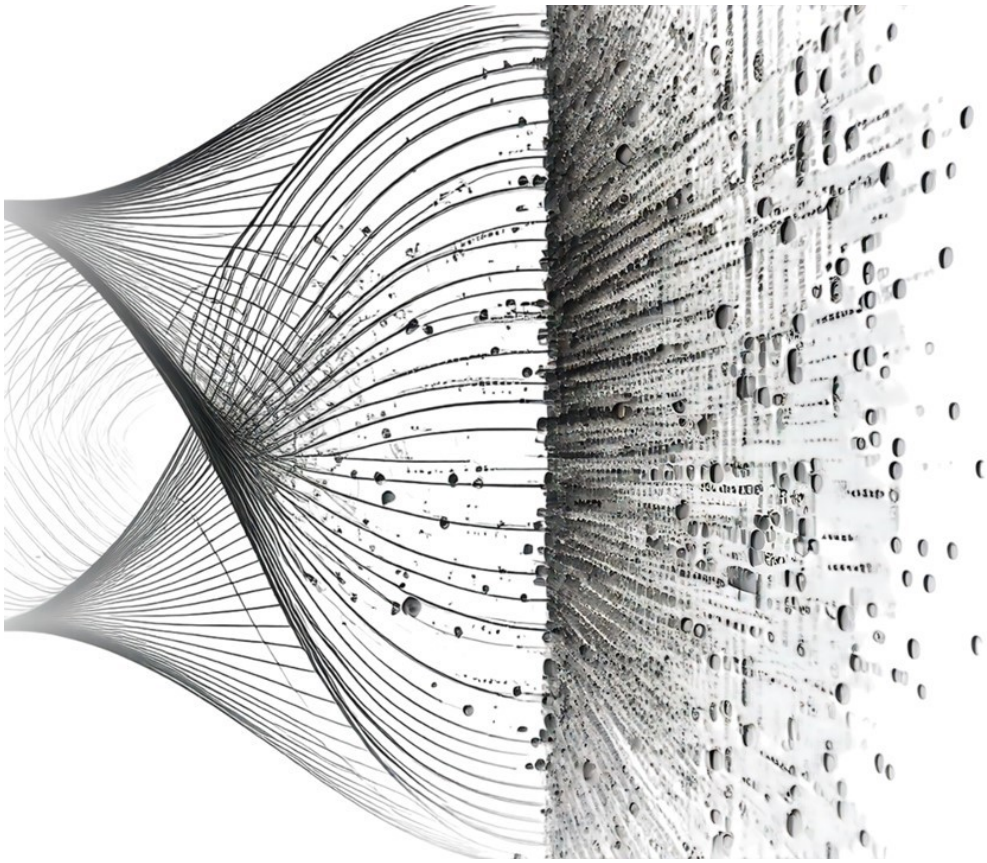

Optimising Large Language Models for Vehicle Classification

A study on machine learning for large-scale applications and implementation in the insurance industry

Master's thesis in Engineering Mathematics and Computational Science

FILIP HJÄRTSTRÖM & SIMON JONÄNGEN

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Master's thesis 2025

Optimising Large Language Models for Vehicle Classification

A study on machine learning for large-scale applications and implementation in the insurance industry

FILIP HJÄRTSTRÖM & SIMON JONÄNGEN

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Applied Mathematics and Statistics
Chalmers University of Technology
Gothenburg, Sweden 2025

Optimising Large Language Models for Vehicle Classification
A study about machine learning for large-scale applications and implementation in the insurance industry
FILIP HJÄRTSTRÖM & SIMON JONÄNGEN

Supervisor: Jonas Meddeb, Data Scientist, If P&C Insurance Company Ltd.
Supervisor: Oskar Molin, Pricing Analyst, If P&C Insurance Company Ltd.
Examiner: Tobias Gebäck, Senior Professor, Department of Mathematical Sciences

Master's Thesis 2025
Department of Mathematical Sciences
Applied Mathematics and Statistics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Wind visualization constructed in Matlab showing a surface of constant wind speed along with streamlines of the flow.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2025

Optimising Large Language Models for Vehicle Classification
A study on machine learning for large-scale applications and implementation in the insurance industry
FILIP HJÄRTSTRÖM & SIMON JONÄNGEN
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

This thesis examines the use of Large Language Models optimised for vehicle classification within the insurance industry. Traditional methods at If P&C Insurance suffer from inaccuracies and scalability issues due to unstructured text data from manual inputs and relatively basic techniques. To address this, our study utilises Parameter-Efficient Fine-Tuning and Low-Rank Adaptation on standardised, manually labelled vehicle data. Our approach combined careful prompt engineering, dataset preprocessing, hyperparameter optimisation via the Nondominated Sorting Genetic Algorithm II, and thorough evaluation across different base models, including various Llama model sizes, DeepSeek, Mistral, and Phi. Through this integrated methodology, we achieved a final model accuracy of 96.8% on hold-out data, using a fine-tuned Llama 3.1 model with 8 billion parameters. Despite the model's relatively modest size, targeted adaptation enabled it to outperform larger proprietary models such as GPT-4o on the specific classification task. Implementation aspects, including computational needs, cost efficiency, and human-in-the-loop strategies, are also discussed. Our deployment framework emphasises selective human review based on model confidence, enabling a sustainable balance between automation and accuracy. Financial and infrastructure considerations showed that fine-tuned open-source models could offer significant cost savings compared to API-based solutions. In conclusion, this research presents a scalable, cost-effective, and high-performing solution that enhances vehicle classification, leading to improved risk segmentation and pricing precision in the insurance sector. The results demonstrate that fine-tuned open-source LLMs, when carefully adapted, can rival and even surpass much larger commercial models in domain-specific applications, offering a viable path for modernising traditional insurance workflows.

Keywords: AI, LLM, PEFT, LoRA, machine learning, supervised learning, unsupervised learning, big data, classification, risk-based pricing.

Acknowledgements

We would like to extend our sincere gratitude to our supervisors at If P&C Insurance, Oskar Molin and Jonas Meddeb, for their invaluable guidance, continuous support, and insightful feedback throughout this project. We are also grateful to Tobias Gebäck, our supervisor and examiner at Chalmers University of Technology, for his expertise, valuable advice, and encouragement. Additionally, we thank If P&C Insurance for hosting our research and providing the necessary computational resources essential for the completion of our thesis.

Filip Hjärtström and Simon Jonängen, Gothenburg, May 2025

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

LLM	Large Language Model
FT	Fine-tuning
PEFT	Performance Efficient Fine-tuning
LoRA	Low-Rank Adaptation
ML	Machine Learning
AI	Artificial Intelligence
GA	Genetic Algorithm

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background and Motivation

Risk-based pricing is a fundamental principle in the insurance industry and refers to the practice of setting insurance premiums proportionate with the risk profile of each insured. In essence, insurers charge different premiums for the same coverage based on risk factors specific to the insured individual or asset. This approach relies on statistical analysis of historical data to predict expected claim costs for a given set of risk characteristics [1, 2]. By estimating correlations between past losses and observable attributes (e.g., age, location, vehicle type), insurers can determine an appropriate premium that reflects the likelihood and magnitude of future claims [1, 3]. The fundamental goal is to charge an "accurate" price for insurance, one that closely aligns with the insured's expected losses [1, 4].

In automobile insurance, insurers consider a range of key risk factors when determining premiums. These factors typically include characteristics of the driver, the vehicle, and usage of the vehicle. Common driver-related rating variables are age, driving experience, driving record (history of accidents or violations), and sometimes credit score or socio-economic indicators [2]. On the vehicle side, characteristics of the insured automobile are critical in pricing. Insurers note the vehicle's make, model, year of manufacture, engine size or power, body type, safety features, and value, among other attributes [1]. Vehicle model, which can affect both the probability of an accident and the expected severity of a claim, will be the focus of this paper. For instance, high-performance sports cars may encourage faster driving and thus higher accident frequency, whereas vehicles with advanced safety features may mitigate injury severity [3].

If P&C Insurance is the leading provider of insurance solutions to both private customers and corporate clients in the Nordic region. One of If's central offerings is automobile insurance for private individuals. Currently, the company faces a significant challenge: the received vehicle model names data for registered vehicles are entered manually as free-form text from the vendor, resulting in inconsistent and non-standardized data entries. This issue is a well-known challenge across the industry, where unstructured and vendor-specific descriptions complicate vehicle classification efforts [J.D. Power, 2023]. For instance, the model "Volvo V70" can appear in numerous variations, and occasionally, essential model information, such as "V70," is omitted entirely. See Table 1.1 for examples of what the data may look

like. Given the vast number of vehicle models on Swedish roads, coupled with the constant introduction of new models each year, the existing classification approach has become inadequate [5, 6].

Table 1.1: Example data

Raw Data	Actual Vehicle Model
cee'd_sw 16 crdi ko	Kia Ceed
tfr 69 p	Opel Campo
ba	Renault Megane Scenic
megane scenic	Renault Megane Scenic
1441341	Volvo 140

Furthermore, the growing volume of data and the increasing complexity of modern pricing methods underline the need for a more scalable and intelligent classification solution [Wilson et al., 2024]. The current classification system employed by If relies heavily on rudimentary methods like regular expressions, which are outdated and limited in their effectiveness. Considering the substantial advancements in machine learning over the past decade, and specifically the introduction of large language models (LLMs), there is now an opportunity to significantly upgrade and modernize this methodology [7, 8].

Recent research has demonstrated that LLMs can be highly effective in parsing, understanding, and structuring unstructured text inputs such as free-form accident reports or insurance claims [Zappa et al., 2021; Li et al., 2025]. These findings suggest that similar techniques could be applied to classify vehicle model names with higher accuracy and lower maintenance demands compared to traditional rule-based systems. In particular, leveraging embeddings and semantic analysis provided by LLMs may allow for better handling of incomplete, abbreviated, or miswritten vehicle descriptions [8].

1.2 Problem Definition

The outdated classification model leads to suboptimal risk segmentation and imprecise pricing, potentially resulting in unfair premiums or financial losses. The objective of this project is to develop a more accurate and scalable classification model, enabling more granular segmentation and improved risk-based pricing.

