Intelligent Code Editor

Final Report

Revised April 18, 2020

sdmay20-46
Professor Ali Jannesari — Client & Adviser

Keaton Johnson — Systems Lead
Jonathan Novak — Machine Learning Lead
Matthew Orth — Meeting Facilitator
Garet Phelps — Report Manager
Isaac Spanier — Test Lead
John Jago — Software Lead

Team Email: sdmay20-46@iastate.edu
Team Website: https://sdmay20-46.sd.ece.iastate.edu

Table of Contents

List of Figures

Figure	**Page Number**
Figure 1. Detailed Design Diagram	13

List of Tables

Definitions

- UI = User Interface
- GUI = Graphical User Interface
- NLP = Natural Language Processing
- NLTK = Natural Language Toolkit, a Python library for working with English

1. Introduction

1.1 Acknowledgement

We would like to thank Professor Ali Jannesari for his guidance on this project not only as the primary client but also as the faculty advisor. We would also like to thank PhD student Hung Phan for sharing his knowledge and for his involvement.

1.2 Project and Problem Statement

1.2.1 Problem Statement

With software becoming more prevalent in fields where it previously did not exist, more people must learn how to write programs to accomplish their work. One example is bioinformatics. At Iowa State, COM S 444: Bioinformatic Analysis is cross-listed with Biology and Genetics, among other majors. Students in these majors do not necessarily have to become top programmers to do their work, but their work involves programming nonetheless. A lack of experience in programming can become an unnecessary hindrance to getting their work done, especially when trying to figure out the syntax of a programming language.

1.2.2 Solution Approach

Our project provides the foundation of a system for users to type what they want to accomplish in natural language and convert that natural language to the software code. Our foundational system focuses only on translating natural language statements to Java method invocations. Future systems will expand this functionality to other programming constructs and languages. This would allow someone who doesn't know how to write a particular statement in a programming language to type an approximation of that statement in English and have the editor convert the English to executable code.

The primary deliverable is a plugin to an existing integrated development environment (IDE) or text editor, which the user will interact with to translate the natural language to code. Behind the scenes is some preprocessing and a classification/translation engine that handles the actual conversion of English to code using a trained model. Also produced is a dataset of possible natural language inputs that someone might type and the expected code translations using automatic code mining methods, which allows us to train our model and test the accuracy of our final product. Our solution is tailored towards users who have some programming knowledge and will support simple Java method invocations.

1.3 Operational Environment

The IDE plugin along with the other software components resides on computers and remote servers. One major risk is that malicious software might be able to read data from our services. While we are not storing personal information, is it possible that people might not want their English (or the resulting code) exposed, as this could leak enough information for an attack on the program that the user is writing or release of confidential information. Therefore, considerations to the security of our software, especially if it communicates over a network, were taken.

1.4 Requirements

1.4.1 Functional Requirements

- User can select or otherwise input the text they wish to translate to code
- User can trigger a translate action
- The textual descriptions are replaced by the translated code fragments
- The translated code fragments can be executed as regular code

1.4.2 Non-Functional Requirements

- Translation time should be fast so it does not slow down the user’s development pace
- User must be connected to the internet

1.4.3 UI Requirements

- Translation action should be easily accessible from the text editor area
- The user interface should be clean and easy to understand

1.5 Intended Users and Uses

Users and their uses of this product:

- Someone who has a basic knowledge about the Java programming language (variables, methods, classes, etc.), but does not know the exact method necessary to perform their specific operation

Future iterations of this project may allow users who do not have any programming knowledge to use this system.

1.6 Assumptions and Limitations

1.6.1 Assumptions

- The user will enter their natural language statements in the general format we expect
    - No punctuation, no commas between parameters, writing hardcoded string surrounded by double quotes (“”)
- The user will enter their natural language statement in the location in the editor they want the translated code inserted
- The user will enter only natural language statements supported by the system’s dataset (Java method invocations)
- The user has knowledge about the basics of Java programming such as methods, variables, classes, etc.

1.6.2 Limitations

- Translations will only be supported for common code syntax, method invocations, and classes.
- The translation may not generate the proper code fragment on the first attempt or at all for certain inputs
- Translation will not occur for natural language statements not represented in the dataset (non-Java method invocation)
- Translations are only supported from English to Java source code

- Only one translation will be supported at a time

1.7 Expected End Project and Deliverables

Note: These were the parts of the project that we were able to implement, which closely resembled our original goal and schedule for the project.

User Interface (November 2019):

The user interface (UI) will present the user with an Integrated Development Environment (IDE) text editor where the user can enter their natural language statements representing the translated code. The user interface will also allow the user to interact with the system to translate their natural language statements to equivalent code. Once the translate button is interacted with, the natural language statement will be preprocessed and passed to the classification/translation engine. The UI will then accept the translated code from the Translation Engine and display it in the editor at the location where the original natural language statement is.

Classification and Translation Engine (December 2019):

The classification engine will be trained from a dataset of natural language, expected code translation pairs. This model (along with the natural language statement from the UI) will be passed to the translation engine. The model will then use the input to produce an output of the expected code translation. Finally, the translation engine will pass the expected translation to the UI.

Dataset (March 2020):

The dataset will contain the natural language statement, expected code translation pairs. This dataset will contain a source (natural language statements) and target (expected code translation) files for training, validation, and testing. The Java code side of the dataset will be automatically generated by mining Java method invocation statements from GitHub using the GitHub API with the C# Octokit library. These mined Java method invocation samples will be labeled using a labeling method described later in the document. The dataset will be created locally and will be stored on the translation server.

Model and Dataset Optimization (March-April 2020):

Throughout the development process, the training model (architecture and hyperparameters) and dataset will be optimized to achieve the best performance. Preprocessing of the entered natural language statement will be done to convert the input natural language statement into verb-noun format (only selecting the verbs and nouns of the input sentence) and converting the parameters into their Java type. The performance will be measured through neural machine translation metrics, research, and observed translation results.

Final Product (April 2020):

The final Intelligent Code Editor project will connect the User Interface, Dataset, and Classification/Translation Engine using a dedicated server. This will create a complete end-to-end translation system supporting Java method invocations.

2. Specifications and Analysis

2.1 Proposed Design

User Interface (UI)

An IntelliJ plugin was created for the user interface. This plugin will present the user with an IDE text/code editor where they can enter their natural language statements. It will also give the user the ability to translate the natural language statement to code through some interaction button. When the translation functionality is interacted with, all method parameters will be converted into their type and NLTK preprocessing (see next section) will be done. The UI will then connect to the classification and translation engine while passing the preprocessed natural language statement and receiving and displaying the expected code translation. Finally, the translated code will be executable by the user.

NLTK Preprocessing

Preprocessing of the input natural language statement is executed to remove all non-verb and non-noun words from the sentence, convert all words to present tense, and convert all characters to lowercase. This allows the translation engine to focus on the most relevant parts of the sentence (verbs being the method name and nouns being the parameter types).

Classification and Translation Engine

The Classification and Translation engine were implemented using OpenNMT-py. The classification engine OpenNMT-py model was trained using a dataset containing natural language statement, expected code translation pairs. The preprocessed natural language statement from the UI is then given to the translation engine that will output the expected translated code to the UI.

Dataset

The dataset contains the natural language statement, expected code translation pairs. The dataset contains a source (natural language statements) and target (expected code translation) files for training, validation, and testing. The Java code side of the dataset was automatically generated by mining Java method invocation statements from GitHub. These mined Java method statements were preprocessed to separate the words into separate tokens and converting the method parameters and variables to their Java types. The mined Java method invocation samples are then labeled using a labeling method described later in the document. The dataset will be created locally and is stored on the translation server.

2.2 Design Analysis

Strengths

For the user interface, documentation for creating an IntelliJ plugin is abundant and there is a large community for it. An IntelliJ plugin is also easy to install and use in the IntelliJ editor, making it easier and more likely for a user to utilize. NLTK was chosen for preprocessing as it is a well-known and high performing Python language processing library. NLTK also supported part of speech (PoS) tagging that is required for our project.

OpenNMT-py is a well-documented and widely used Neural Machine Translation engine that will convert the natural language statement to the expected code. One major strength of OpenNMT-py is that it comes with an easy interface for model architecture configuration, resembling modern techniques for neural machine translation. This simplifies the process of creating a model from scratch as that is something that requires a significant amount of experience and original research.

Weaknesses

When creating an IntelliJ plugin, it does not allow the user to modify everything about the code editor, so we had to ensure that we can modify what is necessary for our requirements. Furthermore, the built-in part of speech tagger included with NLTK does not perform particularly well for programming domain sentences, so modifications were made to support programming domain words such as creating a custom part of speech tagger.

One weakness of using OpenNMT-py is the models take a long time to train. However, this would be a problem regardless of the system we chose to implement it with. Careful consideration for how to create a representative dataset that allows us to effectively train the model was also given. Additionally, OpenNMT-py is more effective when translating from a certain type of statement to another, so preprocessing using NLTK was done to further improve this functionality.

A further detailed list of challenges associated with each part of the system is shown below:

- Text Editor:
    - Required Input Format: Since English natural language is ambiguous, we have required the user to enter some of the natural language statement in a certain format
        - Specify only the outermost method in a nested method call or the last method in a chained method call in natural language and format all other method invocations with their original variable names. Also, do not include parameter spaces, punctuation, and surround hardcoded Strings with double quotes.
- User Interface:
    - Java Parameter Type Automation: We need to create a way to automatically label the Java parameters with their types.
        - This will be done using a Java script that will take the original Java file the mined method was contained in and the mined statement. The script will output the statement with their parameters and variables converted to their types.
    - Custom Java Type Translation: Custom Java data types will be handled the same way build-in Java types are.
- NLTK Script:
    - Correct PoS Tagging of Method Information and Parameters to Verb and Nouns: We need to ensure that all Java types and other useful information are correctly classified as Nouns or Verbs to avoid them being removed.
        - This will be done by adding all Java types into the custom PoS tagger.
- OpenNMT-py:
    - Uniform Dataset Representation: Since we will have six people labeling the dataset, there needs to be a defined, uniform method to label the dataset.
        - This technique is outlined later in the Design Plan section (2.4).

2.3 Engineering Standards and Design Practices

Our team utilized the IEEE standards IEEE 1028-2008, IEEE 16326-2009, and IEEE 1008-1987 related to project management and testing. Our team followed the Agile

Development Process with Test Driven Development. Our team had weekly meetings where we reviewed our project progress, updated our task board, and planned future work. Bi-weekly meetings were held where demos of our progress were shown and feedback was received to further improve our design. These meetings ensured there was clear communication between our group and our client. Testing our design throughout the design process was also incorporated to validate our design. This means that our requirements were continuously changing and improved throughout the development process.

