Unsupervised Translation of Programming Languages

Medium article: *https://medium.com/@priyanka.math/unsupervised-translation-of-programming-languages-9d538c64096f*

SlideShare Link: *https://www.slideshare.net/PriyaM781673/transcoder-250754329*

A transcompiler is a system that converts source code from one high-level programming language (such as C++ or Python) to another. Transcompilers are primarily used for interoperability, and to port codebases written in an obsolete languages (e.g. COBOL, Python 2) to a modern one. They typically rely on handcrafted rewrite rules applied on source code Abstract Syntax Tree (AST) that often lack readability, fail to comply with the target language conventions, and require manual modifications in order to work properly. The overall translation process is time-consuming and requires expertise in both the source and target languages. Although neural models significantly outperform their rule-based counterparts in the context of natural language translation, their applications to transcompilation have been limited due to the scarcity of parallel data in this domain. In this article, we will leverage recent approaches in unsupervised machine translation to train a fully unsupervised neural transcompiler.

'TransCoder' model is trained on source code from open source GitHub projects, and show that it can translate functions between C++, Java, and Python with high accuracy. This approach relies exclusively on monolingual source code, requires no expertise in the source or target languages, and can easily be generalized to other programming languages. To evaluate the model, a test set of 852 parallel functions, along with associated unit tests are used to check the correctness of translations and show that the model outperforms rule-based approaches by a significant margin.

For 'TransCoder', we consider a sequence-to-sequence (seq2seq) model with attention [2], composed of an encoder and a decoder with a transformer architecture [3]. We use a single shared model for all programming languages.

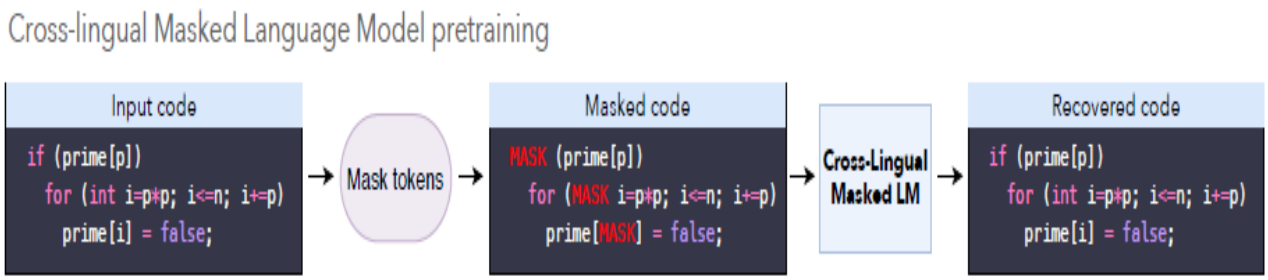
'TransCoder' is trained using three principles of unsupervised machine translation,

1. Initialization
2. Language modeling
3. Back-translation

Let's summarize these principles and learn how we instantiate them to translate programming languages.

1. Cross Programming Language Model pretraining (Initialization)

The first principle initializes the model with cross-lingual masked language model pretraining. As a result, pieces of code that express the same instructions are mapped to the same representation, regardless of the programming language.



Pretraining is a key ingredient of unsupervised machine translation. It ensures that sequences with a similar meaning are mapped to the same latent representation, regardless of their languages. Originally, pretraining was done by initializing the model with cross-lingual word representations. Subsequent work showed that pretraining the entire model (and not only word representations) in a cross-lingual way could lead to significant improvements in unsupervised machine translation.

In particular, 'TransCoder' follows the pretraining strategy where a Cross-lingual Language Model (XLM) is pretrained with a masked language modeling objective on monolingual source code datasets.

The cross-lingual nature of the resulting model comes from the significant number of common tokens (anchor points) that exist across languages. In the context of English-French translation, the anchor points consists essentially of digits and city and people names. In programming languages, these anchor points come from common keywords (e.g. for, while, if, try), and also digits, mathematical operators, and English strings that appear in the source code.

For the masked language modeling (MLM) objective, at each iteration we consider an input stream of source code sequences, randomly mask out some of the tokens, and train 'TransCoder' to predict the tokens that have been masked out based on their contexts. We alternate between streams of batches of different languages. This allows the model to create high quality, cross-lingual sequence representations.