1.3 Aim and Scope

The aim of this project is to improve If P&C Insurance's vehicle model classification system by developing a more accurate and scalable solution using machine learning and large language models. The enhanced classification model will enable better customer segmentation, thereby supporting more precise and risk-based pricing for

car insurance.

The scope includes data preprocessing, model development, and performance evaluation using historical vehicle data provided by If and external sources. The study is limited to the classification component of the pricing model and does not involve the full actuarial pricing process, real-time deployment, or direct integration with production systems. A detailed overview and analysis of how the deployment and strategy management could be structured will be considered.

1.4 Large Language Model

LLMs have emerged as a transformative AI paradigm, consisting of models pretrained on vast amount of data and adaptable to a wide array of downstream tasks [9]. Notable instances include OpenAI's GPT series and Meta's Llama, which contain tens of billions of parameters and are trained on trillions of tokens of data [10, 11]. Initially developed for language understanding and generation, these GPT-style and Llama models now serve as general-purpose engines in numerous domains. With appropriate scaling and prompting, LLMs exhibit remarkable abilities across domains [9]. This broad applicability has led to LLM deployments in everything from search engines and chatbots to document summarization and beyond, underscoring their status as a major advancement in AI text comprehension and generation technology.

In recent years, researchers have begun to expand LLMs into structured data applications. Studies are evaluating how well these models can interpret tabular or otherwise structured inputs [12], and several works have applied LLMs to classification and entity recognition problems in specialized domains [13]. For example, GPT-4 has been used to automatically generate intuitive labels and descriptions for clusters in a geodemographic segmentation study [14] – a novel use of an LLM to interpret and explain data-driven groupings. These developments suggest that LLMs can serve as powerful tools for tasks like categorization and information extraction, effectively acting as general-purpose classifiers or knowledge bases even outside traditional natural language processing (NLP) settings. Given this potential, it is natural to investigate applying LLM technology to the problem at hand. This thesis explores the role of state-of-the-art LLMs as an innovative approach to vehicle classification, bridging the gap between unstructured intelligence and structured data analysis in a formal pricing framework.

1.5 Use of AI Tools

Throughout the writing process of this report, AI tools were occasionally used to assist with grammar. These tools supported improvements in sentence structure, clarity, and overall readability. However, all critical thinking, argumentation, and content development have been done by the writers, and the AI support has been limited to language enhancement only, rather than idea generation.

2

Theory

2.1 Large Language Models

LLMs achieve remarkable capabilities by combining several key components in a coherent architecture. Initially, embeddings are used to represent tokens as dense high-dimensional vectors. These embeddings are then processed through a transformer architecture, which applies self-attention mechanisms and feed-forward networks to model complex contextual relationships. Finally, the output is projected back into the vocabulary space through an unembedding layer, with a softmax function producing a probability distribution over possible next tokens [15].

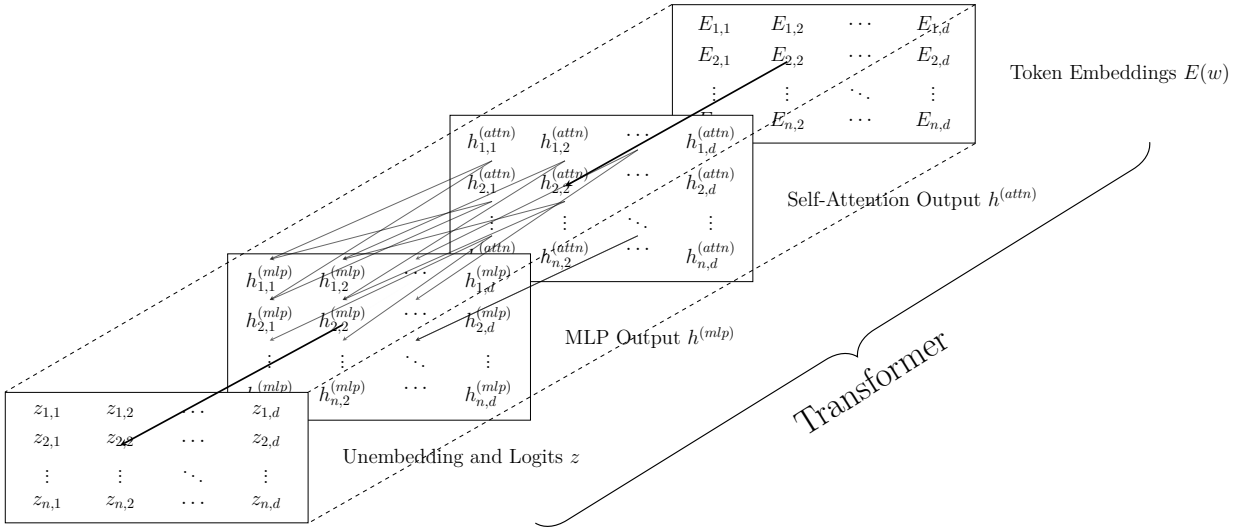

Figure 2.1: Illustration of the Transformer architecture.

2.1.1 Embeddings

Embeddings map discrete tokens into continuous vector spaces that capture semantic relationships and syntactic properties [16, 17]. Mathematically, an embedding function is defined as:

$$E : \mathcal{V} \to \mathbb{R}^d$$

where $\mathcal{V}$ is the vocabulary and d is the embedding dimension [18]. These embeddings are trainable parameters, optimized during model pretraining to capture linguistic

patterns [19]. In modern architectures such as GPT, BERT, and RoBERTa, embeddings are enhanced with positional encodings to incorporate word order information [20, 21]. This provides an insight in not just the word itself, but also in what context they are used.

2.1.2 Transformer Architecture

The transformer serves as the core processing unit of modern LLMs [20]. It consists of repeated blocks that refine the token representations through attention and feed-forward computations.

2.1.2.1 Self-Attention Mechanism

Self-Attention is the core mechanism that enables transformers to dynamically model relationships between all tokens in a sequence. It operates by projecting the input embeddings into three distinct matrices: Q (queries), K (keys) and V (values).

Each token embedding is linearly transformed to produce its own query, key, and value vector. Formally, for an embedding matrix $E \in \mathbb{R}^{n \times d}$ (where n is the number of tokens and d the embedding size). Here $\mathbb{R}^{n \times d}$ contains the entire 'sentence' we have provided the model with. This sequence of words naturally forms a matrix of size $n \times d$. We compute:

$$Q = EW_Q, \quad K = EW_K, \quad V = EW_V$$

where W_Q, W_K, and W_V are learned weight matrices of shape $d \times d_k$ [20].
These projections define different roles for each token:

Table 2.1: Roles of Q, K and V matrices in the Self-Attention mechanism.

Component	Role
Q	Determines what a token is querying: "what information am I looking for?"
K	Represents how a token presents itself: "what information do I contain?"
V	Contains the actual information the token provides to others during attention computation.

The core self-attention operation computes how much attention each token should pay to every other token, based on the similarity between queries and keys:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

$QK^\top$ computes similarity scores between tokens. The softmax normalizes these scores into probabilities, see section 2.1.2.4. These probabilities are used to weigh the value vectors V, producing a new context-aware representation for each token.

Thus, each output vector is a weighted combination of all input tokens' value vectors, allowing the model to aggregate information across the entire sequence. It is important to note that the product $QK^\top$ produces a set of similarity scores representing the dot products between the query and key vectors. These scores are then used to determine attention weights in the model. Conceptually, this can be interpreted as: *the next predicted token should be the one that, based on the input, shares the highest contextual similarity (or historical identity) with the current token.*

Table 2.2: Flow of data through the Transformer architecture corresponding to Figure 2.1.

Step	Description
1	Embeddings $E(w)$ are produced for each input token, mapping discrete tokens into continuous vector spaces.
2	The embeddings are projected into three matrices: Queries Q, Keys K, and Values V via learned linear transformations.
3	Self-Attention computes pairwise relationships by matching queries against keys to generate attention scores.
4	The values V are aggregated according to the computed attention scores, producing context-aware representations $h^{(\text{attn})}$.
5	The resulting context-enriched representations $h^{(\text{attn})}$ are passed into a Multi-Layer Perceptron (MLP) block for further transformation.

Thus, Self-Attention is the process that transforms the plain embeddings into contextualized embeddings, allowing each token to gather relevant information from the entire input sequence.

2.1.2.2 Feed-Forward Networks

After the self-attention mechanism produces context-aware representations for each token denoted as $h^{(\text{attn})}$ these vectors are passed through a position-wise feed-forward neural network (FFN). The FFN operates independently on each position (i.e., each token's representation), applying the same weights across all positions. The input x to the FFN is thus the context-enriched embedding produced by the attention layer for a specific token:

$$\text{FFN}(x) = W_2 \,\text{SeLU}(W_1 x)$$

Here, $x \in \mathbb{R}^d$ represents the updated embedding for a single token after self-attention, and $W_1 \in \mathbb{R}^{d \times d_{\text{ff}}}$, $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d}$ are learned weight matrices. The activation function, SeLU (Scaled Exponential Linear Unit), introduces nonlinearity, enabling the network to model more complex interactions [20].

This feed-forward step allows each token's representation to be further refined by capturing higher-level features, independent of other tokens but based on the globally informed embedding it received from the attention mechanism.

2.1.2.3 Unembedding and Output Projection

After the input tokens have been processed through multiple layers of self-attention and feed-forward networks, each token is associated with a rich, context-aware hidden state. However, these hidden states are still in an internal model space and not directly understandable as vocabulary words. To transform these hidden states back into actual words or tokens, the Transformer applies an unembedding layer:

Table 2.3: Workflow of the Unembedding and Output Projection process in the Transformer.

Step	Description
1	**Final Hidden States**: After processing through multiple Transformer layers, the model produces contextually rich hidden representations H for each token. These vectors encode semantic and syntactic information from the entire sequence.
2	**Projection into Vocabulary Space**: The hidden states are projected into a space where each dimension corresponds to a vocabulary token by multiplying them with a learned weight matrix $W^\top$ [20].
3	**Weight Tying**: Typically, the weight matrix W is tied to the input embedding matrix, a method called weight tying, which reduces model parameters and improves generalization [22, 23].
4	**Output**: The resulting logits Z are passed through a softmax function to generate a probability distribution over the vocabulary [24, 25].

