SE4AI - greet (Linguistic Antipatterns) report

Domenico Antonio Gioia
d.gioia7@studenti.unisa.it
Matricola: 0522501518

Giovanni Scorziello
g.scorziello5@studenti.unisa.it
Matricola: 0512108926

Antonio Scognamiglio
a.scognamiglio32@studenti.unisa.it
Matricola: 0522501496

Repository link: https://github.com/xrenegade100/ProjectSunshine

1 INTRODUCTION

1.1 Context of the project

Linguistic antipatterns are problems related to the naming of elements in the source code of a software project. They cause inconsistencies, ambiguity, and confusion in the code, making it less easy to understand, hence it may be difficult for developers to maintain it. For this reason, they are considered a bad code practice. In the context of AI systems, these antipatterns are even more crucial due to their built-in complexity. Unfortunately, there is currently no tool available that can detect these antipatterns in Python code, and notify developers about them. Furthermore, Python is a critical language for AI-based systems, being widely considered the most used language for this kind of software. The only tool supporting this analysis is *IDEAL*[1], which supports only *C#* and *Java*; it's written in Python and it is also packaged as *Intellij Idea* plugin. This tool analyzes the code to search for antipatterns defined by [1], which are divided into the following categories:

(1) **Ambiguity antipatterns**: These antipatterns occur when the names of code elements are ambiguous, making it difficult to understand their meaning and purpose.
(2) **Synonym antipatterns**: These antipatterns occur when synonyms are used for the same concept in different parts of the code, creating inconsistencies and difficulties in understanding.
(3) **Homonym antipatterns**: These antipatterns occur when different names are used to describe same elements in the code, creating confusion and errors.
(4) **False friend antipatterns**: These antipatterns occur when code element names seem to correspond to a familiar concept, but in reality represent something different.
(5) **Verbose antipatterns**: These antipatterns occur when code element names are too long and verbose, making them difficult to understand.

1.2 Goals

The aim is to offer developers a tool that assists them in avoiding introducing linguistic antipatterns in Python source code, based on AI techniques instead of parsing and POS tagging techniques like the current tool, in order to experiment with a different approach to solving the problem and evaluate its results compared to the current solution and thereby preventing all the issues described in the preceding paragraph. To ensure the tool can be used in real-world projects, that often include thousands or even millions of lines of code, it's important to focus on its scalability and efficiency. The tool can also be integrated into an IDE, development tools, or development workflows so that developers can receive real-time feedback when a problem arises and address it right away, potentially increasing the tool's adoption. It will be available to resolve four particular types of linguistic antipatterns among the 19 available in IDEAL. The selected antipatterns are:

(1) **Get more than accessor**: A getter that performs actions other than returning the corresponding attribute.
(2) **Not implemented condition**: The comments of a method suggest a conditional behaviour that is not implemented in the code. When the implementation is default this should be documented.
(3) **Method signature and comment are opposite**: The declaration of a method is in contradiction with its documentation.
(4) **Attribute signature and comment are opposite**: The declaration of an attribute contradicts its documentation.

Based on the goals defined for the project, the following research questions arise:

(1) **RQ-1** *How accurate is the tool in identifying linguistic antipatterns in Python code?*
This research question is critical because it addresses the main objective of the tool. Accuracy is crucial to ensure that the identified issues are valid and so they can be addressed by developers. Therefore it is essential to evaluate the accuracy of the tool to ensure that it is reliable.
(2) **RQ-2** *How scalable is the tool when analyzing large-scale Python codebases?*
Large software projects, often consist of thousands or even millions of lines of code. Besides accuracy, it is also important for the tool to handle large-scale codebases to ensure it can be used in real-world scenarios, otherwise, it may become unusable in practical applications, making it useless even if it is accurate enough.
(3) **RQ-3** *In what measure can the tool, integrated with existing development tools, provide developers prompt feedback about linguistic antipatterns?*
Since the issues identified by the tool must be addressed by developers quickly and efficiently, one of the focuses must be on usability, which can be increased by integrating the tool with existing development tools. Providing real-time feedback is critical because even if the tool's accuracy and scalability are good, it would not be used actively to address the problem of linguistic antipatterns.