2.4 Design Plan

The below diagram shows our design plan:

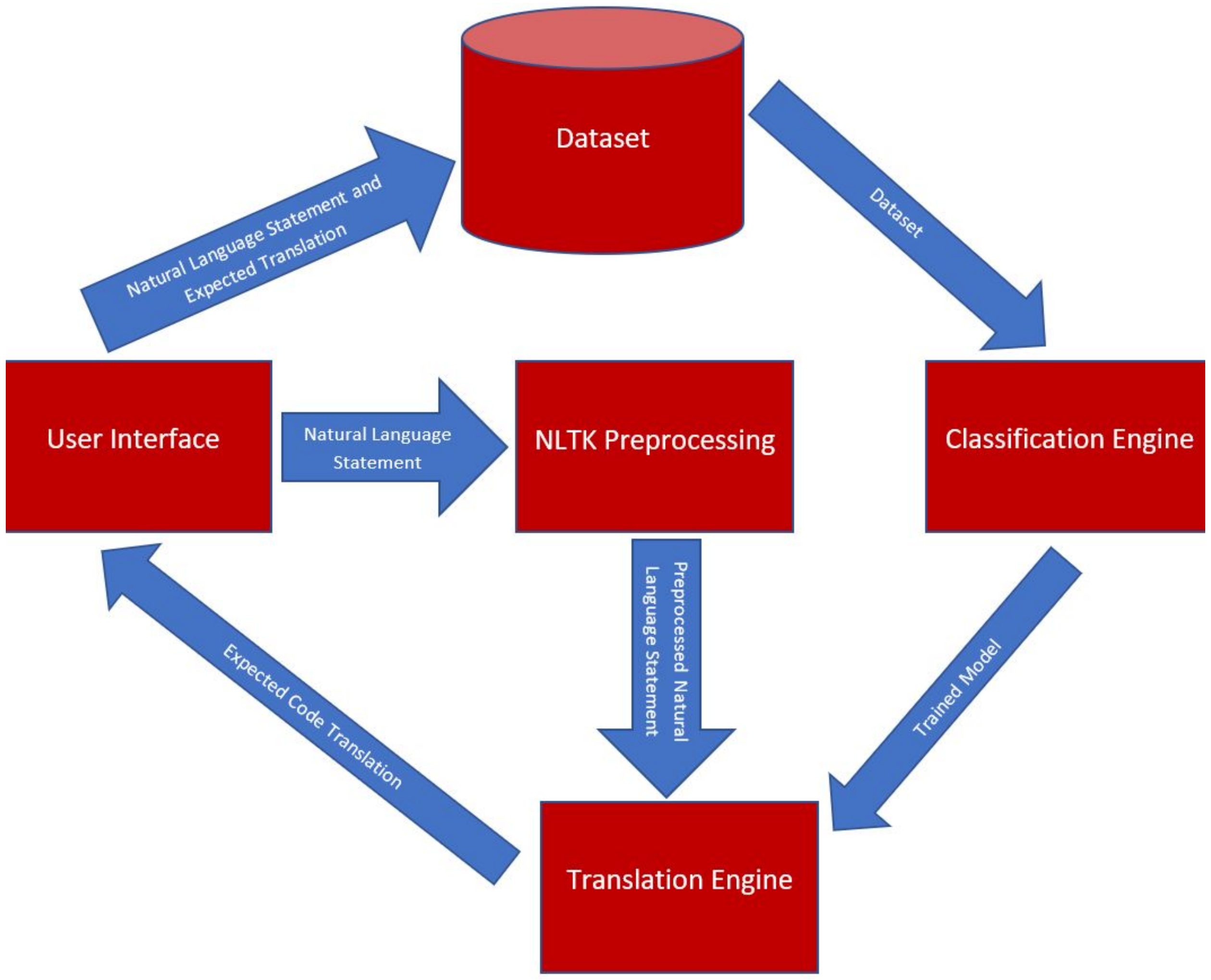

Dataset
Natural Language Statement and Expected Translation
Dataset
User Interface
Natural Language Statement
NLTK Preprocessing
Classification Engine
Preprocessed Natural Language Statement
Expected Code Translation
Trained Model
Translation Engine

Figure 1. Detailed Design Diagram

User Interface

The user will first enter their natural language statement into the code editor. Then after the user clicks the translate button, the User Interface will convert the method parameters to their Java types, storing the mapping of parameters to their types in a table, and then will pass the natural language statement to NLTK for preprocessing.

After the Translation Engine runs the preprocessed statement through, it will return the translated Java code to the User Interface. The User Interface will then convert the Java type parameters to their original parameter names and display the code translation in the text editor. The user interface implements many of the functional requirements.

Dataset

The Dataset (located on the classification engine) stores the natural language statement (in labeled format) and expected code translation in the respective source (natural language) and target (expected code translation) files that are split into training, validation, and test sets. These dataset files will then be passed to the Classification Engine for training.

Below is the detailed dataset labeling method we used for labeling the mined Java method invocations:

*Note: Steps in **Bold** have been automated*

Simplified Natural Language Labeling Steps:

1. Determine sentence label representation of Java code (start from the Java Documentation description of the method)
    a. Modify Java Documentation description slightly to make it sound more like what a person would say while keeping the same general original format
2. Convert Parameters to their type

3. **Run sentence through Script:** https://git.linux.iastate.edu/hungphd/sdmay19-intelligent-code-editor-gitlab/blob/pos-verb-noun-integration/nltk-scripts/Sentence_Preprocessing.py
    a. **Put all sentences from steps 1 and 2 into the Statement.txt file (the results will be contained in the Statement_processed.txt file**
        i. **This step will convert the sentence to Verb-Noun format, convert all characters to lowercase, and convert all words to present tense**
4. *(complete steps 1-3 for ALL other statements first)* Incorporate synonyms and different statement structures (multiple dataset entries for each mined Java code entry)
    a. Using WordNet synsets, thesaurus, and different word orderings while keeping the original meaning (likely done manually)

Java Code (all automated):

1. **Put correct spacing format between method names, parenthesis, and parameters**
    a. **Automated using Script:** https://git.linux.iastate.edu/hungphd/sdmay19-intelligent-code-editor-gitlab/blob/pos-verb-noun-integration/nltk-scripts/Java_Preprocessing.py
        i. **Put the mined Java statements into the Code.txt text file (the results will be contained in the Code_preprocessed.txt file)**
2. **Convert parameter names to their type**

Reminders:

Dataset Labeling:

- Format natural language descriptions using Java Documentation descriptions
    - Start with only this statement and then we can add more generalized statements later
- Do not include commas between method parameters in natural language input

Java Dataset Labeling:

- Convert all chained method calls to their type except for the last method call

Following this dataset labeling process helped to ensure that uniform dataset labeling occurred between all six members and will result in the best OpenNMT-py training and translation results as the labels generalize to other formats.

Classification Engine

The Classification Engine trained the OpenNMT-py model on the provided dataset, using the natural language statement as the input and the expected code translation as the output. After training has been completed, the model will calculate translation metrics (such as BLEU score) on the testing dataset to determine the effectiveness of the results. The Classification Engine will then pass the trained model to the translation engine to use for translation.

NLTK Preprocessing

The natural language statement that the user wants to translate is first passed through NLTK preprocessing. During this process, the sentence will have all non-verb and non-noun words removed, convert all words to present tense, and convert all characters to lowercase. This preprocessing will convert the input sentence into a format that the Translation Engine will be able to more effectively run through the model.

Translation Engine

The Translation Engine takes the trained model from the Classification Engine and the input from the User Interface to generate the expected code translation. This expected code translation will be passed to the user interface to display. The dataset, classification, and translation engine satisfy many of the non-functional requirements.

Architecture Diagram:

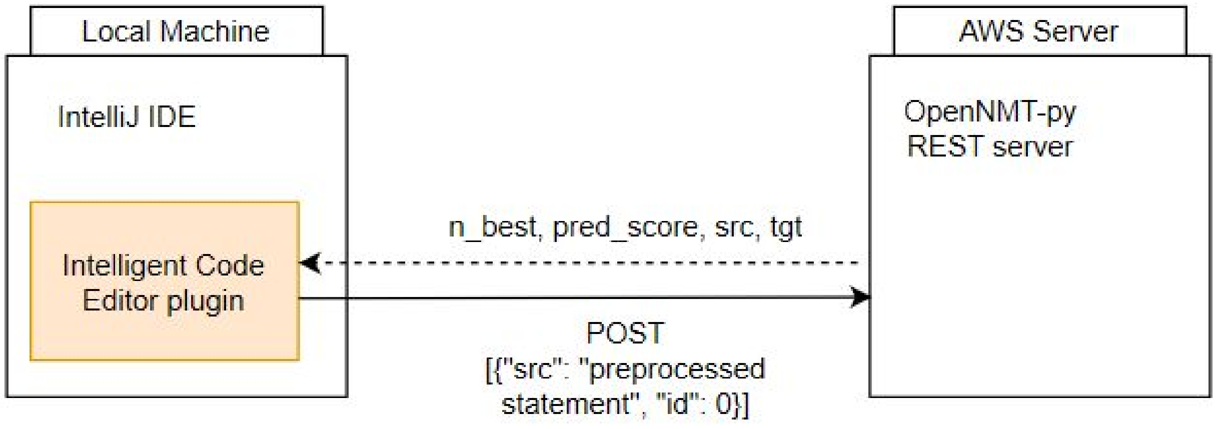

Local Machine
IntelliJ IDE
Intelligent Code Editor plugin
AWS Server
OpenNMT-py REST server
n_best, pred_score, src, tgt
POST [{"src": "preprocessed statement", "id": 0}]

Figure 2. User Interface and Translation Server Architecture Diagram

The above architecture diagram shows in more detail the communication between the UI and the classification/translation server.

User Flow Diagram:

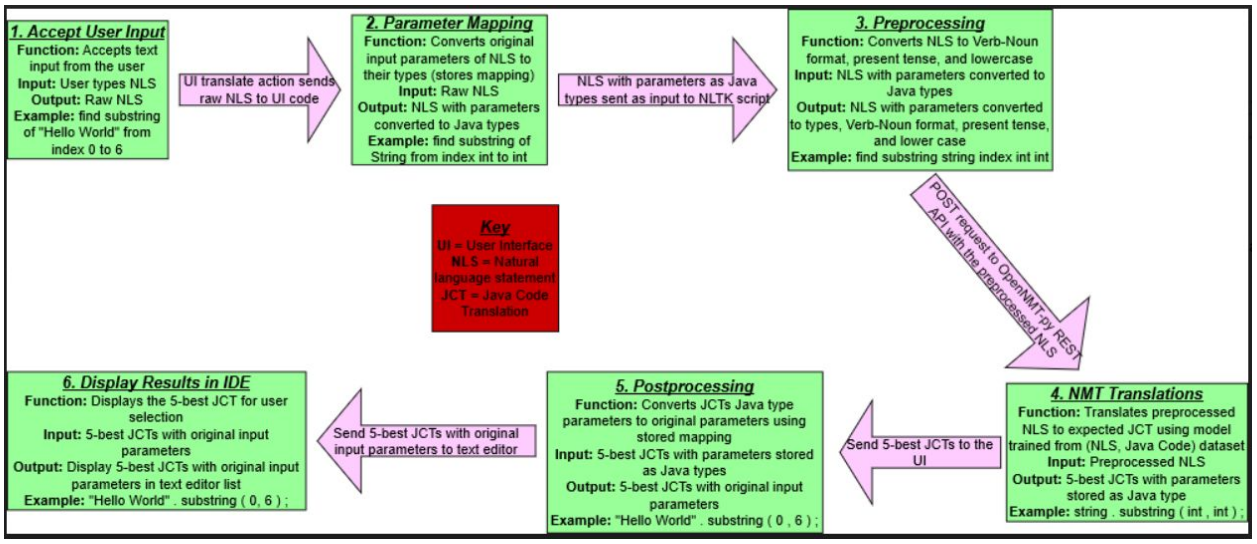

1. Accept User Input
Function: Accepts text input from the user
Input: User types NLS
Output: Raw NLS
Example: find substring of "Hello World" from index 0 to 6
UI translate action sends raw NLS to UI code
2. Parameter Mapping
Function: Converts original input parameters of NLS to their types (stores mapping)
Input: Raw NLS
Output: NLS with parameters converted to Java types
Example: find substring of String from index int to int
NLS with parameters as Java types sent as input to NLTK script
3. Preprocessing
Function: Converts NLS to Verb-Noun format, present tense, and lowercase
Input: NLS with parameters converted to Java types
Output: NLS with parameters converted to types, Verb-Noun format, present tense, and lower case
Example: find substring string index int int
POST request to OpenNMT-py REST API with the preprocessed NLS
Key
UI = User Interface
NLS = Natural language statement
JCT = Java Code Translation
4. NMT Translations
Function: Translates preprocessed NLS to expected JCT using model trained from (NLS, Java Code) dataset
Input: Preprocessed NLS
Output: 5-best JCTs with parameters stored as Java type
Example: string . substring (int , int) ;
Send 5-best JCTs to the UI
5. Postprocessing
Function: Converts JCTs Java type parameters to original parameters using stored mapping
Input: 5-best JCTs with parameters stored as Java types
Output: 5-best JCTs with original input parameters
Example: "Hello World" . substring (0 , 6) ;
Send 5-best JCTs with original input parameters to text editor
6. Display Results in IDE
Function: Displays the 5-best JCT for user selection
Input: 5-best JCTs with original input parameters
Output: Display 5-best JCTs with original input parameters in text editor list
Example: "Hello World" . substring (0, 6) ;

Figure 3. User Flow Diagram

The above user flow diagram shows more detailed information about the input and output for each component when the user triggers a translation operation with a running example for reference.

3. Statement of Work

3.1 Previous Work and Literature

User Interface

Our research for the user interface started with the IntelliJ plugin API documentation [1]. Here we found useful information about what functionality was available for creating a custom IntelliJ plugin. We also utilized various documentation resources for creating an Eclipse Plugin [2] and Visual Studio Code Extension [3]; however, we did not find as many documentation resources and a large community for these plugins. This led us to choose an IntelliJ plugin.

Natural Language Preprocessing

The AnyCode project is also a system that converts natural language statements into Java code statements [7]. Through reading this report, we determined that the best preprocessing would be to have the resulting natural language statement only contain the verbs and nouns from the original sentence. This is because AnyCode found that the verbs commonly represent the action or method name and the nouns commonly represent the method parameters [7]. Using this method allows our system to remove all the less important words for the original sentence, allowing the OpenNMT-py system to focus on the more important words in the sentence.

Classification/Translation Engine

We read many different research papers to determine methods to implement neural machine translation. From these papers, we determined that modern Neural Machine Translation mechanisms like the Transformer model are some of the best performing models [4]. With this information, we researched tools that can execute this kind of neural machine translation. We then found OpenNMT-py [5], which has easy interaction with modern neural machine translation models as we were able to train a basic model for language translation within a couple of days.

Dataset

The dataset will be modeled in the way OpenNMT-py expects: a source (natural language statements) and target (expected code translation) files for training, validation, and testing.

While there are current systems that rely solely on Deep Learning or solely on preprocessing to achieve the translation results, our system combines both preprocessing with Deep Learning to achieve a result that takes the best part from each system. Additionally, our system aims to allow the user to enter any natural language statement and preprocess it into a format that our system can use effectively while other systems require the user to enter their natural language in a certain format or have poor translation results.

3.2 Technology Considerations

Strengths

For the user interface, documentation for creating an IntelliJ plugin is abundant and there is a large community for it. An IntelliJ plugin is also easy to install and use in the IntelliJ editor, making it something a user would more likely use. NLTK was chosen for preprocessing as it is a well-known and high performing language processing Python library.

OpenNMT-py is a well-documented and widely used Neural Machine Translation engine that will convert our natural language statement to the expected code. One major strength of OpenNMT-py is that it comes with training model templates resembling modern techniques for neural machine translation while also giving more customization options. This means we could use a model like the Transformer model without having to configure this ourselves from scratch. This simplified the process of creating a model from scratch as that is something that requires a significant more amount of experience and original research with Machine Learning and Natural Language Processing.

Weaknesses

When creating an IntelliJ plugin, it does not allow the user to modify everything about the code editor, so we had to ensure that we could modify what is necessary. This was solved by either modifying our features or determining other mechanisms aside from IntelliJ plugins to implement those features. Furthermore, the built in part of speech tagger included with NLTK does not perform well for programming domain sentences, so modifications were made to support programming domain words by creating a custom part of speech tagger.

Some weaknesses of using OpenNMT-py are the models take a long time to train. However, this would be a problem regardless of the system we chose to implement it with. This issue was resolved by requesting more powerful computers or servers to execute the training and classification of the model. Creating a representative dataset to effectively train the model was also a challenge. This took a long time to produce, but automating some of the dataset labeling steps helped to shorten this time. Additionally, OpenNMT-py can only accurately translate more specific kinds of statements. This is why we implemented the NLTK preprocessing before running the natural language statement through the translation engine.

3.3 Task Decomposition

- Research
    - Research and test tools for creating IDE plugin/extension
    - Research tools for natural language to code translation
    - Research tools for creating a dataset
- User Interface (IntelliJ plugin)
    - Modify the IntelliJ IDE to allow the user to enter both natural language statements and code
    - Create a mechanism that allows the user to translate the natural language statement to code
    - Consider the code context to translate variables, methods, class names, and method parameters to generic names or Java types when passing into OpenNMT-py

    - Connect the UI to the Dataset, Classification, and Translation Engine
- NLTK Preprocessing
    - Implement the verb-noun, present tense, and lowercase natural language statement preprocessing
    - Create a part of speech tagger that will accurately tag programming-domain sentences
    - Configure an AWS server to host these processes and OpenNMT-py
- Dataset
    - Research word synonyms, word permutations, and English dialects
    - Research current datasets that could be used to train the classification model
    - Automatically mine Java method invocations from sources like GitHub
    - Label the mined Java method invocation dataset using labeling method
    - Store the natural language statement, expected code translation pairs for the source and target training, validation, and testing dataset
    - Optimize the dataset for best performance
    - Pass the training dataset to the Classification Engine
- Classification Engine
    - Use a dataset to train the natural language statement to expected code translation model
    - Optimize the model architecture and hyperparameters for best performance
    - Pass the trained model to the Translation Engine
- Translation Engine
    - Take as input the trained model from the Classification Engine and the natural language input from the User Interface
    - Run the preprocessed input natural language statement through for translation
    - Pass the expected code translation to the UI
- Testing
    - Write unit, GUI, and integration tests
    - Run usability tests
    - Make improvements based on testing results

3.4 Possible Risks and Risk Management

A potential risk is a lack of knowledge in natural language processing. This starts with creating a large and representative enough dataset for training the classification model. Usually, datasets require thousands of entries, which could take a long time to create. For this, we mined the Java method invocation statements from online sources like GitHub. Additionally, preprocessing of the natural language statements was done to ensure that the model can more accurately translate simpler types of statements.

Since a general system that can translate any natural language statement was very ambitious, restrictions were made to only support translation to Java method invocation statements (outlined in 2.4 Design Analysis section). Additionally, a limitation of use case may require users with some programming knowledge, so they can format their answers in a way that would make them easier to translate.

Once we have our dataset, training the classification model is a lengthy process, requiring powerful computing. We will need a dedicated computer or server for this purpose. Finally, once the model is trained, the accuracy of the translation may be an issue since even modern neural machine translation systems are unlikely to correctly translate the natural language to code accurately [6]. To compensate, we give the user the five most confident translation options to compensate.

3.5 Project Proposed Milestones and Evaluation Criteria

Note: These were the milestones set by us earlier in the project and were achieved.

The first milestone for the project will be researching and determining which user interface (UI), dataset, and classification/translation engine tools we want to use and create. This will uncover the initial direction of the project.

After the UI, dataset, and classification/translation tools are selected, the next milestone will be creating simple demos of using the UI, Dataset, and Classification/Translation engine for our purpose in isolation. This will involve giving fake data to these systems to ensure they behave correctly in isolation. The UI should allow the user to enter natural language statements and be able to convert them to code using a fake back-end system. The classification/translation engine should take a pre-made dataset containing natural language statements and equivalent code to train and test the model. The model translations should be better than 36% accuracy.

After everything is working in isolation, the next milestone will be to connect everything to ensure the system works end-to-end. This means the user will provide the natural language statement to the editor and the translation engine will provide the UI with the expected translated code to display.

Major milestones after this will involve improving the dataset and classification/translation engine to translate more complicated natural language statements (methods, classes, algorithms, etc.). Improvements will be made to the dataset to include automatically mined natural language, code pairs from online GitHub repositories that are labeled using our labeling method. Preprocessing of the natural language statement will also be done before passing the statement to the translation engine to allow the translation engine to only focus on simpler sentence constructs. Once completed, the classification/translation engine will meet our accuracy and functional requirements.

The final milestone for the project will involve testing and verification of the design. This will involve writing tests and running usability tests to ensure our design works as expected. Successful completion will have all required functionality tested, verified, and accepted.

3.6 Project Tracking Procedures

Our group utilized GitLab, Trello, and GroupMe to track our progress throughout the semester. GitLab was used to manage our project code. Each team member developed in their individual branch and merged that into the master branch when ready.

Trello was used to manage task creation and assignment. Each task has a title, description, assigned member, and due date. There is a backlog, doing, done, and completed column. These represent tasks yet to be assigned, currently being worked on, done, and verified respectively. It is expected each member accomplishes their tasks assigned for each sprint.

GroupMe was utilized for immediate communication with the team. Here, we communicated meeting times, quick questions, and other communication.

3.7 Expected Results and Validation

Note: This was the expected results of our system, which we were mostly able to achieve.

Our desired outcome is to create an end-to-end system where the user can enter a natural language statement into the code editor and the system will be able to translate the statement to equivalent code that will be displayed and be executable by the user in the editor.

Our implementation will be validated by creating unit and GUI tests and through usability tests. This will ensure that our system behaves as expected and is easily usable by the users. These testing and usability tests will be developed throughout the development process to ensure we are creating an optimal solution.

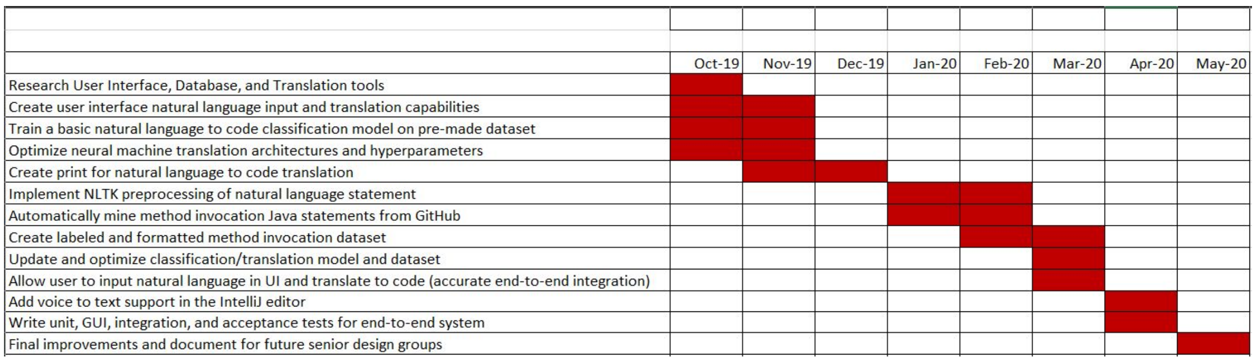
In addition, our training model and dataset’s performance will be validated using neural machine translation performance measures, research, observation of translation, and comparisons against similar systems on the same scale.