In Figure 2.1, this unembedding step corresponds to the transition from the MLP Output $h^{(\text{mlp})}$ to the Unembedding and Logits block, preparing the final outputs for softmax normalization and token prediction.

2.1.2.4 Softmax Function

Softmax is a function that converts raw model outputs (logits) into probabilities by normalizing them into a probability distribution [18]. In the context of LLMs, softmax is primarily used in the final output layer to determine the likelihood of each token in a vocabulary given the input context. Mathematically, given a vector of raw logits $z = [z_1, z_2, ..., z_n]$, the softmax function is defined as:

$$\sigma_i(z) = \frac{e^{z_i}}{\sum_{j=1}^{|\mathcal{V}|} e^{z_j}}$$

where e^{z_i} represents the exponentiation of each logit value. The softmax function ensuring that the output probabilities across the vocabulary sum to one [26, 24]. The largest value in z receives disproportionately higher importance because the exponential function grows rapidly, increasing differences between values.

2.2 Fine-tuning an LLM with Low-Rank Adaptation

The three main methods of enhancing LLMs is retrieval-augmented generation, prompt engineering and fine-tuning. Fine-tuning is by far the most resource-intensive since the training processes and data preparations are so computationally heavy and time-consuming. It also has the potential to generate the most accurate models. [27]

Low-Rank Adaptation (LoRA) is a fine-tuning methodology designed for efficiently adapting LLMs, rooted in linear algebra and optimization theory. It addresses computational limitations associated with traditional fine-tuning by constraining the parameter updates to a lower-dimensional subspace, drastically reducing computational cost and enhancing scalability [28].

Formally, LoRA operates through the introduction of a structured low-rank decomposition to the update step of the pre-trained model's weight matrices [28]. Consider a pre-trained model with weight matrix $W \in \mathbb{R}^{d\times k}$, representing parameters in fully connected or attention layers. LoRA applies a low-rank update ΔW defined as:

$$\Delta W = BA,$$

where $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$ are matrices with a predefined rank $r \ll \min(d, k)$. These matrices encapsulate a low-rank approximation of parameter updates, effectively projecting the high-dimensional parameter space onto a much lower-dimensional space [28].

During fine-tuning, the original weights W remain constant, and only matrices A and B undergo optimization [28]. Thus, the optimization objective of fine-tuning using LoRA can be formally expressed as minimizing a loss function $\mathcal{L}$ over a dataset D:

$$\min_{A,B} \mathcal{L}(W + BA; D).$$

Due to the extensive number of hyperparameters involved in both LoRA and conventional training, efficiently identifying high-performing configurations becomes a complex, multi-objective problem. To address this, evolutionary algorithms offer a promising solution due to their robustness in navigating large and intricate search spaces.

2.3 Nondominated Sorting Genetic Algorithm II

Genetic Algorithms (GAs) are a class of evolutionary optimization techniques inspired by natural selection and genetics [29]. A GA maintains a population of candidate solutions which evolves over successive iterations called generations. In each generation, individuals are evaluated by a fitness function, and the algorithm uses genetic operators to create a new population of potentially better solutions.

This makes GAs well-suited for complex or poorly understood search landscapes.

Multi-objective optimization extends the GA concept to problems with multiple conflicting objectives, where the goal is to approximate the Pareto-optimal front, which is the set of solutions where no objective can improve without worsening another, rather than a single optimum [30]. NSGA-II builds on the basic GA framework with specialized mechanisms to handle multiple objectives effectively. Its key innovations include:

- Fast non-dominated sorting: NSGA-II ranks individuals by Pareto dominance in an efficient manner and reducing complexity to $\mathcal{O}(MN^2)$ for M objectives and population size N [30].
- Crowding distance for diversity: Within each Pareto front, NSGA-II introduces a crowding distance metric to estimate how isolated a solution is relative to its neighbors in objective space [31]. Solutions in less crowded regions of the front get a higher selection priority, encouraging a well-spread distribution of solutions along the Pareto front.
- Elitist selection: NSGA-II uses an elitism approach, which means that it retains top-performing solutions across generations to prevent losing optimal individuals. It does this by combining the parent and offspring populations and then selecting the top N individuals for the next generation based on Pareto rank and crowding distance [30, 31].

While NSGA-II is slightly more computationally heavy than basic GAs because of sorting and crowding distance calculations, it is suitable for a complex task such as hyper-parameter search for LLM tuning with LoRA configuration, where multiple criteria must be optimized simultaneously and fast convergence is ideal.

3
Methods

3.1 Python Libraries

Python libraries Unsloth, Optuna, and TRL were employed to facilitate and optimize the fine-tuning pipeline. Unsloth is an open-source Python framework built on the Hugging Face Transformers library that streamlines the fine-tuning of LLMs such as Llama, Mistral, and Gemma by providing a memory-efficient, high-speed training interface and supporting Parameter-Efficient Fine-Tuning (PEFT) methods like LoRA, thereby enabling fine-tuning with minimal computational resources. According to its documentation, Unsloth can reduce memory usage by up to 70% and shorten training time by up to 30-fold compared to traditional full-parameter fine-tuning approaches [32]. The framework also integrates seamlessly with the Hugging Face Hub, allowing direct loading of pre-trained models (such as Llama 2, Llama 3, and Mistral) for fine-tuning on custom datasets without extensive engineering overhead, and its API remains fully compatible with PyTorch [33].

To identify optimal training configurations, Optuna was utilized for automated hyperparameter optimization. The library enables efficient configuration of search space and application of an appropriate search algorithm. For the supervised fine-tuning stage, the Transformers Reinforcement Learning library's SFTTrainer module provided a structured training framework that facilitated integration with Hugging Face datasets, simplified training management, and enabled efficient tracking of performance metrics and checkpoints throughout the fine-tuning cycle. Finally, all training computations were executed on GPU hardware, since GPUs are designed for highly parallel computation and offer significantly higher memory bandwidth than CPUs, substantially accelerating deep learning tasks. As a result, training workloads that might require days on a CPU can often be completed within hours on a GPU, and modern deep learning frameworks such as PyTorch and TensorFlow are optimized for GPU acceleration and work seamlessly with the fine-tuning libraries [34].

3.2 Training Data

To support effective fine-tuning of the base language model, a dataset consisting of 1,500 high-quality, hand-labelled rows was constructed. This aligns with the recommendations provided by Unsloth, which suggest a minimum of 1,000 well-curated examples for tuning base models and approximately 300 examples for instruction-

tuned variants. Each data row was carefully annotated to ensure clarity, consistency, and relevance to the target task, thereby enhancing the model's ability to generalize. The manual labelling process allowed for precise control over the quality and structure of the training data, which is critical when working with limited yet impactful datasets in PEFT scenarios.

To ensure the dataset accurately reflected the distribution of the target domain, we stratified our sampling based on car brand. This approach helped us maintain a representative distribution across brands, reducing the risk of bias and improving the model's performance on under-represented segments.

3.2.1 Preprocessing

The raw vehicle model data is distributed across three columns: Make, Vehicle Designation, and Trade Name. Due to inconsistencies in how make and model information is recorded, such as the make appearing in the Vehicle Designation or the model in the Trade Name, standardization is necessary. All three columns are concatenated into a single string, converted to lowercase, and processed to remove duplicate words. This ensures each make and model appears only once, regardless of its original placement. The final string is then cleaned by removing unnecessary double spaces and inappropriate commas, such as those not located between digits.

Table 3.1: Result of preprocessing

Make	Vehicle Designation	Trade Name	Cleaned Data
Volvo	a + s80	S80	volvo a s80
Volkswagen	volkswagen,	kombi	volkswagen kombi
Tesla		Modely Y	tesla model y

3.3 Base Models

n initial decision was made to utilize a Llama model with 8 billion parameters, guided by insights from the literature, considerations regarding model size, and performance benchmarks. Recent studies have indicated that smaller pre-trained models, with ≤ 8 billion parameters, can achieve highly competitive results when fine-tuned for specific tasks. In particular, these models have demonstrated the potential to surpass larger models such as GPT-4o, which is estimated to have approximately 1.4 trillion tunable parameters, in specialized tasks including text summarization, reasoning, named entity recognition, and sentiment detection [35]. This highlights the advantage of task-specific fine-tuning in achieving strong performance even with relatively smaller model sizes. This choice provided a reasonable balance between computational efficiency and representational capacity for early-stage experimentation. Once the data pipeline and training setup were finalised, the evaluation process became significantly more structured, allowing for systematic comparison of various base models under consistent conditions. This enabled more objective assessments of model performance and suitability within the established framework.

Specifically, the finalized evaluation pipeline compared the following base models: Llama-3.3-70B, DeepSeek-70B, Phi-4-14.7B, DeepSeek-8B, Mistral-7B, Llama-3.1-8B, Llama-3.2-3B, and Llama-3.2-1B.

3.4 Prompt Engineering

Prompt engineering is typically regarded as a distinct methodology for enhancing the performance of LLMs. However, even in the context of fine-tuning, prompt engineering remains relevant, as each training example must be structured in a manner that clearly communicates the desired task to the model. In this case, the prompt was carefully designed to include three key components: an instruction, an input, and a fill-in-the-blank style response. The instruction is explicit and task-oriented:

```
Extract the car model from the input.  Make sure the
output only consists of the car model and nothing else.
```

This provides a clear directive that defines the scope and expected output of the task, which is a common prompt engineering technique aimed at reducing ambiguity and guiding the model toward the intended behaviour. The prompt further employs an example-based approach, often referred to as few-shot prompting. A single example is given to illustrate the task format:

```
Input: "vw volkswagen id.4 pro"
Response: "id.4 pro"
```

This inclusion of an example serves two purposes: it reinforces the instruction by demonstrating the expected transformation, and it anchors the model's behaviour for subsequent inputs. The final component is the fill-in-the-blank template:

```
Now, complete the task:
Input: (variable content for each row)
Response:
```

This structure ensures consistency across training samples and facilitates clear mapping between the input and the expected output. By combining instructional prompting with example-based formatting and a standardized input-output pattern, the approach integrates several foundational prompt engineering techniques to effectively guide the model's fine-tuning process.

