Leveraging ClimateBERT for Climate Legislation Analysis: Evaluating Sampling Strategies on the POLIANNA Dataset

Sven Lutz

February 14, 2025

Abstract

Climate and energy policies are closely interlinked, with frameworks such as the *United Nations Framework Convention on Climate Change (UNFCCC)* providing the basis for global climate governance. Despite significant legislative efforts, scalable and systematic evaluations of climate policies remain a critical research challenge. This work uses advanced *Machine Learning (ML)* techniques, in particular *Natural Language Processing (NLP)*, to analyse legislative measures. Using the domain-specific transformational model *ClimateBERT* and the *POLIANNA* dataset, this work develops a robust methodology to address challenges such as data imbalance, hierarchical text structures, and the complexity of predicting legislative *Features*. The methodology, illustrated in 3.1, includes dataset adaptation, resampling techniques and advanced modelling approaches. Preprocessing steps included text cleaning, reorganising data by *SpanID*, excluding incomplete curations, and extracting annotations, as detailed in 2. To mitigate the effects of class imbalance (2.2, 2.3, 2.4), resampling techniques were applied, including *Oversampling*, *Undersampling*, and hybrid methods such as *SMOTE*, *ADASYN*, and *SMOTEENN*. *HPO* with *Grid Search* and *Nested Cross-Validation (NCV)* further ensured a fair and systematic evaluation of the different sampling strategies. Initial *Percentage Split* baselines were improved using this structured approach. Guided by three key *Research Questions*, the results show that combining *SMOTEENN* with *ClimateBERT* significantly improves classification performance for policy *Layers* and *Features*. Despite these advances, direct *Feature Prediction* without hierarchical context remains challenging, highlighting the need for further refinement in *Feature Engineering* and model architectures. These findings highlight the value of hierarchical methods and robust sampling strategies for *NLP*-based policy analysis. This work contributes to the intersection of *Artificial Intelligence (AI)* and climate policy research by providing a structured pipeline for evaluating legislative texts. Future work should explore larger and more diverse datasets, improve *Feature Extraction* techniques, and develop frameworks for human-*AI* collaboration to improve policy evaluation and support evidence-based climate policy.

List of Abbreviations

ADASYN Adaptive Synthetic Sampling

AI Artificial Intelligence

BERT Bidirectional Encoder Representations from Transformers

ENN Edited Nearest Neighbors

EU European Union

HPO Hyperparameter Optimisation

IAA Inter-Annotator Agreement

ML Machine Learning

NCV Nested Cross-Validation

NLP Natural Language Processing

SMOTE Synthetic Minority Oversampling Technique

SMOTEENN Synthetic Minority Oversampling Technique with Edited Nearest Neighbors

UNFCCC United Nations Framework Convention on Climate Change

t-SNE t-Distributed Stochastic Neighbor Embedding

Contents

Chapter 1

Introduction

Climate policy and energy policy are closely linked in legislation, and the terms are sometimes used interchangeably [1]. The origins of international climate policy go back further than the milestones known to many [2, 3], such as the 2015 Paris Agreement [4] or the Kyoto Conference [5]. An important starting point was the adoption of the United Nations Framework Convention on Climate Change (UNFCCC) [6], agreed at the 1992 Earth Summit in Rio de Janeiro. This Convention forms the basis for many other international and national climate change measures. At the national level, for example, the German Climate Protection Act of 2019 [7] plays a key role [8]. Another important example is the 'Integriertes Energie- und Klimaprogramm' adopted by the Federal Cabinet in August 2007 [9]. This programme, which consists of 29 specific measures, aims to reduce greenhouse gas emissions in Germany by 40% by 2020 compared to 1990 levels [8].

However, such policies and legislation require regular evaluation to ensure their effectiveness and efficiency [10]. To this end, a monitoring process has been in place since 2011, documenting climate policy developments in annual and triennial progress reports [11, 12]. This process is supported by an independent panel of eminent energy experts, whose assessments help to categorise the monitoring results and formulate recommendations for future climate policy. Given the complexity of climate legislation and its impact, modern analytical approaches are becoming increasingly relevant. *Machine Learning (ML)* offers a powerful means of systematically evaluating the effectiveness of economic and climate policies [13].

By using *Artificial Intelligence (AI)* technologies, large amounts of political and economic data can be processed efficiently, revealing complex relationships that might otherwise go unnoticed. Building on these developments, a key question arises: How can technological innovations - especially in the field of *AI* [14] - support and improve the assessment of climate laws? One promising approach is human-*AI* collaboration, where experts and *AI* systems work together to analyse complex legal texts more efficiently and assess their impact more accurately [15]. In particular, modern *Natural Language Processing (NLP)* methods based on transformer technology [16] have significant potential to advance this field. An important step in this direction is the development of the *POLIANNA* dataset [13], which was created specifically for the analysis of legal texts in the field of climate change.

The aim of this work is to investigate how this dataset can be best used to produce sound predictions and analyses using *NLP* methods. The focus will be on the application of a specialised transformer model, *ClimateBERT* [17], which has particular strengths in the processing of climate-related texts due to its domain adaptation. The central *Research Questions* of this work can be formulated as follows:

1. ***Research Question 1:*** **Assessing the *Precision* of *NLP* Methods:** Are *NLP* methods capable of predicting the characteristics of legislative measures, and if so, how precise are these predictions?