1.3 Model development

The first step in the model development phase concerns the selection of the model. Given the nature of linguistic antipatterns, a technique capable of analyzing and understanding natural language text is necessary.
Therefore, among the NLP techniques, the deep learning model *Transformer* will be considered due to its characteristics:

[1] https://github.com/SCANL/ProjectSunshine

- **Self-Attention mechanism:** the *transformer* model has this mechanism that analyzes and weights the importance of a word within a sentence allowing it to be more accurate in understanding dependencies between words in a sentence, compared to other NLP models.
- **Performance:** due to its structure *transformer* is able to process sentences faster and in a parallel way so even a large amount of data is processed very efficiently.
- **Encoder-Decoder:** the model uses two components, namely *encoder* and *decoder* which enable the use of a wide variety of NLP tasks.
- **Pretrain-finetune:** typically the training phase is divided in 2 stages: *pretrain* and *finetune*. During *pretrain* the model uses the unsupervised learning paradigm, while during *finetune* it switches to supervised fine-tuning where the model is fed with labeled data.

For the implementation, *CodeBERT* will be used, a pre-trained bimodal model for programming language (PL) and natural language (NL). This model has been trained on 6 different programming languages (Python, Java, JavaScript, PHP, Ruby, Go), so it is a perfectly usable model for implementing the solution. Transfer Learning mechanisms will be applied to specialize the model in recognizing Linguistic Antipattern within Python code, it will then be trained using a dataset containing labeled Python code snippets, so that the model can be capable of identifying the different constructs that can be encountered in the source code. Data Engineering mechanisms will be applied, in order to maximize the quality of the data, preparing, cleaning, and labeling them with the aim of making the system able to comprehend the input it is given, determine whether there are linguistic antipatterns, and subsequently inform the developer through a report containing information on which and how many of them have been found in the code and their location. The dataset that will be used will be created ad hoc, comprising data from various datasets. This will allow for a better understanding of the dataset, enabling easier identification of errors in the model.

1.4 Methodical steps to address the goals

As shown in Figure 1, the steps that will be taken to ensure the project goals are addressed are:

(1) **Study analysis**: Studying articles that explain the problem of linguistic antipatterns in source code, and examining code examples that exhibit such antipatterns.
(2) **Current tool analysis**: The next stage involves analyzing the IDEAL tool to gain a deeper understanding of its capabilities and functionalities. A practical test will also be conducted by using the tool to analyze a small project and review the output generated. Valuable insights into how the IDEAL tool works and its effectiveness in real-life scenarios will be obtained through this process. The aim is to identify the strengths and weaknesses in IDEAL to create a set of functional and non-functional requirements for the novel system, that will be defined in a way to address at least one of the weaknesses found.
(3) **Model study:** To identify and analyze linguistic antipatterns, a text comprehension and analysis model is required. Thus, various NLP models will be evaluated to determine the most suitable one for accomplishing the system's objective. During the *fine-tune* phase, the model will be fed with data to increase its accuracy in predictions. In particular, the extraction of useful data will occur through the study of selected examples of instances of linguistic antipatterns to create a dataset that is accurate and diverse. This will increase the quality and accuracy of the analyses performed by the system. For this reason, to speed up the data collection and analysis process, the IDEAL tool itself will be used and Java and C# projects on GitHub will be analyzed. The aim is to identify linguistic antipatterns within the projects, select those that conform to the types that our model should recognize, and analyze them, to replicate them as accurately as possible within our dataset.
(4) **Design the new tool**: The fourth stage involves defining the design goals for the new system, based on the goals defined in Section 1.2 and on the weaknesses found in analyzing the current tool. Among these requirements, identifying the four antipatterns mentioned earlier is a priority.
(5) **Quality testing**: The tool will undergo testing to fulfill specified requirements. The system's accuracy will be evaluated by testing it against a collection of Python projects that have been created, containing linguistic antipatterns. A reference point or "oracle" will be generated from this testing, which can be used to compare the analysis results produced by the system. A dataset was created from scratch because a set of projects suitable for the specified use was not available after an analysis of the available data. Testing on a dataset with more confidence will make it easier to identify any analysis errors made by the system.
(6) **Deployment**: The new tool will then be made available to developers to be used both as a stand-alone system and as an integration for an IDE according to the defined design goals.

