UNIVERSITY OF TORONTO

ECE 1786 – Creative Applications of Natural Language Processing

“Eng2Py” Project Final Report

10th December 2022

Course Instructor: Prof. Jonathan Rose

Team Members

Dhairya Parmar - 1006859516
Yuchen Dai - 1008242313

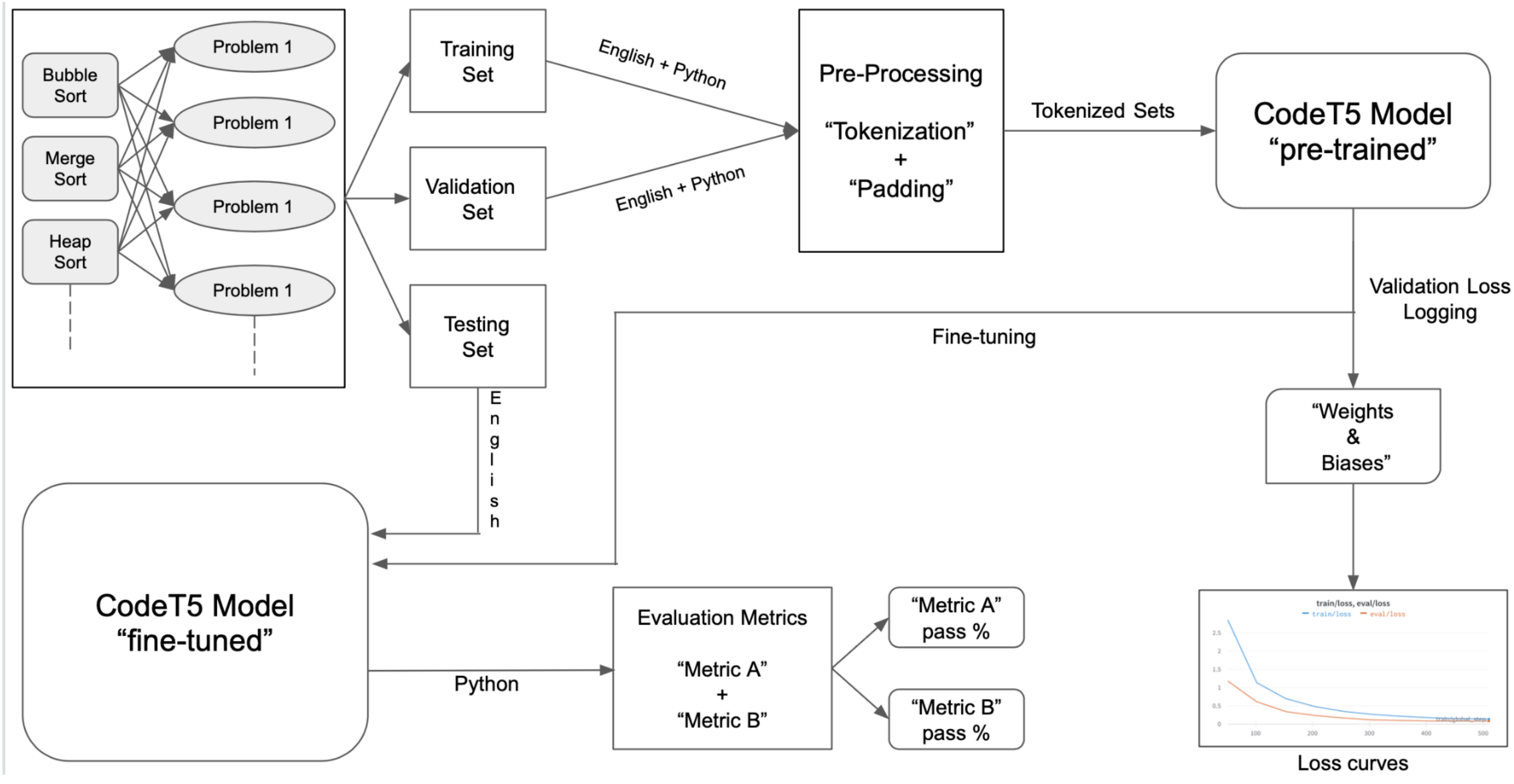
Word Count - 1982 words

Introduction:

The goal of our project is to generate code solutions to basic python programming problems pertaining to sorting algorithms using natural language queries. In very simple terms, the problem that we aim to solve is, given an English instruction to write a Python program for a sorting problem, to generate python code that attempts to solve the problem. The purpose of focusing on sorting problems is to explore if a language model can learn a specific class of problems better.

The motivation behind our project is creating a piece of software that can help software developers save keystrokes or avoid writing dull pieces of code and help non-programmers in other fields, who require computation in their daily work, in creating data manipulation scripts.

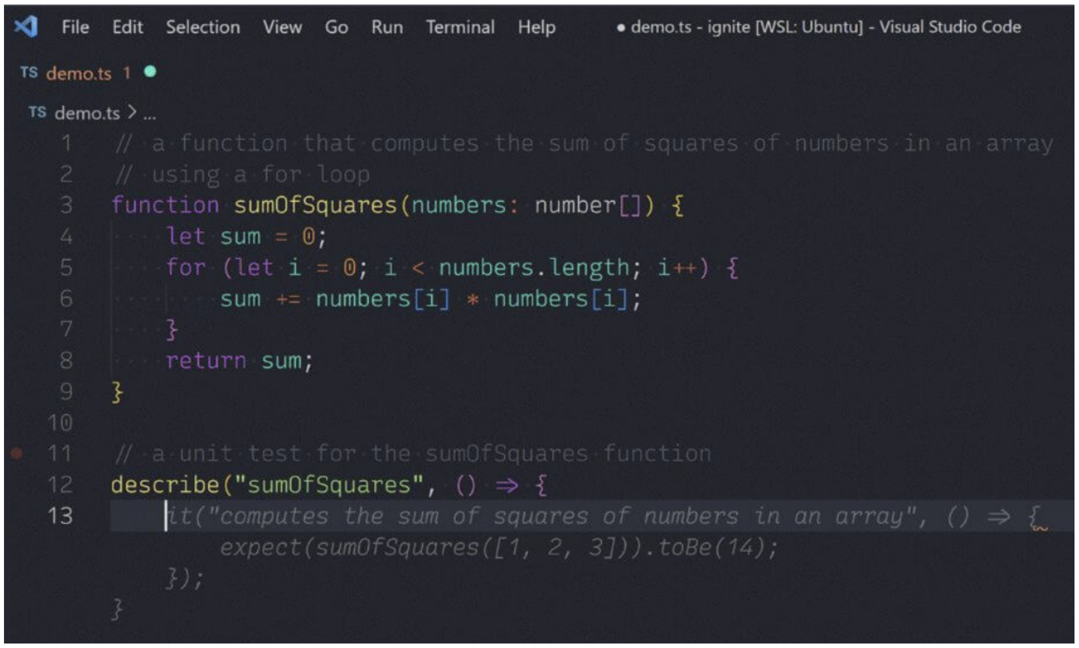
Project Illustration:



Background:

Code Generation is a long standing goal of artificial intelligence. Recent developments in achieving generation of functionally correct code has shown great potential. Wang, Yue et al. **[1]** proposes an encoder-decoder style transformer model CodeT5 for different types of natural language to programming language and vice-versa tasks. The literature also proposes a novel identifier-aware pre-training task that enables the model to distinguish which code tokens are identifiers and to recover them when they are masked. After pre-training this model is then fine-tuned for downstream tasks like Code Summarization, Code Generation, Code Translation, Code Refinement and Defect Detection. The literature also evaluates this model on standard code-related benchmark tasks and compares its performance against other models designed for programming language related tasks.