3.5 Hyperparameter Search

Hyperparameter optimisation was performed for each separate base model using Optuna, a flexible and efficient framework for automated hyperparameter search. The optimisation process followed a 5-fold inner cross-validation loop applied to a 15% stratified sample subset of the full training dataset. This subset was selected to reduce computational overhead while preserving the diversity of the data, under the assumption that optimal hyperparameters identified on this smaller set would

generalise well to the full training run.

A total of 15 optimisation trials were conducted, with the goal of minimising the average evaluation loss across validation folds. The evaluation loss was based on the cross-entropy loss, computed as follows: for each input x_j with corresponding label y_j, the model with parameters θ produces logits $f_\theta(x_j)$, and the per-sample loss is given by

$$\mathcal{L}(y_j, f_\theta(x_j)) = -\log\left(\text{softmax}(f_\theta(x_j))_{y_j}\right).$$

Within each batch, the batch loss is the mean of the individual sample losses, and the final evaluation loss reported is the average of the batch losses across the validation set.

The search space included both standard training hyperparameters and parameters specific to LoRA, reflecting the hybrid nature of the fine-tuning strategy:

Table 3.2: Overview of tunable hyperparmeters

Hyperparameter	Type	Tuned
gradient_accumulation_steps	Trainer	Yes
optim	Trainer	No
learning_rate	Trainer	No
weight_decay	Trainer	No
num_train_epochs	Trainer	Yes
warmup_ratio	Trainer	No
lr_scheduler_type	Trainer	No
r	LoRA	Yes
lora_alpha	LoRA	Yes
lora_dropout	LoRA	Yes (Separately)

- Gradient Accumulation Steps: controls how many steps to accumulate gradients before performing a backward pass, sampled as an integer between 1 and 5.
- Number of Train Epochs: determines how many full passes through the training data, sampled between 2 and 4 epochs.
- LoRA Rank: the rank r of the low-rank decomposition matrices in LoRA, sampled from categorical values [8, 32, 64]. This controls the expressive capacity of the adaptation layers.
- LoRA Alpha: a scaling factor for the LoRA updates, sampled from [16, 32, 64]. This affects how much influence the LoRA-adapted weights have during fine-tuning. The adapted weight update is given by:

$$\Delta W = \alpha \cdot (BA)$$

By combining both types of parameters in the optimisation process, we aimed to jointly tune the general training behaviour (e.g., learning dynamics and convergence

speed) and the effectiveness of the PEFT mechanism. This mix ensured that the model not only trains efficiently but also leverages LoRA in a way that maximises performance while keeping the number of trainable parameters low.

To navigate this mixed and relatively high-dimensional hyperparameter space, we employed the NSGA-II evolutionary algorithm, which is known for its ability to maintain diversity in the search population and avoid local optima.

3.6 Full Training

After completing hyperparameter optimisation, the final models were trained on the full dataset using the optimal hyperparameter configuration for each specific model. The base model was first transformed into a parameter-efficient version. This involved applying LoRA to selected attention and feedforward projection layers within the transformer architecture.

The training was carried out using the SFTTrainer from the TRL library, which facilitates supervised fine-tuning of language models. The trainer was initialised with the LoRA-modified model, tokeniser, and preprocessed full training dataset. In addition to the optimal hyperparameter configuration, the remaining setup parameters are as follows: a learning rate of 1e-4, the paged 8-bit Adam optimiser [36], a weight decay of 0.01, a cosine learning rate scheduler with 0.03 warm-up fraction, and a LoRA dropout rate of 0.1. All training metrics and configuration details were tracked using Weights & Biases (WandB), allowing for detailed monitoring of the training dynamics.

3.7 Validation

3.7.1 Accuracy

To evaluate model performance, we used accuracy as the primary metric. Accuracy is calculated by dividing the number of correct predictions by the total number of predictions made:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

In our case, each prediction was compared against the manually labelled validation dataset. A prediction is considered correct if the generated string output exactly matches the true label. This means that for example a single extra blank space would be determined a false prediction. This provides a straightforward and interpretable measure of how well the model performs on unseen data.

3.7.2 Confidence Score

To measure how confident the model is in its predictions, a confidence score was calculated. Let $z_i \in \mathbb{R}^{T \times \mathcal{V}}$ be the logits for sample i where T is the number of

generated tokens in the sample and $\mathcal{V}$ the vocabulary size. For each sample:

1. Convert all logits into probabilities $p_{i,t,v}$ using the Softmax function 2.1.2.4 for each tokens $t \in 1, ..., T$ and possible predicted word $v \in \mathcal{V}$.
2. Extract the most confident prediction, i.e. the highest probability of each token
$$c_{i,t} = \max_{v} \; p_{i,t,v}.$$
3. Average these maximum probabilities across all tokens in the sample to get a single confidence score per sample
$$\text{Confidence Score}_i = \frac{1}{T} \sum_{t=1}^{T} c_{i,t}$$

Taking the mean of the probabilities rather than multiplying them together ensures that longer vehicle model names do not automatically receive a lesser confidence score.

3.7.3 Buckets

By combining confidence scores with accuracy, predictions were grouped into confidence intervals based on their confidence scores, allowing an evaluation of how model performance, in terms of accuracy, changes with varying degrees of certainty. This approach highlights how accurate the model is when it shows high confidence compared to when it is less certain.

3.7.4 Class-specific Accuracy

To evaluate our model's performance more thoroughly across all categories, we examined both the mean and median class-specific accuracy. This involved calculating the accuracy for each individual class, defined as the proportion of correctly predicted instances within that class, and then computing the mean and the median of these values across all classes. The mean class-specific accuracy provides an overall sense of performance but can be skewed by outlier classes with particularly high or low accuracies. In contrast, the median offers a more robust measure that is less sensitive to such extremes, giving insight into how the model performs for a "typical" class. By considering both metrics, we aimed to capture a more balanced view of the model's performance, especially in the presence of class imbalance.

3.8 Iterations

The experimental nature of this project led to an iteration-based approach, where smaller changes to the model were made incrementally to test and evaluate the strengths and weaknesses of each version. To ensure consistency, the same validation data was used across all iterations, while a separate dataset of 50,000 stratified sampled rows was used for broader model evaluation to determine potential changes. This dataset allows for more granular performance insights, such as model accuracy for specific vehicle models, thanks to the increased sample size.

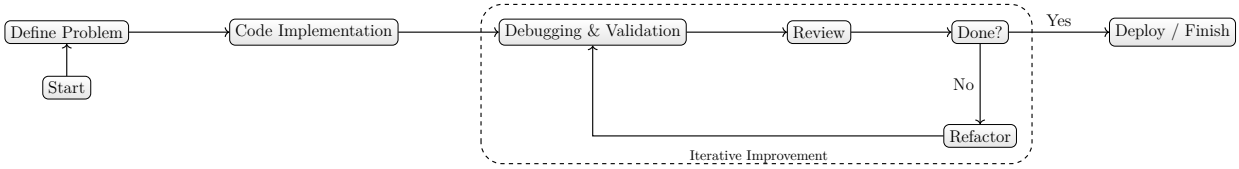

Figure 3.1: Illustration of the method chart with iteration procedure shown inside dashed box.

However, due to the large volume of data, manual labelling was not feasible. As a workaround, the previous classification model from If was used as a reference point for evaluating changes. To avoid overfitting to the validation set, special care is taken to monitor that the original predictions do not shift excessively between iterations, ensuring that improvements reflect generalisable performance gains rather than training and memorisation of the validation data.

3.8.1 Augmenting data

A key feature of the production model is its low maintenance requirements and ease of scalability. Maintenance primarily involves incorporating additional training data and retraining the model when encountering new vehicle models that exhibit lower prediction accuracy or confidence scores when, for example, a new product launches on the market. Maintainability and scalability are evaluated by augmenting the training dataset with data specific to such vehicle models and subsequently measuring improvements in accuracy and confidence scores across successive iterations.

3.8.2 Hyperparameter Tuning

LoRA dropout rate was iteratively tuned to avoid overfitting and underfitting. After each adjustment, we evaluated the impact by tracking key performance metrics, including training and validation accuracy, loss, and confidence scores. Based on these observations, we selected the hyperparameter setting that achieved the best balance between predictive accuracy and robustness to unseen data.

- To counteract overfitting, LoRA can employ dropout on the low-rank updates by randomly zeroing out portions of B and A during training, which serves as a sparsity regulariser [37].

3.8.3 Prompt Engineering

Using a similar iterative approach, we experimented with different prompt structures, such as few-shot and chain-of-thought, to optimize the model's interpretability and generalization. After each modification, we monitored metrics such as accuracy, confidence scores, and indicators of overfitting or underfitting. Comparing these results across prompt variations provided valuable insights into the effects of different prompt designs, finally resulting in the prompt shown in Section 3.4.

4
Results

4.1 Model Selection

In this study, eight distinct LLMs were evaluated after individual hyperparameter optimisation and fine-tuning, each exhibiting unique characteristics in terms of input/output behaviour and size. A comparative analysis was conducted to assess their overall accuracy in relation to their respective model sizes. The results of this comparison are presented visually in Figure 4.1 where the bars are ordered in descending order in terms of total parameters. All models were fine-tuned with the exact same pipeline and datasets. Based on the results, the final model chosen was Llama-3.1-8B.

