carry out the contrastive learning on top of representations, but also perfom generative tasks at the same time. We remark however that in all cases our main goal is to be able to search/retrieve effectively, and generative tasks are added to assess to what extent they affect retrieval performance.

Natural Lang.	**Programming Lang.**			**Total**
	Python	*Java*	*JavaScript*	
Spanish	1298	598	631	2527
Portuguese	1376	694	450	2520
German	543	928	83	1554
French	779	164	199	1142
Total	3996	2384	1363	**7743**

Table 2: Row count of M^{2}CRB per combination of the underlying natural language of docstrings and programming language of corresponding code. Further details on data filtering and processing can be found in Appendix A.

Our main models then correspond to either a sequence-level encoder, and an encoder-decoder pairs, both illustrated in Figure 3. A decoder-only architecture is additionally considered. The encoder, denoted $\mathcal{E}$ from now on, is responsible for actually embedding data from whichever modality (*i.e.*, code or text). The embedding corresponds to $\mathcal{E}$'s output at the end-of-sequence special token `[EOS]`, appended to all inputs prior to feeding $\mathcal{E}$. In Figure 3, embeddings used to represent input sequences are denoted z_{EMB}. The decoder, denoted $\mathcal{D}$, on the other hand, accounts for the generative tasks by auto-regressively predicting tokens. Finally, the decoder-only model we consider is illustrated in Figure 4. Given the two types of losses mentioned above, we then train with their convex combination:

$$\mathcal{L} = \alpha \mathcal{L}_{contrastive} + (1 - \alpha)\mathcal{L}_{generative}, \quad (1)$$

where $\alpha \in [0, 1]$ is a hyperparameter weighing the importances of the two components. The encoder-only case, which will be further discussed in Section 5, corresponds to simply setting $\alpha = 1$. Further details on the training procedure as well as the training losses are described in the following.

[2] https://github.com/saffsd/langid.py
[3] https://github.com/google/cld3
[4] https://huggingface.co/papluca/xlm-roberta-base-language-detection.

4.1 CONTRASTIVE TASK

Given a batch of n pairs of sequences $[x_i, y_i]_{i=1}^n : x_i, y_i \sim (S \in \mathcal{S}, T \in \mathcal{T})$, our contrastive objective relies on the similarity matrix Sim given by rescaled cosine similarities measured between embeddings of x and y:

$$Sim[i, j] = 2 * \cos(\mathcal{E}(x_i)^{eos}, \mathcal{E}(y_j)^{eos}) - 1. \tag{2}$$

We then force Sim to be an identity matrix, that is, for the similarity between paired data to be greater than that between unpaired instances, *i.e.*:

$$\mathcal{L}_{contrastive} = ||Sim - I_n||_2. \tag{3}$$

Similar to CLIP (Radford et al., 2021) however, in practice, we implement a variation of $\mathcal{L}_{contrastive}$ that shares its minimizers since it's easier to train against this variation, as observed empirically. The loss we use treats Sim as a batch of logits, and places labels on the main diagonal to define a cross-entropy objective. An implementation of the $\mathcal{L}_{contrastive}$ objective used during training of our models is shown in Figure 8.

4.2 GENERATIVE TASKS

We consider the rather standard autoregressive maximum likelihood objective commonly used in sequence-to-sequence setups. Assuming access to a batch of paired sequences from source and target distributions, the objective will be given by:

$$\mathcal{L}_{generative} = \frac{1}{n} \sum_{i=1}^{n} \prod_{t=1}^{L'} p_{\mathcal{D}}(y_i^t | y_i^1, ..., y_i^{t-1}, \mathcal{E}(x_i)). \tag{4}$$

We then take advantage of the fact that all training datasets are parallel, and define translation auxiliary tasks. In addition, we further introduce a denoising objective in which the original sequence is to be recovered from its noisy version where tokens are randomly dropped out, and the dropping out probability is assumed to be a tunable hyperparameter.

