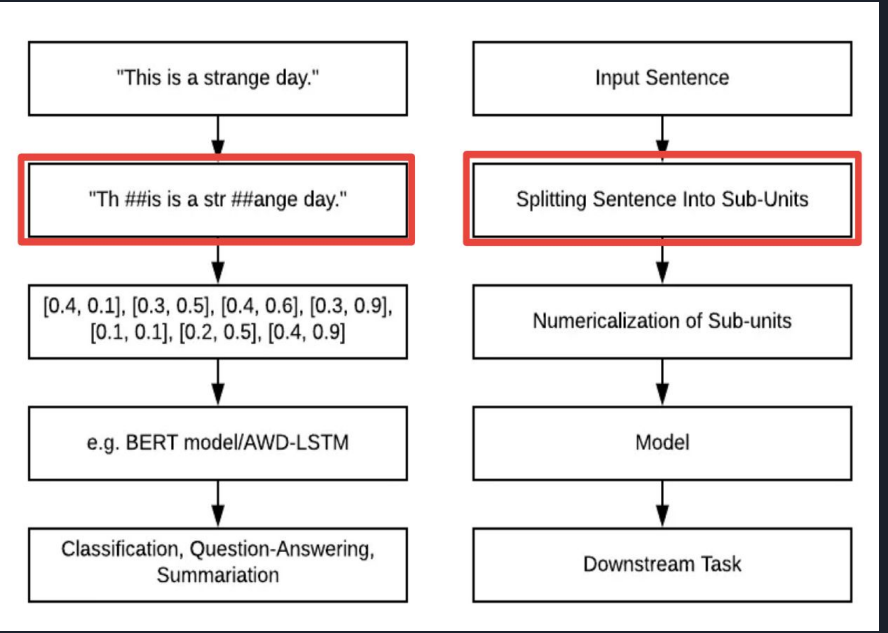
# Tokenizers

- Big code != big vocabulary: Open-vocabulary models for source code
- Code completion with neural attention and pointer networks
- Code prediction by feeding trees to transformers
- IntelliCode compose: Code generation using transformer

# Why tokenization?

The goal is to convert text into numeric vectors so machine can understand. Therefore, we break it, then create vectors and process the large numeric matrices. And this involves, Splitting sentence into words or sub-units called tokens. So Tokenization is part of NLP pipeline



# We are familiar with

- Punkt Sentence Tokenize: Not useful in all cases, for exp with space splitting it does not split I’m — into I and m.
- Gensim’s preprocess_string & simple_processes: Does not have sentence tokenizer
- BertTokenizer from HuggingFace: Good found so far

Let’s explore more Tokenizers

# IntelliCode compose: Code generation using transformer

**Authors:** Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, Neel Sundaresan

# SentencePiece Tokenizer

SentencePiece is used to train tokenization models directly from raw code, without preprocessing like splitting by whitespace.

In IntelliCode Compose: They have used

- SentencePiece tokenizer with 2 variations
  1. Byte-Pair Encoding (BPE) tokenization
  2. Tokenization by splitting programming language identifiers using case conventions

## Byte-Pair Encoding (BPE) tokenization

- Learns tokenization rules by iteratively merging the most frequent pairs of tokens.
- Handles out-of-vocabulary tokens by breaking them into subtokens.

# SentencePiece Tokenizer

## Tokenization by splitting programming language identifiers using casing conventions

- Breaks programming identifiers (e.g., `camelCase`, `snake_case`) into meaningful subtokens (e.g., `myVariable` → `my`, `Variable`).

**How and why?**

- Use regex to split by case changes (e.g., `camelCase`, `PascalCase`) and delimiters (e.g., `_` in `snake_case`).
- Improves model generalization to unseen identifiers by learning subtoken patterns.
- Reduces vocabulary size while preserving semantics.
- Adapts to multilingual programming conventions (e.g., Python, JavaScript, C#).
- Outperforms word-level tokenization (avoids out-of-vocabulary issues) and character-level tokenization (retains high-level structure).

# Why SentencePiece?

- **Closed Vocabulary Problem:** Codebases often have unique identifiers (e.g., variable names) not seen during training. Subtokens from BPE and SentencePiece mitigate this issue.

- **Multilingual Modeling:** A shared tokenizer across multiple programming languages improves efficiency and generalization.

- Flexible framework with built-in pre-processing.

- General-purpose tokenization, Google Translate.

# Why SentencePiece?

- IntelliCode has **perplexity of 1.82,** lower it is, better the model is
- Average edit similarity **86.7%**

**AST based**

- Requires parsing code into Abstract Syntax Trees, which is computationally expensive and language-specific.

**Whitespace-Based Tokenization**:

- Fails to generalize to unseen tokens or languages.
- Struggles with inconsistent formatting (e.g., spaces vs. tabs)

**Character-Level Tokenization**:

- Leads to very long input sequences, increasing computational cost.
- Loses semantic context of words or identifiers.

# References


More on SentencePiece
https://github.com/google/sentencepiece?tab=readme-ov-file
https://colabdoge.medium.com/understanding-sentencepiece-under-standing-sentence-piece-ac8da59f6b08

Byte pair encoding
https://huggingface.co/learn/nlp-course/en/chapter6/5

Lexical analysis
https://www.geeksforgeeks.org/introduction-of-lexical-analysis/

Beam Search
https://www.width.ai/post/what-is-beam-search

Implementation
https://colab.research.google.com/drive/16t-p6W_ag-438aqhpakXNN3FiCL1jN-K?authuser=6#scrollTo=2uqIlCZvxY0O

# Code Prediction by Feeding Trees to Transformers

**Authors:** Seohyun Kim, Jinman Zhao, Yuchi Tian, Satish Chandra

# Research Goal

- Improve next-token prediction for code autocompletion.
- Leverage Abstract Syntax Trees (ASTs) in Transformer-based models.
- Achieve state-of-the-art accuracy.

# Abstract Syntax Tree (AST)?

An Abstract Syntax Tree (AST) is a tree representation of the syntactic structure of source code. It represents the hierarchical structure of code based on the rules of the programming language.

# Model Design

**A. SeqTrans**

- **Input Representation:** A simple sequence of source tokens (e.g., `["x", "=", "2"]`).
- **How It Works:**
  - Tokens are embedded into vectors and fed into a Transformer.
  - The Transformer processes the sequence to predict the next token.

**B. PathTrans**

- **Input Representation:** Root-to-leaf paths from the AST.
- **How It Works:**
  1. Each root-to-leaf path is embedded using an LSTM to capture local syntactic context.

# Model Design

**C. TravTrans**

- **Input Representation:** A pre-order traversal of the AST.
  1. Example: For a small AST, the sequence might be `[Assign, Name, Load, x, Num, 2]`.

- **How It Works:**

  This sequence is processed by the Transformer, which learns both the hierarchical relationships and token dependencies.

# Why is the Author Using ASTs?

**Code is Hierarchical**

- **Problem:** Code isn't purely sequential like natural language; it has a hierarchical structure (e.g., nested function calls, loops).

- **Solution with ASTs:** ASTs capture this hierarchy, showing how different elements of code are related, which is critical for accurate predictions.