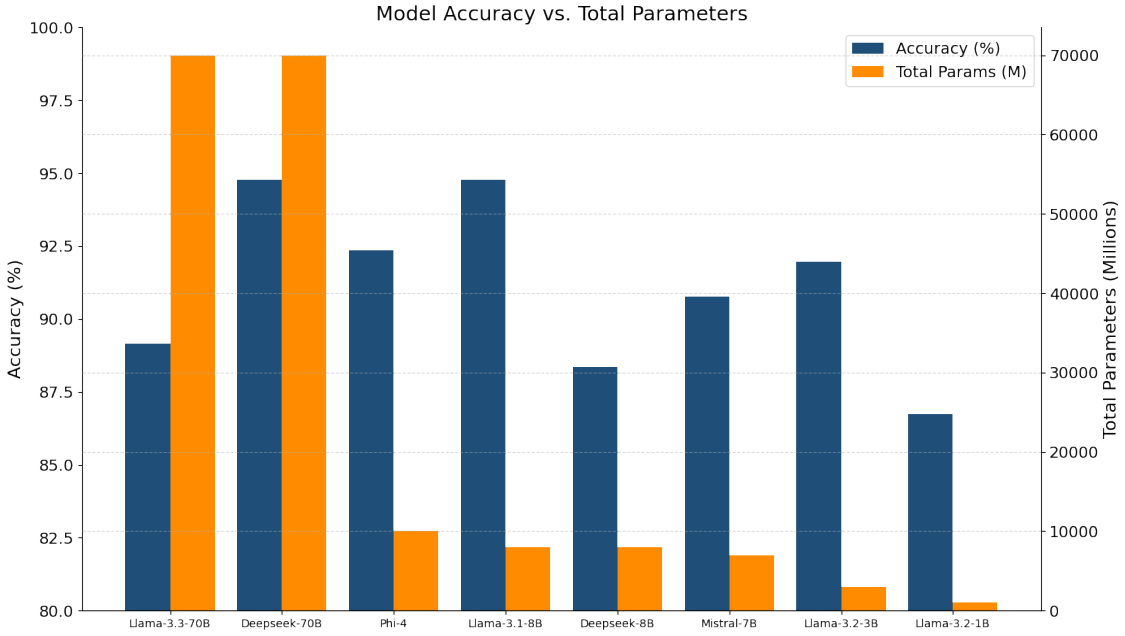

Figure 4.1: Base model comparison on validation data with accuracy metric alongside with number of total parameters.

4.2 Final Model Evaluation

Table 4.1 shows the configuration of hyperparameters used when training the final model, Llama-3.1-8B.

Table 4.1: Overview of final model's hyperparameters

Hyperparameter	Type	Tuned	Value
`gradient_accumulation_steps`	Trainer	Yes	1
`optim`	Trainer	No	`adamw_8bit`
`learning_rate`	Trainer	No	1×10^{-4}
`weight_decay`	Trainer	No	0.01
`num_train_epochs`	Trainer	Yes	2
`warmup_ratio`	Trainer	No	0.03
`lr_scheduler_type`	Trainer	No	`cosine`
`r`	LoRA	Yes	8
`lora_alpha`	LoRA	Yes	16
`lora_dropout`	LoRA	Yes (Separately)	0.1

Figure 4.2 illustrates a comparative analysis of model accuracy across four variants of large language models. The Base Llama 3.1 model achieves an accuracy of 53.7%, while GPT-4o demonstrates a moderately higher performance at 61.3%. Notably, the Fine-tuned Llama 3.1 model significantly outperforms both, reaching an accuracy of 96.8%. A version of the fine-tuned model without hyperparameter search, using base values for all hyperparameters, achieved 86.1%, highlighting the importance of tuning. A total of 83,886,080 weights were updated from the base model in the fine-tuning process to reach this performance. This substantial improvement suggests that fine-tuning the base model on domain-specific data or tasks can yield considerable gains in performance. The results emphasize the effectiveness of targeted model optimization and the varying capabilities of different LLMs when applied to the same evaluation criteria.

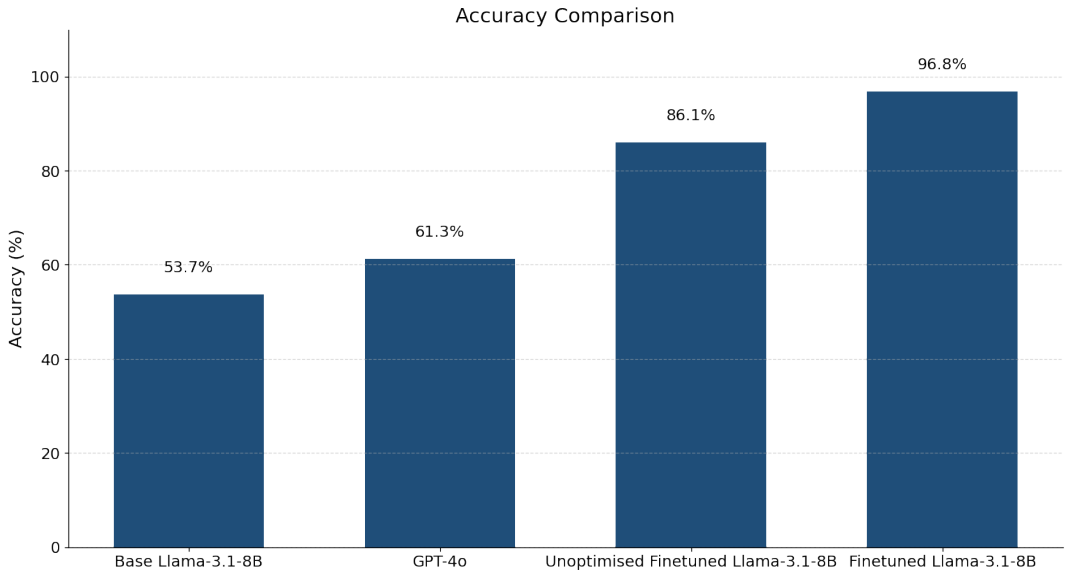

Figure 4.2: Comparison between base mode of Llama-3.1-8B, the non-open source GPT-4o and our fine-tuned Llama-3.1-8B.

Below in table 4.2 is a summary of the final model's properties and metrics.

Table 4.2: Model Metrics for fine-tuned Llama-3.1-8B.

Metric	Value
Base Model	Llama 3.1
Parameters	8 billion
Model Memory Usage	5.90 GB
Accuracy	96.8%
Median Model Accuracy	100%
Mean Model Accuracy	96.0%

The confidence score distribution in Table 4.3 provides insight into model certainty and performance calibration. Among samples with a predicted confidence score of exactly 1.0, the model achieved a perfect 100% accuracy on 230 samples, underscoring a high correlation between model confidence and correctness. In contrast, samples with lower confidence levels (<0.95) displayed markedly reduced accuracy, highlighting the importance of thresholding confidence in critical applications.

Table 4.3: Confidence Score for fine-tuned Llama-3.1-8B.

Bucket	Accuracy	Samples
<95	50%	12
0.95-1	67%	6
=1	100%	230

The iteration-based improvements shown in Table 4.4 further support the efficacy of targeted data augmentation and fine-tuning. For instance, adding just one additional training sample per class led to substantial gains in accuracy e.g., Audi A4 Allroad and Citroën C4 Cactus both improved from below 70% to 100%. These results suggest that the model remains highly responsive to even small-scale data enhancements and can generalize effectively from minimal yet relevant additional input.

Table 4.4: Results from iteration based training data augmentation.

Model	Rows added	Change
Citroën C4 Cactus	1	67⟶100% accuracy
Citroën C5 Aircross	1	95⟶100% accuracy
Citroën C4 Picasso	1	75⟶92% accuracy
Audi A4 Allroad	1	65⟶100% accuracy
Volvo 140	1	61⟶96% accuracy

5

Discussion

5.1 Final Model Considerations

From table 4.2, the chosen Llama-3.1-8B model exhibits a favourable balance between scale and efficiency, maintaining a relatively modest memory footprint of 5.90 GB during inference. The model achieved a high overall accuracy of 96.8% across a test set of 248 instances, reflecting its strong generalization capabilities. Notably, the median accuracy reached 100%, suggesting that a majority of samples were classified correctly without error. The proximity of the mean accuracy (96.0%) to the overall value further indicates a low variance in model performance across the dataset. This narrow dispersion between mean and median implies that the model does perform well on average, but is also reliably consistent across individual cases.

Figure 4.2 provides an overview of the base Llama-3.1-8B, GPT-4o, and the fine-tuned Llama-3.1-8B variant. The base model achieved an accuracy of 53.7%, which serves as a baseline reference for unadapted performance. GPT-4o, a state-of-the-art commercial model, reached 61.3%, offering a moderate improvement. However, the fine-tuned Llama-3.1-8B model significantly outperformed both, attaining a peak accuracy of 96.8%. The unoptimised fine-tuned Llama-3.1-8B model achieved a moderately strong accuracy of 86.1% which is significantly lower than the fine-tuned model with optimal hyperparameters, showcasing the effect of proper hyperparameter optimisation.

Furthermore, the performance gap between GPT-4o and the fine-tuned Llama model suggests that open-source alternatives, when appropriately optimised, can not only rival, but potentially surpass proprietary models in specific domains. This has important implications for accessibility, cost-efficiency, and customization in real-world applications where full model control and data privacy are essential considerations. It is also worth noting that LLMs are primarily designed and optimised for tasks involving natural language understanding, such as sentiment analysis, text summarization, and content generation. Consequently, employing LLMs directly for a classification task without fine-tuning could be the reason for the comparatively low performance observed in figure 4.2 for untuned models like GPT-4o.

5.2 Resource and Infrastructure Considerations

5.2.1 Limitations

Due to constraints in time, financial resources, and computational capacity, an outer cross-validation loop was deemed inefficient, as it would necessitate fully training all eight models for each fold of the cross-validation. Consequently, only an inner cross-validation loop was employed for hyperparameter tuning. The model validation presented in Figure 4.1 was therefore conducted a single time for each model, without cross-validation. As a result, some degree of unwanted variance may be present in determining which model performs best in terms of accuracy.

5.2.2 Computational Power and Time

Fine-tuning LLMs is a computationally intensive task. Even when employing PEFT methods with LoRA, training still requires considerable GPU resources to handle forward and backward passes over billions of parameters. In our experiments, training times ranged from a couple of minutes to days depending on the batch size, number of epoch, evaluation strategy. This computational overhead affects not only development timelines but also operational feasibility for production deployment.

From a business perspective, extended training durations translate directly into higher cloud costs or GPU infrastructure investments. This is particularly relevant for insurance companies like If P&C that operate in highly competitive markets with tight cost margins. Time-to-market also becomes a critical factor, as delays in model updates can hinder responsiveness to emerging data trends or new vehicle models, ultimately impacting pricing accuracy and competitiveness.