5 EVALUATION

We leverage pre-trained `codeT5` (Wang et al., 2021) models in order to implement $\mathcal{E}$ and $\mathcal{D}$ for most cases except for the decoder-only case, where we start from `CodeGen` Nijkamp et al. (2022a). The `codeT5` model builds on the `T5` architecture (Raffel et al., 2020) by including pre-training tasks that rely solely on code, such as denoising or identifier tagging, but also translation. In this case, models perform bimodal dual generation, *i.e.*, back and forth translation between programming and natural language. In addition to the contrastive task performed on top of $\mathcal{E}$'s outputs, we also consider generative tasks such as denoising, in which case we ask the model to "in-paint" dropped out tokens. Translation tasks are also performed in the dual generation setting. We remark that, in our case, generative auxiliary tasks are performed on top of all training datasets, including those containing only natural language. `CodeGen` on the other hand is a decoder-only language model trained on a mix of natural and programming language.

Our experiments focus in the comparison between the discriminative case where the full model is dedicated to retrieval versus the generative setting where translation is also performed, and then some of the model capacity is moved to a decoder. That is, under a not widely varying parameter budget, we assess whether adding functionality to a model, i.e., by enabling it to generate in addition to retrieve, will hurt search performance relative to an encoder that devotes its full capacity to a discriminative objective. We further consider ablations where we drop training tasks or data. As for pre-processing, subword tokenization is performed using the same strategy employed by Wang et al. (2021), i.e., a byte-pair encoding (BPE) Sennrich et al. (2015) is used and code-specific tokens are included in the vocabulary such as brackets and whitespace sequences. Moreover, given a training pair (x, y), we append special tokens and feed the model with the following template:

$$(\{\text{lang}[y]\} : \{x\}[EOS], \{y\}[EOS]),$$

where the operator $\text{lang}(\cdot)$ returns the underlying language of its argument.

Prog. Lang	Nat. Lang.	Disc.	Gen.+Disc.	*Ablations*		
				Gen.	**No denoising**	**Dec.**
Python	*de*	58.1%	60.2%	4.0%	53.2%	65.7%
	es	75.5%	74.8%	4.1%	65.8%	63.8%
	fr	64.5%	70.3%	4.3%	65.0%	68.2%
	pt	70.6%	66.9%	3.7%	58.0%	51.0%
Java	*de*	31.3%	23.2%	3.5%	24.2%	21.7%
	es	37.2%	27.3%	3.6%	31.0%	22.0%
	fr	47.4%	39.3%	8.0%	40.7%	49.9%
	pt	34.8%	26.4%	4.5%	27.4%	26.6%
JavaScript	*de*	61.6%	62.2%	12.2%	59.2%	55.7%
	es	28.6%	22.2%	4.6%	21.8%	19.6%
	fr	30.8%	30.5%	6.7%	31.4%	28.0%
	pt	27.2%	20.8%	4.4%	22.5%	14.3%
Avg.		47.3%	43.7%	5.3%	41.7%	40.5%

Table 3: M^{2}CRB's results for code search from natural language queries in different languages. Results correspond to the area under the MRR curve (*au*MRR*c*, the higher the better).

Data augmentation is further performed during training so as to avoid spurious solutions, able to retrieve by simply matching keywords present in both docstrings and actual implementations. That is, in early experiments, we noticed well performing models would simply retrieve code snippets for which variable or function names would appear in the docstrings. To counter that and enforce actually semantic retrieval, we remove this "shortcut" by randomly assigning meaningless function names in codes snippets, and via randomly replacing variable names by uninformative strings.

5.1 Evaluation metric

Given paired sets of n points from the source and target distributions as denoted by $x, y = x_1, ..., x_n, y_1, ..., y_n \overset{\text{i.i.d.}}{\sim} (\bigcup_{i=1}^{|\mathcal{S}|} \mathcal{S}_i, \bigcup_{i=1}^{|\mathcal{T}|} \mathcal{T}_i)$, evaluations will require computation of the mean reciprocal rank defined by

$$\text{MRR}(x, y) = \sum_{i=1}^{n} \frac{1}{\text{rank}(x, y, i)}, \tag{5}$$

where the rank corresponds to the position/index of Y_i in the ordered set of similarities

$$\text{rank}(x, y, i) = \text{index}(Sim(x_i, y_i), \{Sim(x_i, y)\}_{sorted}), \tag{6}$$

and $\text{index}(\cdot, \{\cdot\})$ returns the position of its first argument in the ordered set given as the second argument. Note that we overload Sim, defined in the same way as that used to compute entries of the similarity matrix in (2), and compute it both for data pairs, where the output is a scalar, and between a source data point and a set of target domain instances, in which case a set of similarities is output. We then evaluate models under varying sizes of the underlying sets. That is, we compute the area under the MRR curve, obtained for increasing sizes of retrieval sets, as indicated in the following:

$$au\text{MRR}c = \int_0^1 \text{MRR}(x, y_{1:\lfloor n*t \rfloor}) dt. \tag{7}$$

To approximate (7), we discretize t so that $t = \{5\%, 10\%, 20\%, 30\%, 50\%, 75\%, 100\%\}$.

5.2 Multi-language evaluation with M^{2}CRB

We perform evaluations on M^{2}CRB in terms of *au*MRR*c* as reported in Table 3 for different models, while MRR curves as a function of t are displayed in Figure 5 in Appendix C.2. In particular, we are concerned with assessing to what extent models manage to generalize to source/target pairs unseen during training. We further seek to compare fully discriminative encoder-only models with models able to both retrieve and generate. Model identifiers reflect the model architecture and training objective such that (**Disc.**) stands for discriminative and corresponds to the encoder-only model. Similarly, the (**Gen.+Disc.**) case refers to encoder-decoder settings where models are trained against

the discriminative/generative objective defined in (1). In addition, we consider ablation cases where training objectives are dropped such as (**Gen.**) which corresponds to an encoder-decoder pair trained with the generative objective only (*i.e.*, $\alpha = 0$ in 1), a (**Gen.+Disc.**) model where the denoising objective is dropped, and finally, a decoder-only model (**Dec.**) trained against both discriminative and generative objectives. Results show that most models do manage to achieve good retrieval performance even if all combinations of source/target languages were never presented to the models during training. Interestingly, despite the significant difference in encoder sizes, we do not observe a drastic gap between encoder-only and encoder-decoder models on average. That is, if generation is important downstream, using multi-task models will not hurt retrieval performance in the multi-source-language case to a very large extent as observed in the language combinations we considered. In terms of the ablation cases, we highlight that the results of encoder-decoder architectures trained against generative objectives only (**Gen.**) make it clear that contrastive learning is required in order for embeddings to be semantically structured and language invariant. Moreover, removing generative tasks such as denoising might hurt performance, and the decoder-only model doesn't reach the level of the encoder-decoder setting, and thus the strategy of concentrating the full parameter budget in a decoder is not enough to reach the performance of encoder-only models.

5.3 CODE-TO-CODE EVALUATION

We now push our models further and test their ability to generalize to unseen pairs of languages in the case where target/target combinations are considered. To do so, we use the Python-Java paired data collected from GeeksforGeeks and discussed in Roziere et al. (2020) given by implementations in both language of the solution of a given problem. In other words, models are tasked with querying a codebase in Java using Python snippets. During training however, only natural-programming language or natural-natural language combinations are observed by the models. Results in terms of *au*MRR*c* are reported in Table 4 while MRR curves are displayed in Figure 6. Once more, results indicate that models are clearly able to generalize to unseen combinations of domains, however the gap between encoder-only and the multitask encoder-decoder models grows significantly now that tasks became more difficult. Also, the decoder-only models observed a much bigger gap relative to the other cases in this more challenging scenario. Removing capacity from the encoder seems particularly detrimental to retrieval performance when the task difficulty increases.

Disc.	**Gen.+Disc.**	*Ablations*		
		Gen.	**No denoising**	**Dec.**
81.4%	66.4%	4.7%	61.7%	19.0%

Table 4: Results for code search from code queries in terms of *au*MRR*c* on the Python-Java data of (Roziere et al., 2020).

5.4 CODE SEARCH FROM QUERIES IN ENGLISH

Finally, we run a more standard English to code evaluation using the test set of CodeSearchNet to assess the effect of including extra language combinations in the training set. We thus include a further ablative case where a model is trained using the same objective as the (**Gen.+Disc.**) case, but only the English to code datasets are used for training while all other datasets are dropped. This case is indicated by (**En.**) in Table 5 and in the MRR curves partially displayed in Figure 7. Models that were trained on multiple datasets involving different natural languages perform better than models trained in English to code data only. Moreover, similarly to the multi-source language evaluation discussion in Section 5.2, we did not observe too large a gap between encoder-only

Prog. Lang.	**Disc.**	**Gen.+Disc.**	*Ablations*			
			Gen.	**No denoising**	**En.**	**Dec.**
PHP	33.2%	33.9%	0.3%	33.5%	31.9%	39.0%
JavaScript	40.4%	39.5%	0.7%	39.8%	31.2%	41.0%
Python	65.6%	71.4%	0.3%	66.3%	54.9%	67.2%
Go	61.0%	61.6%	0.9%	60.1%	47.3%	61.1%
Java	37.6%	36.3%	0.5%	37.1%	31.3%	36.4%
Ruby	48.2%	37.0%	1.0%	42.2%	41.0%	47.9%
Avg.	47.7%	46.6%	0.6%	46.5%	39.6%	48.8%

Table 5: Code search from queries in English in terms of *au*MRR*c* for the CodeSearchNet test set.

and encoder-decoder models, suggesting once more that evaluation tasks where source/target pairs are closer to what was observed during training will result in a smaller gap between the two types of models. Finally, contrary to the other evaluations, the decoder-only model was the best performer on average in this case, which leads to the overall recommendation of using this kind of model only if the evaluation source/target pairs match those observed during training.

6 Related work

Other datasets. Acquiring aligned code and natural language pairs is key to solve tasks such as code retrieval and code summarization. Towards this goal, Yin et al. (2018) proposed CoNaLa, a dataset mined from StackOverflow consisting of English intent-snippet pairs with Python and Java code. The same methodology was later used in MCoNaLa (Wang et al., 2022) to provide English, Spanish, Japanese and Russian text to Python pairs. A donwside of these datasets is that they are based on Q&A pairs from StackOverflow and thus they are not suitable for open-ended code generation. Differently, CodeSearchNet (Husain et al., 2019) leverages English comments from GitHub repositories in 6 programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). We follow the same data gathering procedure as CodeSearchNet, extending it by a factor of approximately 25, and adding a new test set with textual queries in Spanish, Portuguese, German, and French.

Code models. The success of transformers for natural language modelling (Vaswani et al., 2017) has motivated researchers to explore their usefulness for code modelling. CodeParrot (Tunstall et al., 2022) was trained with data from GitHub for code completion on a single language. CodeGen (Nijkamp et al., 2022b), PolyCoder (Xu et al., 2022), Codex (Chen et al., 2021), and CodeT5 (Wang et al., 2021) were trained on additional programming languages as well as natural language queries in English. In this work, we enhance a pre-trained CodeT5 to support new additional natural languages such as Spanish, Portuguese, German, and French.

Contrastive methods. Recent work has leveraged contrastive techniques to learn a useful sentence- or document-level embedding space where retrieval can be performed efficiently. GTR Ni et al. (2021) for instance, showed that dual-encoder settings can benefit from scaling up model size. SimCSE, on the other hand, showed that simple strategies such as dropping out tokens leads to surprisingly effective augmentation approaches to create positive pairs for unsupervised learning of sentence embeddings. In settings where the focus lies on finding shared embedding spaces between text and code, DocCoder (Zhou et al., 2022) uses a contrastive schemes to match representations from natural language queries to retrieve documentation later used for code generation. Most similar to our work, `cpt-code` (Neelakantan et al., 2022) was pre-trained with contrastive learning to learn a common space between English text and code. In our work, we add an additional element to the text-code pair consisting of text-text pairs from different languages, resulting in a multilingual model.

7 Conclusion

We showed that one can overcome the lack of multi-language paired data by introducing indirect paths from source to target languages, *i.e.*, a model able to map from source A to source B and from source B to a target, can also map directly from source A to the target. To show evidence of that, we introduced a new dataset referred to as $\texttt{M}^2\texttt{CRB}$ where paired data is available containing multiple natural languages as source and multiple programming languages as target. An extensive empirical evaluation in the code search/retrieval setting was carried out in order to indicate how different design choices influence performance under this out-of-distribution generalization condition we consider. Interestingly, we identified cases where one can safely give up capacity from the encoder to enable generative capabilities without affecting retrieval performance significantly. Similarly, our experiments showed that once tasks become more difficult, then the gap enlarges and having single-task dedicated encoders might be preferable. Moreover, results suggest that adding different data sources that do not correspond to English to code improves English to code performance, such that adding multi-language capabilities in this case does not affect performance in the original task.

REFERENCES

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*, 2022.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

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 preprint arXiv:2102.04664*, 2021.

Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005*, 2022.

Jianmo Ni, Chen Qu, Jing Lu, Zhuyun Dai, Gustavo Hernández Ábrego, Ji Ma, Vincent Y Zhao, Yi Luan, Keith B Hall, Ming-Wei Chang, et al. Large dual encoders are generalizable retrievers. *arXiv preprint arXiv:2112.07899*, 2021.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *ArXiv preprint, abs/2203.13474*, 2022a.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*, 2022b.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pp. 8748–8763. PMLR, 2021.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33: 20601–20611, 2020.

Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*, 2021.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

Jörg Tiedemann. The tatoeba translation challenge–realistic data sets for low resource and multilingual mt. *arXiv preprint arXiv:2010.06354*, 2020.

Lewis Tunstall, Leandro von Werra, and Thomas Wolf. *Natural language processing with transformers*. " O'Reilly Media, Inc.", 2022.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F Xu, and Graham Neubig. Mconala: A benchmark for code generation from multiple natural languages. *arXiv preprint arXiv:2203.08388*, 2022.

Wikimedia-Foundation. Acl 2019 fourth conference on machine translation (wmt19), shared task: Machine translation of news, 2019. URL `http://www.statmt.org/wmt19/translation-task.html`.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*, pp. 476–486. IEEE, 2018.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao JIang, and Graham Neubig. Doccoder: Generating code by retrieving and reading docs. *arXiv preprint arXiv:2207.05987*, 2022.

A Additional data preparation details

M^2CRB as well as the training dataset we built were both mined from The Stack Kocetkov et al. (2022), which contains over 3TB of permissively-licensed source code files from GitHub. The workflow to mine and prepare data, as illustrated in Figure 2, roughly consists of (1)-filtering out repositories that appear in CodeSearchNet, (2)-filtering the files that belong to the programming languages of interest, (3)-filtering the files that likely contain text in the natural languages of interest, (4)-AST parsing, and (5)-performing language identification of docstrings in the resulting set of functions/methods. In further detail, for (2), we simply check files extensions. For (3) on the other hand, for each file, we check the fraction of overlaps between words and the union of vocabularies on the natural languages of interest. We keep only the files for which the fraction is greater than a threshold. Remaining files are AST parsed, and we finally perform language identification with three independent classifiers on each docstring, i.e., we ensemble three different off-the-shelf open-source language classifiers from text.[5,6,7] If a docstring is in English, that function/method is added to traing set. Otherwise, we perform a further step and ask a human to verify the ensemble's prediction. The resulting non-English functions are added to M^2CRB. We remark that we do not exaust The Stack and stop once we find a certain number of natural/programming language combinations.

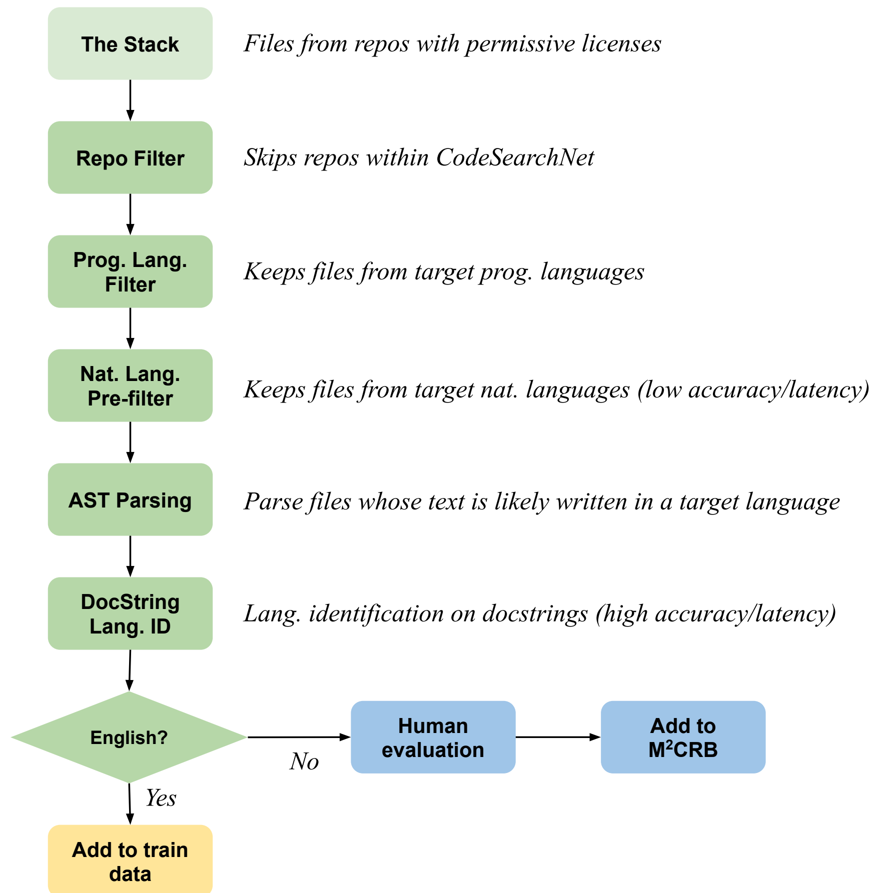

Figure 2: Data preparation workflow.

[5]https://github.com/saffsd/langid.py
[6]https://github.com/google/cld3
[7]https://huggingface.co/papluca/xlm-roberta-base-language-detection.

B MODEL DETAILS

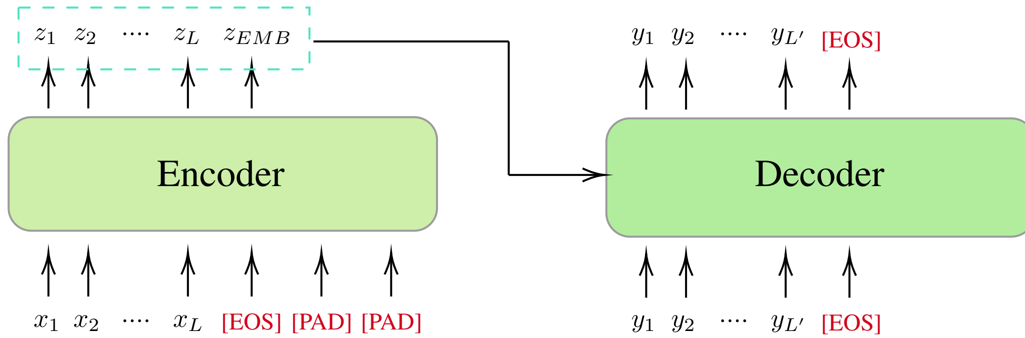

Figure 3: Encoder-decoder pair we consider in our experiments. Contrastive training is performed on top of embeddings obtained at the [EOS] token output by the encoder. Generative tasks, on the other hand, are performed with standard sequence-to-sequence maximum likelihood estimation.

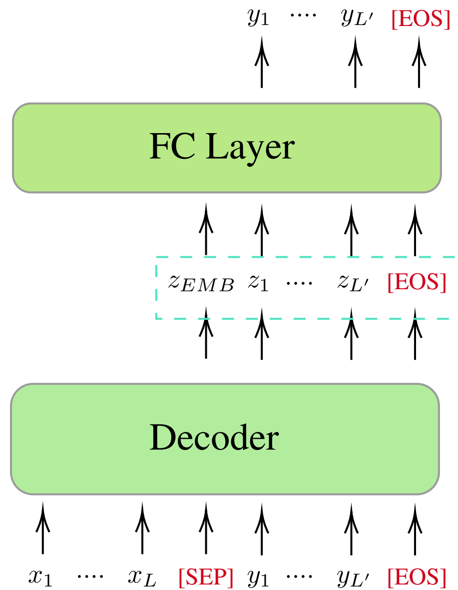

Figure 4: Decoder-only setting we consider in our experiments. Contrastive training is performed on top of embeddings obtained at the separator token [SEP] prior to final fully-connected layer. Generative tasks, on the other hand, are performed with standard sequence-to-sequence maximum likelihood estimation.

Prog. Lang	Nat. Lang.	Disc.		Gen.+Disc.		En.
		Direct	*Translate*	*Direct*	*Translate*	
Python	*de*	58.1%	63.4%	60.2%	65.0%	54.2%
	es	75.5%	69.4%	74.8%	66.9%	60.2%
	fr	64.5%	64.5%	70.3%	66.1%	57.2%
	pt	70.6%	63.5%	66.9%	61.6%	54.7%
Java	*de*	61.6%	36.8%	62.2%	28.5%	25.5%
	es	28.6%	39.1%	22.2%	37.0%	34.5%
	fr	30.8%	55.4%	30.5%	45.9%	48.8%
	pt	27.2%	40.6%	20.8%	35.0%	33.7%
JavaScript	*de*	31.3%	60.4%	23.2%	61.9%	52.2%
	es	37.2%	30.8%	27.3%	28.5%	22.9%
	fr	47.4%	45.5%	39.3%	40.8%	37.3%
	pt	34.8%	35.0%	26.4%	30.6%	23.8%
Avg.		47.3%	50.4%	43.7%	47.3%	42.1%

Table 6: M^{2}CRB's results with an auxiliary translator. Results correspond to the area under the MRR curve (*au*MRR*c*, the higher the better).

C Additional results

C.1 Results with auxiliary translator

In Table 6, we report additional results on M^{2}CRB when an external translator is used. That is, given a textual query, we first translate it into English using some external service, and then query the model using the result. In particular, we used the public Google translate API[8] and compared our models with a model trained with only English to code data. We further report side by side comparisons of our models when we directly translate in the original language versus the cases where we translate beforehand. Both the encoder-only and encoder-decoder models perform better on average than a model trained exclusively on English to code, and that's the case no matter whether or not an auxiliary translator is used. However, both our models benefited from the extra translation component on average, though that's not the case for every language combination.

[8] https://pypi.org/project/googletrans/

C.2 MRR CURVES

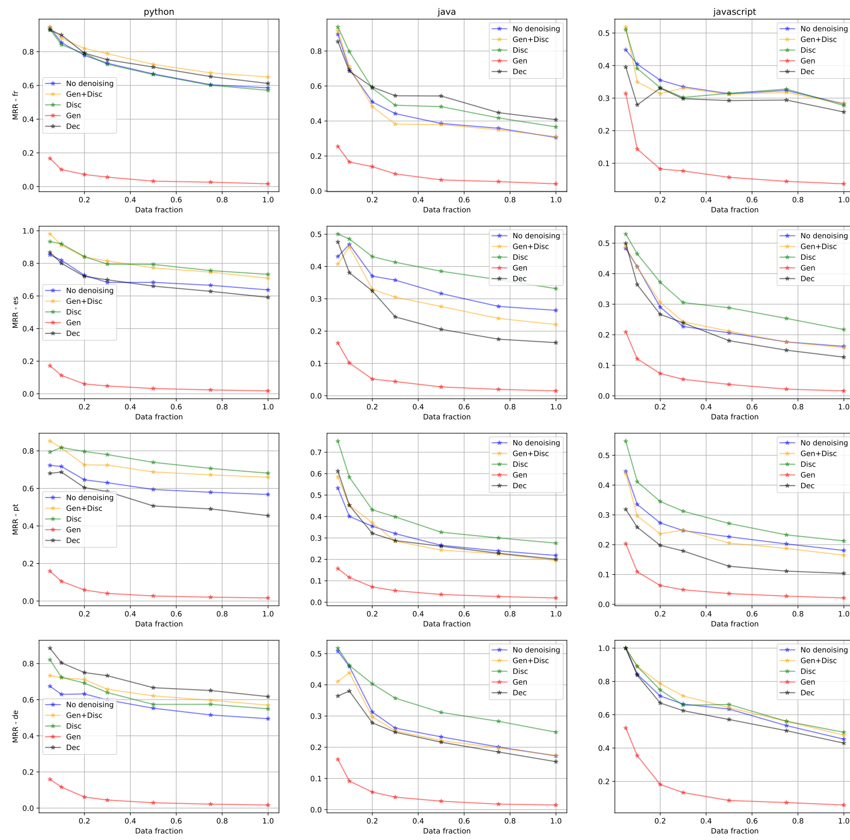

Figure 5: MRR curves for cross natural language evaluation.

