POLITECNICO DI TORINO

MASTER's Degree in CYBERSECURITY

MASTER's Degree Thesis

Using Explainability Methods to Uncover Shortcuts in Language Models

Supervisors

Prof. Luca VASSIO

Prof. Idilio DRAGO

Prof. Marco MELLIA

Candidate

Fabio MARMELLO

DECEMBER 2025

Using Explainability Methods to Uncover Shortcuts in Language Models

Fabio Marmello

Abstract

The increasing use of Machine Learning (including Large Language Models) in critical domains such as cybersecurity calls for stronger model reliability and trustworthiness. A key challenge in these applications is the potential for models to learn spurious correlations or shortcuts – patterns that yield high accuracy on training data but fail to capture genuine relationships, leading to poor performance in real-world deployments.

This thesis addresses the identification of shortcuts learned during end-to-end supervised training of language models applied to cybersecurity tasks. The recognition of these patterns is essential to validate the robustness of the model and ensure safe deployment.

To approach this challenge, we conduct a comparative analysis of explainability techniques across two paradigms: traditional feature-attribution methods (LIME and SHAP) that analyze input-output correlations, and concept-based approaches that probe the model's internal representations. For the latter, we examine Testing with Concept Activation Vectors (TCAVs), adapted from computer vision to allow manual definition and measurement of concept influence on predictions, and Sparse Autoencoders (SAEs), which automatically extract interpretable features from model activations to reveal internal reasoning patterns.

We build a systematic methodology for shortcut identification and conduct a case study with two RoBERTa-based classifiers trained on cybersecurity classification tasks.

Our study reveals that feature-based explainability techniques effectively identify keywords used by classifiers, which can also be exploited to alter predictions. In contrast, concept-based methods highlight semantically coherent tokens that contribute to class decisions, offering deeper insight into model reasoning.

Ai miei genitori, che mi hanno sempre sostenuto
A mia sorella, che mi ha spinto a dare il massimo
E al Me del futuro

Table of Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence.
BERT	Bidirectional Encoder Representations from Transformers.
CAV	Concept Activation Vector.
CNN	Convolutional Neural Network.
GPT	Generative Pre-trained Transformer.
LIME	Local Interpretable Model-agnostic Explanations.
ML	Machine Learning.
MLP	Multi-Layer Perceptron.
NLP	Natural Language Processing.
RoBERTa	Robustly Optimized BERT Approach.
SAE	Sparse Autoencoder.
SHAP	SHapley Additive exPlanations.
TCAV	Testing with Concept Activation Vectors.
XAI	Explainable Artificial Intelligence.

Chapter 1

Introduction

AI is a term that is increasingly being introduced across a wide range of domains, sometimes even in contexts where its utility is questionable. However, there are also environments in which it is driving major advancements.

Among these, the adoption of transformer-based models has drastically increased: the ability of these models (such as RoBERTa) to process and interpret textual data brought unprecedented performance[1] on Natural Language Processing tasks, such as sentiment analysis[2] or threat detection[3].

In Cybersecurity, language models can be particularly helpful: messages such as network traffic log, alerts, or any type of textual report can contain nearly invisible patterns that might reveal malicious activity. Misclassifying a packet, however, might not mean just accuracy loss, but also severe breaches, financial losses, or compromised infrastructures. Modern adversaries are increasingly exploiting the opacity of AI models, launching adversarial attacks to bypass protections. This makes transparency and interpretability essential for cybersecurity systems. Language models, however, are notorious for their lack of transparency, and the research community moved to solve this problem[4], further influenced by regulatory initiatives such as the European Union AI Act[5], which mandates transparency, accountability, and trustworthiness in AI systems.

That's why the field of Explainable Artificial Intelligence (XAI) has emerged, providing methods and frameworks to allow researchers and practitioners to interpret the model's decisions and inner workings. Employing these techniques not only enhances user confidence, but also enables the identification of biases, errors, vulnerabilities and patterns that may compromise the model's capabilities.

This thesis investigates four prominent XAI techniques, applied to RoBERTa-based models, to capture such correlations: Local Interpretable Model-agnostic Explanations[6] (LIME), Shapley Additive exPlanations[7] (SHAP), Testing with Concept Activation Vectors[8] (TCAV), and Sparse Autoencoders[9] (SAE). Specifically:

- LIME: this technique generates local explanations by approximating the model with a simpler interpretable mathematical surrogate in the locality of a given input. In this way, it allows us to identify which features most influence a

specific prediction.

- SHAP: this technique builds on cooperative game theory, assigning an importance score to the input features based on the Shapley values. It differentiates from LIME due to the use of a theoretically grounded framework to define the most relevant features.
- TCAV: this technique introduces for the first time the term "concept" in the XAI world. It enables the quantification of the importance of human-defined concepts within model predictions by studying the relations between the model representations and the outputs.
- SAE: this technique leverages sparse autoencoders to untangle model representations and uncover patterns within the hidden layers. By discovering those patterns, the models can highlight links between meaningful internal representations.

Our objective is to evaluate each technique's contribution in explaining the global behavior of RoBERTa-based models, defining whether the models found meaningful patterns or learned spurious correlations that degrade real-world performance or can be exploited to trick the model.

To compare these techniques, we performed the analysis on two case studies within a similar domain of text classification. The first case study focuses on code classification, where the objective is to categorize source code snippets according to predefined labels. Based on a high-accuracy model, this scenario serves as a controlled setting to verify the functionalities of the methods and to gain proficiency in handling RoBERTa-based models. The second case study addresses packet classification, specifically the identification of threats within HTTP packets. This scenario, characterized by high training accuracy but low testing accuracy, investigates the utility of the techniques in uncovering potential shortcuts and vulnerabilities, highlighting the challenges of applying language models to cybersecurity tasks where misclassifications may have severe consequences.

Chapter 2

Background and Related Work

Transformer-based architectures have achieved remarkable success across a wide range of natural language processing (NLP) tasks, but they also shown the ability of learning shortcuts and superficial patterns that yield high accuracy on training data but fail to generalize on unseen datasets[10]. In this chapter we introduce the field of Explainable AI, highlighting the distinction between post-hoc techniques and approaches that aim to uncover the actual computational mechanisms inside neural network. We provide a literature overview of the recent most used explainability techniques with particular attention to the applications on Language Models. Finally we present the theoretical groundings of the selected approaches for this work.

2.1 Explainable AI

Explainable Artificial Intelligence (XAI) is a topic that sees its origin in more than forty years ago[11], where a program that models an expert was explained using its embedded rules. However, in recent years, it has caught the attention of the research community as more complex models achieving astonishing results, such as Transformer-based Language Models, are born. The problem is that a trend is emerging: more accurate results mean less explainable results[12]. For this reason, researchers are trying to decipher the black-box nature of AI systems by designing tools for post-hoc explanations and creating more transparent models. The field of Explainable AI aims at developing tools to help developers improve AI algorithm by detecting data bias, discovering mistakes in the models, and remedying the weaknesses.

In the recent years, a new branch of XAI, known as mechanistic interpretability[13], is gaining wide attention. While traditional explainability methods often provide post-hoc approximations of a model's behavior, mechanistic interpretability aims to uncover the actual computational mechanisms inside neural networks. The objective is not merely to describe correlations between inputs and outputs, but to identify how internal components such as neurons, attention heads, or latent representations contribute to specific functions.

2.2 Transformer Paradigm and BERT

Transformers were first introduced by Vaswani[14] and revolutionized natural language processing by replacing recurrent and convolutional structures with attention mechanisms, enabling efficient modeling of long-range dependencies. The transformer

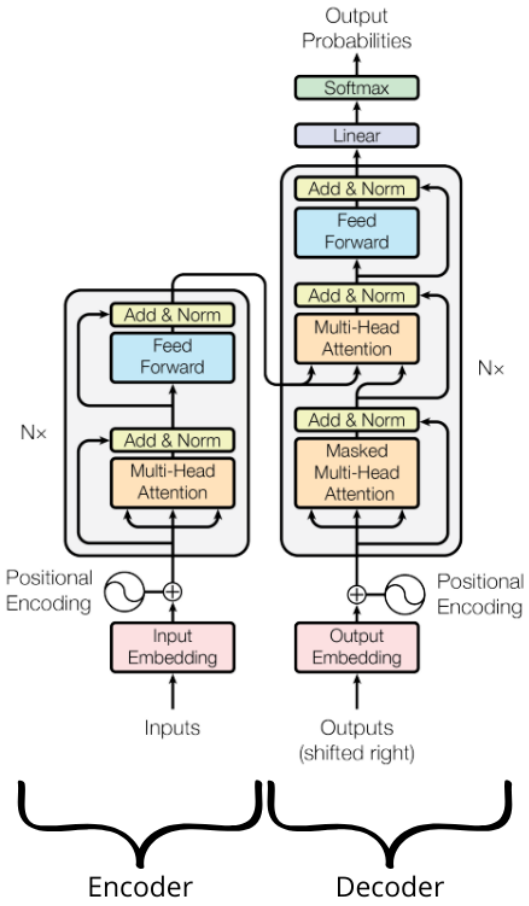

Figure 2.1: Transformer Model architecture[14]

models are based on an encoder-decoder architecture (Figure 2.1) and exploit the mechanism of self-attention to capture semantic relations between elements of a sequence. This was previously done through recurrent and convolutional structures, while the new architecture allows a more efficient representation. The objective of the encoder is to transform the input sequence into an internal representation capable of identifying a context. The decoder uses this representation, together with previous ones, to generate the output. The sequence information is maintained through the use of a positional encoding since the model is not sequential. The mechanism of self-attention allows the model to give different weights to each token, basically embedding an importance metric. This process is done in parallel through the use of multiple attention heads.

2.2.1 BERT

BERT, Bidirectional Encoder Representations from Transformers, was presented in 2018[1], extending the transformer paradigm by learning bidirectional representations of text. Previous models processed text on a left-to-right or right-to-left basis, while BERT simultaneously exploits both directions in all layers, achieving state-of-the-art results across multiple NLP benchmarks. BERT is a multi-layer bidirectional encoder-only transformer, which means that its structure is composed of a stack of encoder blocks, each integrating multi-head self-attention. The focus of this model is not generating text, but instead Masked Language Modeling (MLM) or Next Sentence Prediction (NSP). BERT training process is divided into two stages:

- Pre-training: the model is trained on a large amount of unlabeled text on multiple tasks, such as Masked Language Modeling and Next Sentence Prediction.
- Fine-tuning: the pre-trained parameters are adapted to a specific downstream task using labeled data. For the classification task, a single classification layer is added at the end of the model, with the representation of the special token `CLS` as input.

The fine-tuning process allows each model to quickly and easily adapt to many different situations, while sharply reducing training costs. By leveraging knowledge acquired during large-scale pre-training, fine-tuning enables the model to specialize in domain-specific tasks with relatively small datasets, ensuring high accuracy and robustness without the need for extensive computational resources. This approach not only accelerates deployment in practical applications, but also enhances flexibility, as the same pre-trained backbone can be efficiently repurposed across diverse contexts such as medicine, cybersecurity, or natural language understanding.

2.2.2 RoBERTa

RoBERTa (Robustly Optimized BERT Pre-training Approach) was introduced as a refinement of BERT[15]. The authors argued that the original BERT model was significantly under-trained, proposing several modifications to the pre-training methodology aimed at improving efficiency and generalization. Specifically, the changes introduced are:

- Removal of the Next Sentence Prediction (NSP) objective: BERT used this task to learn inter-sentence relationships, but an in-depth analysis revealed that NSP does not contribute much to downstream performance.
- Training on larger mini-batches and longer sequences: by increasing the batch size and the maximum sequence length, RoBERTa showed to be capable of learning better dependencies a improve stability during optimization. Furthermore, this modification allowed the model to learn better long-range dependencies in text.

- Use of significantly more training data: RoBERTa overall training dataset included several text dataset more than BERT, such as Common Crawl News[1] and OpenWebText[2].
- Dynamic masking of tokens during pre-training: BERT training process involved the use of fixed masking patterns, RoBERTa introduces dynamic patterns that change across epochs. This prevents the model from memorizing specific masked tokens, allowing a more robust contextual learning.

These adjustments led to substantial improvements across multiple benchmarks, demonstrating that BERT's original training regime was not fully exploiting the potential of the architecture. RoBERTa thus established itself as one of the most robust and widely adopted models in natural language processing, serving as the foundation for numerous subsequent works and applications. Its design emphasizes the importance of careful optimization and large-scale training, showing that performance gains can be achieved not only through architectural innovations but also through methodological refinements.

2.3 Tokenizer

In order for AI models to understand text, an important step is the tokenization. This step divides a text sequence into smaller units, called tokens, which are then mapped to numerical indices within a vocabulary. The method in which this task is performed has a direct impact on the model's capabilities to represent natural language, influencing the robustness of the model across different domains, such as different languages. It is important to understand that a single word may be split into multiple tokens or into a single one.

BERT employs the WordPiece tokenizer[16], which segments words into frequent subunits and utilizes prefixes (such as `__`) to indicate internal splits, with the addition of 3 special tokens `CLS`, `</s>` and `PAD` (respectively with ID 0, 1 and 2). RoBERTa instead introduced a Byte-Level BPE (Byte-Pair Encoding) tokenizer, which operates at the byte level, providing greater flexibility in handling special characters and multilingual text. It also introduced explicit encoding for whitespace through the use of`Ġ and Ċ`, enabling the model to distinguish between words with or without preceding spaces.

2.4 Related Work

In recent years, the need for Explainable AI has risen exponentially due to the growing adoption of machine learning models in critical environments, such as medicine and computer science[17]. To address this challenge, research has moved in two directions: developing new explanation techniques or designing inherently

[1] Available at: `https://commoncrawl.org/`
[2] Available at `https://github.com/jcpeterson/openwebtext`

interpretable models[18]. Nevertheless, the application of Language Models and Deep Neural Networks has gained a predominant role in critical scenarios such as image captioning or text classification. In this section we analyze the principal contributions in the literature related to black-box models and post-hoc techniques.

[19] gives a comprehensive overview of current available techniques, highlighting their characteristics and applicability. For our use case, we are interested only in black-box compatible techniques, in particular those applicable to textual data. Several techniques applicable to our use case have been presented in the literature [20], but two stand out: LIME and SHAP. These two methods are among the most widely used explainability techniques with broad applicability ([21], [22], [23]). LIME, for example, has proven effective in generating local explanations by highlighting and quantifying the contribution of individual input features to a model's prediction [24].

However, those techniques fall short when it comes to capturing the model's global behavior or accounting for complex feature interactions—factors that are often crucial in high-dimensional decision-making scenarios. For this reason, much of the literature has turned to the analysis of attention mechanisms in transformer-based architectures such as BERT: for instance, it was demonstrated that attention heads are capable of capturing a considerable amount of syntactic information[25]. Yet, while these techniques can reveal whether the model has learned structural relationships between tokens or not, they offer limited insight on the actual reasoning behind the outputs. This limitation is particularly noticeable in contexts where syntactic cues are sparse or less informative and has motivated researchers to explore more invasive techniques, focusing on internal activations and representations, particularly in domains like computer vision.

The most common technique in this field is TCAV [8] which has received a lot of considerable attention and development. For example, some studies have attempted to combine TCAV with saliency maps (a technique to highlight the most relevant pixels in images) to improve concept visualization [26]. TCAV saw application also in the field of cybersecurity: one approach proposed a way to use SHAP to define the concepts and then verify them by using TCAV[27], however this approach doesn't suit Language Models due to the input variability. A statistical adaptation of TCAV, designed to be compatible with Language Models, has been successfully implemented, though its focus remained on data augmentation efficiency rather than interpretability [28]. The limitation of TCAV-derived techniques is that concepts must be human-defined: for this reason, new approaches such as ACE[29] have been proposed to automatically detect them.

Following this trend, mechanistic interpretability gained popularity. This subfield saw interest from many of the big competitors in the AI race, from OpenAI[30] to Antropic[31], that demonstrated how sparse autoencoders (SAEs) are capable of disentangle neuron activations to help reach more understandable features.

Taken together, these contributions illustrate the evolution of XAI from local, post-hoc explanations toward deeper investigations of model internals. This trajectory frames the motivation of our work, which aims to analyze which technique is suited

the best for determining overall global behavior and finding shortcuts.

2.5 Techniques

As previously outlined, we chose to assess the utility of four techniques: LIME, SHAP, TCAV, and SAE. LIME and SHAP are designed to provide local explanations, with their main strength lying in the identification of the most relevant input features for a given prediction. They were chosen due to their ability to adapt to many models (they consider the model a black-box) and due to their widespread use among all the research community.

In contrast, TCAV and SAE operate on the internal representations of the model. TCAV was selected due to its ability to evaluate the model sensitivity to human-defined concepts, whose application is easy to realize from the structured nature of internet protocol. SAE, instead, was selected due to the impressive results shown from the main competitors in the AI scenario, along with its suitability in uncovering patterns within the activations, which is our goal.

2.5.1 LIME

LIME is a tool that was first presented in 2016[6]. It was introduced as a general-purpose framework capable of explaining any classifier by providing interpretable and faithful explanations. Its primary goal is to help practitioners build trust in individual predictions, and through repeated analysis, gain confidence in the model as a whole. As highlighted in their work [6], these are two different challenges and solving one, does not imply solving the other.

LIME is designed to generate local explanations for individual predictions by approximating the model's behavior in the vicinity of a specific input. This is achieved by creating a set of perturbed samples around the original input and analyzing the corresponding outputs. A simple and interpretable model—typically a linear classifier—is then trained on this neighborhood to mimic the behavior of the original model locally. This allows users to understand which input features most influenced the prediction, without needing to inspect the model's internal mechanisms.

One of the key challenges in applying LIME to text classification lies in finding a representation that is easily understandable by humans: Language Models transform text into tokens which may not always correspond directly to words, as words can be split into multiple tokens or include punctuation. To address this, the input is mapped to a binary vector indicating the presence or absence of words. The original representation of the instance to be explained is denoted as $x \in \mathbb{R}^d$, while the binary vector is represented as $x' \in \{0,1\}^{d'}$.

The explanation is formally defined as a model $g \in G$, where G is a class of potentially interpretable models. The domain of g is $\{0,1\}^{d'}$ since g operates over the presence or absence of interpretable components. The explanation provided by

LIME is obtained by solving:

$$\xi(x) = \arg\min_{g \in G} \mathcal{L}(f, g, \pi_x) + \Omega(g) \tag{2.1}$$

This formula consists of two components: the unfaithful parameter $\mathcal{L}(f, g, \pi_x)$ and the interpretability constraint $\Omega(g)$. The objective of the technique is to minimize $\mathcal{L}(f, g, \pi_x)$ while keeping $\Omega(g)$ as low as possible to ensure both local fidelity and human interpretability. The unfaithful parameter is defined as:

$$\mathcal{L}(f, g, \pi_x) = \sum_{z \in \mathcal{Z}} \pi_x(z) \cdot (f(z) - g(z))^2 \tag{2.2}$$

where $\mathcal{Z}$ is the dataset of perturbed samples and π_x is a locality kernel. The perturbed samples are generated around x' by drawing uniformly at random nonzero elements of x' and generating $z' \in \{0, 1\}^{d'}$, which is then reconstructed into the original representation $z \in \mathbb{R}^d$. This representation is then used as $f(z)$ to obtain a label for the explanation model. The locality kernel has the objective of measuring the distance between any sample $z \in \mathcal{Z}$ and the original instance x to properly define a locality around x. It is computed as:

$$\pi_x(z) = exp\left(\frac{-D(x, z)^2}{\sigma^2}\right) \tag{2.3}$$

where D is a distance metric, that, in the case of a text classifier, is the cosine distance. The exponential ensures that samples closer to the original instance receive higher weights, thereby focusing the explanation on the local behavior of the model. It is important to note that, due to the randomness of this process, the perturbed samples are not generated only in the proximity of x (with high weight from π_x), but also far away (with low weight from π_x).

The interpretability constraint parameter (2.4) has the objective of ensuring that the explanation is interpretable: in the case of a text classifier this is performed by setting a limit to the number of words that a user can sustain, by following

$$\Omega(g) = \infty \cdot \mathbb{1}[||w_g||_0 > K] \tag{2.4}$$

where w_g is the weight vector of the linear model g, and K is a user-defined threshold on the maximum number of non-zero coefficients. This constraint ensures that the explanation uses at most K features, promoting sparse and human-readable explanations. If the number of active features exceeds K, the penalty becomes infinite, effectively making the model unusable.