4. Project Timeline, Estimated Resources, and Challenges

4.1 Project Timeline

Table I

Intelligent Code Editor Project Timeline



	Oct-19	Nov-19	Dec-19	Jan-20	Feb-20	Mar-20	Apr-20	May-20
Research User Interface, Database, and Translation tools	🟥							
Create user interface natural language input and translation capabilities	🟥	🟥						
Train a basic natural language to code classification model on pre-made dataset	🟥	🟥						
Optimize neural machine translation architectures and hyperparameters	🟥	🟥						
Create print for natural language to code translation		🟥	🟥					
Implement NLTK preprocessing of natural language statement				🟥	🟥			
Automatically mine method invocation Java statements from GitHub				🟥	🟥			
Create labeled and formatted method invocation dataset					🟥	🟥		
Update and optimize classification/translation model and dataset						🟥		
Allow user to input natural language in UI and translate to code (accurate end-to-end integration)						🟥		
Add voice to text support in the IntelliJ editor							🟥	
Write unit, GUI, integration, and acceptance tests for end-to-end system							🟥	
Final improvements and document for future senior design groups								🟥

The first major stage of the project was to conduct the initial research of the UI, dataset, and classification/translation engine tools we want to use, which was completed in late October. This research set the foundation for the remainder of the project.

The second major stage involved setting up and creating the basic functionality for the UI, dataset, and classification/translation engine. For the UI, the natural language input and translation interaction were configured. The classification/translation engine was configured and trained on a basic print dataset, and set up as a server. Finally, we created a targeted Java print dataset. This work was completed in late November, and it was used to confirm our design for the remainder of the project.

Connectivity between the UI, dataset, and classification/translation engine was the next major task. This task involved connecting the UI to the dataset and classification engine, the dataset to the classification/translation engine, and the classification/translation engine to the UI. We then ensured that we can correctly send information between these systems. This work was completed in late-March / early-April once we got access to a dedicated AWS server.

The final stage involved optimizing, testing, and validating our results. This started with fine-tuning the dataset and classification/translation engine for the best results. Automatic dataset generation from GitHub and NLTK preprocessing was done in this phase. These steps created a more accurate translation system. We also continued writing the automated tests, end-to-end tests, and user validation testing. Finally, feedback from those tests were used to improve our final design and document our results for future teams that may work on this project.

4.2 Feasibility Assessment

The end-to-end natural language to code translation system explained above was very ambitious. Current systems struggle obtaining high accuracy for this purpose. Because of this, a more reasonable solution to this project was made to have the user enter more structured statements that the classification model can more easily learn from. Input preprocessing before passing the natural language input into the translation engine helped to improve our accuracy.

Since a general system that can translate any natural language statement is very ambitious, restrictions were made to only support Java method invocation translations. Additionally, a limitation of use cases may require users with some programming knowledge, so they can format their answers in a way that would make them easier to translate.

Another challenge was creating a large and representative enough dataset that will allow the trained model to be more accurate. Automation scripts were created to help automate parts of the manual dataset labeling process.

4.3 Personnel Effort Requirements

Table II

Intelligent Code Editor Personal Effort Requirements

Research	**30 hours**
Research IDE Plugin/Extension	10 hours
Research Natural Language to Code Translation	10 hours
Research Creating Dataset	10 hours
User Interface (IntelliJ Plugin)	**40 hours**
Enter Natural Language Statements and Code Functionality	10 hours
Create Mechanism to Translate Natural Language to Code	10 hours
Consider Code Context Before Passing Natural Language to OpenNMT-py	15 hours
Connect the UI to the Dataset, Classification, and Translation Engine	5 hours
NLTK Preprocessing	**45 hours**
Implement the verb-noun natural language statement preprocessing	15 hours
Create a part of speech tagger that will accurately tag programming-domain sentences	15 hours
Create and setup AWS Server to run these preprocessing scripts and OpenNMT-py	25 hours
Dataset	**110 hours**

Research Word Synonyms, Word Permutations, and Different English Dialects	5 hours
Research current datasets that could be used for training	10 hours
Automatically mine natural language, code pairs from GitHub	25 hours
Label the mined Java method invocation dataset using labeling method	50 hours
Store dataset in source and target training, validation, and testing datasets	5 hours
Optimize dataset for best performance	10 hours
Pass the Training Dataset to the Classification Engine	5 hours
Classification Engine	**35 hours**
Use Dataset to Train The Natural Language Statement to Expected Code Translation Model	10 hours
Optimize Model Architecture and Hyperparameters for Best Performance	20 hours
Pass the Training Model to the Translation Engine	5 hours
Translation Engine	**15 hours**
Receive Input and Trained Model from User Interface and Classification Engine	5 hours
Pass the Expected Code Translation to the UI	10 hours
Testing	**60 hours**
Write Unit, GUI, and Integration Tests	25 hours
Run usability Tests	10 hours
Make Improvements Based on Testing Results	25 hours

4.4 Other Resource Requirements

A dedicated GPU server was required and provided to train our classification model on the dataset. Our group has been given access to the Pronto GPU server at Iowa State University [8]. This server is a shared computing resource that allows access to powerful computing resources, including multiple GPUs. This sped up our training and development process as well as allowed us to train models with more complex architectures.

Additionally, a dedicated AWS server was provided to run our classification/translation engine's REST server. This server connects with the front-end to allow data to be sent from the user interface to the classification/translation server.

4.5 Financial Requirements

This project does not provide or require any financial resources. All the required resources will be provided to us as needed.

5. Testing and Implementation

5.1 Interface Specifications

This project does not contain any interfacing between hardware and software components, as it consists entirely of three software parts: the IntelliJ plugin (client), OpenNMT-py (the translation engine into which we feed our model), and the OpenNMT-py REST server (hosted on an AWS server). However, there are interfaces between the user and the plugin as well as the plugin and the OpenNMT-py system, which are described below.

The chosen IDE for this project is an IntelliJ plugin. This interface is a good way to present our plugin to a user that is wanting to write code without using an actual programming language such as Java. IntelliJ's user interface is easy to become familiar with but has many advanced options for more adept users. An IntelliJ plugin is needed to integrate our software with the IDE. NLTK is used to preprocess the original natural language statement to remove unnecessary words from translation.

To convert the natural language to code within the IntelliJ editor, the user selects the natural language they typed and triggers our plugin. This process interfaces with the end-to-end system that consists of the dataset and classification/translation REST engine, which is located on a dedicated server.

5.2 Hardware and Software

For hardware, the Pronto shared computing resource was used to train and test the classification model, and we used our personal computers for the rest of the development, which includes creating the user interface. In order to provide our translation engine as a service, we hosted it on an AWS EC2 server.

In terms of software, we used the IntelliJ Platform SDK to develop the IntelliJ plugin that was used for the user interface. OpenNMT-py was used to implement the classification/translation engine and server. NLTK was used to preprocess the natural language statements before passing them to OpenNMT-py. GitLab was used for source control, collaboration, and maintaining the project history. Trello was used for project management and distributing the work among team members.

5.3 Functional Testing

We wrote automated unit, GUI, and integration tests for the user interface component of the project. This ensured that the system both meets our specified requirements and behaves correctly as an end-to-end system. These tests were written throughout the development process to catch bugs early. Additionally, where appropriate, the test-driven development methodology was used in front-end development.

Our client and graduate student performed acceptance testing on the plugin at weekly meetings to evaluate the direction our software is going, specifically the user interface aspect, to confirm that the product meets requirements, and to make recommendations for future improvements.

Following standard Java practices, the Intellij plugin has tests located in src/test/java. These tests cover functionality within the plugin itself, such as whether the translate action calls an external API and whether text preprocessing is performed correctly.

5.4 Non-Functional Testing

As mentioned in the feasibility section of this design document, the success rate of similar projects is not high. Our application aims to improve on the AnyCode accuracy benchmark of around 36% [7]. Most of the current systems, which support more general statement translations, obtain around a 25%-35% success rate.

In terms of other non-functional requirements for the application, the first requirement we focused on is the appearance of the plugin application. The plugin should be easy to use, and easy to access and should have a user interface that is fairly simple and fast. This user interface was essentially designed for a user who is adept with a computer but shouldn’t overwhelm them with information.

Another area of non-functional testing that is considered is security. Given that we will have a public REST server running to serve clients the results of our translated model, we ensured that best practices in security are followed, such as blocking unused ports from the outside internet and delivering data over secure TLS connections.

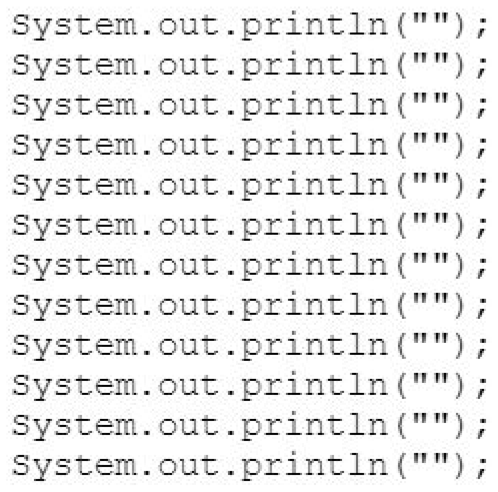
5.5 Process

Unit, GUI, and integration tests were written throughout the development process to test the functionality of the user interface and the end-to-end system. This ensured that we caught bugs or missed requirements early. During the development of our dataset and classification/translation engine, we tested the effectiveness of our dataset and classification/translation engine architecture and hyperparameters. This was done by running our dataset through our designed neural machine translation architecture and observing the results according to research comparison and neural machine translation metrics such a BLEU and calculating the translation accuracy. Modifications to our system were made when unexpected results were observed. This cycle continued until we achieved all requirements and achieved accurate translation results.