5.2.3 Hardware

Our setup relied on NVIDIA A100 GPUs for training, which offered sufficient memory and compute throughput for running LoRA-adapted LLMs. However, such hardware is not trivial to access or maintain at scale. Deploying LLM-based systems across an organisation may require a significant overhaul of existing infrastructure, including GPU-capable servers, faster networking, and upgraded cooling and power systems in on-premise environments.

This hardware dependency also imposes a barrier to entry for less mature organizations or smaller teams within If. For wide adoption, the IT department would need to ensure scalable access to GPU clusters and shared training environments. Additionally, the risk of hardware obsolescence emerges large, as newer architectures (e.g., NVIDIA Hopper, AMD MI300X) offer considerable performance gains, but come with new cost and integration challenges [34].

On the positive side, employing LoRA and related PEFT techniques drastically reduces the number of trainable parameters—by more than 90% in some cases.

This enables organisations to fine-tune models using consumer-grade GPUs or rented cloud GPUs at a fraction of the cost [28].

5.2.4 Financial Resources

The financial implications of deploying LLMs can go beyond compute costs. Some models operate on a pay-per-token pricing model, where users are charged based on the number of tokens they use. To illustrate, GPT-4 with a 32k token context window charges $60 per million prompt tokens and $120 per million generated tokens [38]. In our case, each prompt contains roughly 71 tokens, and the generated output averages around 2 tokens per vehicle model prediction. Given a dataset of approximately 7.3 million entries and using a USD to SEK exchange rate of 10, a single full run of the GPT-4 model on the entire dataset would result in a cost of:

$$(71 \cdot 60 + 2 \cdot 120) \cdot 7.3 \cdot 10 = 328500 \text{ SEK}.$$

This substantial expense would recur every time the dataset is updated by the vendor or whenever adjustments to the model are required. Given these recurring costs, it becomes clear that relying on a pay-per-token API for large-scale, frequent predictions is not financially sustainable. By contrast, fine-tuning and deploying a custom LLM, despite requiring an initial investment in compute resources and engineering effort, can drastically reduce marginal costs over time. Once deployed, a fine-tuned model allows for unlimited inferences without additional per-token fees, making it a far more economical and scalable solution in the long run.

Other factors such as model development incur expenses in data annotation, developer time, model training, validation, and system monitoring. In particular, human-labelled data remains a bottleneck, both in terms of time and money, especially when working with domain-specific datasets like vehicle classifications.

5.2.5 Bottlenecks

Although fine-tuning is inherently resource-intensive, a considerable portion of the total computational cost and time arises from hyperparameter optimization. During this process, evaluation loss must be computed repeatedly to identify the optimal hyperparameter configurations, significantly prolonging training time and increasing GPU utilization. However, this overhead can be substantially reduced by employing well-established default values for key hyperparameters or by making informed estimates based on prior experimental results. For instance, training a final model without evaluation loss computation, using a single epoch and a warmup ratio of 0.03 on an A100 GPU, only takes approximately one minute. In contrast, the complete training pipeline, including hyperparameter optimisation and continuous evaluation metric tracking, extends to around 24 hours. As shown in Figure 4.2, the model fine-tuned with standard hyperparameter settings maintains relatively strong accuracy (86.1%) while achieving a drastically reduced overall training time. Furthermore, omitting evaluation loss computation in the final training stage lowers computational demands; however, it removes the possibility of applying early stopping or other types of pruning based on evaluation metrics. Under these conditions,

the primary constraint becomes GPU memory capacity, which must be sufficient to accommodate and process the large model weights during training.

5.3 Performance versus Size

Balancing model accuracy and computational efficiency is crucial when selecting an LLM for deployment in production environments. Our evaluation of several LoRA-adapted LLMs reveals an inherent trade-off between the number of trainable parameters and model accuracy. Specifically, we considered models ranging from large-scale variants such as DeepSeek-70B, with approximately 70 billion trainable parameters, down to smaller alternatives like Llama-3.1-8B, which features significantly fewer parameters while maintaining commendable accuracy.

While it is a valid consideration that a much smaller model could achieve comparable performance with lower computational costs, our empirical evaluation, as illustrated in Figure 4.1, demonstrates a consistent decline in accuracy as model size decreases. In particular, while Llama-3.1-8B sustains high performance levels, the smaller Llama-3.2-3B and Llama-3.2-1B variants exhibit progressively lower accuracy scores. This suggests that, for our specific task, reducing the number of parameters beyond the 8B scale entails a non-negligible sacrifice in model effectiveness. Furthermore, it is noteworthy that larger models such as DeepSeek-70B achieve accuracy levels comparable to Llama-3.1-8B, implying that beyond a certain parameter threshold, additional scale yields diminishing returns. These findings underscore that model selection should not merely prioritize minimizing size but must carefully balance computational efficiency with task-specific performance requirements.

Given these results, Llama-3.1-8B was selected as the most suitable model. As indicated in our performance benchmarks (see Figure 4.1 and Subsection 4.2), it maintains competitive accuracy while substantially reducing the number of trainable parameters compared to larger counterparts. This choice aligns strategically with the computational and business constraints outlined in Section 5.2.

5.4 Human-in-the-loop

Despite the advances in LLM capabilities, human oversight remains a critical component in ensuring robustness, interpretability, and accountability. By analyzing prediction confidence scores (see Section 3.7.2), we observed that the model's outputs vary significantly in reliability. A practical approach is to establish confidence thresholds beyond which human verification is triggered.

For instance, predictions with confidence scores below 1.0 may be flagged for manual review by pricing analysts, see table 4.3. This human-in-the-loop mechanism provides a safeguard against erroneous classifications, especially for rare or newly released car models where training data is sparse or non-existent. It also allows for

continuous improvement: low-confidence or incorrect predictions can be added back into the training dataset for future fine-tuning cycles, forming a self-improving loop.

Organizationally, this requires alignment between data science and underwriting teams to define actionable policies around confidence scores, error tolerance, and retraining schedules. human-in-the-loop systems can be implemented within internal dashboards or labelling tools, allowing non-technical staff to contribute feedback. Several studies have shown that hybrid AI-human systems outperform fully automated or fully manual workflows in high-stakes domains such as finance and healthcare [1].

Moreover, by only reviewing uncertain cases, the human labor involved remains manageable and cost-efficient. This ensures a sustainable AI deployment model that balances automation with control which potentially maximising business value while preserving model trustworthiness.

5.5 Deployment

Deploying a fine-tuned LLM in a large-scale corporate environment, such as an insurance company, necessitates a strategic framework that extends beyond technical feasibility. Strategy management in this context encompasses organizational readiness, infrastructure integration, lifecycle planning, risk governance, and stakeholder alignment to ensure the model delivers consistent value over time [39].

A successful deployment strategy begins with alignment to organisational objectives. In the case of If P&C Insurance, the LLM is intended to improve vehicle model classification, thereby enhancing the accuracy of risk segmentation and pricing. [1]. Next, cross-functional integration must be established. The LLM should be embedded within the existing pricing and product development pipelines, either through API endpoints or as part of the internal microservices architecture. [34].

Monitoring and feedback mechanisms must be a core part of the deployment strategy. These include logging prediction confidence scores (as described in Section 3.7.2), detecting model drift, and incorporating human-in-the-loop feedback systems for low-confidence classifications (Section 5.4). Monitoring tools like WandB can provide real-time performance dashboards to monitor prediction failures or data anomalies [32].

To address scalability and maintainability, the strategy should incorporate scheduled re-training cycles, informed by version control and data logging infrastructure. Retraining could be triggered periodically or event-based (e.g., when confidence scores drops below a defined threshold). New car models entering the market can be flagged through data augmentation (see Section 3.8.1), enabling proactive updates to the classification system [28].

From a governance and compliance perspective, the deployment strategy must comply with internal data protection policies, industry regulations, and AI risk frameworks. Ensuring the interpretability of model predictions, maintaining records of all model versions, and documenting ethical considerations are essential steps toward responsible AI deployment [40].

5.6 Impact of Sampling Strategy

The initial training dataset was stratified based on car make, leading to a disproportionate representation of certain brands. The rationale behind stratification was to ensure adequate representation and accurate model predictions across all car makes. However, upon closer inspection, significant improvements in model accuracy were observed when even a single training data point was added for specific car models (see table 4.4).

These substantial accuracy gains from minimal additional data suggest that exhaustive stratification by car make might not have been necessary. Our findings imply that including even sparse examples of less represented car models could significantly enhance the model's predictive performance. In retrospect, a more effective approach could involve a balanced or even random sampling strategy complemented by targeted inclusion of rare or uniquely represented models.

6
Conclusion

This thesis has demonstrated that LLMs, when fine-tuned with PEFT techniques such as LoRA, offer a powerful, scalable, and cost-effective solution for vehicle classification with complex and unstructured data. Importantly, the results showed that fine-tuning is not merely an enhancement but a critical requirement for achieving high performance; out-of-the-box models, without adaptation to the task-specific data, failed to deliver satisfactory results.

By addressing the shortcomings of existing classification methods at If P&C Insurance, a fine-tuned Llama-3.1-8B model was developed that achieved a classification accuracy of 96.8% on unseen hold-out data. The approach combined a carefully curated and preprocessed dataset, prompt engineering, and systematic hyperparameter optimization using NSGA-II. This enabled us to maximize performance while maintaining a modest memory footprint and computational cost.

Moreover, we showed that the model's confidence scores closely aligned with its predictive accuracy, supporting the viability of a human-in-the-loop framework to maintain reliability and promote ongoing improvement. Key deployment considerations including infrastructure needs, financial trade-offs, and organizational alignment were critically assessed.

Notably, the results underscore that even relatively small, open-source LLMs can outperform larger, proprietary models like GPT-4o in domain-specific tasks, but only when fine-tuned appropriately. This highlights the strategic importance of domain adaptation through fine-tuning, especially in settings where the performance of a base models is insufficient.