Chen, Mark et al. **[2]** introduces a new set of GPT style models fine-tuned on code called Codex models. The inspiration for these models was to investigate whether it was possible to train large language models to produce functionally correct code bodies from natural language docstrings. These Codex models displayed impressive performance in code generation tasks. Moreover the literature also proposes an evaluation metric called 'pass@k' to evaluate functional correctness, where k code samples are generated per problem and a problem is considered solved if any sample passes a set of unit tests, and the total fraction of problems solved is reported. The latter GPT-3 style Codex models power the most state-of-art code generation application Github Copilot, a demo of which can be seen in the image below,



```
// a function that computes the sum of squares of numbers in an array
// using a for loop
function sumOfSquares(numbers: number[]) {
    let sum = 0;
    for (let i = 0; i < numbers.length; i++) {
        sum += numbers[i] * numbers[i];
    }
    return sum;
}

// a unit test for the sumOfSquares function
describe("sumOfSquares", () => {
    it("computes the sum of squares of numbers in an array", () => {
        expect(sumOfSquares([1, 2, 3])).toBe(14);
    });
}
```

Img Ref. "https://www.onmsft.com/news/microsoft-github-copilot-openai"

This project is inspired by the results of Github Copilot and the supervised fine-tuning approach used to build the Codex models.

Data and Data Processing:

All of our data is hand-made, the strategy for data creation involves combining different sorting algorithms with simple problems that can be solved using sorted arrays. A total of 331 samples were created using 33 sorting algorithms and their variants and 10 simple problems. An example data sample looks like below,

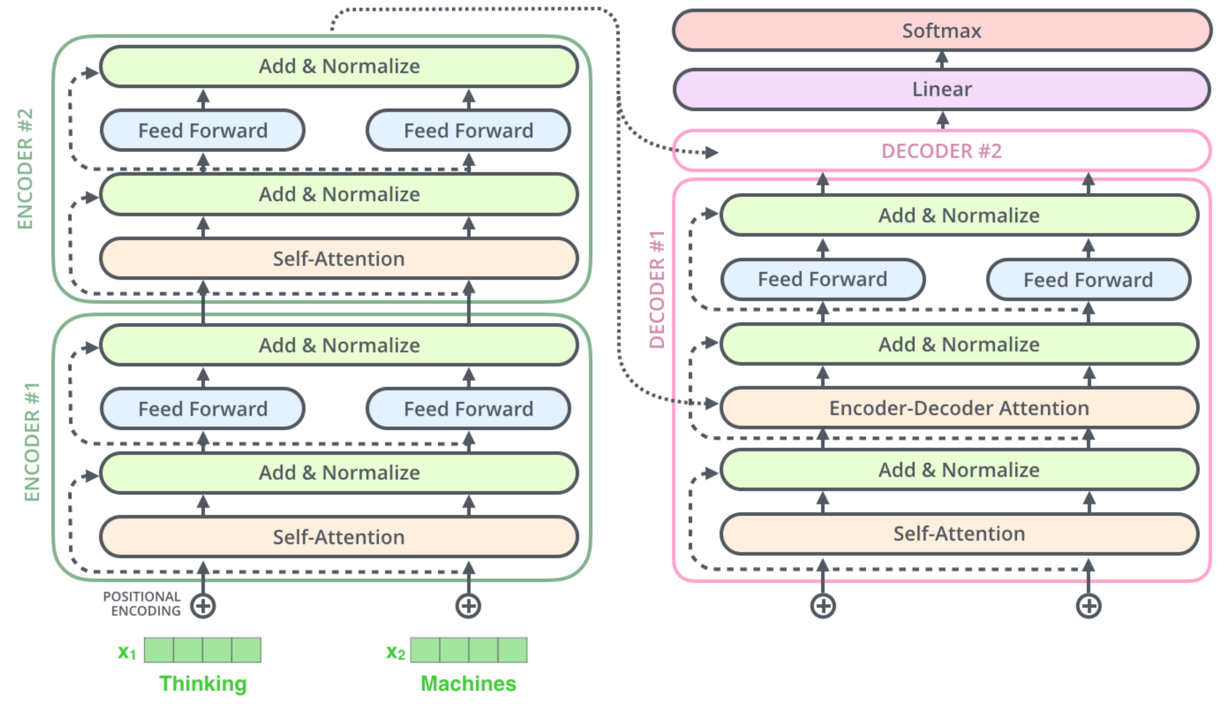
```python
#218 Write a program to print an array of the squares of each number sorted
#    in non-decreasing order of a given array using Gnome Sort

def gnomeSort(arr):
    n = len(arr)
    index = 0
    while index < n:
        if index == 0:
            index = index + 1
        if arr[index] >= arr[index - 1]:
            index = index + 1
        else:
            arr[index], arr[index-1] = arr[index-1], arr[index]
            index = index - 1

    return arr

arr = [-2,-1,0,3,4]
for i in range(len(arr)):
    arr[i] *= arr[i]
print(gnomeSort(arr))
```

The training, validation and test splits were manually created from this dataset since it was desired to have splits that were stratified on sorting algorithms and evaluation sets to be composed of truly unseen combinations not used in the training set. In addition to these, we added 14 new samples to the testing set. Finally, the splits are as follows, 202 training samples, 28 validations samples and 101 testing samples.

To prepare data for fine-tuning we tokenized the data, implemented padding to the longest sequence and assigned a separate token for the words to be ignored, for both english and python inputs.