5.6 Results

Dataset Creation

First, we generated a simple Java print statement dataset that contained different ways to say print with some different values to print. The translated results can be seen below:

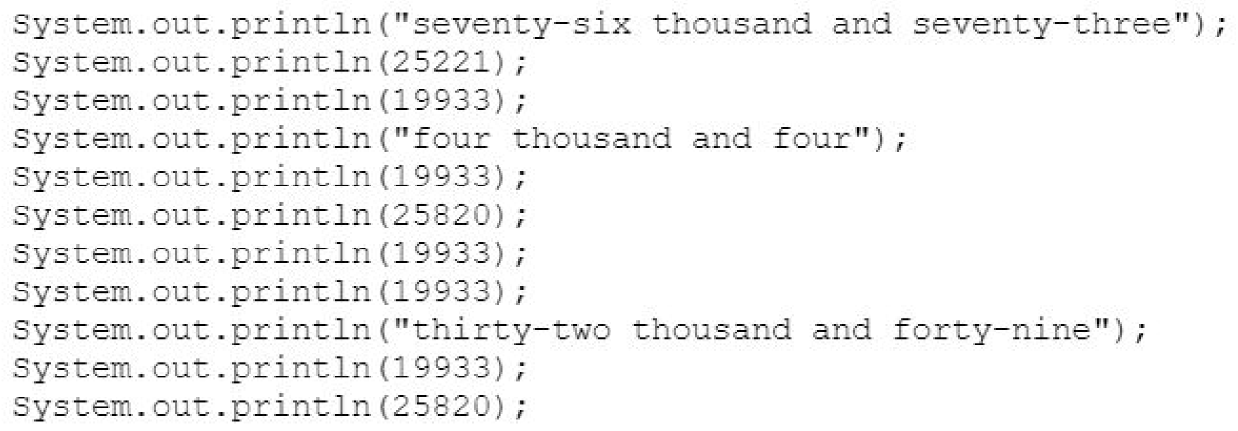


```
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
```

Figure 4. Initial Java Print Dataset Output

We observed that System.out.println(“”); was being correctly generated for each entry; however, the value that was supposed to be printed was not. From this testing, we learned that the default OpenNMT-py configuration can correctly translate common code (in this case the Java print statement).

We then created a more targeted Java print statement dataset consisting of “print number” where number represents all numbers from 1 to 100,000 written in words and in numerical representation. After running this new dataset through our translation model, we achieved the following results:



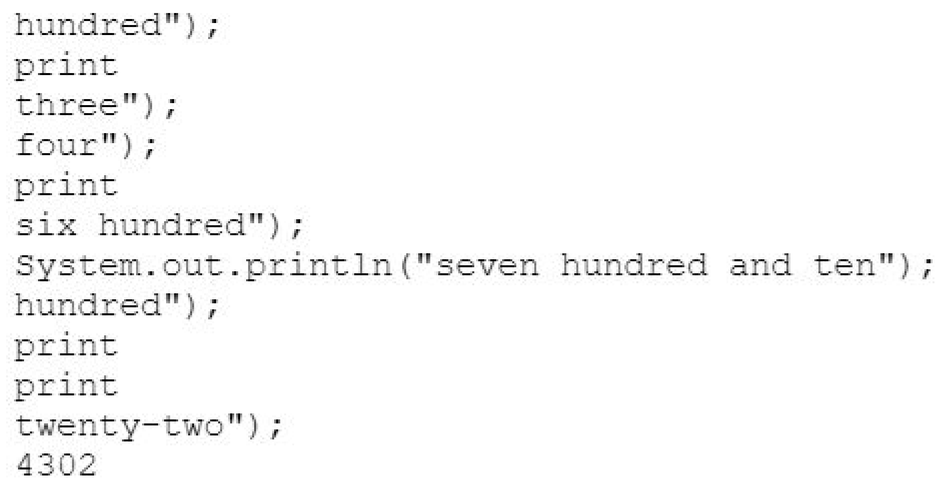
```
System.out.println("seventy-six thousand and seventy-three");
System.out.println(25221);
System.out.println(19933);
System.out.println("four thousand and four");
System.out.println(19933);
System.out.println(25820);
System.out.println(19933);
System.out.println(19933);
System.out.println("thirty-two thousand and forty-nine");
System.out.println(19933);
System.out.println(25820);
```

Figure 5. Java Print Number Output

We could see that the model is now populating the print statement with values; however, these values were not correct (they were supposed to be numbers 1-10). After observing these results, we started experimenting with the OpenNMT-py network architecture and hyperparameters.

OpenNMT-py Configuration

First, we tried using the Transformer architecture. Passing the same dataset through, we achieved the following results:



```
hundred");
print
three");
four");
print
six hundred");
System.out.println("seven hundred and ten");
hundred");
print
print
twenty-two");
4302
```

Figure 6. Java Print Number Transformer Dataset

For some reason, the results achieved were worse than the basic network architecture. This could have been due to the Transformer model requiring many GPU resources, so we have to scale down some of the parameters.

Next, we tried using an RNN network architecture. After running the same dataset through the RNN architecture, we achieved the following results:

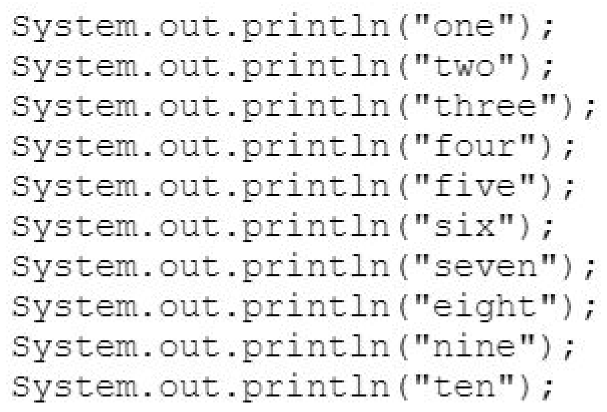

```
System.out.println("one");
System.out.println("two");
System.out.println("three");
System.out.println("four");
System.out.println("five");
System.out.println("six");
System.out.println("seven");
System.out.println("eight");
System.out.println("nine");
System.out.println("ten");
```

Figure 7. Java Print Number RNN Output

These results are exactly what we expect. From this experiment, we learned that values that are contained within the training dataset have a high probability of being translated correctly during testing.

Complex Dataset

After confirming our results using a basic dataset, we then tested our results on more complex datasets. The first dataset we tried was the Python Parallel Code Corpora [9]. This dataset consisted of automatically generated natural language to code translations generated from Stack Overflow. After running this dataset through OpenNMT-py, we achieved the following results:

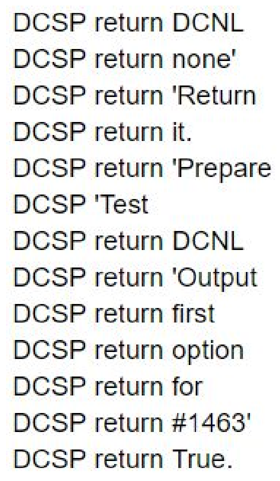
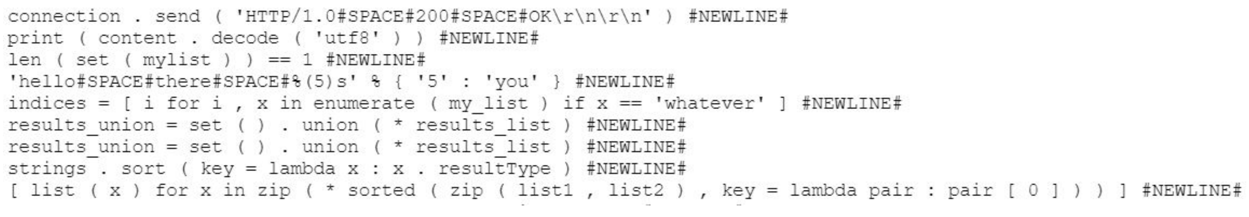

```
DCSP return DCNL
DCSP return none'
DCSP return 'Return
DCSP return it.
DCSP return 'Prepare
DCSP 'Test
DCSP return DCNL
DCSP return 'Output
DCSP return first
DCSP return option
DCSP return for
DCSP return #1463'
DCSP return True.
```

Figure 8. Python Parallel Corpora Output

The results achieved for this dataset are not ideal. It did pull out common syntax (DCSP and return), but the results were not accurate. This is likely due to the natural language statements and expected code translation being very complex (many lines with uncommon syntax).

We then began researching other datasets when we found the Conala dataset [10]. This dataset again was created using automated mining methods from Stack Overflow. However, this dataset also contained manually generated data points that the automatic generation system used when mining. After running this dataset through our system with the Transformer architecture on more powerful GPU resources, we achieved the following results:

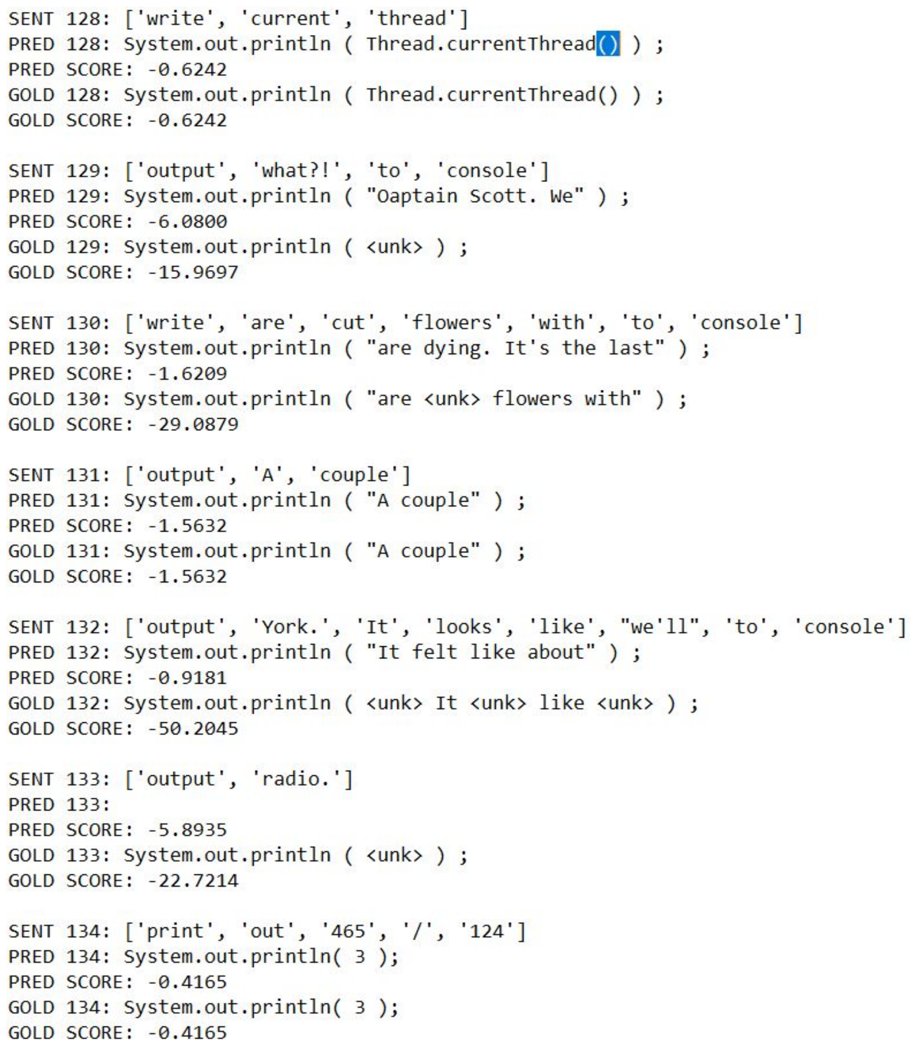


```
connection . send ( 'HTTP/1.0#SPACE#200#SPACE#OK\r\n\r\n' ) #NEWLINE#
print ( content . decode ( 'utf8' ) ) #NEWLINE#
len ( set ( mylist ) ) == 1 #NEWLINE#
'hello#SPACE#there#SPACE#%(5)s' % { '5' : 'you' } #NEWLINE#
indices = [ i for i , x in enumerate ( my_list ) if x == 'whatever' ] #NEWLINE#
results_union = set ( ) . union ( * results_list ) #NEWLINE#
results_union = set ( ) . union ( * results_list ) #NEWLINE#
strings . sort ( key = lambda x : x . resultType ) #NEWLINE#
[ list ( x ) for x in zip ( * sorted ( zip ( list1 , list2 ) , key = lambda pair : pair [ 0 ] ) ) ] #NEWLINE#
```

Figure 9. Conala Dataset Output

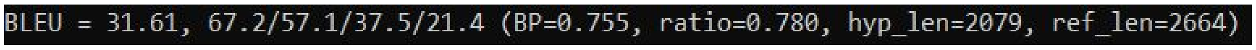
We observed that the results resembled their expected code much more closely. This led us to determine how we can create a more accurate dataset that would achieve greater performance or how we can update our network architecture to achieve better results on the Conala dataset.

Next, we generated a dataset targeting the Java print statement. Our dataset contained various hardcoded strings, arithmetic, variables, and function calls. After running the dataset through our system, we received the following results and metrics:



```
SENT 128: ['write', 'current', 'thread']
PRED 128: System.out.println ( Thread.currentThread() ) ;
PRED SCORE: -0.6242
GOLD 128: System.out.println ( Thread.currentThread() ) ;
GOLD SCORE: -0.6242

SENT 129: ['output', 'what?!', 'to', 'console']
PRED 129: System.out.println ( "Oaptain Scott. We" ) ;
PRED SCORE: -6.0800
GOLD 129: System.out.println ( <unk> ) ;
GOLD SCORE: -15.9697

SENT 130: ['write', 'are', 'cut', 'flowers', 'with', 'to', 'console']
PRED 130: System.out.println ( "are dying. It's the last" ) ;
PRED SCORE: -1.6209
GOLD 130: System.out.println ( "are <unk> flowers with" ) ;
GOLD SCORE: -29.0879

SENT 131: ['output', 'A', 'couple']
PRED 131: System.out.println ( "A couple" ) ;
PRED SCORE: -1.5632
GOLD 131: System.out.println ( "A couple" ) ;
GOLD SCORE: -1.5632

SENT 132: ['output', 'York.', 'It', 'looks', 'like', "we'll", 'to', 'console']
PRED 132: System.out.println ( "It felt like about" ) ;
PRED SCORE: -0.9181
GOLD 132: System.out.println ( <unk> It <unk> like <unk> ) ;
GOLD SCORE: -50.2045

SENT 133: ['output', 'radio.']
PRED 133:
PRED SCORE: -5.8935
GOLD 133: System.out.println ( <unk> ) ;
GOLD SCORE: -22.7214

SENT 134: ['print', 'out', '465', '/', '124']
PRED 134: System.out.println( 3 );
PRED SCORE: -0.4165
GOLD 134: System.out.println( 3 );
GOLD SCORE: -0.4165
```

Figure 10. Conala Dataset Translations



```
BLEU = 31.61, 67.2/57.1/37.5/21.4 (BP=0.755, ratio=0.780, hyp_len=2079, ref_len=2664)
```

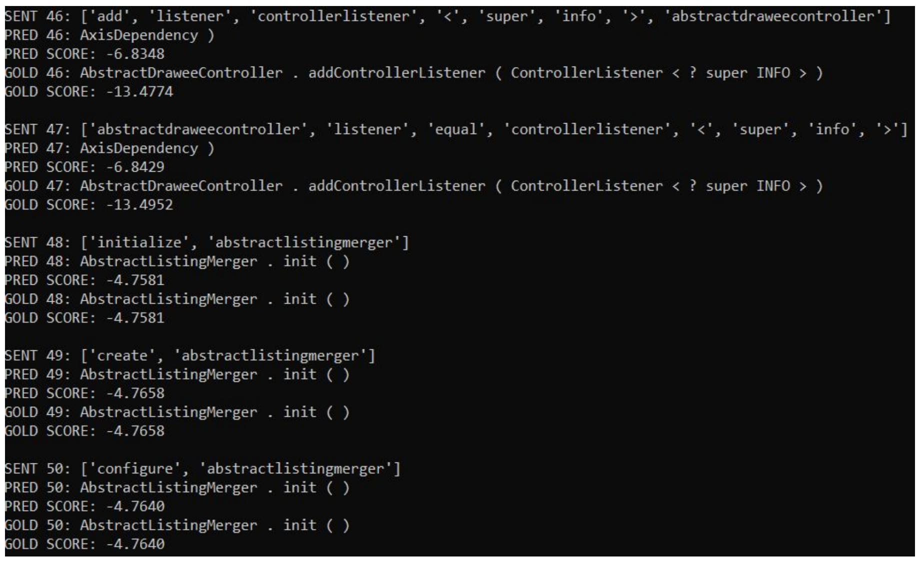
Figure 11. Conala Dataset BLEU Score

Accuracy = 40%

We observed that the arithmetic statements were translated correctly almost all the time, but string values were not. Additionally, variables and function calls had an accurate translation. These results led us to create a dataset that supports more generalized natural language inputs.

The next dataset we created included different types of Java method invocations that were mined from online sources like GitHub using the GitHub API with the Octokit C# library. The natural language part of the dataset consisted of only verbs representing the action or method and nouns representing the method parameters. All other parts of the sentence were removed to allow OpenNMT-py to focus on the more important parts of the sentence. The method parameters and variables were converted to their Java types. Our final dataset consisted of approximately 1,000 Java different method invocations and about 1,500 total samples.

A snippet of the results can be seen below:



```
SENT 46: ['add', 'listener', 'controllerlistener', '<', 'super', 'info', '>', 'abstractdraweecontroller']
PRED 46: AxisDependency )
PRED SCORE: -6.8348
GOLD 46: AbstractDraweeController . addControllerListener ( ControllerListener < ? super INFO > )
GOLD SCORE: -13.4774

SENT 47: ['abstractdraweecontroller', 'listener', 'equal', 'controllerlistener', '<', 'super', 'info', '>']
PRED 47: AxisDependency )
PRED SCORE: -6.8429
GOLD 47: AbstractDraweeController . addControllerListener ( ControllerListener < ? super INFO > )
GOLD SCORE: -13.4952

SENT 48: ['initialize', 'abstractlistingmerger']
PRED 48: AbstractListingMerger . init ( )
PRED SCORE: -4.7581
GOLD 48: AbstractListingMerger . init ( )
GOLD SCORE: -4.7581

SENT 49: ['create', 'abstractlistingmerger']
PRED 49: AbstractListingMerger . init ( )
PRED SCORE: -4.7658
GOLD 49: AbstractListingMerger . init ( )
GOLD SCORE: -4.7658

SENT 50: ['configure', 'abstractlistingmerger']
PRED 50: AbstractListingMerger . init ( )
PRED SCORE: -4.7640
GOLD 50: AbstractListingMerger . init ( )
GOLD SCORE: -4.7640
```

In total, our system has about 50-60% accuracy and a BLEU score of 66.5, showing that our preprocessing method is a good solution to this problem. The accuracy results achieve the about 50% accuracy we were aiming for while the BLEU score highly exceeds our expectations.

6. Implementation

6.1 IntelliJ Plugin

The IntelliJ plugin acts as the user interface of the project. Here, the user is presented with a Java IDE text editor where the user can enter natural language statements or normal Java code. If the user enters a natural language statement, they can interact with the translation button that will preprocess that natural language statement by converting the method parameters and variables to their types and format the sentence into verb-noun format by calling a Python script hosted on our AWS server. An example of this translation can be seen below:

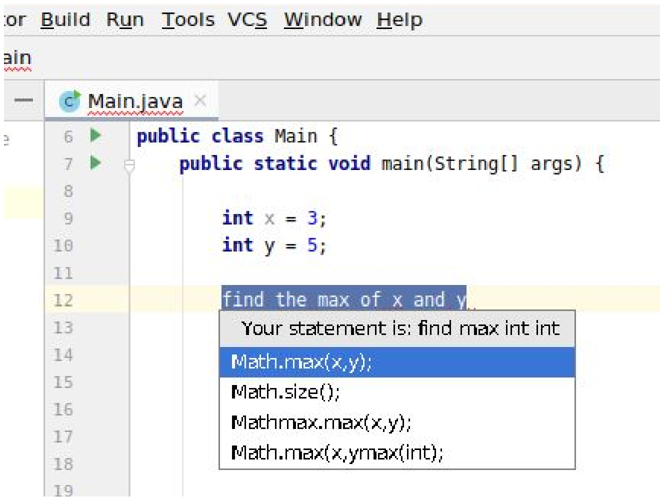
Input Statement: return the char value at index pos+1 for str
Preprocessed Statement: return char value index int string

The above example shows that the variables pos+1 and str are converted into their Java types of int and string respectively. Also, only the verbs and nouns from the original input sentence remain after preprocessing while the other parts of the sentence are removed. While it is not required, to achieve better translation results, the user is encouraged to follow the below steps:

- Do not include spaces between multiple words or operations in raw natural language statement
- Do not include commas between method parameters in natural language input
- Avoid using punctuation in raw natural language statement
- Write hardcoded String surrounded by double quotes in raw natural language statement
- Include all information about objects being called and nested and chained method information
    - Only the last method call will be translated in a chained method call chain
    - Nested method invocations will be written as is in natural language statement

This helps the OpenNMT-py translation engine to focus on the most important words, leading to more accurate translation results. A restriction to our system is that only the

outermost method (in the case of nested method invocations) and the last method (in the case of a chained method invocation) are translated. This means that all other methods in a nested or chained method call must be explicitly stated by the user as input.
A screenshot of the IntelliJ Plugin interface and functionality is shown below:



```
public class Main {
    public static void main(String[] args) {

        int x = 3;
        int y = 5;

        find the max of x and y
```

Your statement is: find max int int

```
Math.max(x,y);
Math.size();
Mathmax.max(x,y);
Math.max(x,ymax(int);
```

Figure 12. IntelliJ Plugin User Interface

The above screenshot shows the text area where there is a mix of normal Java code and a natural language statement. The user then interacted with the natural language statement (find the max of x and y) first selecting it and left clicking on it to select the translate option, triggering the preprocessing and translations actions. During the translation process, the OpenNMT-py translation engine runs the preprocessed natural language statement through the trained model and returns the top-n most confident expected Java code translations. The user can then click on the suggestion they want and that code snippet will be populated in the location in the text editor where the original statement was. In general, the user interface is used to support most of the functional requirements.

6.2 Dataset

A significant part of this project involved creating a dataset that consisted of the natural language statement, equivalent Java code pairs. The natural language statements are in the verb-noun format described above with each parameter converted into their type. This will allow preprocessed user input natural language statements to be in the same format as the dataset natural language labels and generalize better given that our system cannot account for every possible variable or method name. Additionally, the natural language labels for each method were generated by starting with the Java documentation description of the method. This ensured more uniform dataset labeling since six different members were labeling the dataset. The Java code consists of the method format with its parameters stored as their type and each token separated by a space. Creating the dataset in this format allowed our neural machine translation system to focus on the main parts of the sentence during translation, helping to reduce the ambiguity of natural language as the description of the method would map to the method name and the Java type parameters would map to each other.