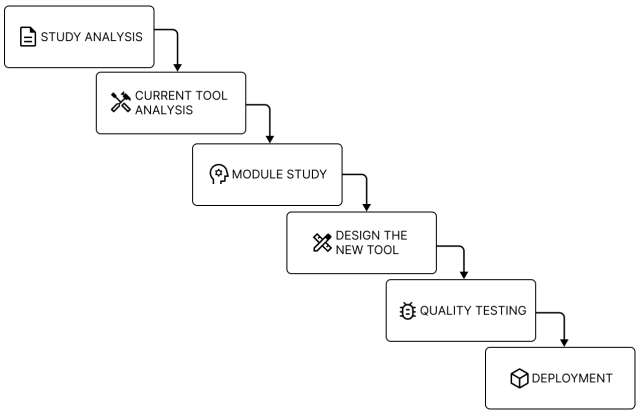

Figure 1: Phase diagram

1.5 System Requirements

After having studied the articles about linguistic antipatterns, and considering the goals of this project, some functional and non-functional requirements have been elicited, with a focus on performance, scalability and reliability. The various aspects of functional requirements have been prioritized according to specific criteria, including how much the market requires the new functionality, how much it increases the impact of the system in the real world and how much it can help the programmer in the development. For non-functional requirements, the aspects considered are: after how long the system can provide a report of analysis, how much the size of a project affects the speed of analysis, and how accurate it can be regardless of the size of the project. The various functional and non-functional requirements have been evaluated on a scale of 1 to 10 to determine their priority. A value of 1 indicates that the functionality has a low development priority, while a value of 10 indicates that it has a high development priority. The functional requirements are:

(1) **FR-1**: The tool must analyze a set of files written in Python code. *Priority: 10/10*
(2) **FR-2**: The tool must detect the linguistic antipatterns of the category "Get more than accessor". *Priority: 10/10*
(3) **FR-3**: The tool must detect the linguistic antipattern of the category "Not implemented condition". *Priority: 10/10*
(4) **FR-4**: The tool must detect the linguistic antipattern of the category "Method signature and comment are opposite". *Priority: 10/10*
(5) **FR-5**: The tool must detect the linguistic antipattern of the category "Attribute signature and comment are opposite". *Priority: 10/10*
(6) **FR-6**: The tool must provide prompt feedback to developers when a linguistic antipattern is detected. *Priority: 8/10*
(7) **FR-7**: The tool must be able to be used as a standalone system. *Priority: 7/10*
(8) **FR-8**: The tool must be packaged as a Visual Studio Code extension. *Priority: 5/10* – According to a survey[2] by Stack Overflow, Visual Studio Code is the most preferred IDE among all developers subscribed to the community. The survey is from 2022, and it remarks that 74.48% of developers who replied prefer Visual Studio Code as their IDE.

The non-functional requirements are:

(1) **NFR-1 Reliability:** The system must be able to recognize 90% of the linguistic antipatterns in the project. *Priority: 9/10*
(2) **NRF-2 Performance:** The tool must be able to parse 10,000 lines of Python code in less than 2 minutes. *Priority: 8/10*
(3) **NRF-3 Usability:** The system must be self-installing, configuring all the necessary components for its operation. This non-functional requirement has arisen from an analysis of the weaknesses of IDEAL. *Priority: 8/10*
(4) **NRF-4 Scalability:** The computation time difference between the analysis of a project with less than 70,000 lines of code and a project with up to 700,000 lines of code should not be more than 50%. *Priority: 6/10*

[2]https://survey.stackoverflow.co/2022/#most-popular-technologies-new-collab-tools

2 SOLUTION IMPLEMENTATION

2.1 Dataset Construction

First and foremost, it is important to highlight that specific datasets containing examples of linguistic antipatterns are not available online, as this is a relatively new research field. Therefore, a dataset was constructed from scratch, utilizing various techniques of data engineering and data cleaning. In the following section, we will delve into the specific methodologies employed during the construction process. The initial phase of data collection involved leveraging the ChatGPT text model from OpenAI, given its characteristics as a generative text model that is well-suited for generating code to be included in the dataset. Specifically, by using specific prompts, the model was asked to generate code snippets containing linguistic antipatterns. To ensure the inclusion of accurate data in the dataset, the prompts included specific descriptions related to the code to be generated. For instance, when requesting an example of the *method opposite comment* antipattern, a prompt could be: "Generate a Python code snippet containing a method and a docstring comment. The comment should include a word that is an antonym of any of the words in the method name." Consequently, the generated text was inserted into the dataset along with its corresponding label. Subsequently, manual efforts were made to create correct versions (without linguistic antipatterns) of the aforementioned code snippets. This was done to ensure a balanced representation of the different classes within the dataset. Another approach that was utilized involved adapting code snippets from the *Python Code Data*[3] dataset, which was uploaded on Kaggle by the user "AIL-UROPHILE". By adapting, it means that examples of code (which in the Kaggle dataset are devoid of linguistic antipatterns) were inserted into our dataset. Alongside these examples, manually modified versions were included to incorporate various categories of linguistic antipatterns, following the techniques mentioned in the previous approach. This dataset comprises approximately 150,000 functions, programs, and variable assignments in the Python programming language. The dataset is stored in a .txt file with a total size of 1.12MB. To ensure the quality of the data, an additional approach was necessary. This was particularly important because the first two techniques resulted in the dataset containing only snippets of code with atomic functions, typically of short length. This could potentially pose challenges as the data did not reflect the majority of code used in real-world projects. Such a limitation could lead to issues in predictions since they need to be made on real projects; to address this issue some functions were extracted from the code of *transformers* and *pytorch* and have been tweaked manually as stated in this paragraph before being put into the dataset. The final dataset is represented by four *.csv* file that consists of two columns: "label" and "code". The "label" column contains the label associated with the code snippet present in the "code" column. The following are the descriptions of the labels utilized in the dataset:

(1) **GET**: Get more than accessor
(2) **NIC**: Not implemented condition
(3) **MOP**: Method signature and comment are opposite
(4) **AOP**: Attribute signature and comment are opposite
(5) **CLR**: No linguistic antipattern detected

[3]https://www.kaggle.com/datasets/veeralakrishna/python-code-data

The four files are divided according to the type of linguistic antipattern, and each of them also contains examples of clean code, i.e., without the presence of linguistic antipatterns. Respectively, the files contain:

(1) 113 code snippets with *AOP* instances and 125 without
(2) 91 code snippets with *GET* and 85 without
(3) 186 code snippets with *MOP* and 195 without
(4) 104 code snippets with *NIC* and 114 without

In total, there are 1013 code snippets, with 494 containing linguistic antipatterns and 519 without.

2.2 Model Training

2.2.1 Environment. The model was trained in the Google Colaboratory environment, also known as Colab. This choice was made, among others, for one crucial reason, being the ability to use GPU hardware acceleration during the training process, to have higher computing power, and not to be limited by personal hardware. In addition, code in Colab is organized in notebooks, which combine formatted text and executable code blocks. This allows for the creation of interactive documents that integrate explanations, code, and results.

2.2.2 Data Preparation for Training. The library used for data processing in the model is Hugging Face's transformers library, widely used for training AI models based on transformers. Specifically, for data preparation in training, the code executed enables:

(1) Loading the dataset from various CSV files, divided by type of linguistic antipattern, wrapped then in specific Python objects to be used in the pipeline.
(2) Splitting the dataset into train, test, and validation sets. The details of the partitions are as follows: 66% of the data used for training, 24% for testing, and 9% for validation.
(3) For tokenization, the usage of the RobertaTokenizer class provided by the transformers module, the tokenizer used by Microsoft's CodeBERT.

The hyperparameters configuration used is as shown in Table 1.

Table 1: Hyperparameters and Values configuration used for training

Hyperparameter	Value
Batch Size	16
NAdam Learning Rate	2e-5
NAdam Epsilon	1e-08
Weight Decay	0.1
Number of Epochs	8

2.2.3 Evaluation Metrics. Both for validation and testing some specific metrics were collected to optimize the model training and overall performance, and also to gain a comprehensive understanding of the model's performance from different perspectives: namely, *train loss*, *accuracy*, *recall*, *precision*, *f1 score* and *matthew's correlation coefficient*. Each of them measures a different aspect of the model's behavior:

- **Train loss**: helps assess how well the model fits the training data and guides in optimizing the model's parameters. A decreasing train loss indicates that the model is learning and improving its performance on the training set.
- **Accuracy**: provides an overall measure of the model's correctness by calculating the percentage of correctly predicted labels. It allows an assessment of the model's ability to make correct predictions across different classes.
- **Recall**: measures the model's ability to correctly identify all positive instances, minimizing the risk of false negatives. A higher recall implies a lower chance of missing positive instances.
- **Precision**: calculates the proportion of true positive predictions among all positive predictions made by the model. A higher precision indicates a lower rate of false positives.
- **F1 Score**: this metric combines *precision* and *recall* in a single measure that reflects the balance between the two. A higher score implies better performance of the classification model with a good balance of precision and recall.
- **Matthew's Correlation Coefficient**: in this metric all possible binary classification outcomes (true positives, false positives, true negatives, false negatives), are taken into account, and a score ranging from -1 to +1 is produced, with a higher score representing a better predictive ability. In particular, +1 indicates a perfect prediction, 0 indicates a random prediction, and -1 indicates a complete discordance. It is particularly useful in cases of dataset balancing issues.

2.2.4 Training. The training phase for the model is actually a loop: the loop is repeated a total number of times equal to the *number of epochs* hyperparameter: for every iteration at the start the model is set to *training* mode; subsequently, the data is processed in batch in which every element consists of input tokens returned by the tokenizer, input masks and labels. These batches are then transferred to the GPU and then passed to the model. After this preparation phase, the model performs a forward pass on the input batch, generating predictions. The loss is calculated based on predictions and provided labels; a backward pass is then performed to compute the gradients and update the model's parameters using the specified optimizer.
At the end of the training stage, for each iteration also a validation phase is performed, in which, similarly to the training, each batch from the validation dataset is transferred to the GPU and then to the model which does a single forward pass, giving as output the predictions to calculate validation metrics.

2.2.5 Testing. After the training loop is completed, the model is tested against the test dataset, to calculate the same metrics as those calculated during the validation phase, with the only difference being that the model is "frozen", meaning that it is no longer learning from the input it receives, so some final metrics about how the model performs are obtained. The overall model's metrics obtained are shown in Table 2.

2.3 CLI Tool

The CLI tool is designed to take a series of command-line arguments, specifically accepting the following flags:

Table 2: Model's testing metrics

Metric
Train Loss
Accuracy
Precision
F1 Score
Recall
MCC

- **-i <path>** to indicate the tool a folder (where it will search for Python files) to analyze or a single file.
- **-o <path>** to indicate the tool a folder where it should store the results of the analysis.
- **-c** can be included if the user wants to include in the results the analyzed entities with no issues found by the model.
- **-s** can be used to run the cli in **server** mode, allowing it to analyze code passed through HTTP requests; this is explained in detail in Section 2.5.

Since one of the weaknesses of IDEAL is related to the complexity of installation, the tool is designed to simplify these operations as much as possible. It is, in a way, "self-installing": it attempts to automatically download the required libraries using the Python pip module and also fetches the text analysis model from the internet if it's not found on the disk. Furthermore, there is the option to save analysis results in JSON format files to keep track of them. In Figure 2, the result of an analysis performed on a test file is shown.

Figure 2: CLI Tool Analysis

2.4 CLI and Model Testing

For the testing of the CLI and the model, a project was devised containing Python functions that connected instances of linguistic antipatterns and non-antipatterns. The creation of these functions was achieved using the same method employed for creating the model's dataset, namely by utilizing ChatGPT, leveraging functions from the *Python Code Data* dataset which was uploaded on Kaggle, and incorporating entire files from *transformers* and *pytorch* projects. Since it was necessary to have an oracle for each tested function, the testing was preceded by an activity focused on establishing oracles for each function. Once the tool was initiated with the "-i" argument, the analysis results were obtained. From the analysis of the obtained results, it can be inferred that the tool is capable of making correct predictions in 79% of the analyzed cases, indicating that the objective of **NRF-1** was not achieved.

2.5 Visual Studio Code Extension

An extension for Visual Studio Code has been developed that is self-installing and self-configuring. Upon installation, a folder containing *greet* CLI, the Python interpreter, and the PIP module is downloaded. During the installation, the machine learning model is also downloaded, which will enable real-time predictions on openings and modifications of Python code documents. By launching the extension, an HTTP server will automatically start, which will be used to send the file to be analyzed in search of linguistic antipatterns. The prediction results will be displayed to users using the diagnostics provided by the *vscode package*. To avoid overloading the request server, given the high frequency of file modifications during code writing, a debounce technique has been implemented. This technique allows forwarding requests to the server after a 1-second delay following the user's typing completion. Moreover, predictions for previously analyzed files are saved to prevent sending a request upon file opening. The extension is available[4] on Visual Studio Marketplace as shown in Figure 3.

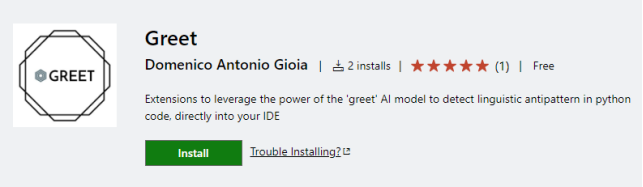

Figure 3: Visual Studio Marketplace

In Figure 4 the feedback provided by the extension while typing is shown.