In summary, LIME offers a powerful and flexible approach for interpreting black-box models, providing locally faithful and interpretable explanations. This makes it one of the most used tool for increasing transparency and trust in AI systems.

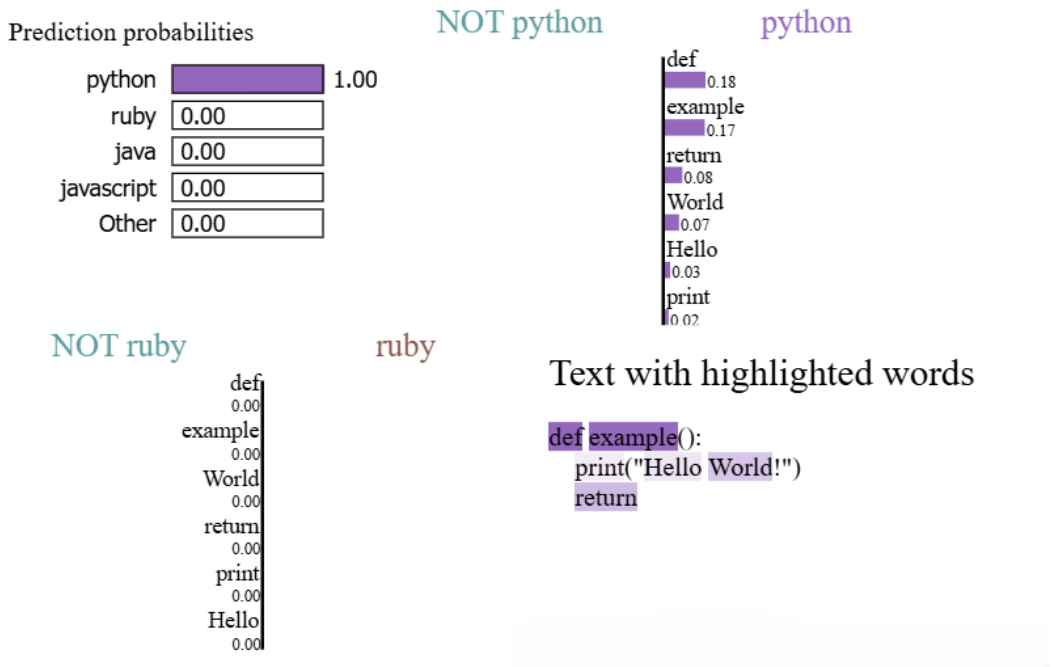

Figure 2.2: LIME interface example

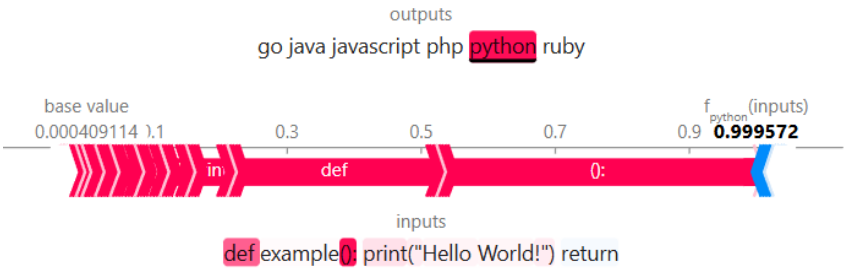

Figure 2.3: SHAP interface example

2.5.2 SHAP

SHapley Addictive exPlanations was first presented in 2017 by [7] as a framework to unite many explainability techniques by defining a class of additive feature attribution methods. This class is identified by "any method that has an explanation model that is a linear function of binary values". This means that the explanation follows the formula. Specifically, any method whose explanation model follows the formula:

$$(z') = \phi_0 + \sum_{i=1}^{M} \phi_i z'_i \tag{2.5}$$

Here, $z'_i \in \{0,1\}^M$ represents a simplified binary input vector of M features. Any method whose explanation model conforms to this function assigns an attribution value ϕ_i to each feature, and the sum of these attributions provides an approximation of the output of the original model $f(x)$. To ensure a single unique solution within this class, SHAP introduces three properties:

- Local accuracy: for a given input instance, the explanation model is required

to match the output of the original model when applied to the corresponding simplified representation.

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^{M} \phi_i x'_i \tag{2.6}$$

- Missingness: features that are not present in the original input provide no influence on the output.

$$x'_i = 0 \Rightarrow \phi_i = 0 \tag{2.7}$$

- Consistency: if the original model changes in such a way that some simplified input's contribution increases or stays the same regardless of any other inputs, that input's attribution should not decrease.

$$f'_x(z') - f'_x(z' \setminus i) \geq f_x(z') - f_x(z' \setminus i) \tag{2.8}$$

for all inputs $z' \in \{0,1\}^M$, then $\phi_i(f', x) \geq \phi_i(f, x)$.

Combining the formula 2.5 and the properties (2.6, 2.7, 2.8) leads to the theorem on which SHAP is based: only one possible explanation model g follows

$$\phi_i(f, x) = \sum_{z' \subseteq x'} \frac{|z'|!(M - |z'| - 1)!}{M!} \left[f_x(z') - f_x(z' \setminus i) \right] \tag{2.9}$$

where $|z'|$ is the number of non-zero entries in z', and $z' \subseteq x'$ represents all z' vectors where the non-zero entries are a subset of the non-zero entries in x'.

The solutions to equation 2.9 are known as Shapley values, named in this framework SHAP values, where $f_x(z') = f(h_x(z')) = \mathbb{E}[f(z) \mid z_S]$ and S represent the expected model output influenced by the subset of nonzero features S in z'. The SHAP values provide a measure of the unique additive feature contribution. Due to the computational cost of these values, the framework introduced several algorithmic implementations tailored for specific cases. For instance, Kernel SHAP is capable of adapting to any model but is computationally expensive, while Tree SHAP offers efficient value computation but only for tree-based models. Our use case saw the application of the Partition Explainer: one of the most recent additions to the SHAP library [3]. This explainer is model agnostic, like LIME, but to improve the computational complexity uses an approximation of the SHAP values: the Owen values.

The Owen values originate from cooperative game theory [32]: the objective is to take into account that certain players (in our case, features) are more likely to work together than others. These "coalitions" are defined by SHAP via hierarchical partitioning of features, and then the Owen values are computed along the tree recursively to determine each feature's contribution within its coalition. While the SHAP library does not explicitly provide the formula used for Owen value

[3] Available at `https://github.com/shap/shap`

computation, the relationship between Shapley and Owen values is theoretically justified, as demonstrated in [33]. The selection of the Partition Explainer is performed automatically by the library based on the model type and computational efficiency: this method has a computational complexity of $O(n^2)$ with respect to $O(2^n)$ of the Kernel Explainer[4], where n is the number of features.

2.5.3 TCAV

Testing with Concept Activation Vectors (TCAV) is a technique introduced in 2017 [8] for Computer Vision models which are typically trained to assign a text label to an image. The objective of TCAV is to quantify how much human-defined concepts influence the model's classification.

To apply TCAV, a dataset of concept-related images is created and passed through the model. The activations of a specific bottleneck layer are extracted for each image and used to compute the Concept Activation Vectors (CAVs). These vectors are obtained by training a linear classifier to differentiate between activations from concept-related images and those from random images. The resulting CAV corresponds to the weights of the linear classifier and is formally defined as the normal vector to the hyperplane separating examples with a concept and without. It is important to notice that the concept is human-defined and it is not limited to the model's labels, training data or existing features.

Once the CAV is computed, it is used to calculate the conceptual sensitivity, by performing the directional derivative of the activations extracted from a general image with respect to the CAV:

$$S_{C,k,l}(x) = \lim_{\varepsilon \to 0} \frac{h_{l,k}(f_l(x) + \varepsilon v_C^l) - h_{l,k}(f_l(x))}{\varepsilon} = \nabla h_{l,k}(f_l(x)) \cdot v_C^l \tag{2.10}$$

This derivative takes inspiration from the saliency maps[34]: a more classical explainability method that uses gradients of logit values with respect to individual input features.

Testing with Concept Activation Vectors expands the conceptual sensitivity to an entire class by analyzing multiple inputs of a specific label:

$$\text{TCAV}_{Q_{C,k,l}} = \frac{|\{x \in X_k : SC_{k,l}(x) > 0\}|}{|X_k|} \tag{2.11}$$

This allows to quantify the average conceptual sensitivity of a class k to a specific concept C (l refers to the layer from which the activations where extracted). However, it is possible that the training of the linear classifier produces a meaningless CAV: to ensure reliability, the test should be repeated multiple times and lead to consistent TCAV scores for meaningful concepts. It was also shown that complex concepts tend to have higher scores within deeper layers of the model, while simpler ones gradually

[4]This is the explainer used by Kernel SHAP, the only other solution for our model

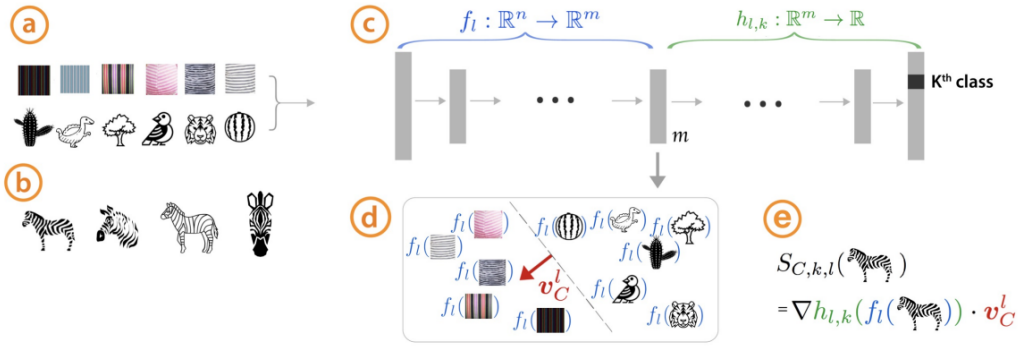

Figure 2.4: TCAV functionality example[8]: a series of images representing the human-defined concept "stripes" and a series of random images (a), are passed to the model (c). The activations of layer m of the network are used to calculate the CAV (d). Then, for the class of interest (as highlighted in b, zebras), the directional derivative is used to compute the conceptual sensitivity (e)

lose importance, highlighting the hierarchical nature of learned representations. TCAV provides a principled way to bridge human-defined concepts and model representation, enabling quantification of how abstract notions influence prediction. It saw many application in the computer vision scenario, while not many applied this technique to Language Models: The varying input length adds a big challenge on its adaptation.

2.5.4 SAE

Sparse AutoEncoders are a novelty in the interpretability scenario. Initially introduced by [9] to address the superposition problem, SAEs offer a mechanism to disentangle neuron activations: many models, not only Language Models, have polysemantic neurons that activate in similar way for tokens that are semantically unrelated, meaning that the neuron is key for multiple tasks. Autoencoders are neural models designed to approximate the input through an internal representation. While the latent dimension is typically smaller than the input to enforce compression, increasing it beyond the input size—combined with a sparsity-oriented loss function—enables the disentanglement of complex activations and the discovery of hidden patterns.

What SAEs learn to represent are dictionary of words that can be manually or automatically linked to a human-defined concept. Usually, to automate this process, the gathered data is fed to an LLM for it to automatically generate the concept that summarizes the results.

Our work was shaped following Antropic's latest design [31]: the SAE is composed by a single-layer encoder, linked to a single-layer decoder through the ReLU activation function.

This architecture is standard across SAEs applications. Antropic's key innovation lies in the loss function: this is the core of the SAE and its objective is to incentivize sparsity. The sparsity allows for disentangling activations by stimulating each input to activate a small number of SAE's neurons. The Antropic's sparsity-regularized

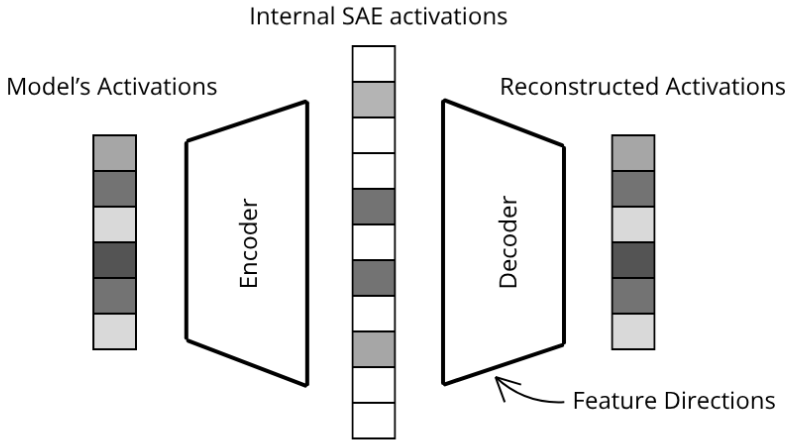

Figure 2.5: This image represent the SAE structure: you give in input the model's activations for a token and it tries to reconstruct them using few neurons as possible.

loss function is defined as follows[35]:

$$\mathcal{L} = \frac{1}{|X|} \sum_{\mathbf{x} \in X} \|\mathbf{x} - \hat{\mathbf{x}}\|_2^2 + \lambda \sum_i \left\| f_i(\mathbf{x}) \right\| W_{\cdot,i}^{dec} \Big\|_2 \tag{2.12}$$

where x are the model's activations normalized so their average squared L2 norm is the residual stream dimension D, λ is the sparsity coefficient, W^{dec} are the decoder weights and f_i is the output of the encoder. The normalization follows this formula:

$$MSN = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{d} x_{ij}^2 \tag{2.13}$$

$$\mathrm{s} = \sqrt{\frac{D}{MSN}}; \quad \tilde{x}_{ij} = x_{ij} \cdot \mathrm{s} \tag{2.14}$$

The encoder's and decoder's outputs are defined as:

$$f_i(x) = \mathrm{ReLU}\left(\mathbf{W}_{i,:}^{enc} \cdot \mathbf{x} + \mathbf{b}_i^{enc} \right) \tag{2.15}$$

$$\hat{\mathbf{x}} = \mathbf{b}^{dec} + \sum_{i=1}^{F} f_i(\mathbf{x}) \mathbf{W}_{,i}^{dec} \tag{2.16}$$

where D is the model's activations dimensions, F is the dimensionality of the SAE's hidden layer, $W^{enc} \in \mathbb{R}^{F,D}$, $W^{dec} \in \mathbb{R}^{D,F}$, $\mathbf{b}^{enc} \in \mathbb{R}^F$ and $\mathbf{b}^{dec} \in \mathbb{R}^D$. From this formulation, the feature activations are computed as $f_i(x) \cdot \left\| \mathbf{W}_{:,i}^{dec} \right\|_2$ and the corresponding feature directions will be $\frac{\mathbf{W}_{\cdot,i}^{dec}}{\left\| \mathbf{W}_{\cdot,i}^{dec} \right\|_2}$, directly extracted from the decoder (as highlighted in figure 2.5). To clarify the feature activations are the SAE internal activations of a specific neuron, relative to a specific token, while the feature direction identifies on which tokens the neuron is more likely to activate.

The token activations for the training of the autoencoder are collected by feeding

complete samples through the model, allowing each representation to maintain contextual information. When passing those values to the SAE instead, the order of the tokens is randomized to avoid biases. After training, by analyzing which tokens activate the same feature, one can construct token dictionaries that, if semantically coherent, can be mapped to a concept.

sparse AutoEncoders provide a scalable and unsupervised framework for uncovering latent structure in neural representations, which could provide important contribution in understanding a model.

Chapter 3

Methodology

The methodology adopted in this thesis is designed to evaluate the ability of state of the art explainability techniques to uncover meaningful patterns between the input of a Language Model and its predicted output. Our goal is to investigate whether these techniques are capable of providing reliable insights into the decision-making process of the model, using our expert eye to evaluate the outcome. To this end, the approach is divided into three pipelines[1] (Figure 3.1) that aim at providing us the most meaningful result possible:

- LIME and SHAP pipeline: being these two techniques really similar in both the explanation type and output style, it is easy to run them in parallel. We will compare their capability of explaining both single instances and global model behavior.
- TCAV pipeline: this pipeline aims at evaluating the correctness of the technique, the relevance of every defined concept, and the visualization of the results to proceed with a manual verification.
- SAE pipeline: this pipeline allows the training of multiple SAEs, the evaluation of their metrics and feature filtering to ease the manual validation phase.

3.1 LIME and SHAP Pipeline

As previously mentioned2.5.1, LIME and SHAP try to establish a direct correlation between input tokens (the features of a Language Model) and the predicted output class. In this thesis, we chose two techniques based on two main factors:

- State of the art: both techniques are widely adopted and continuously supported by the research community, ensuring methodological robustness
- Readiness: their ease of use makes them suitable as baselines and ground truth references for the subsequent comparison with concept-based methods.

[1] All the code related to this analysis is available at `https://github.com/ChehSuccedeh/tesi`

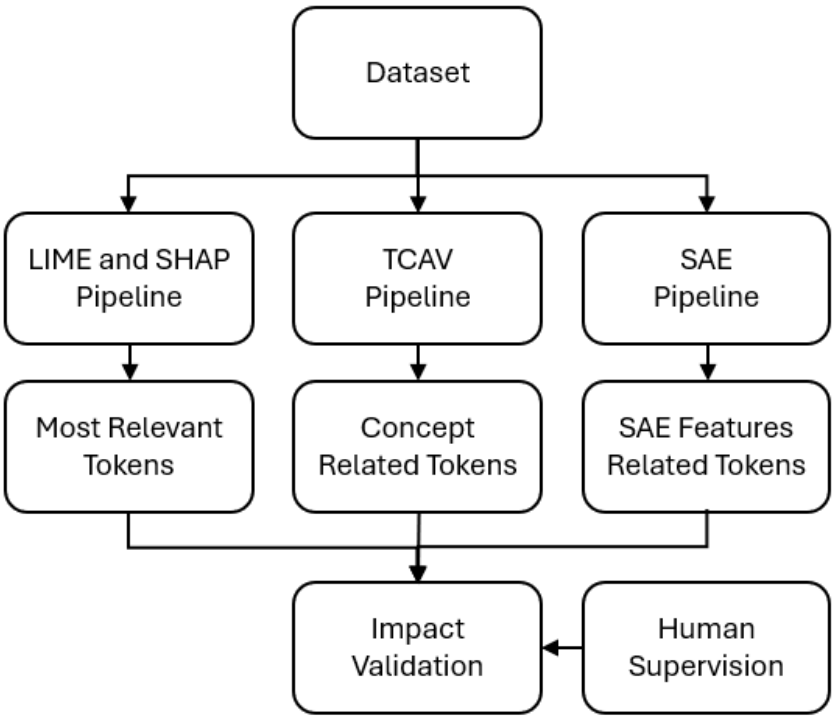

Figure 3.1: Complete Methodology Pipeline

Both these techniques are model-agnostic, meaning that the internal architecture and parameters of the model are not important, and thus making them compatible with a wide range of architectures. Moreover, despite being developed many years ago, they remain central in the interpretability landscape.

We employed these methods in their intended scope of local explanations, analyzing the highlighted features to detect differences and identify potential biases. Subsequently, we extended the evaluation to a large set of samples to assess their ability of capturing the global model behavior. The resulting feature importance distributions serve as a baseline for evaluating the added value of newer, more invasive techniques.

LIME and SHAP are both available through open source Python libraries[2]. This facilitated the implementation, however LIME required the implementation of a model wrapper class that, once initialized with model and tokenizer, provides the technique with a method capable of processing a string and returning the model prediction probabilities for each class (the prediction probabilities are computed through the SoftMax of model logits).

We provide each technique with some text samples and use the explanation to craft some adversarial inputs to verify the effectiveness of the explanation. These examples however are only valid in the locality of the text samples analyzed. To generalize, we provide multiple samples of each class, and collect the average importance of each identified word. We then compare the results of the local explanation to the latter average importance to verify the consistency: if the global explanations do not match

[2]Available at: `https://github.com/marcotcr/lime` and `https://github.com/shap/shap`

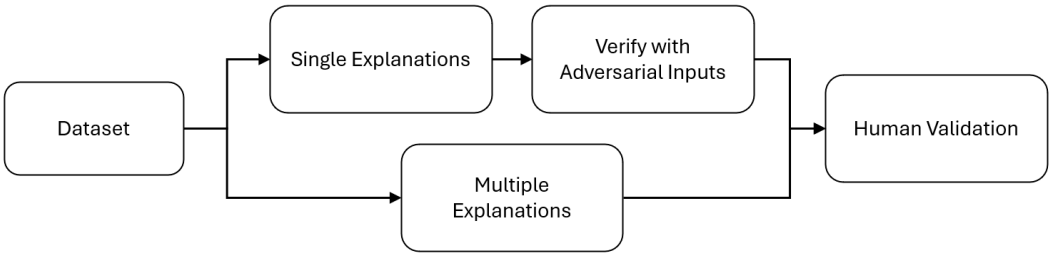

Figure 3.2: Visual representation of the LIME and SHAP pipeline

the local ones, these techniques are not useful in finding overall shortcuts.

3.2 TCAV Pipeline

As opposed to LIME and SHAP, TCAV exploits internal model representations to identify whether human-defined concepts are encoded within the model. To correctly obtain the representations we started by adapting a previous implementation[3][28] which uses PyTorch hooks to grab the token activations from a predefined layer. Another hook is used to collect the gradients to compute the final TCAV score. Their analysis however limited the training of the CAVs to the single `CLS` token, which significantly impacts the utility in finding token relations. For this reason we decided to consider all tokens fed through the model. This raised a challenge: the linear classifier that is used to compute the CAV is not able to deal with varying length inputs. We opted to design three strategies to overcome this:

- Truncating every sample to the maximum length of the concept samples: this would allow to gather as much information as possible from the concepts, but creates problems when calculating the derivative of samples. Those samples have to be truncated to that same length, potentially discarding many tokens that may be related to the concept, effectively lowering the TCAV score. We identify this strategy with "Truncated".

- Padding to the maximum length supported by the model: by casting to the maximum length, we would avoid the problem related to the truncation of the tokens, however, the information related to the CAV would be concentrated on the first tokens[4], introducing a positional bias. We identify this strategy with "Padded".

- Averaging model activations across concept samples: this solution would eliminate all the problems related to the positioning of the tokens, but it is possible that the average could lead to information loss, meaning that the CAV could not be correctly represented. We identify this strategy with "Average".

[3]Available at: `https://github.com/IsarNejad/TCAV-for-Text-Classifiers`
[4]Being the concepts usually short, a lot of space would be taken by meaningless padding

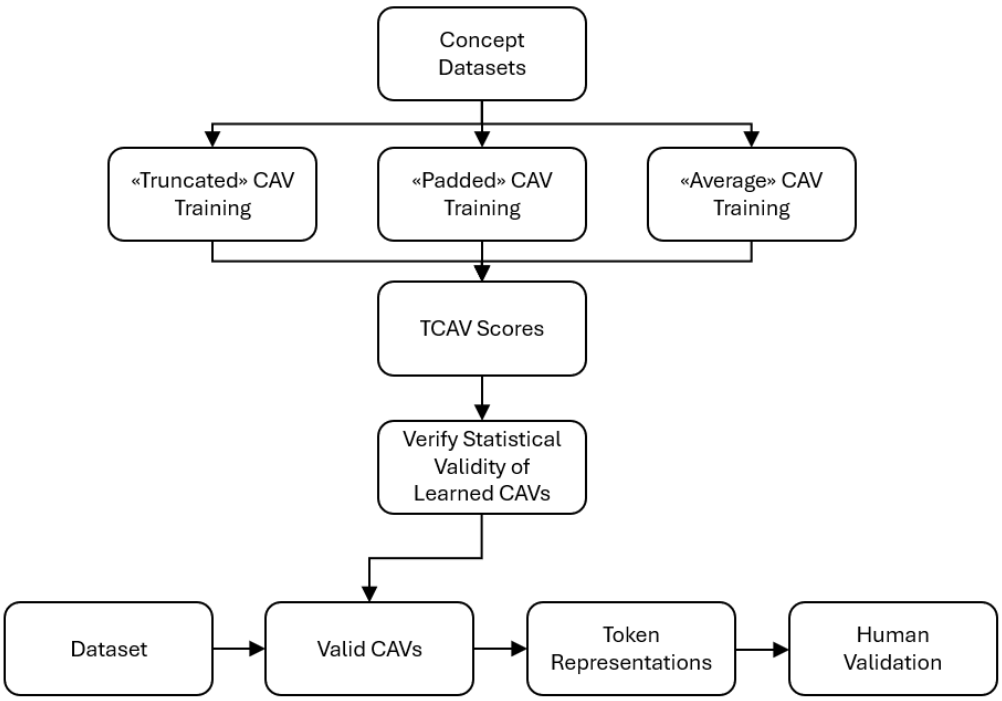

Figure 3.3: Visual representation of TCAV pipeline

These implementation choices reflect the inherent tension between preserving semantic fidelity and ensuring computational compatibility with TCAV. While each strategy introduces its own limitations, their formulation was necessary to adapt the method to the structural constraints of language models. It is important to consider for each concept we trained a different CAV for each layer of the model to gather a complete overview. The concept we defined can be characterized in two types: general concepts and class-related concepts. The first category aims at defining concept that can be found within any sample, despite the class they belong to. The second category instead aims at defining class-specific concepts to define whether the technique is capable of distinguishing them and if the model has learned more class-specific patterns. To analyze the contribution of TCAV, we tested many concepts in parallel, repeating each test 30 times. For each of them, we compared the three strategies by examining their average TCAV score across the layers in search of patterns, such as gradual score improvements, which may suggest that the concept is well captured by the CAV and relevant for the classification, or conversely reveal that the CAV is merely incidental and does not reflect a meaningful representation. This analysis also considered the variability of the TCAV scores: more stable values suggest a reliable CAV, whereas high variance may reveal excessive randomness and undermine its trustworthiness. After performing the visual analysis, we validated our observations through statistical t-tests (see Appendix A.1) to assess whether a defined concept is significantly distinguishable from a random baseline. To ensure robustness, we applied a stringent filter on the p-values (< 0.0001), following the language model implementation of TCAV[28]. If a CAV passes this test, it is subsequently employed

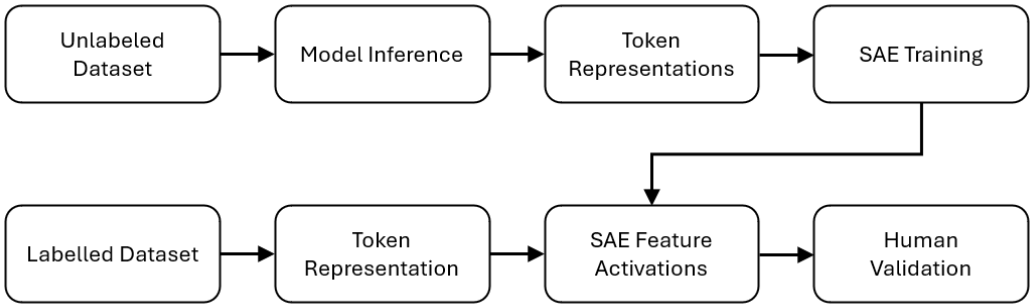

Figure 3.4: Visual representation of the SAE pipeline

to compute the conceptual sensitivity of each token and visualized in a format similar to LIME and SHAP. This allows the verification of the correct concept representation: tokens belonging to the concept are expected to exhibit higher conceptual sensitivity compared to those that do not.

3.3 SAE Pipeline

SAE is another technique that employs internal representation: it tries to untangle them to detect relationships between tokens that may have been learned by the model. There is no available library that implements this technique for BERT-based models. We proceeded with the definition of a class that allows the creation of any sparse autoencoder following the structure defined in Section 2.5.4, by specifying the input dimension and the hidden dimension. In our case the input dimension is 768: this is the number of neurons, and so the number of activations per tokens, of a RoBERTa layer. For completeness purposes and we trained SAEs of sizes 768, 1536, 3072, 6144, 12288 and finally 24576: going higher is possible but the training time gets linearly higher with the size, limiting our capabilities. Each SAE is trained on a single layer of the original model, meaning that the token representations are extracted from a specific layer and fed through the autoencoder in randomized fashion. Following recent studies[36], we trained SAEs on the higher layers of the models, aiming to capture a meaningful progression in the learned representations. Specifically, for the code classification task, based on the RoBERTa-small architecture with 6 layers, we focused on layers 3 to 6. For the packet inspection task, instead, which relies on the RoBERTa-base architecture with 12 layers, we trained on layers 6 to 12, leading to a total of 55 trained SAE. Specifically, the training of these SAEs was performed over 200 epochs using the ADAM optimizer with these parameters (suggested by Antropic's implementation[35]): sparsity coefficient $\lambda = 5$, learning rate $lr = 5 \cdot 10^{-5}$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Once all the SAE have been trained we analyzed them by first comparing them to define whether a better model was present and then we proceeded to filter SAE features to detect meaningful one. The comparison across configuration was based on two primary criteria:

- Sparsity, defined as the number of SAE features that are simultaneously active

on the same token.

- Dead features, which are features whose activations remain consistently below a predefined threshold.

As an additional step, we performed a manual validation of the SAE feature with the highest average activation to get an overview of the interpretability of the results across layers. We then moved to identifying within a SAE the most relevant features, to assess its capability to find meaningful relations that could be exploited by the model to perform its classification. We performed three comparisons:

- Clustering
- Class level TF-IDF filtering
- Token level TF-IDF filtering

The clustering was employed to identify groups of similar concepts and applied on the SAE feature directions (see Section 2.5.4). First, we reduced the dimensionality using UMAP to 10 dimensions, using as parameters 15 neighbors, minimum distance 0.1 and cosine similarity as metric. Then, on top of this representation, we applied HDBSCAN with a minimum cluster size of 10 to automatically detect clusters. The resulting clusters are then manually inspected to verify whether they correspond to similar tokens or classes.

The class-level TF-IDF (see Appendix A.2) was defined to highlight concepts (SAE features) relevant to specific classes. For each feature, the Term Frequency (TF) was defined as the frequency of its activation within a sample, while the Document Frequency (DF) was defined as the number of samples in which the feature was active for at least one token. This allows to down-weight concepts that are always active across samples and highlight rarer concepts that may be more specific for individual classes.

The token level variant of the TF-IDF has the Term Frequency defined as the activation for each concept across all tokens, aggregating all samples together. The Document Frequency is defined as the number of tokens in which a feature is active. For duplicated tokens the average TF-IDF score is computed to avoid bias. This approach allows to filter concepts based on token-level specificity, reducing the influence of ubiquitous or uninformative tokens such as whitespace.

To reference a specific feature, we introduced a compact acronym that enables quick identification: use case / layer of reference / feature identifier / SAE hidden dimension. For instance, the notation P/8/145/768 denotes feature 145 of the SAE model with hidden dimension 768, trained on layer 8 within the packet inspection scenario (where P indicates packet inspection, while C will be used for code classification).

Finally, for each identified important concept, we validated its influence on the model by removing the tokens that most strongly activated it and observing the resulting variation in performance. As a baseline, we compared these results with the

removal of randomly selected tokens. This procedure allowed us to assess whether the identified concepts had a measurable and interpretable impact on the model's behavior.

3.4 Visualization Tool

To assist TCAV and SAE, and achieve a more comparable visualization with the feature-based, we designed a web application using React and TypeScript. It is composed by two pages, each related to one technique, and allows the user to load a JSON file to visualize for each sample the token's impact. For the TCAV it allows to easily cycle through each layer, concept and sample, highlighting at the top the top 5 tokens with highest and lowest conceptual sensitivity within all samples.

To capture the relationship between concepts and model activations, we employ a JSON structure specifically designed for token sensitivities (Table 3.1). Each entry associates a concept (e.g., random) with the corresponding layer identifier, which can be expressed either as a string or a numerical index. The entry also records the "sample_index", an integer that uniquely identifies the analyzed sample. Finally, the "token_sensitivities" field contains an array of pairs, where each pair links a token to its sensitivity value. This representation provides a compact yet expressive format for quantifying how individual tokens contribute to the activation of a given concept across different layers, thereby facilitating interpretability analyses such as TCAV.

Field	Description
concept	String, the concept associated (e.g., "random")
layer	String/number, identifier of the layer (e.g., "1")
sample_index	Integer, index of the analyzed sample (e.g., 42)
token_sensitivities	Array of pairs [token, value]

Table 3.1: Example of JSON structure for token sensitivities

The page dedicated to the SAE analysis allows to load a single SAE's data, due to size limitation (files above 100 MB tend to crash the client's browser. In order to systematically represent the activations associated with code samples, we adopted a hierarchical JSON structure.

Field	Description
sample_index	Integer, index of the analyzed sample (e.g., 42)
tokens	Array of token objects (see Table 3.3)
class	String, category (e.g., "python")

Table 3.2: Structure of the JSON entry for code activations

At the highest level (Table 3.2), each entry corresponds to a single analyzed sample, identified by an integer index and annotated with its categorical class (e.g., python). The entry also contains an array of token objects, which encapsulate the finer-grained information.

Field	Description
token_idx	Integer, position of the token in the sequence (e.g.,“42”)
token_str	String, the token itself (e.g., “CLS”, “def”, “cursor”)
activations	Array of pairs [feature_id, activation_value]

Table 3.3: Structure of each token object

The internal organization of each token (Table 3.3) specifies its sequential position, textual representation, and the corresponding activation values. Activations are stored as pairs of feature identifiers and numerical values, thereby enabling a direct mapping between individual tokens and the features they trigger. This design ensures both readability and extensibility, allowing the data to be efficiently parsed, analyzed, and visualized in subsequent stages of the study.

Chapter 4

Use cases and Datasets

In this thesis we focus on two use cases that employ fine-tuned RoBERTa model with a classification head:

- Code Classification: this model serves as reference. It corresponds to the CodeBERTa-small-v1 model, originally trained for next token prediction across multiple programming languages, and extended with a classification layer to assign language labels. The possible labels are: `Go`, `Java`, `JavaScript`, `PHP`, `Python`, and `Ruby`.
- Packet Inspection: this is the main model analyzed in our work. It is a standard CodeBERTa-base architecture, extended with a classification layer to assign a label to HTTP packets. The label is referred to the attack type within the packet, choosing between: `Adware`, `Backdoor`, `Botnet`, `CGI`, `Code-execution`, `DDos`, `Dir-Traversal`, `Dos`, `Info-Disclosure`, `Injection`, `Other`, `Overflow`, `Ransonware`, `Remote-file-Inclusion`, `Scanner`, `Spyware`, `Trojan`, `Virus`, `Webshell`, `Worm`, `XSS`

The first model[1] achieves very high performance, with testing accuracy and F1 score above 0.999. It is the "small" version of RoBERTa, comprising only six transformer layers, which are sufficient for the relatively simple task of code classification. The second model, instead, is a RoBERTa-base model with twelve layers. It achieves a poor accuracy and very limited information is available beyond the set of outputs, since it was trained on a confidential dataset.

4.1 Datasets

Datasets play a crucial role not only in the training of the model but also in the application of explainability techniques. For the code classification use case, we employed the same testing dataset used to evaluate the model performance: CodeSearch-Net[2]. This dataset was constructed by crawling GitHub repositories and

[1] Available through the HuggingFace Python library `transformers`: `https://huggingface.co/huggingface/CodeBERTa-language-id`

[2] Available at: `https://huggingface.co/datasets/code-search-net/code_search_net`

Class	Elements
Go	14,291
Java	26,909
JavaScript	6,483
PHP	28,391
Python	22,176
Ruby	2,279
Total	100,529

Table 4.1: Distribution of samples within Code-Search-Net dataset

Class	Elements
Backdoor	586
Botnet	978
CGI	192
Code-execution	166,359
Dir-traversal	90,636
DoS	610
Info-Disclosure	113,851
Injection	143,112
Overflow	718
Remote-File-Inclusion	5,515
Scanner	14,654
Trojan	72
Webshell	10,684
Worm	290
XSS	53,261
Total	601,518

Table 4.2: Distribution of the private training dataset

gathering code functions across the six programming languages supported by the model. The distribution of the dataset is shown in Table 4.1.

For the packet inspection use case, instead, the training was performed on a private dataset with samples composed by an IP address followed by a standard HTTP request, for example:

```
203.0.113.10
GET /cgi-bin/test.cgi?input=foo&&cat%20/etc/shadow HTTP/1.1
Host: submit2.com
User-Agent: Mozilla/5.0
Accept: text/html
Accept-Encoding: gzip, deflate, br
Accept-Charset: UTF-8
Accept-Language: en-US
Connection: keep-alive
```

In Table 4.2 are shown the dataset distribution which highlight a clear imbalance

Class	Elements
Dir-Traversal	31
Adware	30
Injection	30
Code-execution	30
Info-Disclosure	26
XSS	20
Remote-file-Inclusion	20
Overflow	17
Spyware	10
Trojan	10
CGI	10
Scanner	7
Ransomware	7
Botnet	6
Backdoor	4
Total	258

Table 4.3: Distribution of samples within the LLM generated dataset

between classes (e.g. CGI: 192; XSS: 53,261). These categories were defined by the private author reflecting its practices, and, not having access to it, the classes may not reflect correctly commonly accepted standards. On this note, we highlight the presence of partially overlapping categories which may hinder our considerations: for example the class `Injection` can comprehend `Code-Execution` and `XSS`. To obtain a dataset to use for testing purposes we prompted LLMs (specifically Gemini 2.5 Flash, GPT-5 and Claude Sonnet 4) with the classes and packet structure to generate new, hypothetic HTTP request, manually verifying each label. This procedure yielded a total of 258 samples: attempts to generate additional data resulted in either repeated entries or incorrectly formatted outputs. As shown in Table 4.3, the classes `Other`, `DoS`, `DDoS` are not present within the dataset. Their exclusion stems from the lack of clear definitions: `Other` is overly generic, while `DoS` and `DDoS` are difficult to characterize within a single HTTP Request as they often resemble legitimate traffic.

As an additional resource for the second use case, specifically to improve the concept datasets and to train the SAEs on more tokens, we employed the CSIC 2010 Web Application Attacks[3]. This dataset contains HTTP packets (without IP information), divided in protocol fields, classified either as `Normal` or `Anomalous`. For our purpose, we restricted usage only to the `Anomalous` samples, automatically assigning each a random IP. Although the true labels are not directly useful for SAE training—which requires model activations rather than class labels—we allowed the classifier to assign labels in order to obtain a general overview of the dataset distribution (Table 4.5).

[3]Available at: `https://www.kaggle.com/datasets/ispangler/csic-2010-web-application-attacks`

Class	Elements
Normal	36000
Anomalous	25065
Total	61.065

Table 4.4: CSIC 2010 Web Application Attacks dataset distribution

Class	Elements
Info-Disclosure	13618
Other	7983
Injection	1551
Code-Execution	1185
XSS	674
Dir-Traversal	26
Scanner	23
Remote-file-Inclusion	5
Total	25065

Table 4.5: Distribution of predicted labels within the CSIC 2010 Web Application Attacks

4.1.1 Concept Datasets for TCAV

To apply the TCAV technique, we first defined a set of concepts and created small datasets for each of them, which were then used to train the linear classifier. Each concept was represented by a collection of samples, allowing the model to learn a direction in the activation space corresponding to that concept.

In the code classification scenario, the defined concepts included:

- `comments`: containing comments in every language (e.g. "//this is a comment")
- `function declarations`: containing function declarations for each language (e.g. "def Example():")
- language-specific function declarations (e.g., "Python function declaration", "Java function declaration"): containing a filtered version of the `function declarations` dataset.

All details about each concept distribution are shown in Table 4.6: each class specific concept contains the same entries as of the function declarations dataset belonging to the class. During training we sampled 300 entries for each concept.
In the packet inspection scenario, the concepts were:

- `IP`: containing IP addresses
- `Methods`: containing HTTP methods (like "GET", "POST", ...)
- `Host`: containing the strings representing the host (like "api.example.org")
- `Path-Traversal`: containing patterns to attempt traversal (e.g. "../")

Function Declarations	
Class	**Elements**
PHP	982
Python	972
Java	955
JavaScript	817
Ruby	653
Go	337
Total	4,716

Comments	
Class	**Elements**
JavaScript	1,902
Python	1,690
Ruby	1,607
Java	1,054
Go	1,017
PHP	899
Total	8,169

Table 4.6: General Concepts dataset distribution

- `Injection`: containing SQL injections, command injections, XSS and other injection example

These datasets were created by extracting information from LLM-generated samples, manually augmented to reach 50 samples each (IP, Method and Host concepts did not require such augmentation). The concepts `Injection` and `Path-Traversal` required manual augmentation to ensure sufficient coverage and representativeness, for this reason their size is significantly smaller than the code classification concepts, attesting at around 50 samples per concept.

Chapter 5

LIME and SHAP Pipeline Results

The analysis follows a structured workflow designed to progressively uncover the behavior of the model under study. First, we examine the local explanations provided by feature-based techniques, which allow us to highlight the contribution of individual features to specific predictions. Building on these local insights, we attempt to exploit such techniques to derive a broader overview of the model's global behavior.

5.1 Explaining Singular Samples

We proceeded with the analysis of the model for the code classification scenario, conducting a basic functionality testing with two hand made samples: an example python function and its equivalent in java.

Both techniques provide a visualization interface that allows to highlight the most important features for a specific class. Analyzing the LIME results, the technique shows that the model relies primarily on language-specific keywords to perform its classification: the Python example (presented in Figure 2.2 in Section 2.5.1 shows "def" as the most relevant word for the `Python` class, while the Java example exhibits (in Figure 5.1) "println", "void" and "public" as the most relevant for the `Java` class. The only outsider is the word "example".

SHAP shows similar results: in red are the features that contribute positively for the selected class, and in blue the ones that contribute negatively. For the same samples analyzed, the Python one (shown as the Lime figure in Figure 2.3 in Section 2.5.2) shows "def" within the most contributing words, while the Java example in Figure 5.2 highlights "System" which is also a keyword of the language. One of the differences between LIME and SHAP is that the latter is capable of analyzing the contribution of punctuation and whitespace: this is important since in the Python example it shows "():" as one of the main contributors, while in the Java examples identifies "\n" (located at the end of the sample) and "!")". To verify whether the model actually relies on keywords to perform its classification, we attempted to design adversarial inputs by embedding keywords from other languages, with the goal of

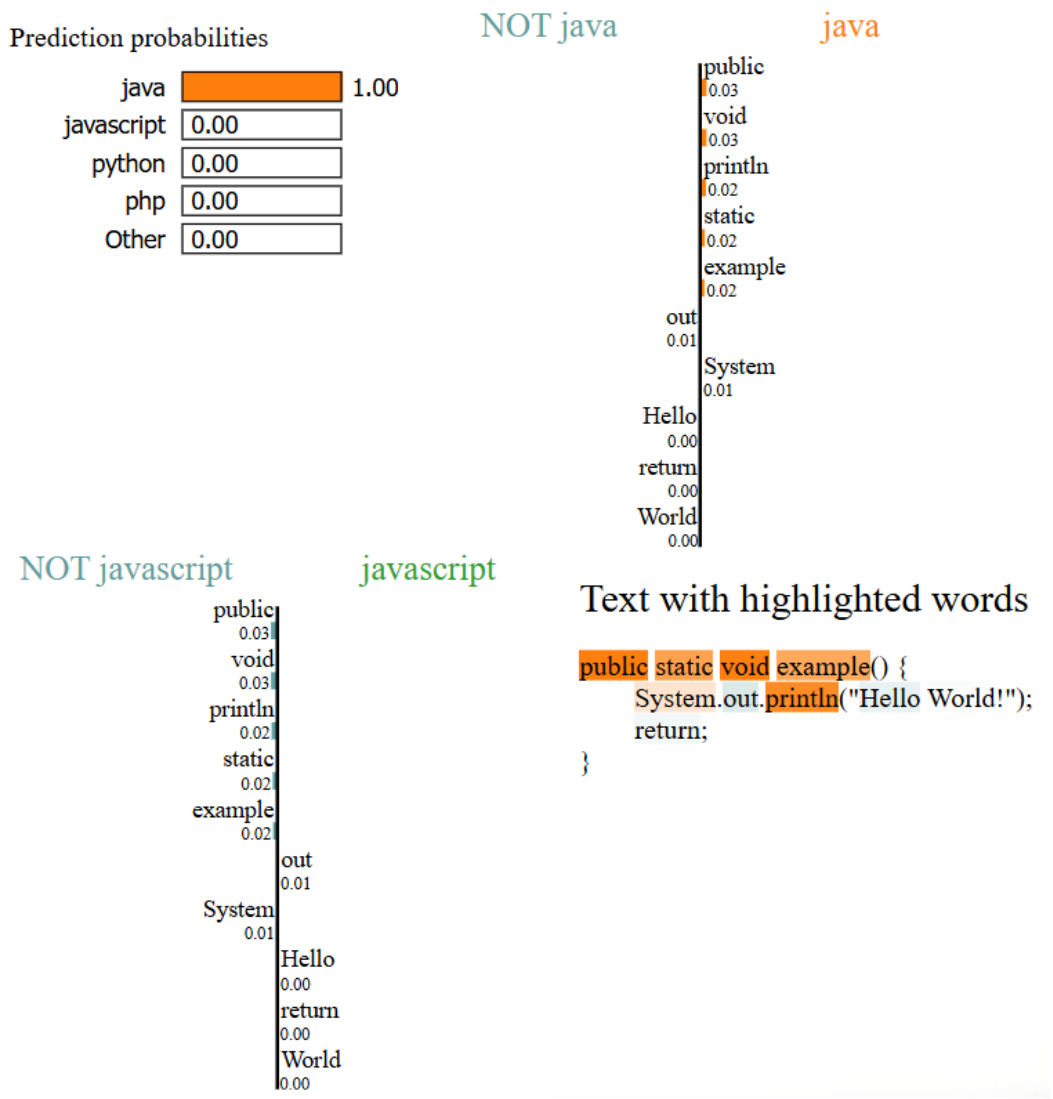

Figure 5.1: LIME Java example

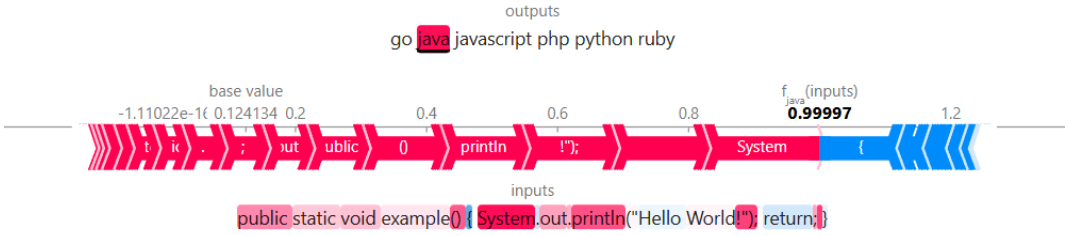

Figure 5.2: SHAP force plot for the Java example

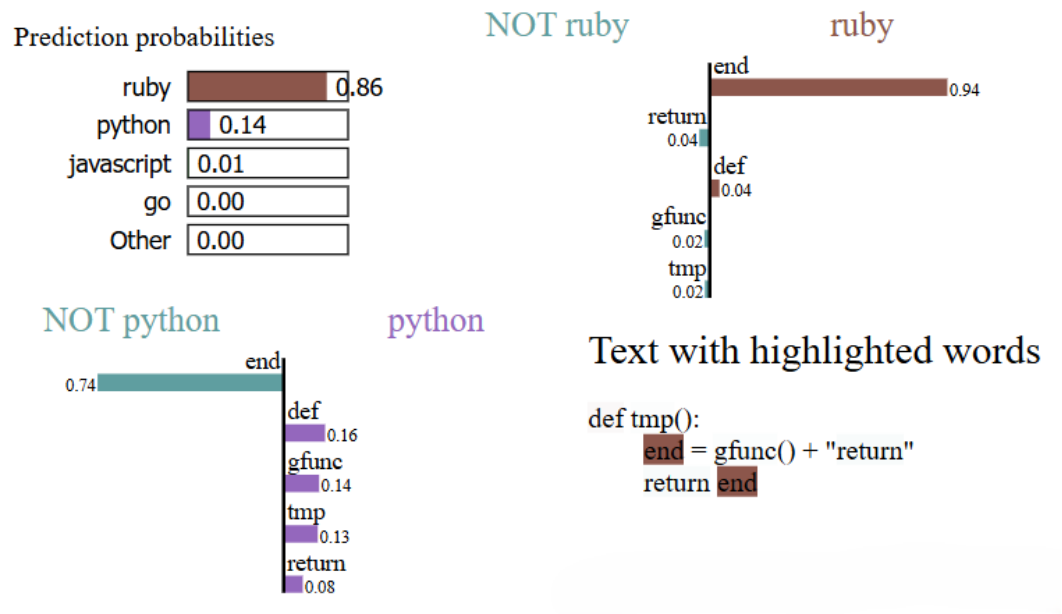

Figure 5.3: LIME explanation for the Python crafted sample

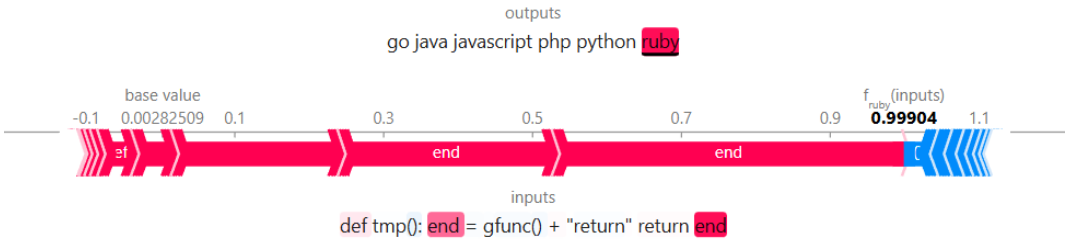

Figure 5.4: SHAP explanation for the Python crafted sample

misleading the model. Among the six supported languages, Ruby shows the greatest similarity to Python. This observation is primarily based on the syntax of function declarations: while Python uses a colon (":") to introduce the function body, Ruby employs the keyword "end" to mark its termination. In both languages, the keyword return is used to explicitly return values, although in Ruby the function structure is always enclosed between "def" and "end". Java, on the other hand, exhibits superficial syntactic similarities with JavaScript, since the latter was originally designed with a Java-like syntax to ease adoption. Based on these analogies, we crafted two examples:

- For Python we use "end" as a variable name and insert the word "return" as a string;
- For Java, we use "let" as a variable name and use "function_" as function name.

The results in Figure 5.6 confirm our assumption that the model heavily relies on keywords, yet fails to gather the global structure of the code. In particular, for the Python sample, LIME tells us that the keyword "end" is the only one important for the class `Ruby`. SHAP gives us the same explanation, however highlighting a second difference with LIME: it is capable of differentiating between two equal words,

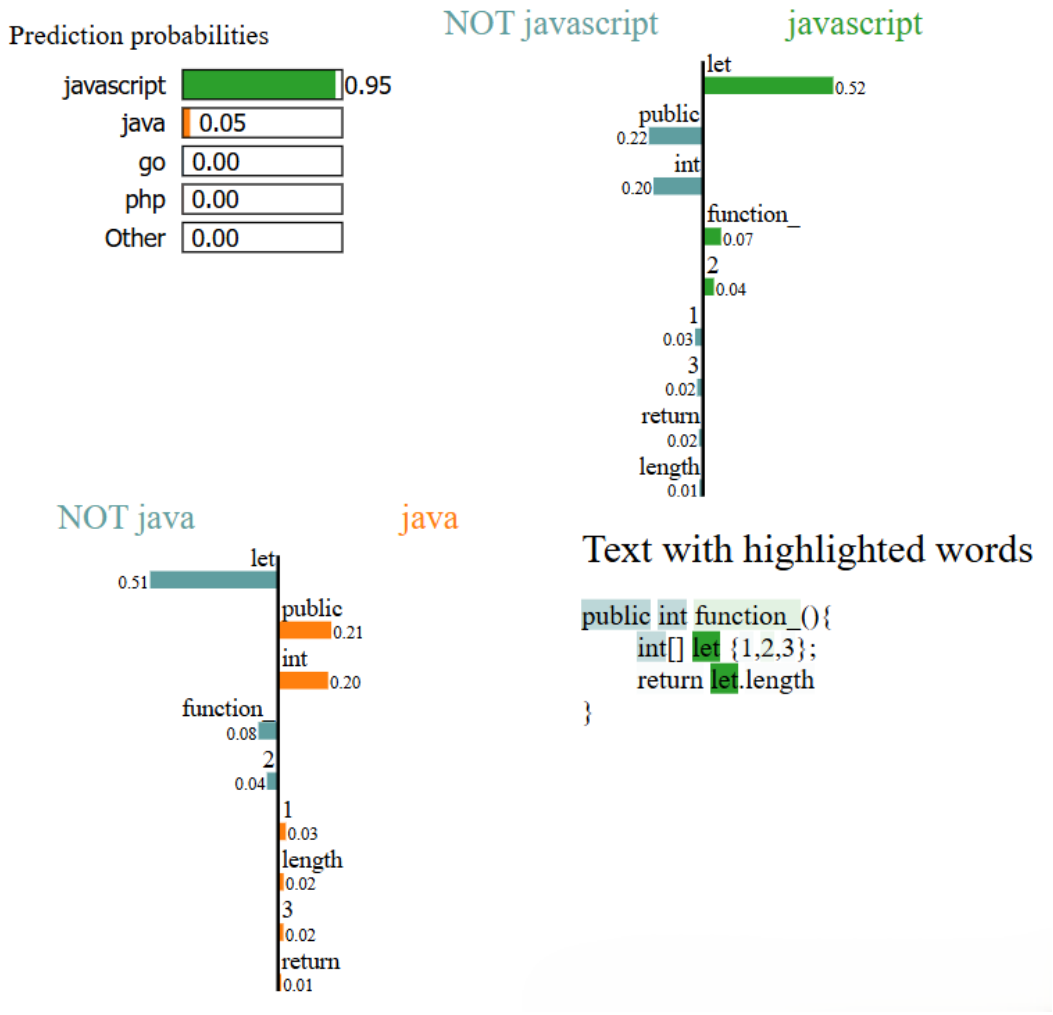

Figure 5.5: LIME explanation for the Java crafted sample

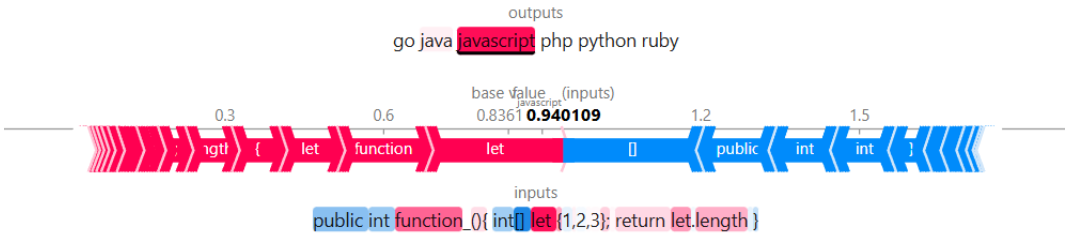

Figure 5.6: SHAP explanation for the Java crafted sample

analyzing the contribution of each one separately. It is important to notice how SHAP marks "():" as against the `Ruby` classification meaning the model might have learned some code structure, however its contribution is too weak compared to the words "end". For the Java and JavaScript examples, in Figure 5.2, the explanation shows little difference from LIME: SHAP identifies "[]" as one of the main contributors for the label `Java`, which is to be expected since that construct in JavaScript is used in a very different context. Following these observation, the overall explanation for both Python and Java matches: we expect the model to use the keywords of each respective language to perform its decision. To verify this assertion,

These results show that the model could have a too heavy dependence on the keywords, or it is possible that such small examples are not suitable for the model, since it might not be able to extract the context. We tested some examples taken from the dataset and the results are similar: the keywords are the most contributing factors; however, their contribution is not higher as before, being slightly above the other parameters. During testing we also observed another behavior: the model is unexpectedly sensible to the indentation. In some cases, using four spaces as indentation instead of the "\t" character substantially varies the prediction and, consequently, the LIME values. This is not expected since the training data varies between the two types of indentation between samples.

We then moved to the packet inspection scenario and, similarly to the code classification case, we began with the analysis of basic samples. Among the manually inspected cases, the most prominent case involved the `Dir-Traversal` class: LIME reveals an important influence over content containing the word "filename" (as shown in Figure 5.7), despite the fact that the ground-truth label of the sample corresponds to `Remote-file-Inclusion`.

Removing the word from the packet content completely shifts the classification towards the `Other` class (second in position in Figure 5.7). This behavior is consistent across other samples: for instance, analyzing an example from the class `Code-Execution`, the LIME results indicates that the classification relies almost only on the word "php". While this feature highlights the presence of a PHP file or script, it does not necessarily imply an actual code-execution attempt.

Figure 5.8 reports the SHAP explanation for the same sample analyzed previously. The results are highly consistent with LIME: the tokens "filename", "=" and "txt" emerge as the most influential features, confirming the local importance of specific content elements in driving the classification.

We see that this behavior of exploiting key words is consistent among the most relevant classes for the model: as mentioned in Section 4.1, from the analysis of our testing dataset, 96% of the samples is classified within only five classes. For the less important ones instead it is difficult to identify clear relevant words. These results are encouraging, showing in full the capabilities of the current techniques for local explanation. However we cannot be sure that the keywords are always what is used in the decision making process.

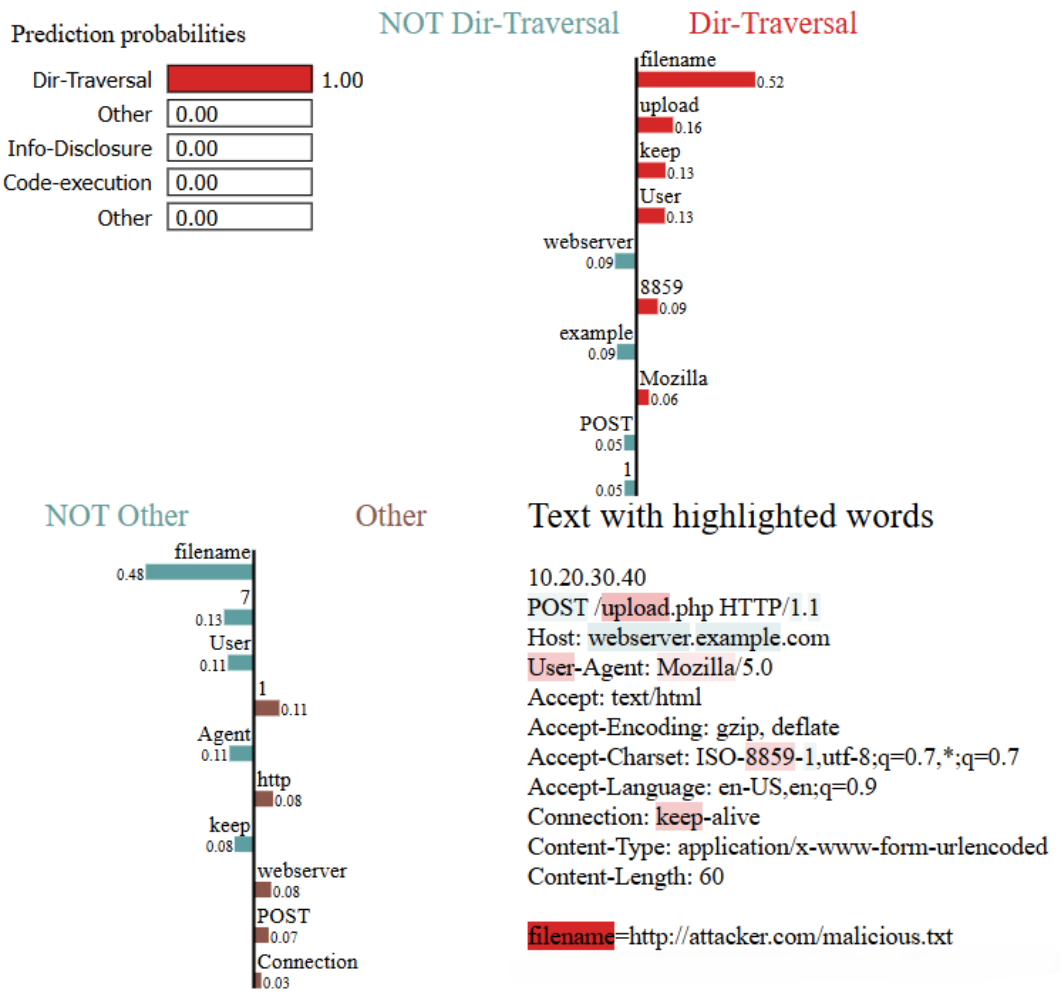

Figure 5.7: LIME explanation of a Remote file inclusion sample.

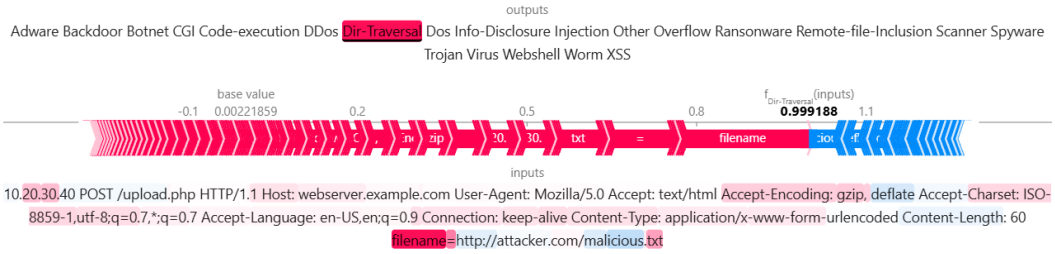

Figure 5.8: LIME explanation of a Remote file inclusion sample.

Go		Java		JavaScript	
Word	Value	Word	Value	Word	Value
"MockTracer"	0.1343	"rgbs"	0.2370	"var"	0.0690
"matchers"	0.0860	"double"	0.2299	"newSlots"	0.0510
"UnixNano"	0.0762	"Configuration"	0.1632	"function"	0.0476
"GomegaMatcher"	0.0727	"mapLabel..."	0.1549	"watcher"	0.0414
"mem"	0.0707	"DEFAULT_C..."	0.1533	"useDom"	0.0398

PHP		Python		Ruby	
Word	Value	Word	Value	Word	Value
"string_arr..."	0.0166	"scores"	0.0841	"md"	0.2082
"alpha"	0.0132	"score"	0.0554	"to_i"	0.0498
"beta"	0.0074	"Stop"	0.0166	"nonce"	0.0441
"backButton..."	0.0072	"stopDistance"	0.0164	"ciphertext"	0.0421
"server"	0.0063	"package_dir"	0.0136	"i_to_s"	0.0404

Table 5.1: LIME most important words for each class in the code classification use case.

5.2 Explaining Multiple Samples

In order to gather information on the global behavior of the models we analyze multiple entries computing the average value of "words". This allows us to compare the elements that appear many time against the sporadic ones, to define if the model has a bias for specific words: if the model consistently exploit keywords, we expect them to be between the highest average values. For the first use case we feed through both LIME and SHAP a series of 360 samples, with 60 samples per class, to obtain the average value for the words. The LIME results are available in Table 5.1, we can clearly see how our previous considerations were indeed dependent on our crafted samples. The general overview highlights how keywords are not as important: we see keyword in the top 5 words only in two classes. In `JavaScript` we see "var" and "function", but their LIME value is really small and not that much bigger than casual terms like "newSlot", and in `Java` we see only double", which is actually used only one time within the test samples analyzed. These LIME results, despite correcting our previous consideration, fail to help us to understand better the global behavior: all of the classes have similar average values meaning there is not a prevalent one and the most important words are in all cases situational. In the SHAP results (Table 5.2) we see a similar condition to LIME: the most influential features correspond to casual words that are present in very few examples. We see however that there are two exceptions: the class `Go` and the class `Ruby`. Those two present "func" and "end" respectively in the first positions highlighting a possible bias of the model: their value however is not significantly higher than the other words.

Moving to the packet inspection use case, we analyze the average word values of the LLM generated dataset. The results we obtain are quite different: the values of the most important features among classes are very imbalanced.

Go		Java		JavaScript	
Word	Value	Word	Value	Word	Value
"funcMap"	0.27	"StrokeStringAt"	0.27	"noinfo"	0.21
"func"	0.23	"RegisterExtractor"	0.25	"UsabillaBridge"	0.19
"gc"	0.20	"GomegaMatcher"	0.22	"cloneDeep"	0.19
"updateVal"	0.18	"Format"	0.19	"(/#/."	0.12
"SetContext"	0.16	"potentialMatch"	0.18	"repeat"	0.12

PHP		Python		Ruby	
Word	Value	Word	Value	Word	Value
"listCampaigns	0.15	"*_\'\'\'"	0.09	"end"	0.14
")$"	0.10	".\"\"\""	0.08	"«-"	0.12
"::$_"	0.09	"def"	0.08	"puts"	0.10
"ucwords..."	0.09	"Compatibility"	0.06	"textile"	0.09
"PAGE"	0.08	"\'\'\'**"	0.05	"://#{"	0.08

Table 5.2: SHAP most important words for each class in the code classification use case.

The aggregated results, limited only to the first three words for readability purposes, are summarized in Table 5.3.

This table reports the average contribution values defined by LIME of words across all test samples[1]. We can see how classes such as `CGI`, `Adware`, `Virus`, and `Backdoor`, although present in the test dataset, are never considered by the model, achieving very low average importance. Looking at the most relevant classes we see that the values are much higher: the most evident being the `Injection` class, with the most important words actually relatable to the label: even if promising, these results probably depend from the poor size and distribution of the testing dataset.

The results of SHAP (in Table 5.4) are really similar, many classes see average feature importance close to zero, while the top 5 classes see relevant values. In general we have a more clearer explanation with SHAP since the words are better represented, for example without including numbers with them, but at the same time we see some fixation on the "word" “’)”, which makes results harder to interpret.

5.3 Conclusions regarding LIME and SHAP

The analysis of LIME and SHAP across multiple samples highlights a fundamental limitation of feature-based explanation techniques: while they are effective in providing local insights on individual predictions, they fail to capture the global behavior of the models. Both methods tend to emphasize sporadic or context-dependent words, which prevents the identification of systematic biases or structural shortcuts. Only in the packet inspection scenario we see sporadic meaningful words, but these results are attributable to the poor testing dataset. The results obtained with the local

[1]It is important to remember that due to the small dimension of the dataset, some words represent really the average, while other appear to sporadically: eliminating those sporadic words would not improve the results since in many case, their presence highly influence the classification.

Info-Disclosure		Dir-Traversal		Scanner	
Word	Value	Word	Value	Word	Value
"ne"	0.7525	"filename"	0.4609	"callback"	0.1214
"2fboot"	0.6081	"resize_image"	0.4306	"overflow"	0.0958
"new_ads"	0.5209	"26file2"	0.4020	"js2"	0.0299

Code-execution		XSS		Remote-file-Inclusion	
Word	Value	Word	Value	Word	Value
"module"	0.6090	"alert"	0.5103	"hacker"	0.1638
"debug"	0.5821	"3Ealert"	0.2841	"http"	0.0648
"echo"	0.4004	"document"	0.2747	"rfi"	0.0644

Botnet		Overflow		Backdoor	
Word	Value	Word	Value	Word	Value
"Nikto"	0.0198	"20hacker"	0.0256	"body"	0.0085
"mysite2"	0.0188	"eval"	0.0164	"onmouseover"	0.0074
"Googlebot"	0.0077	"76"	0.0141	"onload"	0.0069

Webshell		Worm		Trojan	
Word	Value	Word	Value	Word	Value
"shell"	0.1320	"search"	0.0232	"search"	0.0004
"evil"	0.0654	"25"	0.0132	"20hacker"	0.0004
"20192"	0.0483	"webmail"	0.0121	"20sudo"	0.0003

CGI		Adware		Ransonware	
Word	Value	Word	Value	Word	Value
"20hacker"	0.0012	"20hacker"	0.0002	"search"	0.0002
"apisite2"	0.0011	"search"	0.0002	"20hacker"	0.0002
"20sudo"	0.0008	"20sudo"	0.0001	"20sudo"	0.0001

Spyware		Virus		Injection	
Word	Value	Word	Value	Word	Value
"search"	0.0002	"search"	0.0002	"27whoami"	0.8423
"20hacker"	0.0002	"20hacker"	0.0001	"execSync"	0.7588
"61"	0.0001	"shell"	0.0001	"expression"	0.5076

Table 5.3: LIME most important words for each class in the packet inspection use case.

Info-Disclosure		Dir-Traversal		Scanner	
Word	Value	Word	Value	Word	Value
"conf"	0.43	"path"	0.21	"callback"	0.21
"sql"	0.40	"../"	0.16	"cook"	0.16
"words"	0.38	"size"	0.13	"click"	0.11

Code-execution		XSS		Remote-file-Inclusion	
Word	Value	Word	Value	Word	Value
"debug"	0.49	"alert"	0.39	"inc"	0.05
"{{"	0.44	"document"	0.18	"jection"	0.03
"module"	0.37	"img"	0.18	"nd"	0.02

Botnet		Overflow		Backdoor	
Word	Value	Word	Value	Word	Value
"body"	0.08	"=’"	0.01	"load"	0.01
"param"	0.07	"’)."	0.01	"́)"	0.00
"query"	0.07	"Attack"	0.01	"̈"	0.00

Webshell		Worm		Trojan	
Word	Value	Word	Value	Word	Value
"shell"	0.11	"search"	0.01	"’)"	0.00
"uploads"	0.11	"body"	0.00	"search"	0.00
"render"	0.02	"GET"	0.00	"query"	0.00

CGI		Adware		Ransomware	
Word	Value	Word	Value	Word	Value
"́)"	0.00	"search"	0.00	"’)"	0.00
"́)."	0.00	"́)"	0.00	"search"	0.00
"search"	0.00	"́)."	0.00	"’)."	0.00

Spyware		Virus		Injection	
Word	Value	Word	Value	Word	Value
"’)"	0.00	"search"	0.00	"expression"	0.69
"search"	0.00	"’)"	0.00	"assert"	0.56
"null"	0.00	"query"	0.00	"Sync"	0.25

Table 5.4: SHAP most important words for each class in the packet inspection use case.

explanations are situational, offering little guidance on how the model organizes its decision boundaries at a global scale. Their local capabilities instead proved effective in highlighting important tokens, this method can be used to support end users in trusting the model, but does not help in gaining insights on model reasoning.

Chapter 6

TCAV Pipeline Results

The methodological adaptations described in Section 3.2 provide the foundation for evaluating the effectiveness of TCAV in capturing human-defined concepts within language models. In particular, the introduction of three distinct strategies (Truncated, Padded, and Average) was necessary to overcome the structural constraints imposed by variable-length token sequences, ensuring that concept activation vectors could be consistently trained across layers. The following results illustrate how these strategies behave when applied to both general and class-related concepts, highlighting differences in score progression, stability, and statistical significance. By systematically comparing the average TCAV scores, their variability, and the outcomes of rigorous statistical testing, we aim to assess whether the concepts are meaningfully represented within the models or whether the derived CAVs reflect incidental correlations.

6.1 Comparing TCAV adaptations

To compare the three strategies, we trained the CAVs multiple times across all defined concepts and monitored the average TCAV scores across the layers of both models. The results shows that the "Truncated" and the "Padded" strategy behave in similar manner, while "Average" remains close, but consistently separated. Moreover, "Padded" and "Truncated" exhibit greater variability across layers, with scores fluctuating more significantly, whereas "Average" shows a more linear trend. This behavior is illustrated in Figure 6.1, reporting the average score for the concept "comments" in the code classification use case, and in Figure 6.2, which shows the corresponding results for the packet inspection scenario: the behavior is particularly evident on the class `Java`.

Looking globally at all the trained concepts, it becomes evident that none of them exhibit a clear directional trend across layers, as originally suggested by [8]. In our experiments, TCAV scores neither consistently increase nor decrease in deeper layers, making it difficult to infer where — or if — the model internalizes specific concepts.

When examining each individual strategy more closely, a different perspective emerges: "Truncated" and "Padded" show lower variability and more stable scores

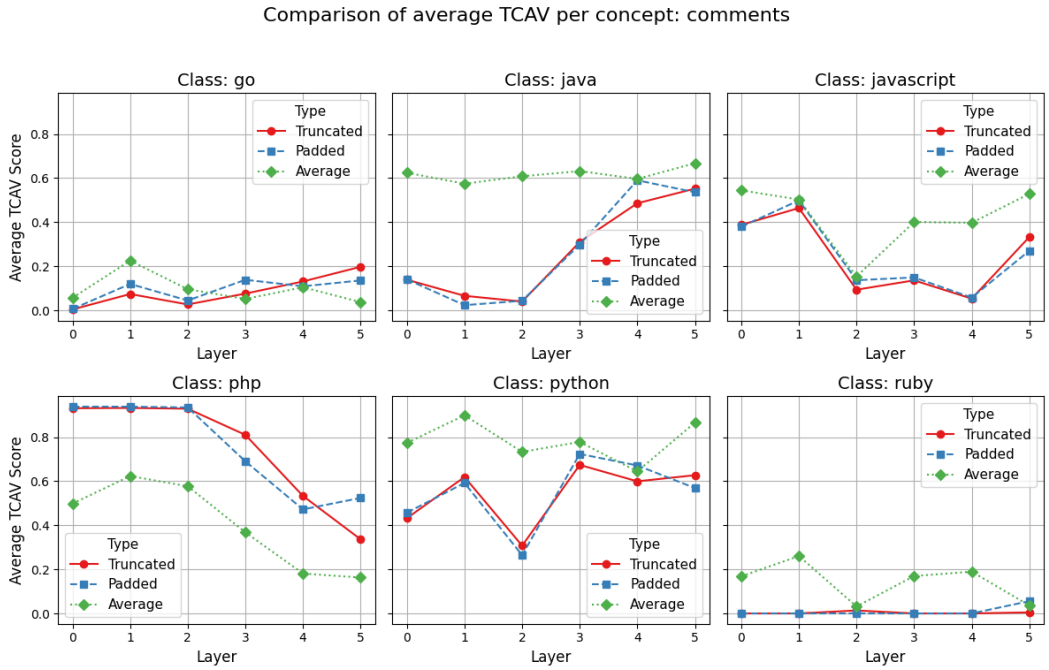

Figure 6.1: Average TCAV score for the concept "comment" on the code classification model

between test runs, while "Average" proves to be more inconsistent. This behavior is observed across most concepts in the code classification use case, whereas in the packet inspection scenario it appears less pronounced. Through the box-plot in Figures 6.3 and 6.4 we can see that within the "Padded" strategy, 50% of the results tend to hover within a range of 0.25 from the median, while if we consider the "Average" strategy, 50% of the samples are almost always in a range of 0.5 around the median. This highlights that "Padded" and "Truncated" might be better suited even if their average value varies a lot between the layers. Nonetheless, the variability of TCAV values remains high, suggesting that the CAVs might not be fully compatible with the internal representations of Language Models.

6.2 Analyzing differences between general and class-specific concepts

To explore all possible concept scenarios we opted to define two types of concepts: general and class-specific. The general concepts contain examples extracted from multiple classes inserted together. Their objective is to understand if the model recognizes structures similar to the one humans use. The class-specific concepts instead contain examples extracted only from one single class. Their objective is to define whether a model is exploiting that concept to perform its classification. In the results of the code classification scenario, we observe that class-specific concepts yield TCAV scores approaching 1 within their corresponding classes. Figure 6.5, showing the "PHP function declaration" concept with "Padded" strategy, exemplifies

Figure 6.2: Average TCAV score for the concept "IP" on the packet inspection model

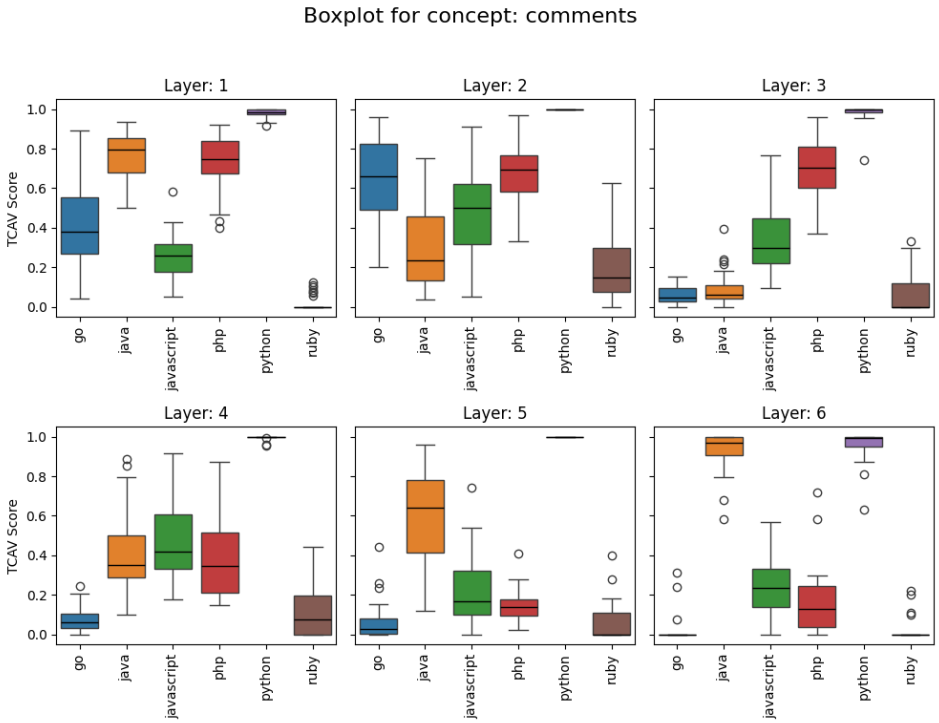

Figure 6.3: TCAV scores for concept "comments" with "Padded" strategy in the code classification scenario

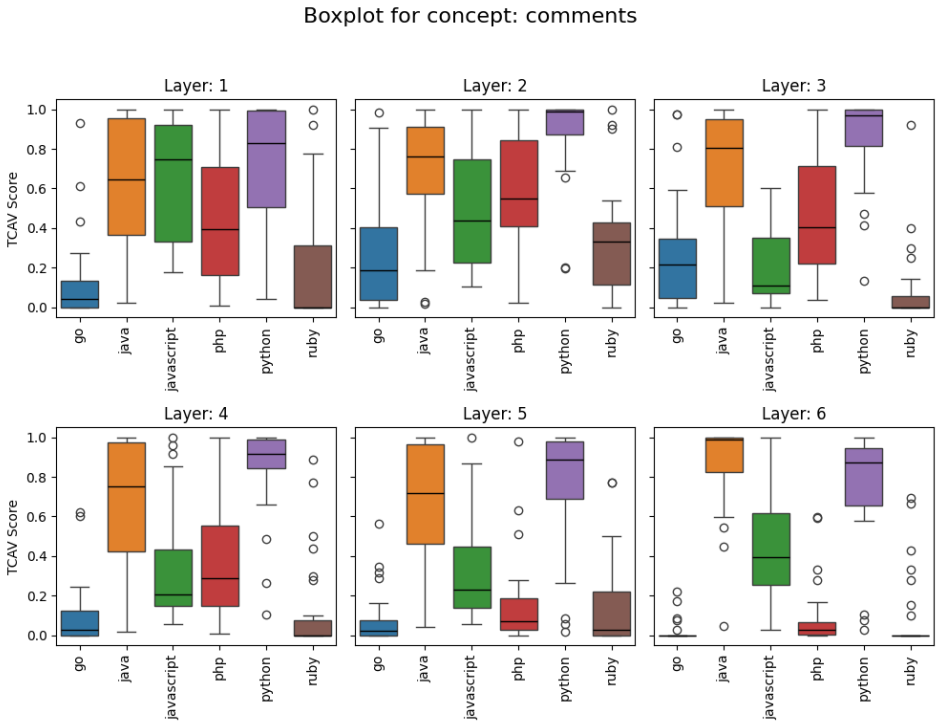

Figure 6.4: TCAV scores for concept "Comments" with "Average" strategy in the code classification scenario

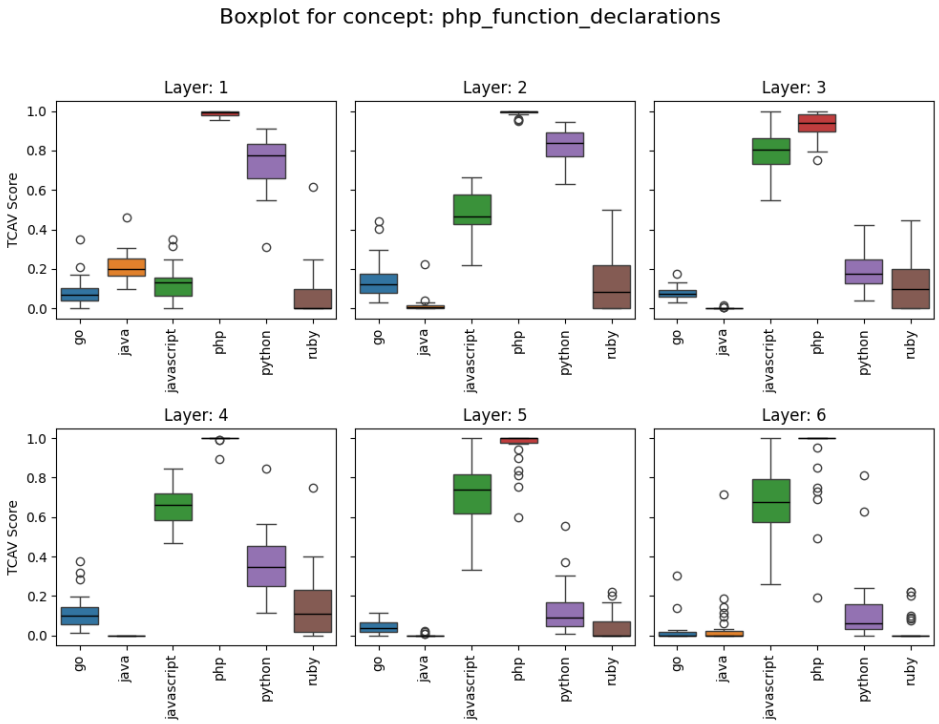

Figure 6.5: TCAV scores for the "PHP function declarations" concept with "Padded" strategy

this behavior for the PHP class, where the TCAV score remains consistently high across all layers in every run. This behavior is consistent also for the "Average" strategy: this outcome may indicate either that the concept is genuinely meaningful and well-represented for that class, or that the CAV has simply learned to distinguish samples of that class from all others, without capturing a truly semantic pattern. Analyzing the general concepts reveals a more nuanced scenario. The concept `Comment`, as illustrated in Figure 6.3, shows a consistent tendency toward the `Python` class, but only under the "Padded" and "Truncated" strategies. In contrast, the concept `Function Declaration` does not exhibit any noticeable class-specific bias. These observations may hint at a preference of the model for associating comments with Python code; however, this hypothesis remains speculative and needs statistical support to be considered conclusive.