The Java method invocations in the dataset were automatically mined using the C# Octokit library and GitHub API. The natural language statements were manually generated but automatically preprocessed to put them in the required dataset format.

6.3 Classification and Translation Engine

OpenNMT-py, a neural machine translation system, was used to convert the natural language statement to its equivalent Java code. The classification engine of OpenNMT-py would take the dataset explained above and train a model using the transformer architecture (a commonly used and well performing neural machine translation architecture). This trained model would then be passed to the translation engine. The translation engine would then take the preprocessed input from the user interface and run it through the trained model. The five most confident expected code translations would then be passed as output to the user interface to be displayed. The communication between the user interface and the translation engine takes place using OpenNMT-py's REST API interface. The OpenNMT-py REST API is hosted on an AWS server.

7. Closing Material

7.1 Conclusion

7.1.1 Development Progress

In the development of the Intelligent Code Editor, we have done the following:

- Looked into existing tools related to NLP (Natural Language Processing) and its use in translating natural language to code
  - Discovered OpenNMT, an open-source neural machine translation system
  - Used for research on various translations (image-to-text, English-to-Spanish) [4]
- Completed a literature analysis on research related to translating natural language to code
  - A paper on a tool called *anyCode* was particularly insightful [7]
- Worked on the development of the plugin
  - Made prototype plugins for Visual Studio Code, IntelliJ, and Eclipse
  - Ultimately decided to develop the plugin for IntelliJ
- Created an IntelliJ plugin
  - Allows the user to enter and select natural language in the code editor
  - Passes the entered natural language statement to the preprocessing script and the translation engine
  - Receives the expected code translation back from the translation engine
  - Uses context to assign variables a generic name myVar to ease the burden on the translation engine
- OpenNMT-py Configuration
  - Configured OpenNMT-py on the Pronto GPU server
  - Determined how to run a dataset through the classification engine
  - Determine how to adjust network architecture and hyperparameters
  - Researched and tested different datasets, architectures, and hyperparameters
- Dataset Creation

    - Created a Java print statement dataset to use as a baseline for our translation results
- NLTK Preprocessing
    - Preprocess the original natural language statement into Verb-Noun format
        - Example: *stop thread* (*stop* is the verb and *thread* is the noun)
    - Change all verbs to present tense
    - Convert all characters to lowercase
- Method Invocation Dataset Creation
    - Automatically mine Java method invocation statements from online sources like GitHub
    - Label the natural language statement part of these mined statements using Verb-Noun format
- Create a Part of Speech Tagger
    - Create a part of speech tagger that more accurately labels the verbs and nouns in programming domain sentences
- User Interface Parameter Mapping
    - Create technique for front-end to convert the input natural language statement variables and method parameters to their Java types.
- AWS Server Integration
    - Hosted the NLTK preprocessing script and OpenNMT-py REST API on AWS
        - Preprocessing script deployed as an AWS Lambda function
        - OpenNMT-py server running continuously on dedicated EC2 instance
- End-to-end System Integration
    - Connected each of the components together to create a fully integrated final system

While we were able to create a system that has about 50-60% accuracy, further improvements could be made to improve our system in the future.

Potential Ways to Improve Accuracy

- **Problem**: <unk> token issues
    - **Information**: <unk> token errors occur when there are words that are not contained in the trained models vocabulary

    - **Solution**: Determine a better way to handle <unk> token errors
        - Could use a system like GloBE word embeddings to find the most likely word given all the words stored in the vocabulary
        - The conala dataset used conditional probabilities to find the most likely word match from the vocabulary
        - Include a greater variety of programming domain words in our trained vocabulary
- **Problem**: Lack of generalization in preprocessing
    - **Information**: Since we are manually generating the natural language statements for the code translations, there are limitations to the results given that we cannot generate all possible word substitutions or sentence structures
    - **Solution**: These could be improved using automated methods to preprocess the sentence.
        - Something similar to AnyCode's WordNet could be used to generate all possible word substitutions
        - NLTK parse trees could be used to generate different sentence structures
            - We could also see about using the part of speech information in our neural machine translation system
- **Problem**: Having the system try to guess the correct output instead of trying to match to the best output
    - **Information**: Since our translations are not guaranteed to be a valid Java method, this could cause the translations to be a little off, but that could result in an invalid Java translation
    - **Solution**: Some other systems try to match to the most likely translation from a list of possible valid translations
            - Again, this could be improved by using something like GloBE word embeddings or Conala dataset conditional probabilities that could choose the translation with the most likely translation
- **Problem**: Our dataset is not trained on every possible Java method invocation

- **Information**: Since our model is only trained on the most frequent Java method invocations from the top GitHub Java projects, not all the Java methods are included
- **Solution**: We could create a system that supports all built-in Java methods or a focus group (like Machine Learning)
    - We could mine our Java methods from the Java documentation instead of GitHub
    - If we use a subset, we could mine our Java methods from that framework's API

- **Problem**: OpenNMT-py not having the best translation results
    - **Information**: Hung showed us Google-NMT that generally had better BLEU results compared to OpenNMT-py
    - **Solution**: We could implement our neural machine translation system using a better translation system
- **Problem**: Lacking supercomputers to train the model on
    - **Information**: Google spent about a week with 96 TPUs training their basic translation model, so our training resources are lacking compared to these
    - **Solution**: This could be solved by trying to find more power computing resources to train the model on
- **Problem**: Manual labeling process is tedious and may lead to inconsistent labeling among different people, even with specific instructions to follow like we had
    - **Information**: Obtaining a large sample of Java method invocations is nearly trivial, as Keaton's work has shown us. We were able to mine only method invocations fairly precisely out of huge Java projects on GitHub.
    - **Solution**: Professor Jannesari suggested techniques such as RenderGAN (https://arxiv.org/pdf/1611.01331.pdf) to generate labeled data. Amazon AWS provides a service for automating data labeling that might use papers like RenderGAN under the hood: https://docs.aws.amazon.com/sagemaker/latest/dg/sms-data-labeling.html
- **Problem**: Unlike the *anyCode* paper, we did not do as much NLP before using the data in the model.
    - **Solution**: Professor Jannesari suggested that building a layer upon BERT (https://github.com/google-research/bert#what-is-bert) might improve results.

7.2 References

[1] JetBrains IntelliJ Platform SDK. (2019). *Creating Your First Plugin*. [online] Available at: http://www.jetbrains.org/intellij/sdk/docs/basics/getting_started.html [Accessed 30 Sep. 2019].

[2] Amsden, J. (2019). *Your First Plug-In*. [online] Eclipse.org. Available at: https://www.eclipse.org/articles/Article-Your%20First%20Plug-in/YourFirstPlugin.html [Accessed 30 Sep. 2019].

[3] Code, V. (2019). *Extension API*. [online] Code.visualstudio.com. Available at: https://code.visualstudio.com/api [Accessed 24 Sep. 2019].

[4] Wu, Y., Schuster, M., Chen, Z., Le, Q. and Norouzi, M. (2016). *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. [online] Google. Available at: https://arxiv.org/pdf/1609.08144.pdf [Accessed 30 Sep. 2019].

[5] Opennmt.net. (2019). *Contents — OpenNMT-py documentation*. [online] Available at: http://opennmt.net/OpenNMT-py/ [Accessed 24 Sep. 2019].

[6] Lin, X., Wang, C., Pang, D., Vu, K., Zettlemoyer, L. and Ernst, M. (2019). *Program Synthesis from Natural Language Using Recurrent Neural Networks*. [online] Seattle, WA, USA: Paul G. Allen School of Computer Science & Engineering. Available at: https://homes.cs.washington.edu/~mernst/pubs/nl-command-tr170301.pdf [Accessed 24 Sep. 2019].

[7] T. Gvero and V. Kuncak, "Synthesizing Java expressions from free-form queries," ACM SIGPLAN Notices, vol. 50, no. 10, pp. 416–432, 2015.

[8] Researchit.las.iastate.edu. (2019). *Pronto Job Manager | ResearchIT*. [online] Available at: https://researchit.las.iastate.edu/pronto [Accessed 24 Oct. 2019].

[9] Valerio, A., Barone, M. and Sennrich, R. (2017). A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. [online] Available at: https://arxiv.org/pdf/1707.02275.pdf [Accessed 24 Oct. 2019].

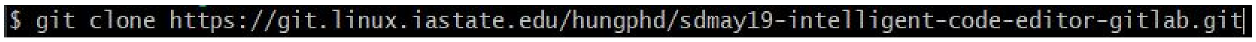
[10] Yin, P., Deng, B., Chen, E., Vasilescu, B. and Neubig, G. (2018). Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. [online] Available at: https://arxiv.org/pdf/1805.08949.pdf [Accessed 24 Oct. 2019].

7.3 Appendices

7.3.1 Operation Manual

Setup/Demo:

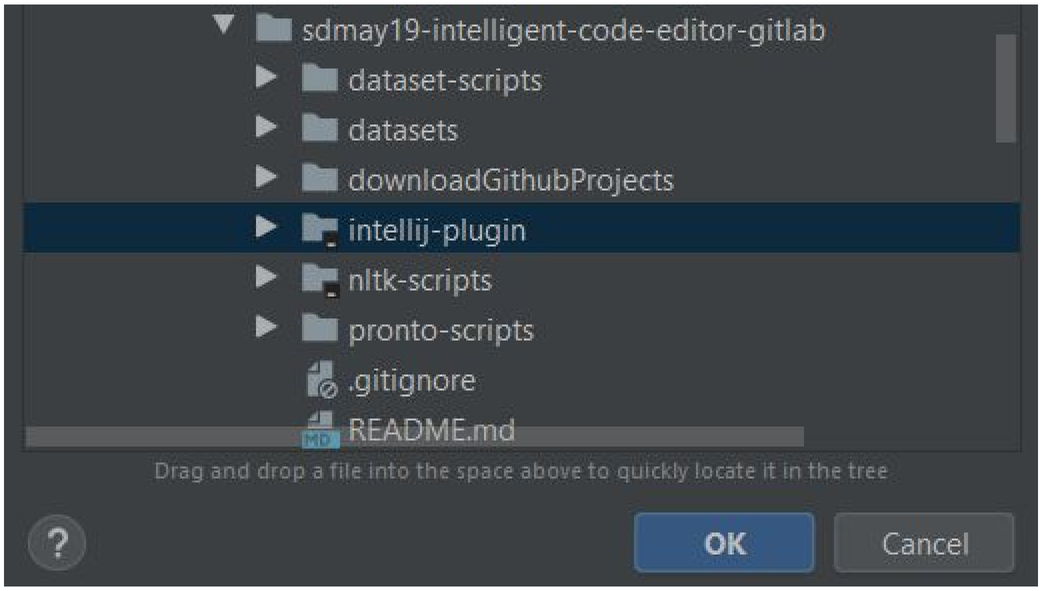
1. After obtaining access, clone our project's GitLab repository at https://git.linux.iastate.edu/hungphd/sdmay19-intelligent-code-editor-gitlab



```
$ git clone https://git.linux.iastate.edu/hungphd/sdmay19-intelligent-code-editor-gitlab.git
```