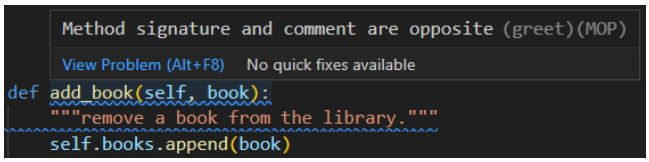

Figure 4: Visual Studio Code Extension

3 PRELIMINARY RESULTS AND FINDINGS

3.1 Model metrics

The final model returned the metrics presented in Table 3. The obtained metrics reveal the effectiveness of the model in detecting and classifying linguistic antipatterns, hence addressing **RQ1**; overall the final testing metrics have shown good results making the model reliable in detecting linguistic issues in Python source code.

3.2 System performance

Regarding **RQ2** and **NRF-4**, since defining a measure of scalability for the system seems to be complex, an analysis was conducted from the file parsing phase to the post-prediction phase, which are the two critical stages of the source code analysis process. The

[4]https://marketplace.visualstudio.com/items?itemName=DomenicoAntonioGioia.greet

Table 3: Final testing metrics for the model

Metric	Value
Train Loss	0.0781
Accuracy	0.916667
Precision	0.904762
F1 Score	0.904762
Recall	0.950000
MCC	0.9001

time taken by the parsing phase can be neglected compared to the total duration of the process because, on various inputs, it accounts for approximately 9/10% of the total time. Specifically, a file with one function and one attribute to analyze is resolved in 1 ms of parsing and 935 ms of prediction, while a file with 17 functions and 6 attributes is resolved in 8.23 ms of parsing and 9417.67 ms of prediction. It is observed that the critical phase for the performance of the system is the prediction calculation, which does not scale particularly well in relation to the number of entities to classify. For this reason, rules have been chosen to decide which entities to include or exclude from the analysis in order to minimize the number of erroneously identified entities to analyze (e.g., a function signature followed by a single-line comment related to the author of that function). The rules are as follows:

- Only entities with comments in docstring format are analyzed
- Only functions where the comment immediately follows the function signature are analyzed
- Only attributes where the comment immediately precedes the declaration of the identifier are analyzed

Since it is not possible to directly modify the performance of the model, this technique has been employed to achieve the objective of non-functional requirement **NRF-2**.

3.3 System Usability

Regarding **RQ-3**, it is challenging to provide a measure of usability. Due to the long time required to obtain prediction results, caused by the code analysis time and the overhead of HTTP, a *debounce* technique has been used. This technique sends the code for analysis 1 second after the user finishes typing in a Python file. The use of this technique significantly optimizes the management of diagnostics for two main reasons. Firstly, sending a large number of requests to the server would have led to its malfunction or degradation of performance. Secondly, it is unnecessary to send code for analysis with every code modification, as it would be redundant and inefficient. By employing the debounce technique, the system ensures that requests are sent at appropriate intervals, minimizing the impact on server performance and avoiding unnecessary analysis for each code change.

4 CONCLUSIONS AND FUTURE WORK

The prototype version *greet-cli-0.0.1-alpha.1* does not support the identification of the *Get more than accessor* antipattern due to time constraints, as additional parsing techniques that have not been implemented would be required. However, the *greet-0.0.1-alpha.1* model in the 5-label version is capable of classifying the mentioned antipattern. Consequently, a 4-label version specifically for the CLI has been created to accommodate this feature.

In conclusion, this work has achieved good overall results, particularly regarding the AI techniques employed. It demonstrates the effectiveness of this approach in identifying linguistic antipatterns. By further enhancing both the quantity and quality of the training data, the system's overall accuracy and performance can be significantly improved.

Possible future developments in this field involve the utilization of generative text models, such as GPT, which belong to the same family as transformer models. These models can be employed to suggest corrections for linguistic antipatterns directly in the source code, providing developers with real-time suggestions in parallel with their identification. This integration of generative text models can further enhance the tool's usability and offer developers an additional layer of support in writing cleaner and more maintainable code.

REFERENCES

[1] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. "Linguistic antipatterns: What they are and how developers perceive them". In: *Empirical Software Engineering* 21 (2016), pp. 104–158.