The situation in the packet inspection scenario is different: we see no big difference between general and class-specific concepts, and in the latter case, the TCAV scores do not indicate a particular tendency towards the related class. For example, in Figure 6.6 the class-specific concept "Path-Traversal" for the class `Dir-Traversal` does not show any significant influence for its class. The same behavior belongs to the generic concepts such as "IP" and "Methods". This could suggest that either the class specific concept is not used to perform the prediction, meaning the model needs to be trained better, or, as suggested in the previous results (Section 6.1), the CAVs does not suit the Language Models.

To verify if the learned CAVs are meaning less, we used the t-test and measured

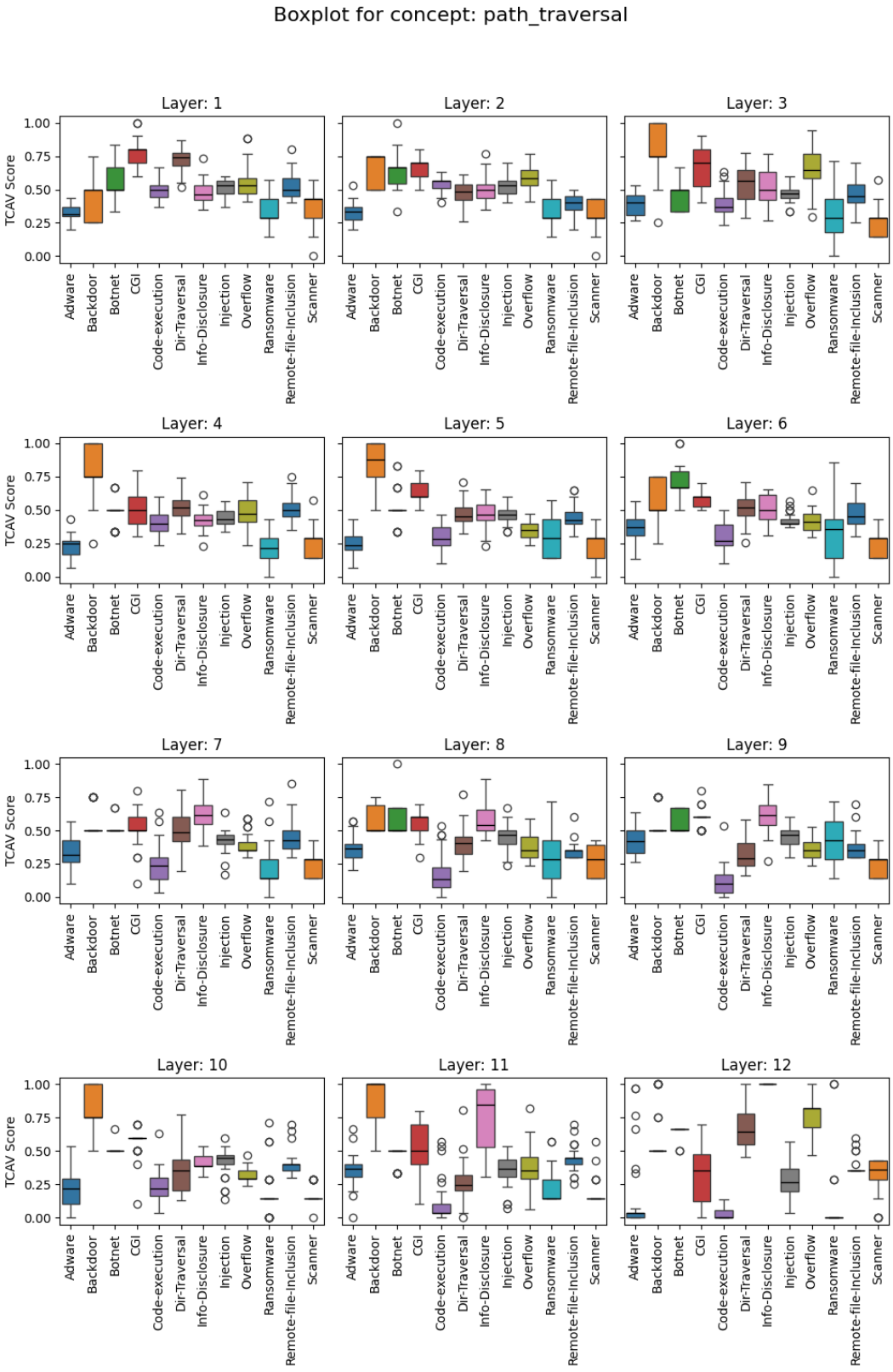

Figure 6.6: TCAV score for the "Path-Traversal" concept with "Truncated" strategy

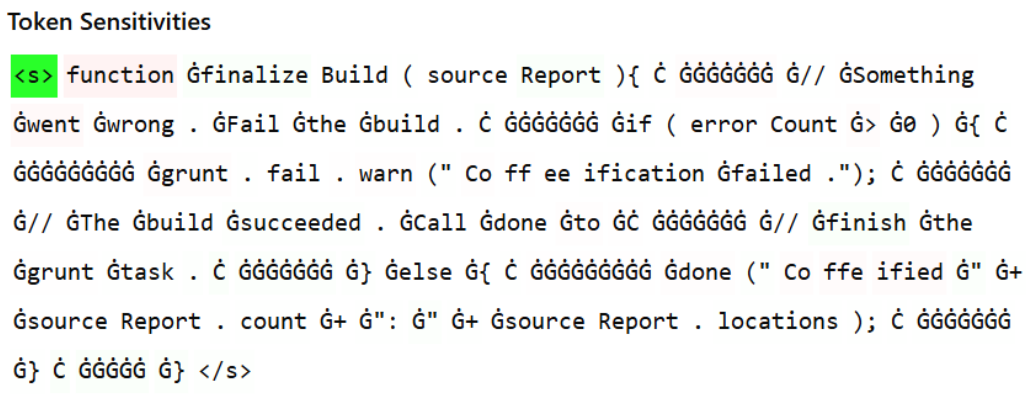

Figure 6.7: TCAV validation for the concept "Python function declaration" in a `JavaScript` sample

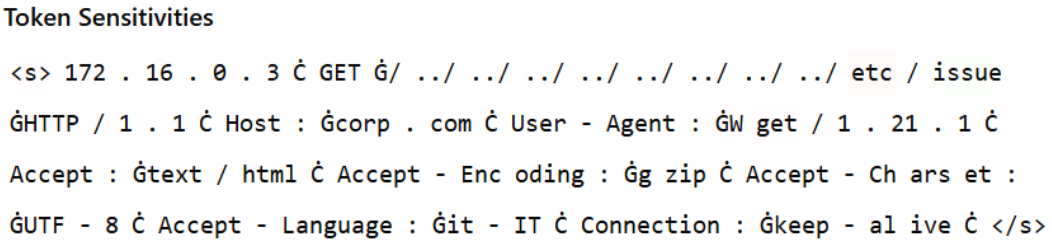

Figure 6.8: TCAV validation for the concept "Path Traversal" in a `Dir-Traversal` sample

which ones obtain a p-value < 0.0001. The results suggests that no CAV is actually meaningful for any class, in both use cases. We see that only in the "Average" strategy we obtain some meaningful concept, but the high variability of them highlighted in Section 6.1 suggests the need of further validation.

6.3 Validating TCAV average strategy

To validate the meaningful CAVs found with the "Average" strategy, we used our visualization tool, allowing us to follow the style of LIME and SHAP. We used the gradient of each singular token to obtain a TCAV score specific for each tokens and the web application allowed us to clearly identify the most meaningful ones. If the concept is correctly represented we expect to see tokens belonging to the concept highlighted positively, while others not belonging negatively. Proceeding with the evaluation we notice that every CAV trained on the last layer of the model provide values only to the special “CLS” token: this is due to the fact that the model is trained to concentrate all information on that to perform classification tasks. This assure us of the correct implementation of the technique, but does not provide information on which tokens represent the concept. In Table 6.1, we can see an example: the CAV seems to have learned some relations thanks to the most important tokens belonging to `JavaScript`, but we cannot extract any information about a potential shortcut. When examining the lower layers, the situation changes: the highlighted tokens

Token	Sensitivity	Class
"CLS"	0.00475	JavaScript
"CLS"	0.00124	JavaScript
"CLS"	0.00015	JavaScript
"CLS"	0.00011	JavaScript
"CLS"	0.00002	JavaScript

Table 6.1: Top 5 most activating tokens for the concept "JavaScript function declaration", in layer 6

Token	Sensitivity	Class
"."	0.00043	JavaScript
"Ġ}"	0.00035	Ruby
"Ġfinalize"	0.00032	JavaScript
"Ġ"	0.00032	Python
"</s>"	0.00031	JavaScript

Table 6.2: Top 5 most activating tokens for the concept "Comment", in layer 4

generally fail to align with the intended concept. For instance, Table 6.2 reports the most activating tokens for the concept `Comments`, yet none of them correspond to actual comment tokens. Interestingly, a token such as "ĠThe"[1], which does belong to a comment, emerges with a negative sensitivity of -0.0002, indicating that it is incorrectly treated as unrelated to the concept. This situation persists across both use cases: in Figures 6.7 and 6.8 we report two examples regarding the packet inspection scenario. In the first image (6.7) the concept `Python function declaration` is shown to activate positively on the starting token of a `JavaScript` sample, while in the second one (6.8) we see that the concept `Path Traversal`, specific for the `Dir-Traversal` class, doesn't activate in a sample clearly containing that concept.

6.4 Conclusions regarding TCAV

These result clearly shows that Testing with Concept Activation Vectors does not suit well the Natural Language Processing scenario. Neither of our implementation strategy proved effective in highlighting meaningful information that could be used to enlighten the decision making process of the models.

[1]The special character Ġ indicate a space before the word, as mentioned in Section 2.3

Chapter 7

SAE Pipeline Results

Finally, we extend the investigation to SAEs. By training multiple autoencoders and analyzing the learned features, we aim to evaluate the capabilities of this technique to uncover latent structures within the activations. The results are organized around three complementary perspectives. First, we compare SAE configurations in terms of sparsity and dead features, to evaluate their efficiency and the presence of redundant or inactive directions. Second, we analyze the interpretability of individual features, both through manual inspection and through automated filtering techniques such as clustering and TF-IDF. These approaches highlight whether certain features correspond to coherent token groups or class-specific concepts. Finally, we validate the relevance of selected features by measuring the impact of their removal on downstream performance, contrasting these results with random baselines.

7.1 Comparison of SAE Trained with Different Dimensions and on Different Layers

In this preliminary analysis we compared Sparse Autoencoders trained on different layers and with varying hidden dimensions, in order to identify the most effective configuration. Our first objective was to define when a feature identified by the SAE is to be considered active. For that we computed the average activation of each feature and selected a value that preserved approximately 25% of the features. This threshold allows to filter low activations that might draw the attention away from the meaningful ones. The choice led to a threshold of 0.02 for the code classification use case, and 0.01 for the packet inspection use case (Figures 7.1 and 7.2). The difference in the value is caused by the first scenario having higher average feature activations than the second one. In both cases no dead features (features that never reach the threshold) were observed, unless the threshold was set to unrealistically high values. This might be due to insufficient feature decomposition, requiring higher SAE dimensionality.

When comparing SAEs of different dimensions trained on the same layer, we expected to observe feature splitting, meaning that a feature in a smaller SAE is decomposed into multiple features in a larger one. However, this phenomenon was not

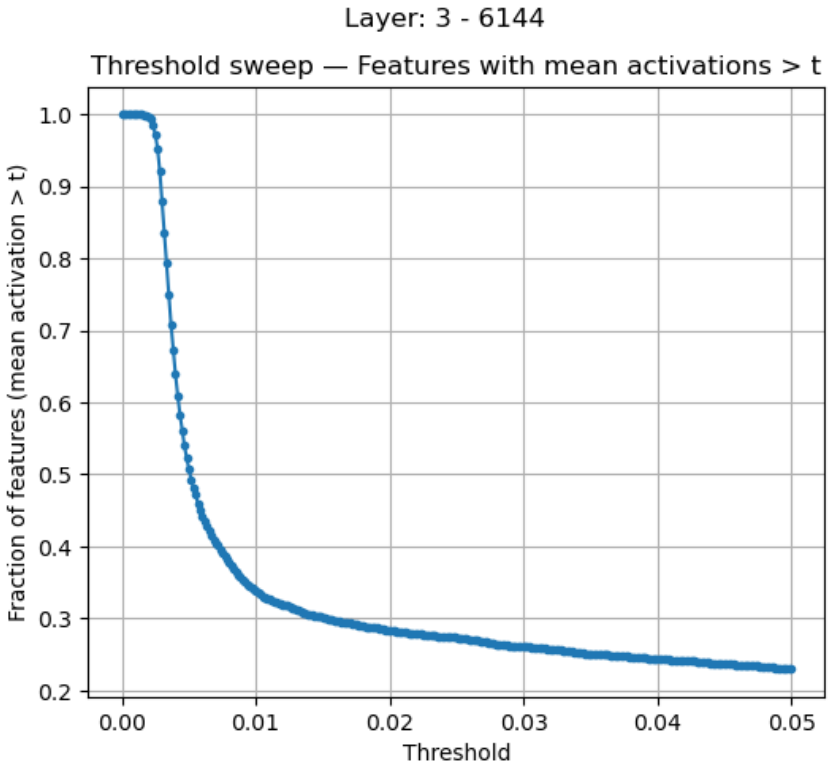

Figure 7.1: Code classification use case: fraction of SAE features with mean activations above the threshold

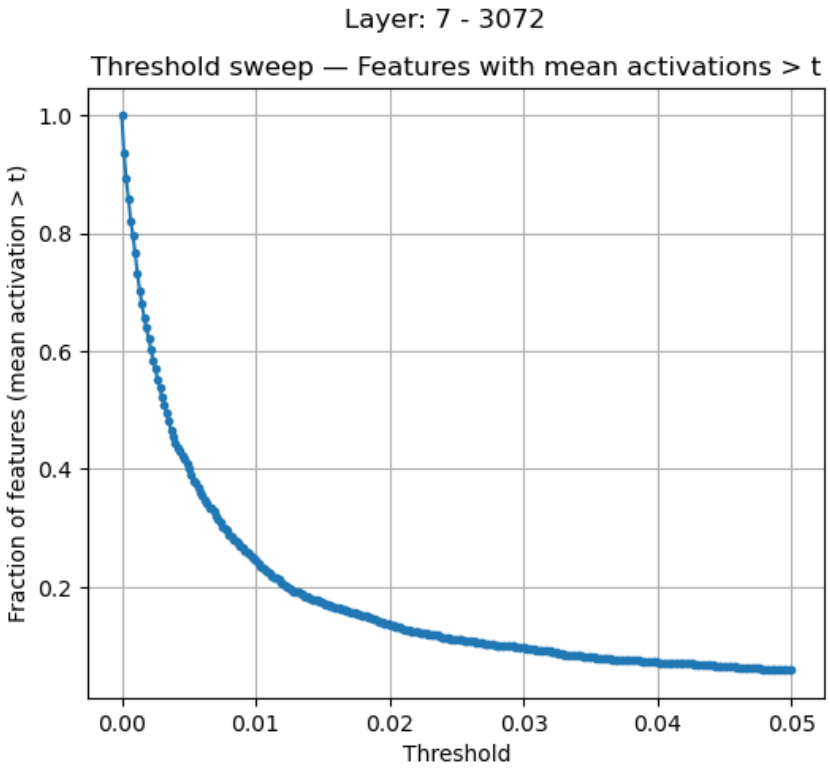

Figure 7.2: Code classification use case: fraction of SAE features with mean activations above the threshold

Go		Java		JavaScript	
Feature	Value	Feature	Value	Feature	Value
9010	0.2471	6377	0.1390	10275	0.5415
6676	0.1888	1883	0.1281	7744	0.1136
4505	0.1717	8359	0.1024	9664	0.0954

PHP		Python		Ruby	
Feature	Value	Feature	Value	Feature	Value
7233	0.1363	10349	0.1213	10480	1.3054
668	0.1219	1262	0.1197	3629	0.2300
4903	0.1142	668	0.1166	4253	0.2154

Table 7.1: Code classification use case: features with highest average value for each predicted class (classes that are not predicted are not shown) for SAE trained on layer 4 of the model and 12288 parameters

directly observed in our experiment. Instead we saw larger SAE have features more specialized for some classes: two different features may activate on the same lexical token, but in different context, suggesting that the model in that layer may have already take the decision. For this reason we tried to identify meaningful features filtering on the highest average activation per class, excluding general features that were active across all classes.

Only a limited number of them exhibited significantly higher activations than the rest. For example in Table 7.1 we see the feature C/4/10480/12288[1] having a high value for the class `Ruby` and in Table 7.2 feature P/8/9570/12288. We can also see how in the packet inspection scenario, the same feature appears in multiple classes, suggesting the RoBERTa classifier is not finding patterns for specific classes, with some exception like the previously mentioned `XSS`. We visualized the two features with our tool obtaining Figures 7.3 and 7.4.

For the feature of the packet inspection scenario, we have a clear identification of tokens related to `XSS`, while the feature of the code classification tends to highlight many tokens, making the interpretability harder. This situation is present for the majority of our SAEs. This behavior highlight the main challenge of interpreting SAE's results, as only a subset of features can be linked to a human understandable result.

These results suggest that SAEs are capable of capturing BERT's layers activation patterns without producing dead features, even with relatively strict threshold. Moreover larger dimensions increase the representational richness but do not guarantee interpretable features. The identification of significant features is limited, but these results motivate further investigation.

[1]Notation: *use case* / *layer of reference* / *feature identifier* / *SAE hidden dimension*.

Info-Disclosure		Dir-Traversal		Scanner	
Feature	Value	Feature	Value	Feature	Value
6652	0.1393	2126	0.1386	2126	0.2475
2126	0.1341	6652	0.1274	5763	0.1418
6311	0.0945	9733	0.0909	4505	0.1344

Code-execution		XSS		Remote-file-Inclusion	
Feature	Value	Feature	Value	Feature	Value
2126	0.1260	9570	0.4285	3233	0.1598
11595	0.1007	7262	0.3171	2126	0.1439
6652	0.1002	3616	0.2534	6311	0.1368

Webshell		DoS		Other		Injection	
Feature	Value	Feature	Value	Feature	Value	Feature	Value
2126	0.1424	2126	0.1469	11294	0.2986	2126	0.1200
6652	0.1038	6652	0.1455	6814	0.2960	6652	0.1005
3233	0.0866	5820	0.1201	4545	0.2717	597	0.0924

Table 7.2: Packet inspection use case: features with highest average value for each predicted class (classes that are not predicted are not shown) for SAE trained on layer 8 of the model and 12288 parameters

Figure 7.3: Feature C/4/10480/12288

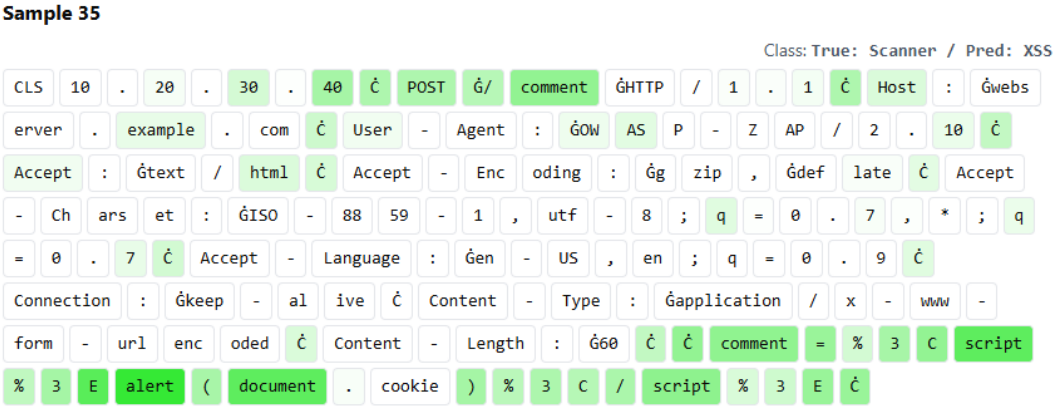

Figure 7.4: Feature P/8/9570/12288

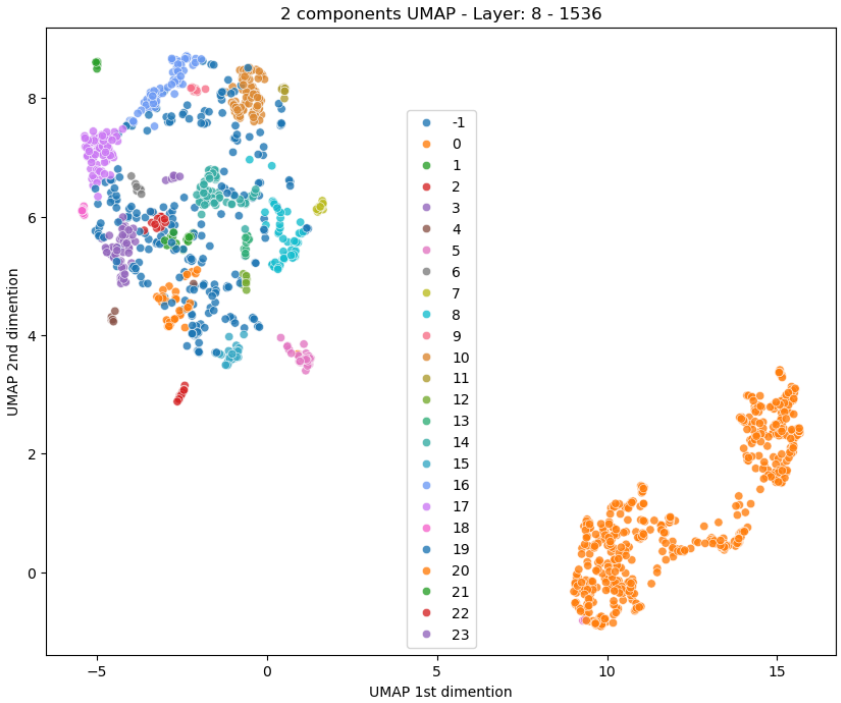

Figure 7.5: Clustering example with HDBSCAN of SAE feature directions

7.2 Identifying important concepts with SAEs

In order to find more meaningful concepts we experimented with several techniques (see Section 3.3). We began by applying clustering to the feature directions using HDBSCAN, sampling representative features from each cluster. To make the clustering tractable, the input data was first reduced to 10 components with UMAP, while for visualization purposes we further projected it into two dimensions. We provide an example in Figure 7.5 related to the SAE trained on Layer 8 with hidden dimension of 1536 related to the packet inspection scenario.

After validating the tokens identified by clustered features, in neither setting did clustering prove effective in consistently identifying similar features. although certain clusters grouped together features that highlighted semantically related tokens, no systematic approach emerged that could reliably detect them. while some clusters effectively grouped feature that highlight semantically related tokens, no systematic method was found to reliably detected them. This does not implies that the clusters are invalid, but rather that two similar features may be specialized for different tokens.

We then attempted to filter concepts to identify those relevant for specific classes using the class-level TF-IDF: we define the Term Frequency as the frequency of the features activation within a sample and the Document Frequency as the number of samples in which the feature is active in at least one token. This allows us to filter the features that are always active and find features that are more rare and potentially activate only in specific classes of samples. This was successful in the code classification scenario since the dataset was extensive enough, but in the packet

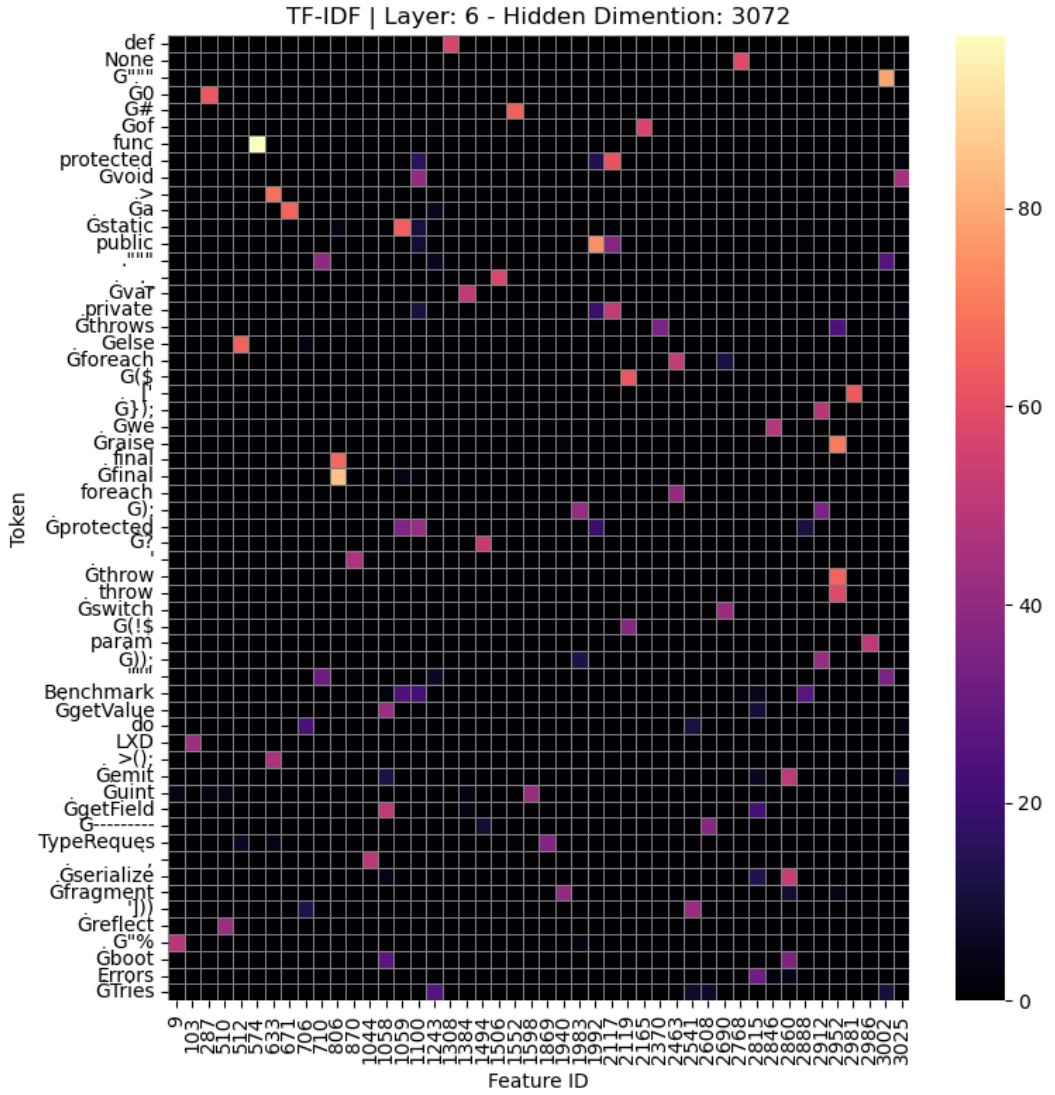

Figure 7.6: Code classification scenario: token-level TF-IDF heatmap of top 1% tokens

inspection use case the only features with high scores where the ones related to the `Overflow` class, which having many repeated tokens obscured other features.

Finally we applied the token level TF-IDF, having as Term Frequency the feature activations of each token (merging all the samples together) and as Document Frequency he number of tokens in which a feature was active. For duplicated tokens, we averaged the TF-IDF score (Figure 7.6). This approach lead to surprising results: in the code classification scenario we expected to obtain a heatmap containing random tokens due to the big variability of the dataset. Instead, we see how in Figure 7.6 many tokens emerged as immediately recognizable. We sorted the tokens based on their highest TF-IDF score, and selected the top 1% to allow an easier visualization. Several domain-relevant keywords consistently appeared among the top-ranked positions across all SAE models trained on this task. Examples include “func”, “def”, “protected”, “public”, and many others, which were repeatedly observed as in the example provided (Figure 7.6). In the packet inspection scenario, we observe

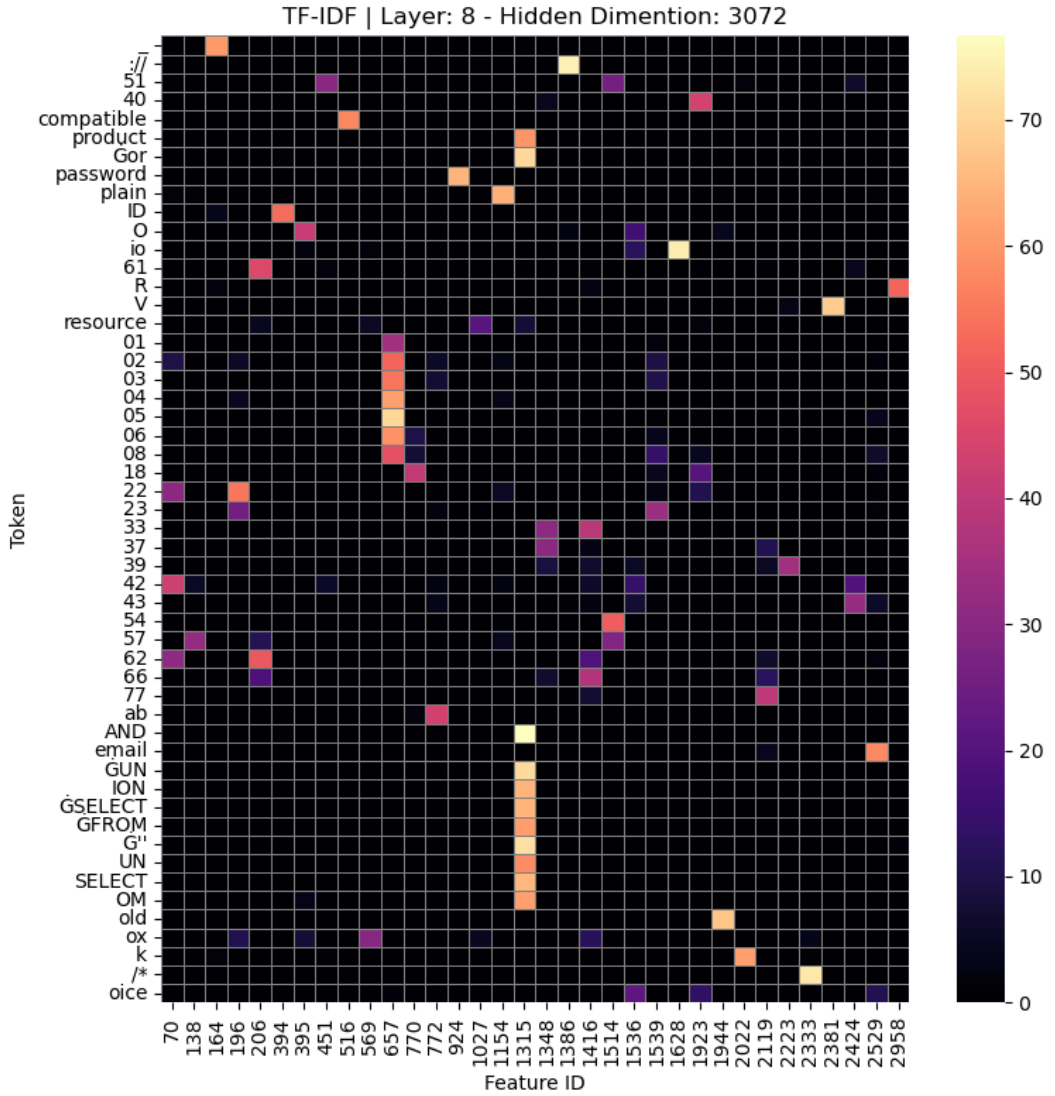

Figure 7.7: Packet inspection scenario: token-level TF-IDF heatmap of top 5% tokens

a comparable outcome: the only difference is that due to the smaller dataset size, we were allowed to visualize the top 5% of the tokens (instead of only 1% as in the code classification scenario). The most frequent tokens across layers and hidden dimensions exhibit strong similarities: starting from layer 8, tokens related to SQL injections begin to emerge consistently, whereas they are absent in layers 6 and 7. This suggest that the model could have learned between those layers the relevance of SQL related tokens. Across all layers, numerical values and hexadecimal strings appear recurrently, with certain features seemingly activating over specific ranges of these tokens. These two patterns, numbers and SQL injection related tokens, indicate that for the RoBERTa model these series carry similar information, however we cannot affirm their importance on the classification.

7.3 Verifying Concepts' Impact

Building on the observation of Section 7.2, we proceeded try verify the impact on model predictions by removing the most important tokens related with specific concepts. From Figures 7.6 and 7.2 we analyze the effect of:

- C/6/1992/3072 for affecting “protected”, “public” and “private”.
- C/6/2952/3072 for affecting “throw” and “raise”.
- P/8/657/3072 for affecting numbers from “01” to “08”.
- P/8/1315/3072 for affecting SQL injection related tokens.

We compared the results of the classification with and without removing any tokens identified by a feature, by confronting the average impact of removing random tokens (Figure 7.8 and 7.9) We choose to remove a fixed number of 50 tokens: SAE features are capable of identifying many different tokens, but to filter the most semantically related we chose to take the 50 most relevant tokens for each analyzed feature. If the feature identifies less tokens we use all of them. In Figure 7.8 and 7.9 we provide the obtained baseline by removing random tokens: in blue, named "Incoming", are shown the average number of predictions that changed from other classes to the one highlighted, while in orange, named "Outgoing", are shown the predictions that changed from the highlighted class to any other shown in the graph.

The results for each SAE feature are shown in Figure 7.10. An important observation is that the tokens identified by the features are not limited to those highlighted in the heatmap, they extend to include many others. Due to the filtering of the token-level TF-IDF, common tokens may not be shown in the first position even with strong activations. In fact we see that feature C/6/2952/3072 includes the tokens “end” and “def”, significantly impacting the classification of the class `Ruby`. Similarly P/8/657/3072 activates strongly also for tokens related to `Code-Execution`, such as “boot”, “echo”, “src”, thereby influencing classification outcomes. Moreover, the class `Info-Disclosure` appears to function as a "safe-zone" for the packet inspection model,

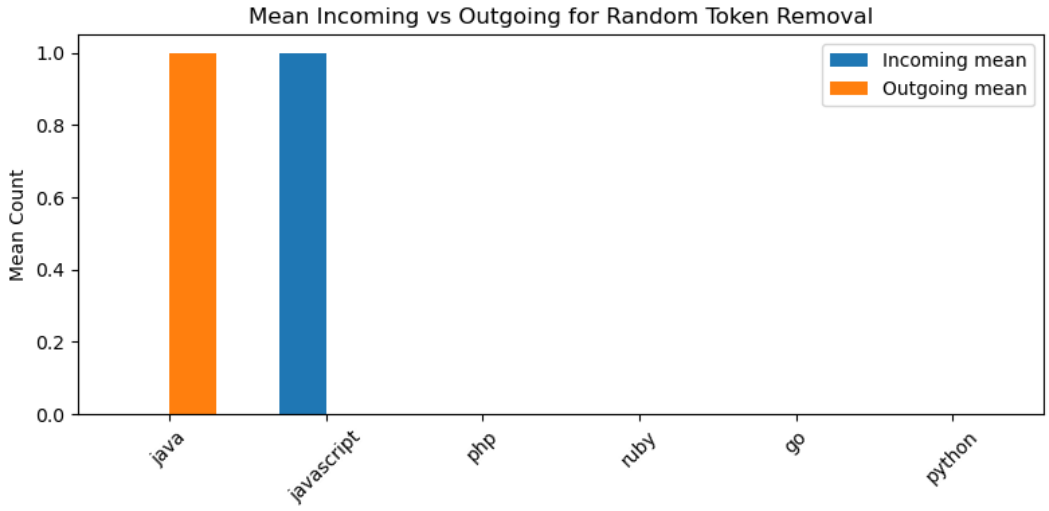

Figure 7.8: Code classification scenario: average impact of random token removal.

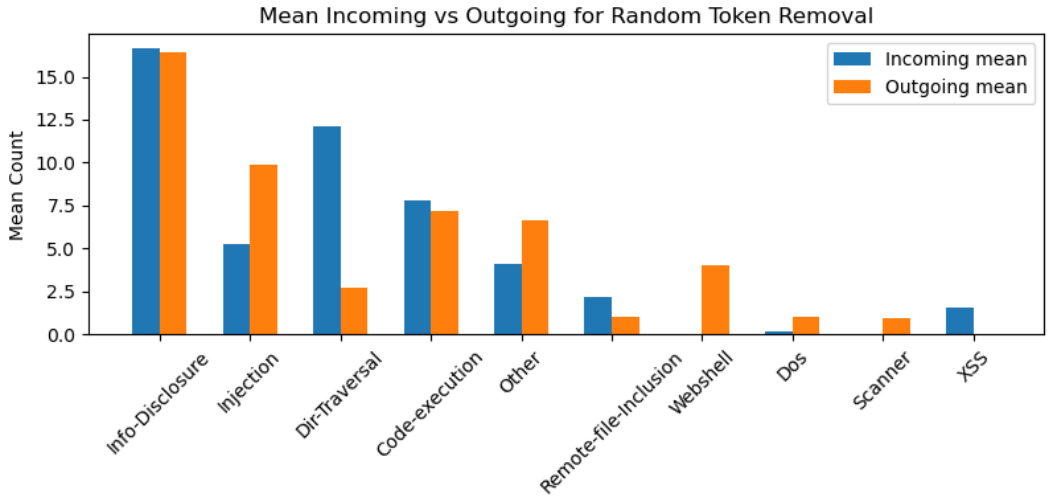

Figure 7.9: Packet inspection scenario: average impact of random token removal.