Figure 13. Git Clone Project Command

2. Open the project in IntelliJ IDEA. (/intellij-plugin)

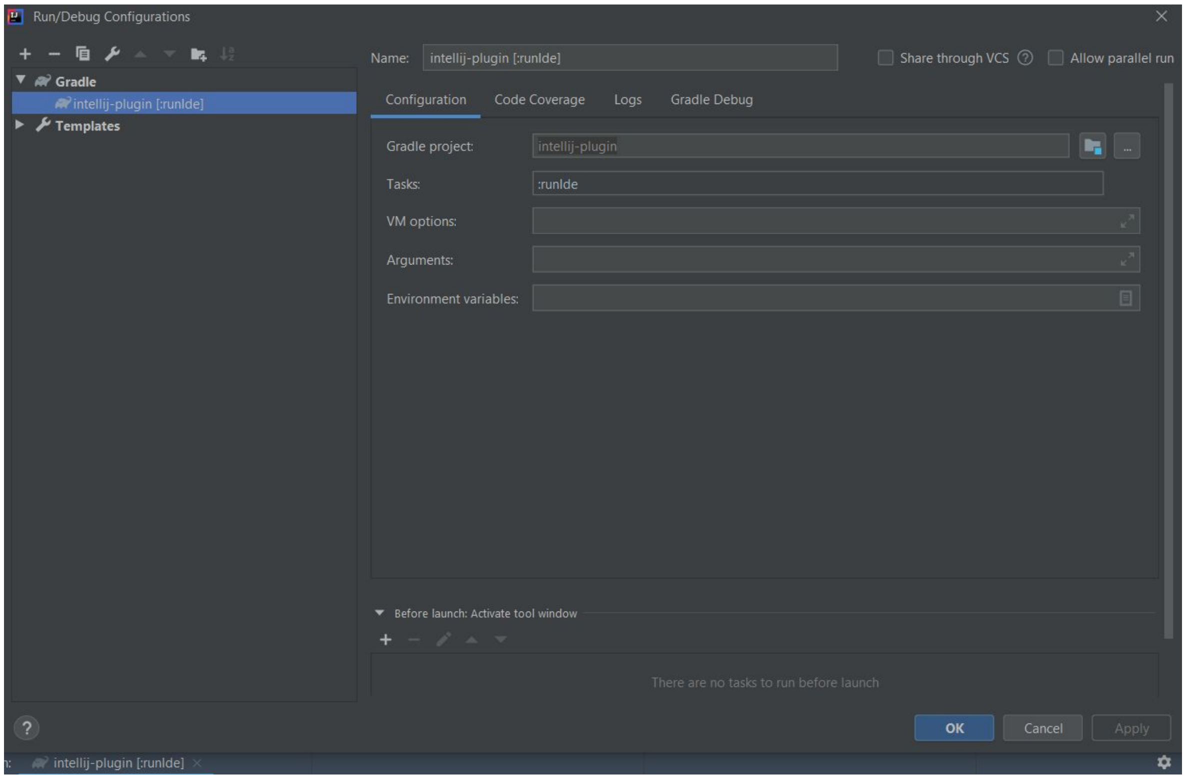


```
▼ sdmay19-intelligent-code-editor-gitlab
    ▶ dataset-scripts
    ▶ datasets
    ▶ downloadGithubProjects
    ▶ intellij-plugin
    ▶ nltk-scripts
    ▶ pronto-scripts
      .gitignore
      README.md
```

Drag and drop a file into the space above to quickly locate it in the tree

Figure 14. IntelliJ Plugin Project Folder

3. Create a new run configuration. From the templates, choose Gradle. Set this project as the Gradle project and add :runIde to Tasks.


Run/Debug Configurations

Name: intellij-plugin [:runIde]

Gradle project: intellij-plugin

Tasks: :runIde

Before launch: Activate tool window

There are no tasks to run before launch

Figure 15. Gradle Build Parameters

4. After running the configuration, you will know that it works if another IntelliJ window opens. This new window has the plugin installed.
5. To obtain translations from the OpenNMT-py server, you'll need to have that running as well. Refer to these installation instructions.

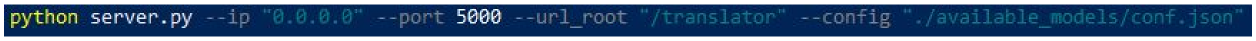
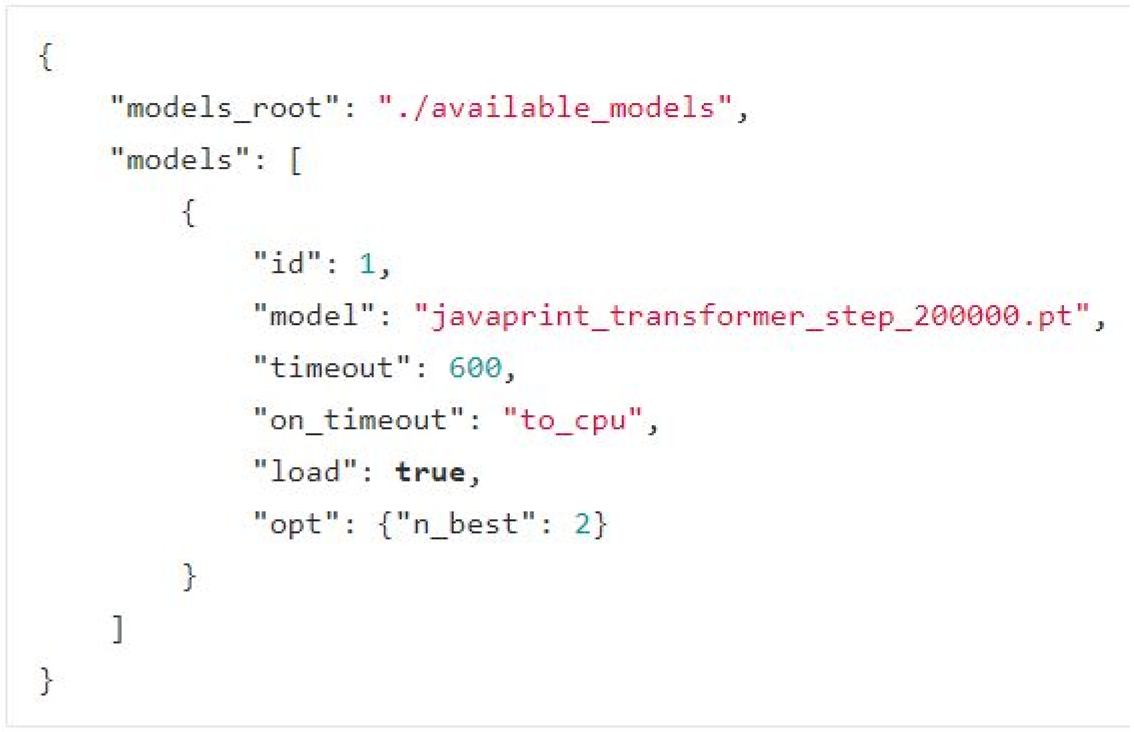

```
python server.py --ip "0.0.0.0" --port 5000 --url_root "/translator" --config "./available_models/conf.json"
```

Figure 16. OpenNMT-py REST API Server Configuration Command

6. Download a trained model from our Google Drive (they end in .pt) and save it to available_models.
7. Create a conf.json in available_models with the following contents:

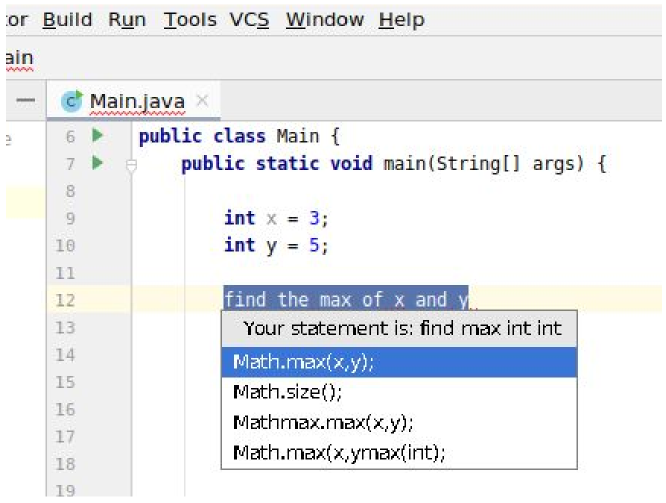


```
{
    "models_root": "./available_models",
    "models": [
        {
            "id": 1,
            "model": "javaprint_transformer_step_200000.pt",
            "timeout": 600,
            "on_timeout": "to_cpu",
            "load": true,
            "opt": {"n_best": 2}
        }
    ]
}
```

Figure 17. conf.json File Contents

Change the model name if needed. Then start the server using these instructions.

After the server starts, you should be able to select text in the development IntelliJ window and use either Shift+T or the context menu to trigger the translation action. Note that only one statement can be translated at a time.



```
public class Main {
    public static void main(String[] args) {

        int x = 3;
        int y = 5;

        find the max of x and y
```

```
Your statement is: find max int int
Math.max(x,y);
Math.size();
Mathmax.max(x,y);
Math.max(x,ymax(int);
```

Figure 18. User Interface Statement Selection and Results

Now the user can select from the list of possible translations. Once they click on a possible translation, it will be displayed and executable in the IDE.

7.3.2 Previous Design Versions and Lessons Learned

The first design version consisted of translations for only a single type of command: the Java print statement (System.out.println()). For this version, we created a dataset that consisted of english statements that were to be translated to some form of a Java print statement. This initial targeted dataset allowed us to learn about what kinds of data resulted in the best translation results. Through creating this dataset, we also learned about the importance of using diverse statement translations to create a well distributed dataset. This allowed us to create our dataset using statement translations from many different resources like GitHub in later revisions instead of creating the entire dataset manually. Doing so gave us better and more general translation results.

For the neural machine translation system, we used OpenNMT-py to train on the Java print dataset. When a user entered the natural language statement, that raw statement would be run through the OpenNMT-py system and would be converted to the respective Java print dataset. An issue that we ran into with this approach was that we were relying on the OpenNMT-py system too much for the translations. During our final presentation for CprE 491, we received a suggestion to only use the neural machine translation system to translate more specific types of statements rather than all types of statements. With this feedback, we then transitioned into the next version of our project, utilizing preprocessing of the English statements.

The second major revision of the project included input and dataset natural language and Java code statement preprocessing and creating a more diverse dataset that contained multiple types of Java method invocations from online code sources like GitHub. The natural language statement preprocessing consisted of only keeping the verbs and nouns from the original statement and converting all words to present tense and all characters to lowercase. Doing so would remove the less important parts of the original statement, which would allow the OpenNMT-py translation engine to focus on the most important words. Later, we also limited the scope of our project by only translating the outermost or last method call in a nested or chained method sequence respectively, allowing for simpler translation results. Future groups may remove this restriction.

Mining the Java method invocation statements from sources like GitHub allows for a more diverse representation in the dataset with real-world use cases. This also allowed us to determine what the most commonly used Java methods are to ensure our system supported them. To label the dataset, we all followed a team-defined set of steps to ensure labeling uniformity between members. During the creation of this version of the project, we learned the importance of data mining, preprocessing, and generalization.

7.3.3 Code

All code related to our project can be found in our GitLab repository. Note that the professor has not given public access to this repository, therefore you will need to contact him first. jannesar@iastate.edu

https://git.linux.iastate.edu/hungphd/sdmay19-intelligent-code-editor-gitlab