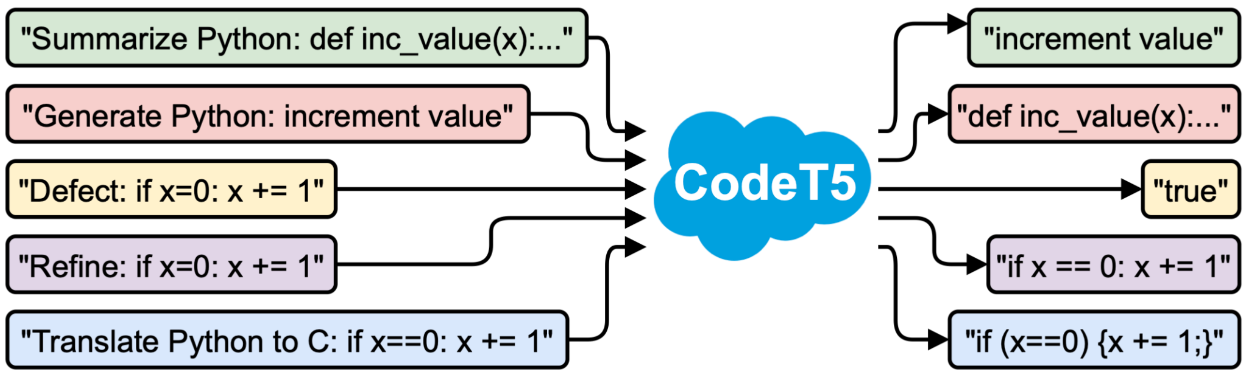
Architecture & Software:

The model architecture used for this project is the CodeT5 model, a model with the same architecture as the standard T5 model (encoder-decoder model as shown in the figure below) but with a special code-specific tokenizer that can leverage the syntactic structure a code snippet possesses



Img Ref. "http://jalammar.github.io/illustrated-transformer/"

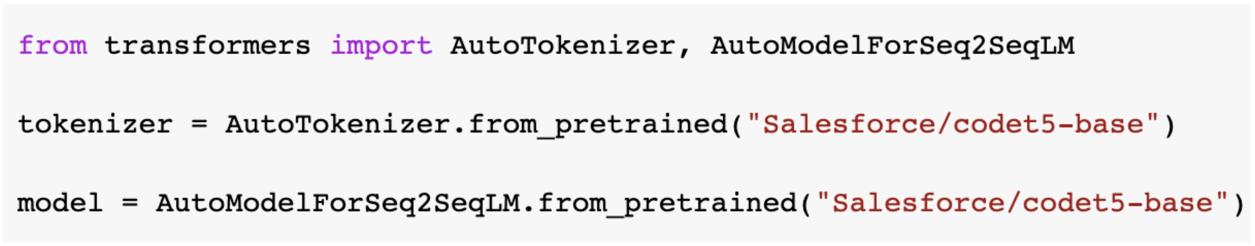
The CodeT5, analogous to the T5 model, can also perform multiple tasks like code summarization, code translation, code generation etc.,



Img Ref. "https://medium.com/analytics-vidhya/t5-a-detailed-explanation-a0ac9bc53e51"

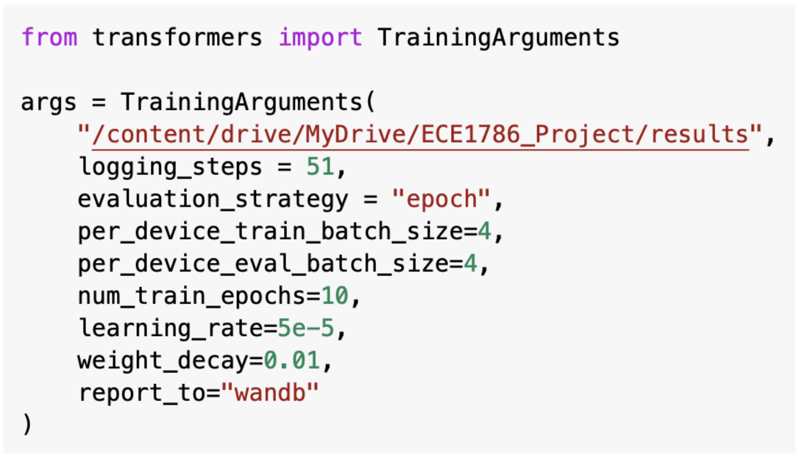
As shown in the figure above, The task “Generate Python:” is a direct application of what we aim to achieve. For this project, the largest CodeT5 model we can implement, complying with the computational resources we have, is the base-sized model. The encoder and decoder layer stacks consist of 12 transformer blocks each (each block comprising self-attention, optional encoder-decoder attention and a feed-forward network). The feed-forward networks in each block consist of a dense layer with an output dimensionality of $d_{ff} = 3072$ followed by a ReLU nonlinearity and another dense layer. The “key” and “value” matrices of all attention mechanisms have an inner dimensionality of $d_{kv} = 64$ and all attention mechanisms have 12 heads. All other sub-layers and embeddings have a dimensionality of $d_{model} = 768$. In total, this results in a model with about 220 million parameters.

This model can be found on huggingface and to reproduce this one can follow the code provided in the figure below,



```python
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

tokenizer = AutoTokenizer.from_pretrained("Salesforce/codet5-base")

model = AutoModelForSeq2SeqLM.from_pretrained("Salesforce/codet5-base")
```

The hyperparameter setting for the fine-tuning mode are as follows,



```python
from transformers import TrainingArguments

args = TrainingArguments(
    "/content/drive/MyDrive/ECE1786_Project/results",
    logging_steps = 51,
    evaluation_strategy = "epoch",
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    num_train_epochs=10,
    learning_rate=5e-5,
    weight_decay=0.01,
    report_to="wandb"
)
```

Apart from the usual pytorch implementation of fine-tuning a model from huggingface we made use of APIs like ‘Pandas’ for data processing and ‘Weights and Biases’ for plotting learning curves.

Baseline Model:

As our project evolved, we decided to include the small variant of the codet5 model as our baseline model. The reason was that we would like to find out if we improved the results in a meaningful way since our main model is a larger variant of the original codet5 model.

The small variant model has the same architecture as our main model, as described in the above section. The only difference is that it has around 60M parameters, compared to 220M of the main model.

Quantitative Results:

Our original thought was to hand-label the results based on the completeness and correctness of the generated code. After generating some outputs of our model, less than half of the results pass this strict metric.

However, we observed that some of the failed samples correctly generated the sorting algorithms, which was our goal. So, besides the strict metric, we proposed another metric that only measures based on the completeness and correctness of the sorting algorithms. This new metric well-fitted our project and would provide a better score.

We named these two metrics Metric A and B as follows:

- Metric A: Pass if the generated code has the correct syntax and completes the task of the input, otherwise, fail

- Metric B: Pass if the generated code has the correct syntax of the sorting algorithm even if it does not complete the task, otherwise, fail

Regarding the quantitative results of the baseline model and the fine-tuned model, we hand-labeled all our test data based on these two metrics and computed a final score as a more direct view. The labels and observations for each one were included in the notebooks.

We concluded that there was no need to compute a final score for the baseline model since the generated results were bad and we could barely find a sample with correct syntax. We will show sample outputs of this model in the Qualitative Results section below.

For our fine-tuned model, we tested 101 samples in total and the accuracy for Metric A was 46.53% and the same for Metric B was 68.31%. We couldn’t say this result was impressive, but compared to that of the original codet5 model, we believe that we improved the results in a meaningful way.