Ultimately, this study demonstrates that integrating LLMs into the insurance pricing pipeline is not only technically feasible but also offers clear business value. By enabling more precise vehicle classification, insurers can significantly enhance their risk-based pricing models, reduce operational inefficiencies, and respond more agilely to market changes.

Bibliography

[1] L Powell. "Risk-Based Pricing of Property and Liability Insurance". In: *Journal of Insurance Regulation* 39.4 (2020).

[2] Martin Eling et al. "Big Data, Risk Classification, and Privacy in Insurance Markets". In: *The Geneva Risk and Insurance Review* 49.1 (2024), pp. 75–126.

[3] Shihui Xie. "Analyzing the Influence of Telematics-Based Pricing Strategies on Traditional Rating Factors in Auto Insurance Rate Regulation". In: *Mathematics* 12.19 (2024), p. 3150.

[4] David Cather. "Addressing Insurance Price Discrimination in an Era of Diversity, Equity, and Inclusion". In: *Risk Management and Insurance Review* 26.3 (2023), pp. 407–429.

[5] J.D. Power Automotive. *The Ultimate Guide to Vehicle Data*. Industry White Paper. 2023.

[6] Siddhartha Das et al. "Vehicle Involvements in Hydroplaning Crashes: Applying Interpretable Machine Learning". In: *Transportation Research Interdisciplinary Perspectives* 6 (2020), p. 100176.

[7] Di Li et al. "Textual Analysis of Insurance Claims with Large Language Models". In: *Journal of Risk and Insurance* (2025). Forthcoming, Early View.

[8] Chris Balona. "ActuaryGPT: Applications of Large Language Models to Insurance and Actuarial Work". In: *British Actuarial Journal* 29 (2024), e15.

[9] Jan Schneider, Christian Meske, and Peter Kuss. "Foundation Models". In: *Business & Information Systems Engineering* 66.2 (2024), pp. 221–231.

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 1877–1901.

[11] Hugo Touvron, Thibaut Lavril, Gautier Izacard, et al. "LLaMA: Open and Efficient Foundation Language Models". In: *arXiv* (2023). URL: https://arxiv.org/abs/2302.13971.

[12] Yuntao Sui et al. "GPT4Table: Can Large Language Models Understand Structured Table Data? A Benchmark and Empirical Study". In: *arXiv* (2023).

[13] Ilyes Keraghel, Sylvie Morbieu, and Mohamed Nadif. "Recent Advances in Named Entity Recognition: A Comprehensive Survey and Comparative Study". In: *arXiv* (2024). URL: https://arxiv.org/abs/2401.10825.

[14] Alex D. Singleton and Seth Spielman. "Segmentation using large language models: A new typology of American neighborhoods". In: *EPJ Data Science* 13 (2024). URL: http://dx.doi.org/10.1140/epjds/s13688-024-00466-1.

[15] Wayne Xin Zhao, Kun Zhou, Jing Li, et al. "A survey of large language models". In: *arXiv preprint arXiv:2303.18223* (2023).

[16] Tomas Mikolov et al. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[17] Jeffrey Pennington, Richard Socher, and Christopher D Manning. “GloVe: Global Vectors for Word Representation”. In: *EMNLP*. 2014, pp. 1532–1543.

[18] Yoshua Bengio et al. “A neural probabilistic language model”. In: *Journal of Machine Learning Research* 3 (2003), pp. 1137–1155.

[19] Yoav Goldberg. *Neural Network Methods for Natural Language Processing*. Morgan & Claypool Publishers, 2017.

[20] Ashish Vaswani et al. “Attention is All You Need”. In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017, pp. 5998–6008.

[21] Alec Radford et al. *Improving language understanding by generative pre-training*. Tech. rep. OpenAI, 2018.

[22] Ofir Press and Lior Wolf. “Using the Output Embedding to Improve Language Models”. In: *arXiv preprint arXiv:1608.05859* (2016).

[23] Hakan Inan, Daniel Khashabi, and Richard Socher. “Tying Word Vectors and Word Classifiers: A Loss Framework for Language Modeling”. In: *International Conference on Learning Representations (ICLR)*. 2017.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016. URL: https://doi.org/10.7551/mitpress/10434.001.0001.

[25] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. “Language Models are Few-Shot Learners”. In: *arXiv* (2020). URL: https://arxiv.org/abs/2005.14165.

[26] J. S. Bridle. “Probabilistic Interpretation of Feedforward Classification Network Outputs”. In: *Neurocomputing: Algorithms, Architectures and Applications*. Springer, 1990, pp. 227–236.

[27] H. K. Chaubey et al. “Comparative Analysis of RAG, Fine-Tuning, and Prompt Engineering in Chatbot Development”. In: *2024 International Conference on Future Technologies for Smart Society (ICFTSS)*. 2024, pp. 169–172.

[28] Edward J. Hu, Yelong Shen, Phil Wallis, et al. “LoRA: Low-Rank Adaptation of Large Language Models”. In: *arXiv* (2021). URL: https://arxiv.org/abs/2106.09685.

[29] Ahmad Hassanat, Khalid Almohammadi, Eyad Alkafaween, et al. “Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach”. In: *Information* 10.12 (2019), p. 390. URL: https://doi.org/10.3390/info10120390.

[30] Kalyanmoy Deb et al. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. URL: https://doi.org/10.1109/4235.996017.

[31] Adeel Zafar, Muhammad Aamir, Nazri Mohd Nawi, et al. “An optimization approach for convolutional neural network using Non-Dominated Sorted Genetic Algorithm-II”. In: *Computers, Materials & Continua* 74.3 (2022), pp. 5641–5661. URL: https://doi.org/10.32604/cmc.2023.033733.

[32] Unsloth Documentation Team. *Fine-tuning Guide*. Mar. 2025. URL: https://docs.unsloth.ai/get-started/fine-tuning-guide.

[33] Andyrasika Singal. *Unleashing the Power of Unsloth and QLora: Redefining Language Model Fine-Tuning*. Hugging Face Blog. Mar. 2024. URL: https://huggingface.co/blog/Andyrasika/finetune-unsloth-qlora.

[34] Jan Schneider and Ian Smalley. *CPU vs. GPU for Machine Learning*. Jan. 2025. URL: https://www.ibm.com/think/topics/cpu-vs-gpu-machine-learning.

[35] J. Zhao et al. "LoRA Land: 310 Fine-tuned LLMs that Rival GPT-4, A Technical Report". In: *arXiv* (2024).

[36] Dettmers T et al. "QLoRA: Efficient Finetuning of Quantized LLMs". In: *arXiv* (2023).

[37] Yao Lin, Xia Ma, Xin Chu, et al. "LoRA Dropout as a Sparsity Regularizer for Overfitting Control". In: *arXiv* (2024).

[38] OpenAI. *How much does GPT-4 cost?* Accessed: 2025-04-10. 2023. URL: https://help.openai.com/en/articles/7127956-how-much-does-gpt-4-cost.

[39] Jan Schneider, Christian Meske, and Peter Kuss. "Foundation Models". In: *Business & Information Systems Engineering* 66 (2024), pp. 221–231. URL: http://dx.doi.org/10.1007/s12599-024-00851-0.

[40] Ilyes Keraghel, Sylvie Morbieu, and Mohamed Nadif. "Recent Advances in Named Entity Recognition: A Comprehensive Survey and Comparative Study". In: *arXiv* (2024). URL: https://arxiv.org/abs/2401.10825.

A

Appendix 1 - Code Hyperparameter Search

```
1
2  import optuna
3  import numpy as np
4  from trl import SFTTrainer
5  from unsloth import is_bfloat16_supported
6  from sklearn.model_selection import KFold
7  from datasets import Dataset
8  from unsloth import FastLanguageModel
9  from transformers import TrainingArguments, Trainer,
       DataCollatorForSeq2Seq
10 from peft import LoraConfig
11 from datasets import load_dataset
12 from optuna.visualization import (
13     plot_param_importances,
14     plot_optimization_history,
15     plot_slice,
16     plot_contour,
17 )
18 import json
19
20
21 # Model Configuration
22 max_seq_length = 2048
23 dtype = None  # Float16 for Tesla T4/V100, Bfloat16 for
       Ampere+
24 load_in_4bit = True  # 4-bit quantization to reduce memory
       usage
25 model_path = "unsloth/meta-llama-3.1-8b-unsloth-bnb-4bit"
26
27
28 # Load Model and Tokenizer
29 model, tokenizer = FastLanguageModel.from_pretrained(
30     model_name=model_path,
31     max_seq_length=max_seq_length,
32     dtype=dtype,
33     load_in_4bit=load_in_4bit,
34 )
```

```

# Define Alpaca-Style Formatting Prompt
alpaca_prompt = """
### Instruction:
{}

### Input:
{}

### Response:
{}"""
```

Listing A.1: Python script for loading LLM

```
EOS_TOKEN = tokenizer.eos_token  # Ensure generation stops at EOS

def formatting_prompts_func(examples):
    instructions = examples["instruction"]
    inputs = examples["input"]
    outputs = examples["output"]
    texts = []
    for instruction, input, output in zip(instructions, inputs, outputs):
        text = alpaca_prompt.format(instruction, input, output) + EOS_TOKEN
        texts.append(text)
    return {"text": texts}

# Load Dataset
dataset = load_dataset("json",  data_files="PATH.json") #  , split="train"
dataset = dataset.map(formatting_prompts_func, batched=True)
dataset = dataset["train"]
```