2. Denoising auto-encoding (Language modeling)

Denoising auto-encoding, the second principle, trains the decoder to always generate valid sequences, even when fed with noisy data, and increases the encoder robustness to input noise.

The encoder and decoder of the seq2seq model is initialized with the XLM model pretrained in the previous step.

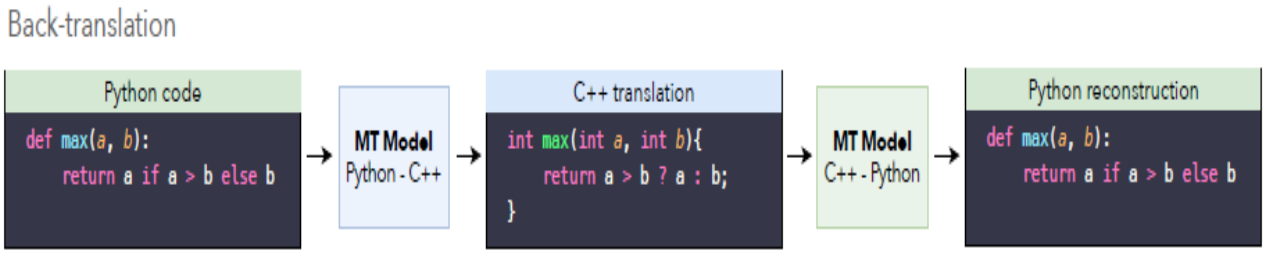
XLM pretraining allows the seq2seq model to generate high quality representations of input sequences. However, the decoder lacks the capacity to translate, as it has never been trained to decode a sequence based on a source representation. To address this issue, we train the model to encode and decode sequences with a Denoising Auto-Encoding (DAE) objective. The DAE objective operates like a supervised machine translation algorithm, where the model is trained to predict a sequence of tokens given a corrupted version of that sequence.

The first symbol given as input to the decoder is a special token indicating the output programming language. At test time, a Python sequence can be encoded by the model, and decoded using the C++ start symbol to generate a C++ translation. The quality of the C++ translation will depend on the "cross-linguality" of the model: if the Python function and a valid C++ translation are mapped to the same latent representation by the encoder, the decoder will successfully generate this C++ translation.

The DAE objective also trains the "language modeling" aspect of the model, i.e. the decoder is always trained to generate a valid function, even when the encoder output is noisy. Moreover it also trains the encoder to be robust to input noise, which is helpful in the context of back-translation where the model is trained with noisy input sequences.

3. Back-translation

Back-translation, the last principle, allows the model to generate parallel data which can be used for training. Whenever the Python → C++ model becomes better, it generates more accurate data for the C++ → Python model, and vice versa.



In practice, XLM pretraining and denoising auto-encoding alone are enough to generate translations. However, the quality of these translations tends to be low, as the model is never trained to do what it is expected to do at test time, i.e. to translate functions from one language to another. To address this issue, we use back-translation(feedback), which is one of the most effective methods to leverage monolingual data in a weakly(semi)-supervised scenario. Initially, it was introduced to improve the performance of machine translation in the supervised setting, back-translation turned out to be an important component of unsupervised machine translation.

In the unsupervised setting, a forward source-to-target model is coupled with a backward target-to-source model trained in parallel. The target-to-source model is used to translate target sequences into the source language, producing noisy source sequences corresponding to the ground truth target sequences. The source-to-target model is then trained in a weakly supervised manner to reconstruct the target sequences from the noisy source sequences generated by the target-to-source model, and vice versa. The two models are trained in parallel until convergence.

Experiments

Training data:

GitHub public dataset available on Google BigQuery is used as dataset. Projects whose license explicitly permits the re-distribution of parts of the project are filtered and the C++, Java, and Python files within those projects are selected. Ideally, a transcompiler should be able to translate whole projects. But, 'TransCoder' translates at function level. Unlike files or classes, functions are short enough to fit into a single batch, and working at function level allows for a simpler evaluation of the model with unit tests. 'TransCoder' pretrains on all source code available, and train the denoising auto-encoding and back-translation objectives on functions only. I was observed that keeping comments in the source code increases the number of anchor points across languages, which results in a better overall performance. Therefore, comments were kept in the final datasets and experiments.