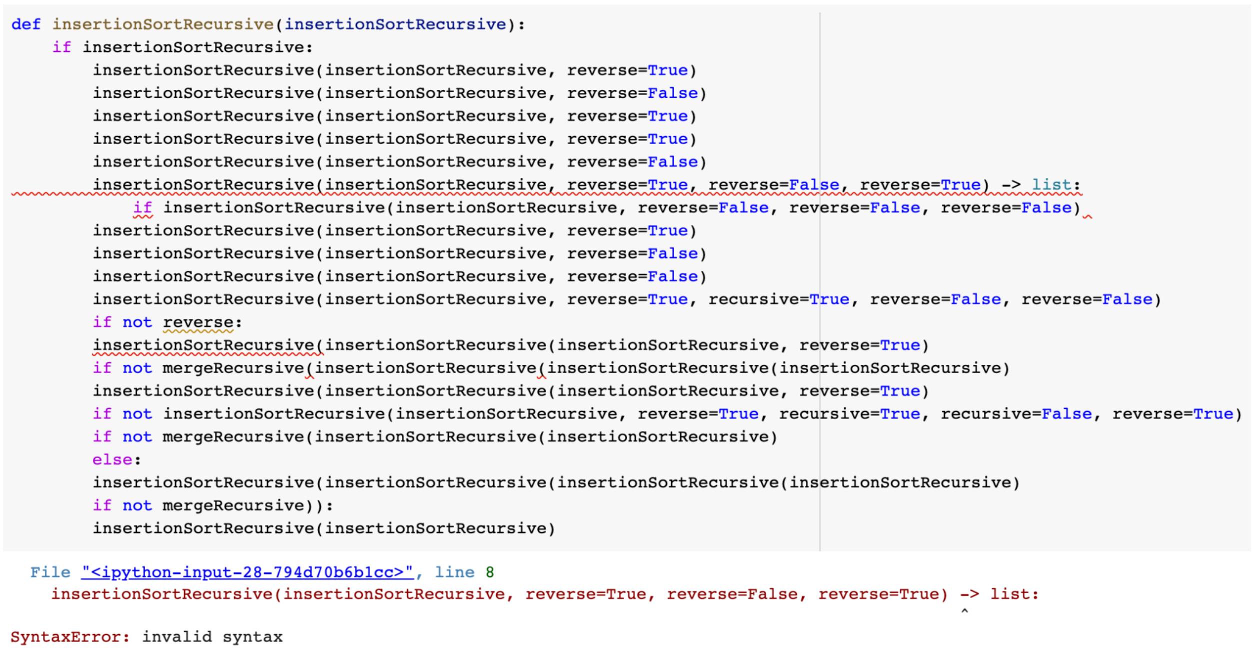
Qualitative Results:

To show the results in detail, several representative samples are shown below:

Sample 1:

Input: write a python program for Recursive Insertion Sort

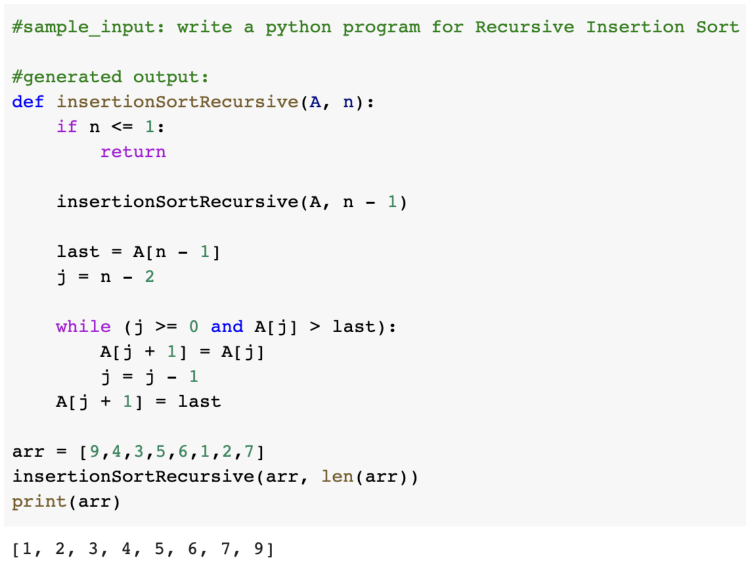
Baseline Result:



```
def insertionSortRecursive(insertionSortRecursive):
    if insertionSortRecursive:
        insertionSortRecursive(insertionSortRecursive, reverse=True)
        insertionSortRecursive(insertionSortRecursive, reverse=False)
        insertionSortRecursive(insertionSortRecursive, reverse=True)
        insertionSortRecursive(insertionSortRecursive, reverse=True)
        insertionSortRecursive(insertionSortRecursive, reverse=False)
        insertionSortRecursive(insertionSortRecursive, reverse=True, reverse=False, reverse=True) -> list:
            if insertionSortRecursive(insertionSortRecursive, reverse=False, reverse=False, reverse=False)
        insertionSortRecursive(insertionSortRecursive, reverse=True)
        insertionSortRecursive(insertionSortRecursive, reverse=False)
        insertionSortRecursive(insertionSortRecursive, reverse=False)
        insertionSortRecursive(insertionSortRecursive, reverse=True, recursive=True, reverse=False, reverse=False)
        if not reverse:
        insertionSortRecursive(insertionSortRecursive(insertionSortRecursive, reverse=True)
        if not mergeRecursive(insertionSortRecursive(insertionSortRecursive(insertionSortRecursive)
        insertionSortRecursive(insertionSortRecursive(insertionSortRecursive, reverse=True)
        if not insertionSortRecursive(insertionSortRecursive, reverse=True, recursive=True, recursive=False, reverse=True)
        if not mergeRecursive(insertionSortRecursive(insertionSortRecursive)
        else:
        insertionSortRecursive(insertionSortRecursive(insertionSortRecursive(insertionSortRecursive)
        if not mergeRecursive)):
        insertionSortRecursive(insertionSortRecursive)
```

```
  File "<ipython-input-28-794d70b6b1cc>", line 8
    insertionSortRecursive(insertionSortRecursive, reverse=True, reverse=False, reverse=True) -> list:
                                                                                               ^

SyntaxError: invalid syntax
```

Fine-tuned Model Result - Pass Metric A & B:



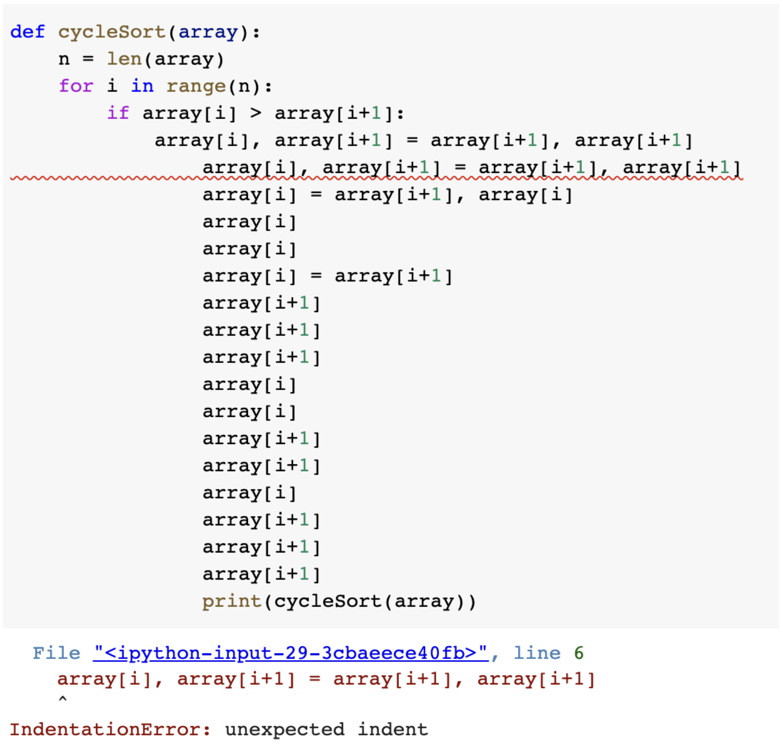
```
#sample_input: write a python program for Recursive Insertion Sort

#generated output:
def insertionSortRecursive(A, n):
    if n <= 1:
        return

    insertionSortRecursive(A, n - 1)

    last = A[n - 1]
    j = n - 2

    while (j >= 0 and A[j] > last):
        A[j + 1] = A[j]
        j = j - 1
    A[j + 1] = last

arr = [9,4,3,5,6,1,2,7]
insertionSortRecursive(arr, len(arr))
print(arr)
```

```
[1, 2, 3, 4, 5, 6, 7, 9]
```

Sample 2:

Input: using Cycle Sort, sort an input array and print the third maximum element

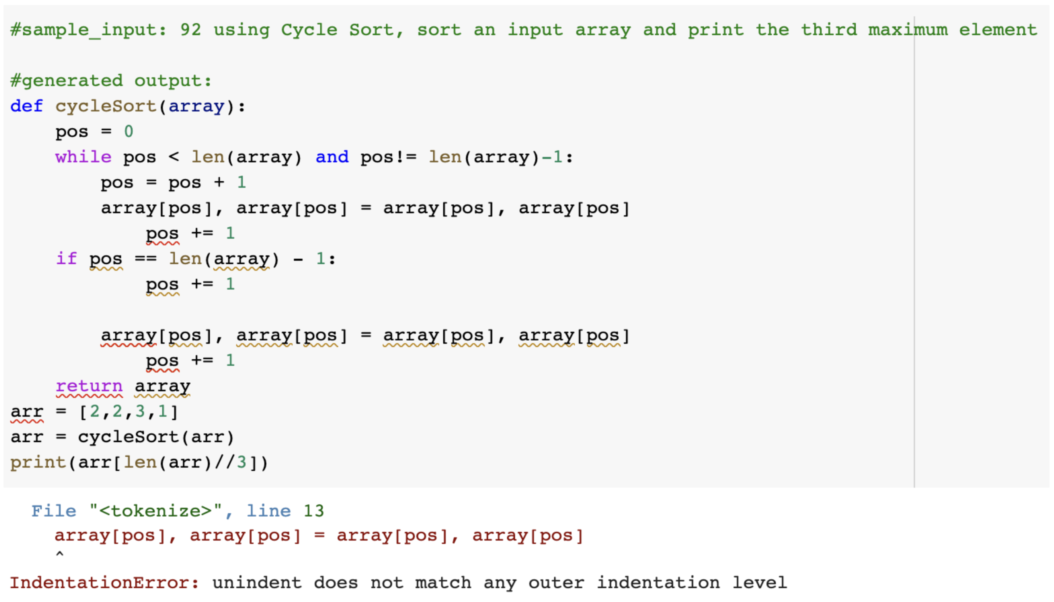
Baseline Result:



```
def cycleSort(array):
    n = len(array)
    for i in range(n):
        if array[i] > array[i+1]:
            array[i], array[i+1] = array[i+1], array[i+1]
                array[i], array[i+1] = array[i+1], array[i+1]
                array[i] = array[i+1], array[i]
                array[i]
                array[i]
                array[i] = array[i+1]
                array[i+1]
                array[i+1]
                array[i+1]
                array[i]
                array[i]
                array[i+1]
                array[i+1]
                array[i]
                array[i+1]
                array[i+1]
                array[i+1]
                print(cycleSort(array))
```

```
  File "<ipython-input-29-3cbaeece40fb>", line 6
    array[i], array[i+1] = array[i+1], array[i+1]
    ^
IndentationError: unexpected indent
```

Fine-tuned Model Result - Fail Metric A & B:



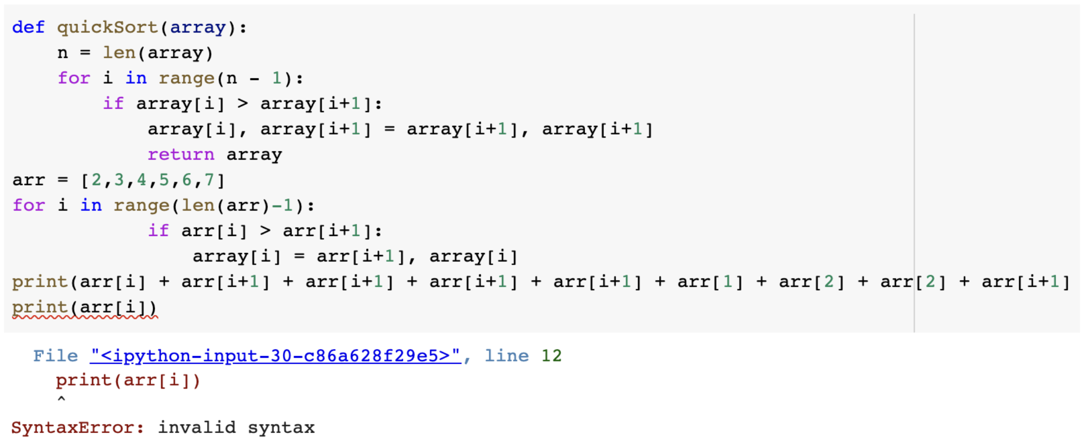
```
#sample_input: 92 using Cycle Sort, sort an input array and print the third maximum element

#generated output:
def cycleSort(array):
    pos = 0
    while pos < len(array) and pos!= len(array)-1:
        pos = pos + 1
        array[pos], array[pos] = array[pos], array[pos]
            pos += 1
    if pos == len(array) - 1:
            pos += 1

        array[pos], array[pos] = array[pos], array[pos]
            pos += 1
    return array
arr = [2,2,3,1]
arr = cycleSort(arr)
print(arr[len(arr)//3])
```

```
  File "<tokenize>", line 13
    array[pos], array[pos] = array[pos], array[pos]
    ^
IndentationError: unindent does not match any outer indentation level
```

Sample 3:

Input: Write a python program that uses the Random Pivot Quick Sort to determine if any value appears in the array at least twice

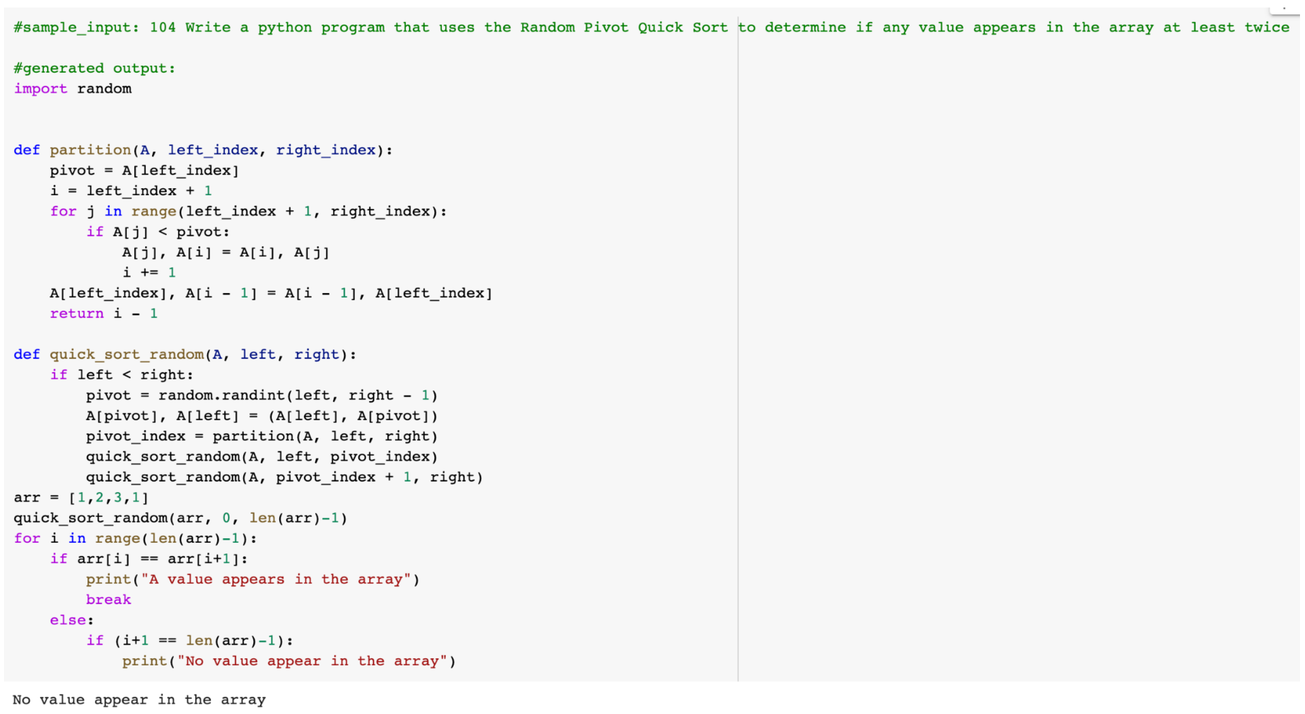
Baseline Result:



```
def quickSort(array):
    n = len(array)
    for i in range(n - 1):
        if array[i] > array[i+1]:
            array[i], array[i+1] = array[i+1], array[i+1]
            return array
arr = [2,3,4,5,6,7]
for i in range(len(arr)-1):
            if arr[i] > arr[i+1]:
                array[i] = arr[i+1], array[i]
print(arr[i] + arr[i+1] + arr[i+1] + arr[i+1] + arr[i+1] + arr[1] + arr[2] + arr[2] + arr[i+1]
print(arr[i])
```

```
  File "<ipython-input-30-c86a628f29e5>", line 12
    print(arr[i])
    ^
SyntaxError: invalid syntax
```

Fine-tuned Model Result - Fail Metric A, Pass Metric B:



```
#sample_input: 104 Write a python program that uses the Random Pivot Quick Sort to determine if any value appears in the array at least twice

#generated output:
import random


def partition(A, left_index, right_index):
    pivot = A[left_index]
    i = left_index + 1
    for j in range(left_index + 1, right_index):
        if A[j] < pivot:
            A[j], A[i] = A[i], A[j]
            i += 1
    A[left_index], A[i - 1] = A[i - 1], A[left_index]
    return i - 1

def quick_sort_random(A, left, right):
    if left < right:
        pivot = random.randint(left, right - 1)
        A[pivot], A[left] = (A[left], A[pivot])
        pivot_index = partition(A, left, right)
        quick_sort_random(A, left, pivot_index)
        quick_sort_random(A, pivot_index + 1, right)
arr = [1,2,3,1]
quick_sort_random(arr, 0, len(arr)-1)
for i in range(len(arr)-1):
    if arr[i] == arr[i+1]:
        print("A value appears in the array")
        break
    else:
        if (i+1 == len(arr)-1):
            print("No value appear in the array")
```

```
No value appear in the array
```

Observations:

The baseline results, as we can see, were not making any sense, they neither had correct syntax, nor the proper sorting algorithms. This is why we thought there was no need to compute a score for it, as mentioned in the Quantitative Results section.

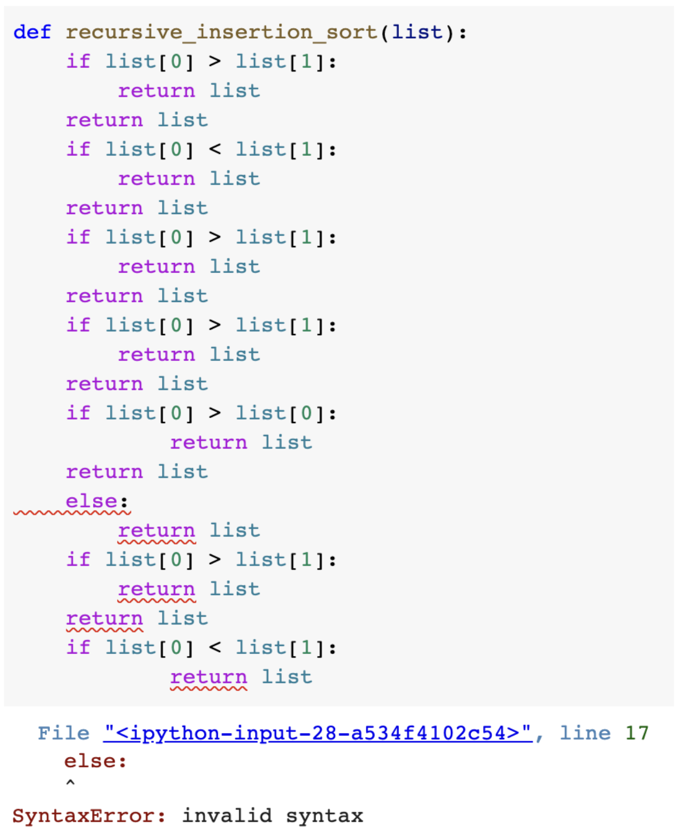
As shown in the above sample, we observed that the results of the fine-tuned model could be generally divided into three groups: both Pass (Sample 1), both Fail (Sample 2), and one Pass one Fail (Sample 3). Metric B mainly affected the third group, one Pass one Fail. Since the accuracy for Metric A is 46% and the same for Metric B is 68%, it means that around 22% of the results belong to the third group. This shows that Metric B indeed provided us with quite different results from Metric A.

We also observed that our model was more likely to generate the correct code for some well-known sorting algorithms but did not do a good job on some uncommon algorithms or the ones with complex structures. Clearly, this is caused by limited training data, the model might not be pre-trained with these algorithms before and we were not able to make the model well-learn these algorithms with our dataset. If we can increase the amount and diversity of our dataset, we expect better results than what we have now.

codet5-base Without Fine-tuning

To see if our fine-tuning step actually improved the results, we tested on the codet5-base model with the default parameters and the results with the same sample inputs as above are shown below:

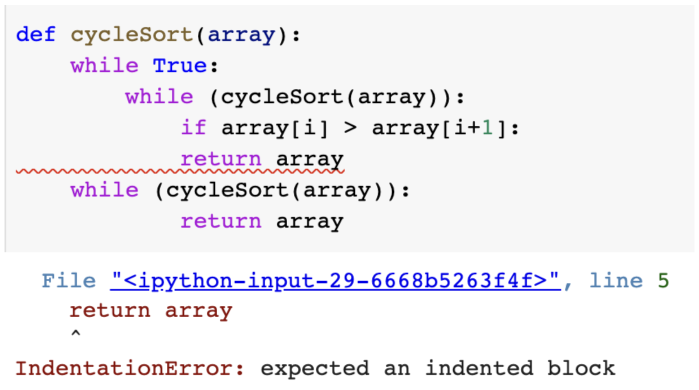
Sample 1:



```
def recursive_insertion_sort(list):
    if list[0] > list[1]:
        return list
    return list
    if list[0] < list[1]:
        return list
    return list
    if list[0] > list[1]:
        return list
    return list
    if list[0] > list[1]:
        return list
    return list
    if list[0] > list[0]:
            return list
    return list
    else:
        return list
    if list[0] > list[1]:
        return list
    return list
    if list[0] < list[1]:
            return list
```

```
  File "<ipython-input-28-a534f4102c54>", line 17
    else:
    ^
SyntaxError: invalid syntax
```

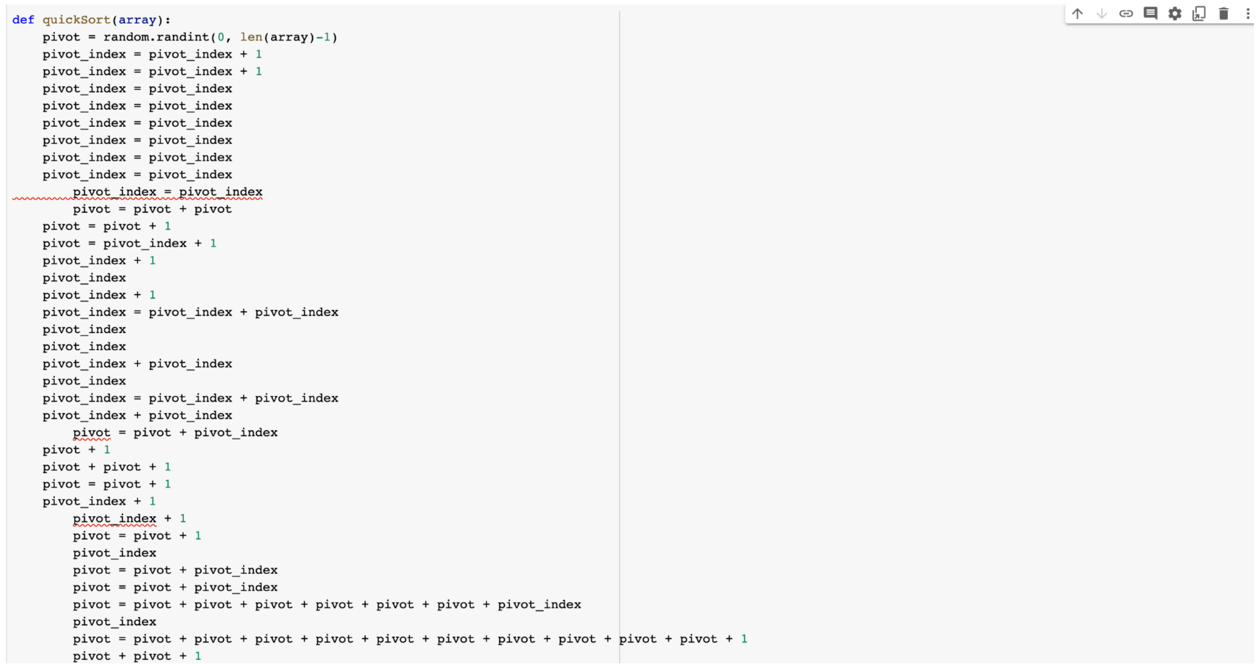
Sample 2:



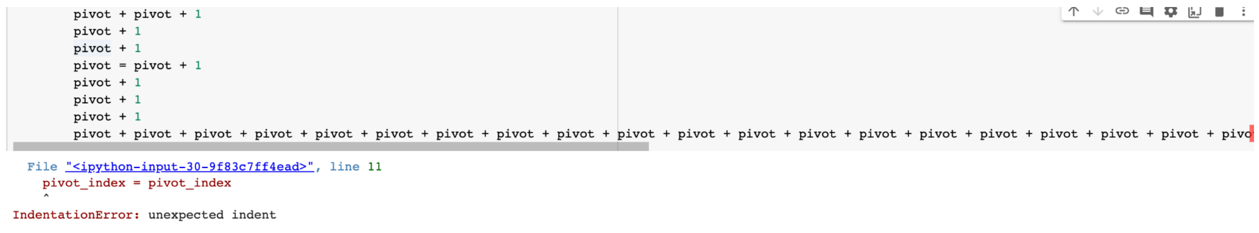
```
def cycleSort(array):
    while True:
        while (cycleSort(array)):
            if array[i] > array[i+1]:
            return array
    while (cycleSort(array)):
            return array
```

```
  File "<ipython-input-29-6668b5263f4f>", line 5
    return array
    ^
IndentationError: expected an indented block
```