Listing A.2: Python script for defining Alpaca Prompt

```
# Define number of folds for cross-validation
NUM_FOLDS = 5

num_samples = len(dataset)
indices = list(range(num_samples))

# Perform KFold splitting
kf = KFold(n_splits=NUM_FOLDS, shuffle=True, random_state=3407)
dataset_splits = list(kf.split(indices))  # Indices of folds

def objective(trial):
    # Hyperparameter suggestions
```

```
    gradient_accumulation_steps = trial.suggest_int("gradient_accumulation_steps", 1, 5)
    num_train_epochs = trial.suggest_int("num_train_epochs", 2, 4)
    lora_r = trial.suggest_categorical("lora_r", [8, 32, 64])
    lora_alpha = trial.suggest_categorical("lora_alpha", [16, 32, 64])

    # Split dataset into k-folds

    fold_losses = []  # Store validation losses for each fold

    for fold, (train_idx, val_idx) in enumerate(dataset_splits):
        print(f"\nFold {fold + 1}/{NUM_FOLDS}...")

        # Inside the loop:
        train_dataset = dataset.select(train_idx.tolist())  # Convert NumPy indices to list
        val_dataset = dataset.select(val_idx.tolist())      # Convert NumPy indices to list


        # Initialize LoRA-adapted model
        p_model = FastLanguageModel.get_peft_model(
            model,
            r=lora_r,
            target_modules=["q_proj", "k_proj", "v_proj", "o_proj",
                            "gate_proj", "up_proj", "down_proj"],
            lora_alpha=lora_alpha,
            lora_dropout=0.1,
            bias="none",
            use_gradient_checkpointing="unsloth",
            random_state=3407,
            use_rslora=False,
            loftq_config=None,
        )

        # Training arguments
        training_args = TrainingArguments(
            save_strategy="no",
            per_device_train_batch_size=2,
            gradient_accumulation_steps=gradient_accumulation_steps,
            warmup_ratio=0.03,
            num_train_epochs=num_train_epochs,
            learning_rate=1e-4,
```

```
            fp16=not is_bfloat16_supported(),
            bf16=is_bfloat16_supported(),
            logging_steps=10,
            optim="adamw_8bit",
            weight_decay=0.01,
            lr_scheduler_type="cosine",
            seed=3407,
            output_dir=f"outputs/trial_{trial.number}_fold{fold}",
            report_to="none",
            eval_strategy="epoch",
        )

        # Initialize the trainer
        trainer = SFTTrainer(
            model=p_model,
            tokenizer=tokenizer,
            train_dataset=train_dataset,
            eval_dataset=val_dataset,
            dataset_text_field="text",
            max_seq_length=max_seq_length,
            dataset_num_proc=1,
            packing=False,
            args=training_args,
        )

        # Train and evaluate

        trainer.train()
        eval_metrics = trainer.evaluate()
        eval_loss = eval_metrics["eval_loss"]
        #trial.report(eval_loss, num_train_epochs)  # Report to Optuna

        # Store validation loss
        fold_losses.append(eval_loss)

    # Return the average loss across k-folds
    avg_loss = np.mean(fold_losses)
    return avg_loss

# Create Optuna study with pruning
study = optuna.create_study(
    direction="minimize"
)
study.optimize(objective, n_trials=15)

# Print best hyperparameters
print("Best hyperparameters found:")
```

```
100 print(study.best_params)
```

Listing A.3: Python script for hyperparameter search

B

Appendix 2 - Code Full Training

```python
from unsloth import FastLanguageModel
import torch

max_seq_length = 2048 # Choose any! We auto support RoPE Scaling internally!
dtype = None # None for auto detection. Float16 for Tesla T4, V100, Bfloat16 for Ampere+
load_in_4bit = True # Use 4bit quantization to reduce memory usage. Can be False.
#model_path = "unsloth/phi-4"
model_path = "unsloth/meta-llama-3.1-8b-unsloth-bnb-4bit"
#model_path = "filip_simon_model"

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = model_path,
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
    # token = "hf_...", # use one if using gated models like meta-llama/Llama-2-7b-hf
)
```

Listing B.1: Python script for loading LLM

```python
import json

with open("PATH.json", "r") as file:
    hyperparameters = json.load(file)

gradient_accumulation_steps = hyperparameters["gradient_accumulation_steps"]
num_train_epochs = hyperparameters["num_train_epochs"]
lora_r = hyperparameters["lora_r"]
lora_alpha = hyperparameters["lora_alpha"]
lora_dropout = 0.1
```

Listing B.2: Python script for loading hyperparameters

```python
    model = FastLanguageModel.get_peft_model(
    model,
```

```
    r = lora_r, # Choose any number > 0 ! Suggested 8, 16, 32, 64, 128
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
                      "gate_proj", "up_proj", "down_proj",],
    lora_alpha = lora_alpha,
    lora_dropout = lora_dropout, # Supports any, but = 0 is optimized
    bias = "lora_only",     # Supports any, but = "none" is optimized
    # [NEW] "unsloth" uses 30% less VRAM, fits 2x larger batch sizes!
    use_gradient_checkpointing = True, # True or "unsloth" for very long context
    random_state = 3407,
    use_rslora = False,  # We support rank stabilized LoRA
    loftq_config = None, # And LoftQ
)
```

Listing B.3: Python script for data preprocessing6

```
from datasets import load_dataset

alpaca_prompt = """
### Instruction:
{}

### Input:
{}

### Response:
{}"""


EOS_TOKEN = tokenizer.eos_token  # Ensure generation stops at EOS

def formatting_prompts_func(examples):
    instructions = examples["instruction"]
    inputs = examples["input"]
    outputs = examples["output"]
    texts = []
    for instruction, input, output in zip(instructions, inputs, outputs):
        text = alpaca_prompt.format(instruction, input, output) + EOS_TOKEN
        texts.append(text)
    return {"text": texts,}


dataset = load_dataset("json",  data_files="PATH.json")
```

```
dataset = dataset.map(formatting_prompts_func, batched=True)
dataset = dataset['train']
```

Listing B.4: Python script for defining Alpaca Prompt

```
import numpy as np
import wandb
from sklearn.model_selection import KFold
from datasets import Dataset
from trl import SFTTrainer
from transformers import TrainingArguments
from unsloth import is_bfloat16_supported
import evaluate

# Initialize WandB for this fold
wandb.init(
    project="my_project_name",  # Change to your WandB project name
    name=f"experiment",
    config={
        "batch_size": 16,
        "gradient_accumulation_steps": gradient_accumulation_steps,
        "learning_rate": 1e-4,
        "num_train_epochs": num_train_epochs,
        "max_seq_length": max_seq_length,
        "evaluation_metric": "accuracy",
    },
)

# Training arguments
training_args = TrainingArguments(
    per_device_train_batch_size=16,
    gradient_accumulation_steps=gradient_accumulation_steps,
    warmup_ratio=0.03,
    num_train_epochs=num_train_epochs,
    learning_rate=1e-4,
    fp16=not is_bfloat16_supported(),
    bf16=is_bfloat16_supported(),
    optim="adamw_8bit",
    weight_decay=0.01,
    lr_scheduler_type="cosine",
    seed=3407,
    output_dir=f"outputs/",
    report_to="wandb",
    save_strategy="no",
    logging_strategy = "epoch",
)

# Initialize Trainer
```

```
trainer = SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=dataset,
    dataset_text_field="text",
    max_seq_length=max_seq_length,
    dataset_num_proc=1,
    packing=False,
    args=training_args,
)

# Train and evaluate
trainer.train()
```

Listing B.5: Python script for train full model

C

Appendix 3 - Code Evaluation with Accuracy and Confidence Score

```python
from transformers import AutoTokenizer
from unsloth import FastLanguageModel
import torch

# Define the path to your saved model
model_path = "PATH.model"

# Load the tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_path)

#  Unpack the model and tokenizer properly
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name=model_path,
    max_seq_length=max_seq_length,
    #dtype=dtype,
    load_in_4bit=load_in_4bit,
)

# Ensure model is moved to the available device
device = torch.device("cuda" if torch.cuda.is_available()
   else "cpu")

print(f"Model type before moving to GPU: {type(model)}")  #
   Should be nn.Module

# Move model to GPU
model = model.to(device)

print("Model loaded successfully on", device)
```

Listing C.1: Python script for loading pre-trained model

```python
import torch
from unsloth import FastLanguageModel
import os
```

```
4
5  # Ensure generation stops at EOS
6
7  # Define batch size (adjust based on GPU memory)
8  batch_size = 64
9
10 # Function to compute confidence scores
11
12 FastLanguageModel.for_inference(model)
13 def compute_confidence_scores_from_logits(logits):
14     """Compute confidence scores directly from logits without
           an extra forward pass."""
15     probabilities = logits.softmax(dim=-1)  # Convert logits
           to probabilities
16     chosen_token_probs = probabilities.max(dim=-1).values  #
           Get max probability for each generated token
17     avg_confidence_scores = chosen_token_probs.mean(dim=-1)
           # Compute mean confidence score per sample
18     return avg_confidence_scores.cpu().tolist()
19
20 # Open output file in write mode
21 with open(output_file, "w", encoding="utf-8") as f:
22     f.write('Input Text,Response Text,Desired Text,Confidence
           Score\n')  # CSV header
23
24     for i in range(0, len(input_list), batch_size):
25         batch = input_list[i:i+batch_size]
26
27         # Format inputs using alpaca_prompt
28         formatted_batch = [
29             alpaca_prompt.format(
30                 "Extract the car model from the input. Make
                       sure the output only consists of the car
                       model and nothing else. Below is an
                       example: \nInput: \"vw volkswagen id.4 pro
                       \". Output: \"id.4 pro\". Now, complete
                       the task.",
31                 text,
32                 ""
33             )
34             for text in batch
35         ]
36
37         # Tokenize batch
38         inputs = tokenizer(formatted_batch, return_tensors="
               pt", padding=True, truncation=True).to("cuda")
39
40         # Generate outputs with token scores
41         with torch.no_grad():
```

```
            outputs = model.generate(
                **inputs,
                max_new_tokens=128,
                output_scores=True,  # Get per-token logits
                return_dict_in_generate=True
            )

        # Decode outputs
        decoded_outputs = tokenizer.batch_decode(outputs.sequences, skip_special_tokens=True)

        # Compute confidence scores from logits
        avg_confidence_scores = compute_confidence_scores_from_logits(torch.stack(outputs.scores, dim=1))

        # Write results to file
        for input_text, output, desired_text, confidence_score in zip(
            batch, decoded_outputs, output_list[i:i+batch_size], avg_confidence_scores
        ):
            response_start = output.find("### Response:") + len("### Response:")
            response_text = output[response_start:].strip()
            f.write(f'{input_text}, {response_text}, {desired_text}, {confidence_score:.4f}\n')

print(f"All outputs saved to {output_file}")
```

Listing C.2: Python script for defining Confidence Score

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY