PAST LIME: Explaining Code Generation via AST-Based Perturbations

Karol Kuźniak[1], Michał Sala[1]

[1]The Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland

Introduction

Code-generation models, such as Stable Code 3B [4], perform well on tasks like completion and bug fixing, but they remain black boxes — offering little insight into ***why*** certain continuations are preferred. Existing explainability methods like LIME [1] struggle with code due to its structured, hierarchical nature and the impracticality of token-level perturbations on long inputs. Naive perturbations often ignore semantics, leading to noisy or misleading explanations.

Our Approach

We address this by using the abstract syntax tree (AST) to segment code into semantically meaningful chunks — such as function definitions, import statements, or comments — which serve as LIME's perturbation units. This allows perturbations to occur at the level of logical code components, improving both efficiency and interpretability.

We call this approach ***PAST LIME (Perturbed Abstract Syntax Tree LIME)***. Instead of focusing on the generated continuation, as is typical in LLM inference, we focus on the **probability** the model assigns to that continuation. This shift allows us to apply LIME to explain **why** a particular continuation — whether valid, arbitrary, or adversarial — receives a certain probability, offering deeper insight into the model's decision-making process.

```
In [5]: %%showast
        def abc():
            for x in 'abc':
                print(x)
            return 0
```

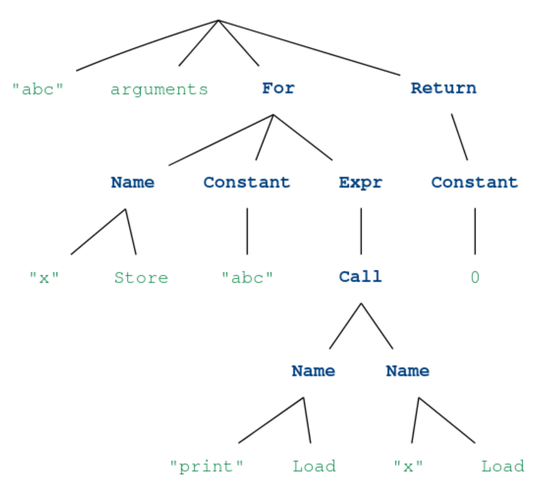

Figure 1: AST visualization for a simple Python function.

Our implementation of this method is available on Gitlab [3].

Methodology

PAST LIME for a given pair *(code, continuation)*, operates as follows:

1. **Parse the Code**: Transform the input *code* into its corresponding abstract syntax tree (AST) representation.
2. **Select Nodes to Perturb**: Choose n AST nodes on which the explanation will be based. This step is policy-based (e.g. selecting top n nodes in the AST).
3. **Generate Perturbed Samples**: Create k perturbed samples of the input code by selectively removing fragments associated with the chosen AST nodes. Each perturbed sample is represented by a binary mask vector $x \in \{0,1\}^n$, indicating which nodes are kept (1) or removed (0).
4. **Calculate Normalized Probabilities**: For each perturbed sample, compute the normalized probability of the model generating the given *continuation*. This yields a target vector $y \in \mathbb{R}^k$.
5. **Train a Surrogate Model**: Fit a simple interpretable linear model using the $X \in \mathbb{R}^{k,n}$ and $y \in \mathbb{R}^k$.
6. **Explanation**: The coefficients of the trained linear model provide the explanation for the model's behavior.

Experiments

We executed ***PAST LIME*** on the Stable Code 3B model, focusing on its Fill-in-the-Middle (FIM) capabilities across three programming languages: Python, JavaScript, and Rust. We used diverse input samples by varying comments, imports, and other structural elements of the code.

Configuration

The node-selection policy used in our experiments chooses the children of ancestors of the smallest node containing the continuation, ensuring no overlap with nodes that contain the continuation itself.

```
    # The following line made me so annoyed with this library that a new one is
    # written. You may not trust me anymore after the next line... in fact, I
    # don't even trust myself... but riakkit-ng is probably going to be better.

    attrs["instances"] = WeakValueDictionary()
    attrs["_references"] = references

    new_class = type.__new__(cls, clsname, parents, attrs)

    bucket_name = attrs.get("bucket_name", None)

    new_class.buckets = {}

    if bucket_name is not None:
      if isinstance(bucket_name, basestring):
        new_class.bucket_name = bucket_name = [bucket_name]

      for bn in bucket_name:
        if bn in _document_classes:
          raise RiakkitError("Bucket name of %s already exists in the registry!"
                             % bn)
        else:
          _document_classes[bn] = new_class

        new_class.buckets[bn] = client.bucket(bn)

      if len(new_class.buckets) == 1:
        new_class.bucket = new_class.buckets.values()[0]
      else:
        new_class.bucket = new_class.buckets[bucket_name[0]]

    for colname, rcls, back_name in references_col_classes:
      rcls._meta[colname] = MultiReferenceProperty(reference_class=new_class)
      rcls._meta[colname].name = colname
      rcls._meta[colname].is_reference_back = back_name
      rcls._references.append(colname)

    return new_class
```

Figure 2: A code snippet with selected nodes highlighted in different colors. The smallest node containing the continuation is marked in black.

As the surrogate model, we employed Lasso regression (with $\alpha = 0.01$) from scikit-learn[6]. For each explanation, we generated $k = 1024$ perturbed samples.

Results

Our method consistently produced reproducible explanations even when the set of selected nodes was significantly larger than $\log_2(k)$. For instance, in Figure 3, as many as 62 nodes were selected, while k remained 1024.

```
    # The following line made me so annoyed with this library that a new one is
    # written. You may not trust me anymore after the next line... in fact, I
    # don't even trust myself... but riakkit-ng is probably going to be better.

    attrs["instances"] = WeakValueDictionary()
    attrs["_references"] = references

    new_class = type.__new__(cls, clsname, parents, attrs)

    bucket_name = attrs.get("bucket_name", None)

    new_class.buckets = {}

    if bucket_name is not None:
      if isinstance(bucket_name, basestring):
        new_class.bucket_name = bucket_name = [bucket_name]

      for bn in bucket_name:
        if bn in _document_classes:
          raise RiakkitError("Bucket name of %s already exists in the registry!"
                             % bn)
        else:
          _document_classes[bn] = new_class

        new_class.buckets[bn] = client.bucket(bn)

      if len(new_class.buckets) == 1:
        new_class.bucket = new_class.buckets.values()[0]
      else:
        new_class.bucket = new_class.buckets[bucket_name[0]]

    for colname, rcls, back_name in references_col_classes:
      rcls._meta[colname] = MultiReferenceProperty(reference_class=new_class)
      rcls._meta[colname].name = colname
      rcls._meta[colname].is_reference_back = back_name
      rcls._references.append(colname)

    return new_class
```

Figure 3: A snippet of a PAST LIME explanation for a sample file extracted from the RiakKit[5] source code. The explained continuation is marked with black background, while the surrounding green highlights indicate the importance scores of the corresponding code fragments. Higher intensity corresponds to greater importance.

Our node-selection policy enabled explaining even highly nested code structures effectively. For instance, in Figure 3, despite the file consisting of 187 lines of deeply nested code, the policy quite successfully isolated the most relevant nodes.

```
import shap
import matplotlib.pyplot as plt
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
import pandas as pd

# Load dataset
boston = load_boston()
X, y = boston.data, boston.target
feature_names = boston.feature_names

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train an XGBoost model
model = xgb.XGBRegressor(objective='reg:squarederror')
model.fit(X_train, y_train)

explainer = shap.Explainer(model, X_train)
shap_values = explainer(X_test)

def prepare_shap_data(shap_values, feature_names):
    """
    Prepares the SHAP values for plotting.
    Returns a DataFrame sorted by the mean absolute SHAP values.
    """
    shap_df = pd.DataFrame(shap_values.values, columns=feature_names)
    shap_mean = shap_df.abs().mean().sort_values(ascending=False)
    return pd.DataFrame({'Feature': shap_mean.index, 'Mean SHAP Value': shap_mean.values})

def plot_shap_summary(df):
    # I hate that plotting library...
    pass

# Prepare data and call the plotting function
shap_plot_data = prepare_shap_data(shap_values, feature_names)
plot_shap_summary(shap_plot_data)
```

Figure 4: An explanation for a comment expressing frustration with a specific plotting library. The green highlights reveal the connection to the library, indicating positive importance scores. Red highlights represent negative importance scores.

The explanations revealed meaningful relationships between code fragments and the model's predictions. When applied to diverse programming languages, our method performed quite robustly. The method's explanations remained consistent across repeated runs.

Conclusion

We proposed a simple, interpretable, and reproducible LIME-based method for explaining code generation. Unlike prior token-level approaches [2] (natural language inference domain), our method operates on the AST level, selecting code fragments mapped to the input's syntactic structure. This produces semantically meaningful and human-readable explanations.

We evaluated our method using Stable Code 3B and found that its explanations aligned well with human intuition. We believe PAST LIME can serve as a useful tool for probing bias and unintended behavior in code generation models.

Future Work

1. Designing alternative node selection policies and perturbation operations.
2. Quantitatively evaluating explanation quality and comparing with baseline methods.
3. Exploring interactive schemes, allowing to recursively refine explanations for greater granularity.
4. Investigating the relationship between continuation length and explanation quality.
5. Testing the method across a wider range of code-generation models.

References

[1] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. arXiv: 1602.04938 [cs.LG]. URL: https://arxiv.org/abs/1602.04938.

[2] James Thorne et al. 2019. arXiv: 1904.10717 [cs.LG]. URL: https://arxiv.org/abs/1904.10717.

[3] URL: https://gitlab.uw.edu.pl/betoniarze/ast-lime.

[4] Nikhil Pinnaparaju et al. URL: https://huggingface.co/stabilityai/stable-code-3b.

[5] URL: https://github.com/shuhaowu/riakkit.

[6] URL: https://scikit-learn.org.