Preprocessing:

Recent approaches in multilingual natural language processing tend to use a common tokenizer, and a shared vocabulary for all languages. This reduces the overall vocabulary size, and maximizes the token overlap between languages, improving the cross-linguality of the model. In our case, a universal tokenizer would be suboptimal, as different languages use different patterns and keywords. The logical operators && and || exist in C++ where they should be tokenized as a single token, but not in Python. The indentations are critical in Python as they define the code structure, but have no meaning in languages

like C++ or Java. We use the javalang5 tokenizer for Java, the tokenizer of the standard library for Python6, and the clang7 tokenizer for C++. These tokenizers ensure that meaningless modifications in the code (e.g. adding extra new lines or spaces) do not have any impact on the tokenized sequence.

Python function v1

```
def rm_file(path):
    try:
        os.remove(path)
        print("Deleted")
    except:
        print("Error while deleting file", path)
```

Python function v2

```
def rm_file(path):

    try:
        os.remove( path )
        print( "Deleted" )
    except  :
        print("Error while deleting file", path)
```

```
def rm_file ( path ) : NEWLINE try : NEWLINE INDENT os . remove (path) NEWLINE print ( " Deleted " )
DEDENT except : NEWLINE INDENT print ( " Error _ while _ deleting _ file " , path ) DEDENT
```

Figure 3: **Example of function tokenization.** We show two versions of the same Python function and their common tokenization. These function versions differ by extra spaces and one extra new line. Our Python tokenizer is robust to extra spaces and extra new lines except in strings. In strings, spaces are tokenized as ▁ (U+2581). Indentation is meaningful in Python: indented blocks are surrounded by INDENT DEDENT tokens.

Evaluation:

GeeksforGeeks is an online platform with computer science and programming articles. It gathers many coding problems and presents solutions in several programming languages. From these solutions, set of parallel functions in C++, Java, and Python are extracted, to create validation and test sets. These functions not only return the same output, but also compute the result with similar algorithm.

C++

```
bool checkDivisibility(string num){
  int length = num.size();
  if(length == 1 && num[0] == '0')
    return true;
  if(length % 3 == 1){
    num += "00";
    length += 2;
  }
  else if(length % 3 == 2){
    num += '0';
    length += 1;
  }

  int sum = 0, p = 1;
  for(int i = length - 1;
            i >= 0; i--){
    int group = 0;
    group += num[i--] - '0';
    group += (num[i--] - '0') * 10;
    group += (num[i] - '0') * 100;
    sum = sum + group * p;
    p *= (-1);
  }

  sum = abs(sum);
  return (sum % 13 == 0);
}
```

Java

```
static boolean checkDivisibility(
                String num){
  int length = num.length();
  if(length == 1 && num.charAt(0) == '0')
    return true;
  if(length % 3 == 1){
    num += "00";
    length += 2;
  }
  else if(length % 3 == 2){
    num += "0";
    length += 1;
  }

  int sum = 0, p = 1;
  for(int i = length - 1; i >= 0; i--){
    int group = 0;
    group += num.charAt(i--) - '0';
    group += (num.charAt(i--) - '0') * 10;
    group += (num.charAt(i) - '0') * 100;
    sum = sum + group * p;
    p *= (-1);
  }

  sum = Math.abs(sum);
  return (sum % 13 == 0);
}
```

Python

```
def checkDivisibility(num):
  length = len(num)
  if(length == 1 and num[0] == '0'):
    return True
  if(length % 3 == 1):
    num = str(num) + "00"
    length += 2
  elif(length % 3 == 2):
    num = str(num) + "0"
    length += 1

  sum = 0
  p = 1
  for i in range(length - 1, -1, -1):
    group = 0
    group += ord(num[i]) - ord('0')
    i -= 1
    group += (ord(num[i]) - ord('0')) * 10
    i -= 1
    group += (ord(num[i]) - ord('0')) * 100
    sum = sum + group * p
    p *= (-1)

  sum = abs(sum)
  return (sum % 13 == 0)
```

Figure 4: **Example of parallel function from our test set.** We extracted parallel functions from GeeksforGeeks to create validation and test sets. Here, we have the parallel implementations in C++, Java, and Python of the checkDivisibility function, which determines whether a long integer represented as a string is divisible by 13.