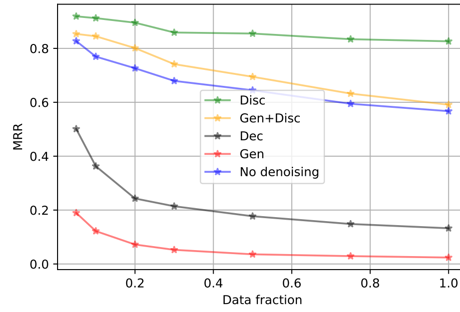

Figure 6: MRR curves for code to code evaluation.

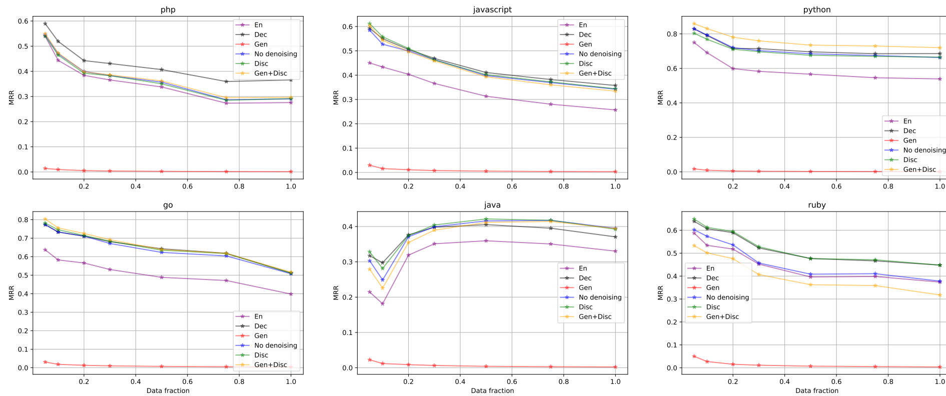

Figure 7: MRR curves for English to code evaluation.

D ADDITIONAL TRAINING DETAILS

In terms of model architectures, our encoder-only models build upon the encoder of the `codet5-base` architecture, while encoder-decoder pairs build upon the `codet5-small`. As discussed in the main text, the encoder of the base configuration has a similar parameter count relative to the full small configuration, so that we study the impact in retrieval given by converting part of an encoder into a decoder to enable multi-tasking. Our decoder-only architecture reuses the `codegen-350M-multi` configuration.

An implementation of the exact variation of $\mathcal{L}_{contrastive}$, similar to CLIP (Radford et al., 2021), is shown in Figure 8. In particular, it shares its minimizers with the loss described in the text, but was observed to be easier to train against in practice. The loss treats the similarity matrix Sim as a batch of categorical log-probabilities, places labels on the main diagonal, and then defines a cross-entropy objective so that entries in the main diagonal are forced to be greater than off-diagonal elements.

Finally, we remark that we found that setting $\alpha = 0.5$ would work well for the encoder-decoder models we consider so that both generative and discriminative objectives are given the same importance. The same was applied for the decoder-only case.

```python
import torch

def contrastive_loss(
   x_source: torch.FloatTensor,
   y_target: torch.FloatTensor) -> torch.FloatTensor:
   """Computes contrastive loss.

   Args:
      x_source (torch.FloatTensor): Batch of normalized source
          embeddings. Expected shape is [batch_size, embedding_dim].
      y_target (torch.FloatTensor): Batch of normalized source
          embeddings. Expected shape is [batch_size, embedding_dim].

   Returns:
      torch.FloatTensor: Contrastive loss.
   """

   # Compute similarity matrix.
   sim = x_source @ y_target.T

   # A row-wise cross-entropy criterion is used
   # with labels placed on the main diagonal.
   ce_labels = torch.arange(sim.size(0)).long()
   contrastive_loss = torch.nn.functional.cross_entropy(
      sim,
      ce_labels)

   return contrastive_loss
```

Figure 8: Pytorch implementation of the contrastive loss we consider taking as inputs normalized representations of source/target data.