Sample 3:



```
def quickSort(array):
    pivot = random.randint(0, len(array)-1)
    pivot_index = pivot_index + 1
    pivot_index = pivot_index + 1
    pivot_index = pivot_index
    pivot_index = pivot_index
    pivot_index = pivot_index
    pivot_index = pivot_index
    pivot_index = pivot_index
    pivot_index = pivot_index
        pivot_index = pivot_index
        pivot = pivot + pivot
    pivot = pivot + 1
    pivot = pivot_index + 1
    pivot_index + 1
    pivot_index
    pivot_index + 1
    pivot_index = pivot_index + pivot_index
    pivot_index
    pivot_index
    pivot_index + pivot_index
    pivot_index
    pivot_index = pivot_index + pivot_index
    pivot_index + pivot_index
        pivot = pivot + pivot_index
    pivot + 1
    pivot + pivot + 1
    pivot = pivot + 1
    pivot_index + 1
        pivot_index + 1
        pivot = pivot + 1
        pivot_index
        pivot = pivot + pivot_index
        pivot = pivot + pivot_index
        pivot = pivot + pivot + pivot + pivot + pivot + pivot + pivot_index
        pivot_index
        pivot = pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + 1
        pivot + pivot + 1
```



```
        pivot + pivot + 1
        pivot + 1
        pivot + 1
        pivot = pivot + 1
        pivot + 1
        pivot + 1
        pivot + 1
        pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivot + pivo
```

```
  File "<ipython-input-30-9f83c7ff4ead>", line 11
    pivot_index = pivot_index
    ^
IndentationError: unexpected indent
```

We can see that the results are similar to the baseline results. Since the codet5 model is not pre-trained on turning natural languages into code, it cannot be directly used to perform this task without fine-tuning. This evidence shows that our fine-tuning step indeed improve the results as expected.

OpenAI Codex

In addition, we compared our results to that of the openAI codex model, which is state-of-art in code generation. The results with the same sample inputs as above are shown below:

Sample 1:

```python
def recursiveInsertionSort(arr):
    if len(arr) <= 1:
        return arr
    else:
        key = arr[0]
        smallList = recursiveInsertionSort(arr[1:])
        bigList = []
        for i in smallList:
            if i < key:
                bigList.append(i)
            else:
                bigList.append(key)
                key = i
        bigList.append(key)
        return bigList

arr = [12, 11, 13, 5, 6]
print("Given array is", end="\n")
print(arr)
print("\nSorted array is: ", end="\n")
print(recursiveInsertionSort(arr))
```

```
Given array is
[12, 11, 13, 5, 6]

Sorted array is:
[5, 6, 11, 12, 13]
```

Sample 2:

```python
def cycleSort(array):
    # traverse array elements
    for cycle_start in range(0, len(array) - 1):
        item = array[cycle_start]
        pos = cycle_start
        for i in range(cycle_start + 1, len(array)):
            if array[i] < item:
                pos += 1

        if pos == cycle_start:
            continue

        while item == array[pos]:
            pos += 1

        if pos != cycle_start:
            array[pos], item = item, array[pos]

        while pos != cycle_start:
            pos = cycle_start
            for i in range(cycle_start + 1, len(array)):
                if array[i] < item:
                    pos += 1

            while item == array[pos]:
                pos += 1
            if item != array[pos]:
                array[pos], item = item, array[pos]

    return array

# Driver Code
arr = [2, 4, 5, 1, 3]
sorted_arr = cycleSort(arr)

# printing the third maximum element
print("Third Maximum element is", sorted_arr[2])
```

```
Third Maximum element is 3
```

Sample 3:

```
import random

def quick_sort(A, low, high):
    # Base Case
    if low >= high:
        return
    # Choose random pivot
    pivot = A[random.randint(low, high)]

    # Initializing left and right pointers
    i = low
    j = high

    # Sorting the array based on pivot
    while i <= j:
        while A[i] < pivot:
            i += 1
        while A[j] > pivot:
            j -= 1
        if i <= j:
            A[i], A[j] = A[j], A[i]
            i += 1
            j -= 1

    # Recursively call Quick Sort on the partitions
    quick_sort(A, low, j)
    quick_sort(A, i, high)

def check_duplicates(A):
    # Sorting the array
    quick_sort(A, 0, len(A) - 1)

    for i in range(len(A) - 1):
        if A[i] == A[i + 1]:
            return True
    return False
```

```
# Driver Code
A = [3, 4, 2, 5, 6, 1, 7, 8, 9]

# Prints true if the array contains a duplicate
if check_duplicates(A):
    print("Yes")
else:
    print("No")
```

```
No
```

As we can see, the codex model generated the almost perfect code of the algorithm and correctly solve the problem. Our model results are not able to reach the level of the codex model results, but we believe if we increase the amount and diversity of our dataset, we will further improve the performance of our model.

Discussion and Learnings:

Based on our results and the above observations, 46% accuracy for Metric A and 68% accuracy for Metric B might not reach the level of the openAI codex model, but compared to the small variant model and the same model without fine-tuning, we improved the results in a meaningful way.

When looking back on our project, we encountered several difficulties. First, we spent some time finding an appropriate model for code generation. Second, we put a lot of effort into hand-labeling our results since there are over a hundred test data with long outputs. Third, which was the most difficult part, data collection. We underestimated the complexity of our data and we cannot find the data that we want. This took us more time to create data by ourselves.

So, if we have a similar project, we will take this into consideration before we actually start the project. By this way, we can save some time on searching data and work on increasing the diversity of the dataset to achieve better results.

Individual Contributions:

Dhairya Parmar:

- Created 216 data samples, includes training, validation and testing samples
- Manually generated the training, validation and testing splits
- Wrote Datasets.ipynb for importing data in a compatible format
- Responsible for adapting the model script for CodeT5
- Incorporated 'Weights and Biases' for plotting learning curves
- Hand-Labeled 50 testing samples using metrics explained above

Yuchen Dai

- Created 100 data samples for training + 14 for testing
- Wrote the initial version of the t5 model
- Trained the baseline model
- Hand-Labeled 51 testing samples
- Compared the results between baseline model, main model, main model without fine-tuning, and codex model

References:

[1] **Wang, Yue, 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).**

[2] **Chen, Mark, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards et al. "Evaluating large language models trained on code."** ***arXiv preprint arXiv:2107.03374*** **(2021).**

[3] **OpenAI Codex API: https://openai.com/blog/openai-codex/**

Permissions

Dhairya Parmar

- permission to post video: wait till see video
- permission to post final report: yes
- permission to post source code: yes

Yuchen Dai

- permission to post video: wait till see video
- permission to post final report: yes
- permission to post source code: yes