Analysis of results:

- TransCoder successfully understands the syntax specific to each language, learns data structures and their methods, and correctly aligns libraries across programming languages. For instance, it learns to translate the ternary operator “X ? A : B” in C++ or Java to “if X then A else B” in Python, in an unsupervised way.
- TransCoder successfully map tokens with similar meaning to the same latent representation, regardless of their languages.
- TransCoder can adapt to small modifications.

Example of unsupervised translation from Python to C++:

Python input

```python
def SumOfKsubArray(arr, n, k):
  Sum = 0
  S = deque()
  G = deque()
  for i in range(k):
    while (len(S) > 0 and arr[S[-1]] >= arr[i]):
      S.pop()
    while (len(G) > 0 and arr[G[-1]] <= arr[i]):
      G.pop()
    G.append(i)
    S.append(i)
  for i in range(k, n):
    Sum += arr[S[0]] + arr[G[0]]
    while (len(S) > 0 and S[0] <= i - k):
      S.popleft()
    while (len(G) > 0 and G[0] <= i - k):
      G.popleft()
    while (len(S) > 0 and arr[S[-1]] >= arr[i]):
      S.pop()
    while (len(G) > 0 and arr[G[-1]] <= arr[i]):
      G.pop()
    G.append(i)
    S.append(i)
  Sum += arr[S[0]] + arr[G[0]]
  return Sum
```

Unsupervised C++ translation

```cpp
int SumOfKsubArray(int arr[], int n, int k){
  int Sum = 0;
  deque <int> S;
  deque <int> G;
  for(int i = 0; i < k; i ++){
    while((int) S.size() > 0 && arr[S.back()] >= arr[i])
      S.pop_back();
    while((int) G.size() > 0 && arr[G.back()] <= arr[i])
      G.pop_back();
    G.push_back(i);
    S.push_back(i);
  }
  for(int i = k; i < n; i ++){
    Sum += arr[S.front()] + arr[G.front()];
    while((int) S.size() > 0 && S.front() <= i - k)
      S.pop_front();
    while((int) G.size() > 0 && G.front() <= i - k)
      G.pop_front();
    while((int) S.size() > 0 && arr[S.back()] >= arr[i])
      S.pop_back();
    while((int) G.size() > 0 && arr[G.back()] <= arr[i])
      G.pop_back();
    G.push_back(i);
    S.push_back(i);
  }
  Sum += arr[S.front()] + arr[G.front()];
  return Sum;
}
```

Figure 2: **Example of unsupervised Python to C++ translation**. TransCoder successfully translates the Python input function `SumOfKsubArray` into C++. TransCoder infers the types of the arguments, of the variables, and the return type of the function. The model maps the Python `deque()` container, to the C++ implementation `deque<>`, and uses the associated `front`, `back`, `pop_back` and `push_back` methods to retrieve and insert elements into the `deque`, instead of the Python square brackets `[]`, `pop` and `append` methods. Moreover, it converts the Python `for` loop and `range` function properly.

Example of failed 'TransCoder' translations:

Input	Java failed translations	Description
`bool isEven (int n){ return (!(n & 1)); }`	`static boolean isEven(int n){ return (!(n & 1)); }`	The `!` operator works on boolean and integers in C++ (it returns `true` if the integer is positive) but it only works on boolean in Java.
`int summingSeries(long n){ return pow(n, 2); }`	`static int summingSeries(long n){ return Math.pow(n, 2); }`	In Java, `Math.pow(n, 2)` returns a `double` which should be cast to `int` to match the function return type.
`def minSum(A): min_val = min(A) return min_val * (len(A) - 1)`	`static double minSum(double[] A){ double minVal = Math.min(A); return minVal*(A.length - 1); }`	`Math.min` is a Java function but does not take as input a `double[]` array but a pair of `double`.

Figure 11: **Examples of failed TransCoder translations.** TransCoder fails to translate these C++ and Python functions into Java, showing its limitations. In these examples, it fails to account for the variable types when using a method or an operator. In particular, the NOT operator `!` in C++ should have been translated to ~ in Java, because it is applied to an integer. Similarly, the `Math.min` function in Java cannot be applied to arrays.

References:

[1] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot and Guillaume Lample. Unsupervised Translation of Programming Languages. arXiv preprint arXiv:2006.03511, 2020

[2] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Advances in neural information processing systems, pages 3104–3112, 2014

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in neural information processing systems, pages 5998–6008, 2017.