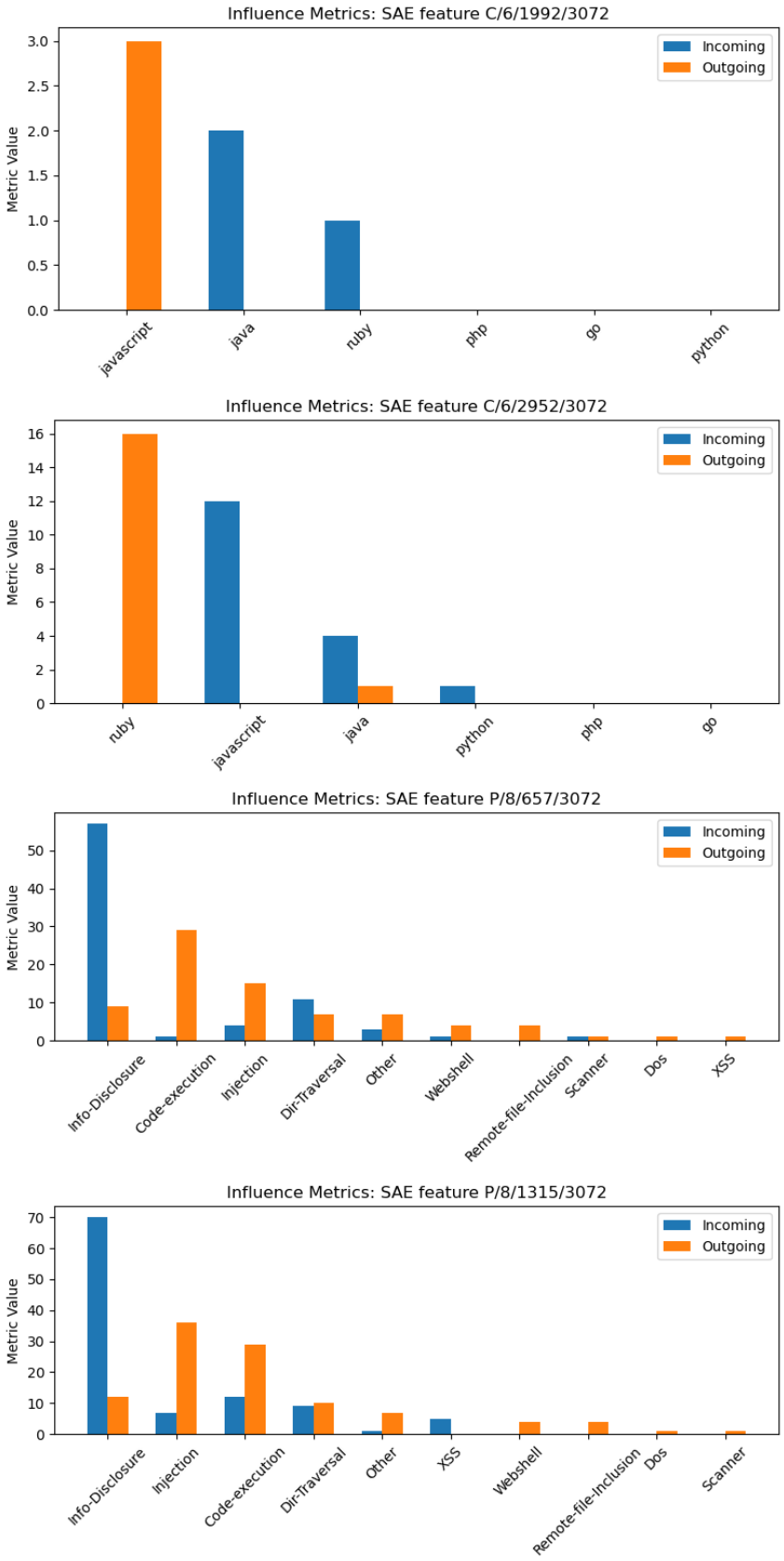

Figure 7.10: Impact of concept-related token removal across selected SAE features.

since even removing random tokens leads to severe degradation in its classification performance.

7.4 Conclusions regarding SAEs

The experiments conducted with Sparse Autoencoders (SAEs) demonstrate both the potential and the limitations of this technique in uncovering latent structures within model activations. Across different layers and hidden dimensions, SAEs avoided producing dead features, even under relatively strict thresholds, leading to consistent results. Larger hidden dimensions increased representational richness, but did not guarantee interpretability, as only a subset of features could be reliably linked to human-understandable concepts.

Attempts to identify meaningful features through clustering revealed that, while some clusters grouped semantically related tokens, no systematic method emerged to consistently detect similar features. This suggests that features may specialize in subtle ways that are not easily captured by unsupervised grouping. In contrast, TF-IDF proved more effective: at both the class and token level, it highlighted domain-relevant tokens such as programming keywords in the code classification scenario and SQL-related tokens in the packet inspection scenario. These findings indicate that TF-IDF filtering can surface interpretable signals, although dataset size and token distribution strongly influence its success.

Finally, the removal experiments confirmed that features identified by SAEs can have measurable impact on downstream classification. Certain features were shown to activate over coherent sets of tokens, and their removal altered predictions in ways that exceeded random baselines. However, the identified features tend to be polysemantic, underlying that the base parameters used, such as the sparsity coefficient of the loss function and the hidden dimensions, might need further exploration.

Chapter 8

Conclusions

This thesis investigated the potential of feature-based techniques and concept-based techniques in uncovering global pattern behavior in Transformer-based language models. The objective was to assess whether existing local explainability methods could provide insights beyond single explanations and compare the outcome against newer more in-depth strategies. Our analysis showed that feature-based explanations such as LIME and SHAP allowed to easily explain single instances allowing to easily impact the model classification by manipulating relevant tokens. Their utility instead proved very limited when exploring a broad range of samples to gather a model overview: in a balanced model, such as in the code classification scenario, the results showed no bias, while, in an unbalanced model, such as in the packet inspection scenario, the technique proved capable of at least finding class biases. TCAV, even if the technique exploits internal representation, proved to be ineffective, failing to correctly learn meaningful CAVs capable of correctly representing human-defined concepts and thus, providing us with relevant information about the models. This could be due to the high variability of Transformer inputs or to the possible absence of human-like concepts. In contrast, SAE proved capable of uncovering relations between tokens and internal activations: it not only was able to highlight important tokens as feature-based techniques, but also revealed structural relations exploited by the model. We discovered through its use that in the code classification scenario, the model presented clearer concepts, with features related to key tokens impacting significantly the classification, while other less important features hindered less the outcome. The outcome on the packet inspection model is instead more complex: we found features with poly-semantic tokens, making it hard to define whether it should be relevant or not for the model, nevertheless the groups of token found revealed high impact on the outcome, confirming the capabilities of the technique. Even if the SAE technique proved proficient, significant limitations emerged: the computational cost of training multiple SAE models is substantial, requiring several days on our systems, while the analysis of a single feature demanded several minutes. Although automation of the pipeline would allow analyzing many features together, it remains computationally expensive and may hinder scalability. In conclusion, feature-based techniques fall short in explaining global behaviors of Transformer

models, and TCAV proved unsuitable for this task. SAE instead proved capable of providing interpretable results to advance the understanding of model reasoning.

Technique	Strengths	Limitations	Overall Outcome
LIME & SHAP	Clear local explanations; easy identification of influential tokens	Limited for global overview; only detects bias in unbalanced models	Useful for local insights, not global behavior
TCAV	Exploits human-defined concepts	Failed to learn meaningful concepts; did not identify relevant tokens	Ineffective for global behavior
SAE	Highlights important tokens; reveals structural relations; supports visualization	High computational cost; scalability issues	Most effective for interpretable results despite complexity

Table 8.1: Compact comparison of explainability techniques applied to Transformer models.

8.1 Future Work

The discovery of Sparse Autoencoders (SAEs) opens several promising research directions. The patterns found through SAE analysis can be used to improve model training by differentiating the datasets, generating new entries that specifically avoid unwanted patterns, or by uncovering biases arising from dataset distributions. Moreover, the ability of feature activations to link tokens with internal representations suggests a path toward model manipulation, where unwanted behavior could be eliminated by directly interfering with the model activation, or conversely, also amplified in case the shortcut are considered valid but with insufficient impact. At the same time, there remain many opportunities for improvement in SAE training itself. In particular, the design of the loss function is still evolving, with multiple variations proposed in recent literature. It is likely that each model architecture will require a tailored loss function to maximize interpretability and efficiency. Further research should therefore investigate adaptive or architecture-specific objectives, as well as strategies to reduce the computational cost of SAE training and inference.

Appendix A

Mathematical Aspects of the Applied Techniques

A.1 T-test

The Student's t-test is a statistical method used to determine whether the means of two groups are significantly different from each other. It is also useful with small sample sizes and when the standard deviation is unknown: for these reasons, it applies perfectly to our case. Formally, the t-statistic is defined as:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{s_p \cdot \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \tag{A.1}$$

where:

- $\bar{x}_1, \bar{x}_2$ are the sample means,
- n_1, n_2 are the sample sizes,
- s_p is the pooled standard deviation, computed as

$$s_p = \sqrt{\frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}}$$

  with s_1^2, s_2^2 being the sample variances.

To verify our results we use the p-value, which is calculated as:

$$p = 2 \cdot (1 - F_{t,\,df}(|t|)) \tag{A.2}$$

where:

- t is the observed test statistic,
- df is the number of degrees of freedom (typically $n_1 + n_2 - 2$ for independent samples),

- $F_{t,\,df}$ is the CDF of the t distribution with df degrees of freedom,
- the factor 2 accounts for the two-tailed nature of the test.

A small p-value means that the difference between the two datasets is unlikely to be attributable to chance, meaning in our case that the concept under analysis is meaningful with respect to the random baseline. We choose 0.001 as threshold following the previous text implementation of TCAV[28].

A.2 TF-IDF

The Term Frequency–Inverse Document Frequency (TF-IDF) is a statistical measure used to evaluate the importance of a term within a collection of documents. It is widely applied in information retrieval and text mining, as it balances the frequency of a term in a document with its rarity across the entire corpus.

Formally, the TF-IDF score of a term t in a document d is defined as:

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \cdot \text{IDF}(t)$$

where:

- Term Frequency (TF) measures how often a term occurs in a document. A common definition is:

$$\text{TF}(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \tag{A.3}$$

  with $f_{t,d}$ being the raw count of term t in document d.

- Inverse Document Frequency (IDF) measures how rare a term is across the corpus:

$$\text{IDF}(t) = \log\left(\frac{N}{1 + n_t}\right) \tag{A.4}$$

  where N is the total number of documents and n_t is the number of documents containing term t.

The idea behind TF-IDF is that terms which are frequent in a specific document but are rare across the corpus, are more meaningful, while terms that appear in many documents (such as stopwords) receive lower scores.

In our scenario is difficult to define a document and a corpus, for this reason we implemented slightly modified versions. We also modified the IDF function to avoid edge cases:

$$\text{IDF}(t) = \log\left(\frac{N + 1}{1 + n_t}\right) + 1 \tag{A.5}$$

Bibliography

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805 (cit. on pp. 1, 5).

[2] Hu Xu, Bing Liu, Lei Shu, and Philip S Yu. "BERT post-training for review reading comprehension and aspect-based sentiment analysis". In: *arXiv preprint arXiv:1904.02232* (2019) (cit. on p. 1).

[3] Keping Yu, Liang Tan, Shahid Mumtaz, Saba Al-Rubaye, Anwer Al-Dulaimi, Ali Kashif Bashir, and Farrukh Aslam Khan. "Securing critical infrastructures: Deep-learning-based threat detection in IIoT". In: *IEEE Communications Magazine* 59.10 (2021), pp. 76–82 (cit. on p. 1).

[4] Maximilian A. Köhl, Kevin Baum, Markus Langer, Daniel Oster, Timo Speith, and Dimitri Bohlender. "Explainability as a Non-Functional Requirement". In: *2019 IEEE 27th International Requirements Engineering Conference (RE)*. 2019, pp. 363–368. DOI: 10.1109/RE.2019.00046 (cit. on p. 1).

[5] *Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 laying down harmonised rules on artificial intelligence (AI Act)*. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32024R1689. Official Journal of the European Union, L 202, 12.7.2024, p. 1–108. 2024 (cit. on p. 1).

[6] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. 2016, pp. 1135–1144 (cit. on pp. 1, 8).

[7] Scott M Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/8a20a8621978632d76c43dfd28b67767-Paper.pdf (cit. on pp. 1, 10).

[8] Been Kim, Martin Wattenberg, Justin Gilmer, Carrie Cai, James Wexler, Fernanda Viegas, and Rory Sayres. *Interpretability Beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV)*. 2018. arXiv: 1711.11279 [stat.ML]. URL: https://arxiv.org/abs/1711.11279 (cit. on pp. 1, 7, 12, 13, 40).

[9] Hoagy Cunningham, Aidan Ewart, Logan Riggs, Robert Huben, and Lee Sharkey. *Sparse Autoencoders Find Highly Interpretable Features in Language Models*. 2023. arXiv: 2309.08600 [cs.LG]. URL: https://arxiv.org/abs/2309.08600 (cit. on pp. 1, 13).

[10] Yuqi Zhao, Giovanni Dettori, Matteo Boffa, Luca Vassio, and Marco Mellia. "The Sweet Danger of Sugar: Debunking Representation Learning for Encrypted Traffic Classification". In: *Proceedings of the ACM SIGCOMM 2025 Conference*. SIGCOMM '25. São Francisco Convent, Coimbra, Portugal: Association for Computing Machinery, 2025, pp. 296–310. ISBN: 9798400715242. DOI: 10.1145/3718958.3750498. URL: https://doi.org/10.1145/3718958.3750498 (cit. on p. 3).

[11] A Carlisle Scott, William J Clancey, Randall Davis, and Edward H Shortliffe. *Explanation capabilities of production-based consultation systems*. 1977 (cit. on p. 3).

[12] Feiyu Xu, Hans Uszkoreit, Yangzhou Du, Wei Fan, Dongyan Zhao, and Jun Zhu. "Explainable AI: A Brief Survey on History, Research Areas, Approaches and Challenges". In: *Natural Language Processing and Chinese Computing*. Ed. by Jie Tang, Min-Yen Kan, Dongyan Zhao, Sujian Li, and Hongying Zan. Cham: Springer International Publishing, 2019, pp. 563–574. ISBN: 978-3-030-32236-6 (cit. on p. 3).

[13] Naomi Saphra and Sarah Wiegreffe. *Mechanistic?* 2024. arXiv: 2410.09087 [cs.AI]. URL: https://arxiv.org/abs/2410.09087 (cit. on p. 3).

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762 (cit. on p. 4).

[15] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: 1907.11692 [cs.CL]. URL: https://arxiv.org/abs/1907.11692 (cit. on p. 5).

[16] Yonghui Wu et al. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. 2016. arXiv: 1609.08144 [cs.CL]. URL: https://arxiv.org/abs/1609.08144 (cit. on p. 6).

[17] Alon Jacovi. *Trends in Explainable AI (XAI) Literature*. 2023. arXiv: 2301.05433 [cs.AI]. URL: https://arxiv.org/abs/2301.05433 (cit. on p. 6).

[18] Cynthia Rudin. *Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead*. 2019. arXiv: 1811.10154 [stat.ML]. URL: https://arxiv.org/abs/1811.10154 (cit. on p. 7).

[19] Rudresh Dwivedi et al. "Explainable AI (XAI): Core Ideas, Techniques, and Solutions". In: *ACM Comput. Surv.* 55.9 (Jan. 2023). ISSN: 0360-0300. DOI: 10.1145/3561048. URL: https://doi.org/10.1145/3561048 (cit. on p. 7).

[20] Mirko Cesarini, Lorenzo Malandri, Filippo Pallucchini, Andrea Seveso, and Frank Xing. "Explainable AI for Text Classification: Lessons from a Comprehensive Evaluation of Post Hoc Methods". In: *Cognitive Computation* 16 (Aug. 2024), pp. 3077–3095. DOI: 10.1007/s12559-024-10325-w (cit. on p. 7).

[21] Ahmed M Salih, Zahra Raisi-Estabragh, Ilaria Boscolo Galazzo, Petia Radeva, Steffen E Petersen, Karim Lekadir, and Gloria Menegaz. "A perspective on explainable artificial intelligence methods: SHAP and LIME". In: *Advanced Intelligent Systems* 7.1 (2025), p. 2400304 (cit. on p. 7).

[22] Diogo Gaspar, Paulo Silva, and Catarina Silva. "Explainable AI for intrusion detection systems: LIME and SHAP applicability on multi-layer perceptron". In: *IEEE Access* 12 (2024), pp. 30164–30175 (cit. on p. 7).

[23] Viswan Vimbi, Noushath Shaffi, and Mufti Mahmud. "Interpreting artificial intelligence models: a systematic review on the application of LIME and SHAP in Alzheimer's disease detection". In: *Brain Informatics* 11.1 (2024), p. 10 (cit. on p. 7).

[24] Melkamu Mersha, Mingiziem Bitewa, Tsion Abay, and Jugal Kalita. *Explainability in Neural Networks for Natural Language Processing Tasks*. 2025. arXiv: 2412.18036 [cs.CL]. URL: https://arxiv.org/abs/2412.18036 (cit. on p. 7).

[25] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. *What Does BERT Look At? An Analysis of BERT's Attention*. 2019. arXiv: 1906.04341 [cs.CL]. URL: https://arxiv.org/abs/1906.04341 (cit. on p. 7).

[26] Antonio De Santis, Riccardo Campi, Matteo Bianchi, and Marco Brambilla. "Visual-tcav: concept-based attribution and saliency maps for post-hoc explainability in image classification". In: *arXiv preprint arXiv:2411.05698* (2024) (cit. on p. 7).

[27] RR Rejimol Robinson, Rendhir R Prasad, Ciza Thomas, and N Balakrishnan. "Validating network attack concepts: A TCAV-driven approach". In: *Journal of Computer Virology and Hacking Techniques* 20.4 (2024), pp. 841–855 (cit. on p. 7).

[28] Isar Nejadgholi, Kathleen Fraser, and Svetlana Kiritchenko. "Improving Generalizability in Implicitly Abusive Language Detection with Concept Activation Vectors". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Smaranda Muresan, Preslav Nakov, and Aline Villavicencio. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 5517–5529. DOI: 10.18653/v1/2022.acl-long.378. URL: https://aclanthology.org/2022.acl-long.378/ (cit. on pp. 7, 18, 19, 62).

[29] Amirata Ghorbani, James Wexler, James Y Zou, and Been Kim. "Towards automatic concept-based explanations". In: *Advances in Neural Information Processing Systems*. 2019, pp. 9273–9282 (cit. on p. 7).

[30] Leo Gao, Tom Dupré la Tour, Henk Tillman, Gabriel Goh, Rajan Troll, Alec Radford, Ilya Sutskever, Jan Leike, and Jeffrey Wu. "Scaling and evaluating sparse autoencoders". In: *arXiv preprint arXiv:2406.04093* (2024) (cit. on p. 7).

[31] Adly Templeton et al. "Scaling Monosemanticity: Extracting Interpretable Features from Claude 3 Sonnet". In: *Transformer Circuits* (May 2024). Anthropic Research. URL: https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html (cit. on pp. 7, 13).

[32] Guilliermo Owen. "Values of Games with a Priori Unions". In: *Mathematical Economics and Game Theory*. Ed. by Rudolf Henn and Otto Moeschlin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1977, pp. 76–88. ISBN: 978-3-642-45494-3 (cit. on p. 11).

[33] S. López and Martha Saboyá. "On the relationship between Shapley and Owen values". In: *Central European Journal of Operations Research* 17 (Dec. 2009), pp. 415–423. DOI: 10.1007/s10100-009-0100-8 (cit. on p. 12).

[34] L. Itti, C. Koch, and E. Niebur. "A model of saliency-based visual attention for rapid scene analysis". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.11 (1998), pp. 1254–1259. DOI: 10.1109/34.730558 (cit. on p. 12).

[35] Tom Conerly, Adly Templeton, Trenton Bricken, Jonathan Marcus, and Tom Henighan. *Update on How We Train SAEs*. Anthropic Interpretability Team, Circuits Updates - April 2024. Apr. 2024. URL: https://transformer-circuits.pub/2024/april-update/index.html#training-saes (cit. on pp. 14, 20).

[36] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. "A Primer in BERTology: What We Know About How BERT Works". In: *Transactions of the Association for Computational Linguistics* 8 (2020). Ed. by Mark Johnson, Brian Roark, and Ani Nenkova, pp. 842–866. DOI: 10.1162/tacl_a_00349. URL: https://aclanthology.org/2020.tacl-1.54/ (cit. on p. 20).

Dedications

I would like to thank all of my friends who have been close to me and that have always given me a smile: Lorenzo Filomena, Marta Gambarotta, Riccardo Rosin, Filippo Gerbino and Angelo Chavez. I would like to thank Asia Ferri for being by my side I would also like to thank Kurram Arshad and Matteo Saglimbeni for being a solid anchor through the years. I would also like to thank all my family, my father, my mother, my grandmothers Irma Raie and Loretta Anteghini for constantly giving me love and support. Lastly I would like to thank my supervisors Idilio Drago and Luca Vassio for guiding me through this journey and their important feedback.