2. ***Research Question 2:*** **Evaluating the Effectiveness of Sampling Strategies:** Are sampling strategies effective in overcoming the challenges of a limited and unbalanced dataset to improve model *Accuracy*?

3. ***Research Question 3:*** **Evaluating Direct *Feature Prediction* vs. *Layer-Level Analysis:*** Does the direct Prediction of *Features* yield better results than performing the analysis at the *Layer* level? Specifically, is it possible to achieve strong predictive performance using direct *Feature Prediction*? How do the results compare to using the *Layer* as an input *Feature* in the model?

To address the *Research Questions*, this work develops a systematic methodology, illustrated in 3.1. The approach includes dataset adaptation, sampling strategies and modelling techniques. First, the

POLIANNA dataset was cleaned and adapted to meet the requirements of *NLP* modelling. The main preprocessing steps included text cleaning, restructuring by *SpanID* and annotation extraction (see 2 for details). Due to class imbalance (2.2, 2.3, 2.4), various resampling techniques - *Oversampling*, *Undersampling* and hybrid methods such as *Synthetic Minority Oversampling Technique (SMOTE)*, *Adaptive Synthetic Sampling (ADASYN)* and *Synthetic Minority Oversampling Technique with Edited Nearest Neighbors (SMOTEENN)* - were explored to improve model performance. The methodology integrates *ClimateBERT* with *Grid Search* and *Nested Cross-Validation (NCV)* to ensure systematic *Hyperparameter Optimisation (HPO)* and fair comparability of sampling strategies. Initially, a simple *Percentage Split* served as a baseline, before moving to *NCV* for a more robust evaluation. Further details can be found in 3, in particular (3.3) and (3.3).

This work is divided into several chapters. First, the *POLIANNA* dataset (2.1) and the preprocessing steps used are described in detail in the Background chapter (2). This is followed by the Approach chapter (3), which explains the methodological approach and the procedures used. The results are then presented in the Results section and critically analysed and placed in a scientific context in the Discussion section (4). Finally, the limitations of the work are discussed and an outlook is given (5), indicating possible future research directions. Supplementary tables and figures are provided in the appendix (5.2).

Chapter 2

Background

This work uses the *POLIANNA* dataset to test the *Research Questions* and hypotheses and to generate an automated policy annotation. The following chapter provides a detailed insight into the dataset by highlighting its creation and explaining the underlying logic (see 2.1), presenting its characteristics (see 2.2) and describing the necessary preprocessing steps (see 2.2).

2.1 The POLIANNA Dataset

The *POLIANNA* dataset (*POLIcy design ANNotATions*) is a comprehensive tool that supports the analysis and evaluation of climate policies through *ML*. This section provides a detailed introduction to the dataset, including its creation, structure, and potential applications in research. The dataset was developed by Sewerin et al. [13] to enable systematic and efficient approaches to the analysis of policy design. It comprises annotated text spans from 18 European Union (EU) directives and regulations that focus on climate change and renewable energy. These texts have been extracted from the *EUR-Lex* database, an official EU platform for access to legal documents. *EUR-Lex* provides EU legal acts such as regulations, directives, decisions and opinions in an open format, making it easier to search and compare documents. A multidimensional coding scheme forms the methodological basis of the dataset. It covers three *Layers* of policy design: *Instrument Types*, *Policy Design Characteristics* and *Technology Specificity*, as outlined in the table 5.1. To better understand the composition of these coding *Layers*, we examine the distribution of annotations and the annotation methodology. To illustrate how these *Layers* are represented in the dataset, we analyze the distribution of annotations, as shown in Figures 2.2, 2.3, and 2.4. The *Instrument Types* include policy instruments such as tax incentives, subsidies or regulatory measures. The *Policy Design Characteristics* include elements such as *Actors*, *Compliance Mechanisms* or *Time Frames*. The *Technology Specificity* focuses on the promotion of low-carbon technologies and energy sources. The annotation process was conducted using the open-source software *INCEpTION*, which supports hierarchical structures and overlapping *Tags*. *INCEpTION* is a platform for machine-aided annotation that enables the annotation of complex hierarchical data structures and thus effectively supports *ML* [18]. The annotated text spans contain an average length of three tokens. A *Token* denotes a unit of text separated by spaces or punctuation, such as a word or a number. The short average length illustrates the *Precision* of the annotation, as individual concepts can be accurately identified and separated. To ensure consistency and quality, Inter-Annotator Agreement (IAA) metrics such as the γ score were applied. The γ score is an extended measure of agreement between different annotators that takes into account both the selection of areas and the categorization. This measure corrects the agreement by the expected random value and is particularly useful for complex annotations with overlapping ranges. In addition, the *POLIANNA* dataset provides a robust basis for the development of methods for the automated analysis of policy texts. Potential applications include the identification of trends in policy design, the study of interactions between policies, the derivation of recommendations for action in climate policy, and the development of tools to support manual coding processes. These *Layers* are hierarchically organized into *Features* and *Tags*, as shown in Table 5.1, Table 5.2, and Table 5.3 in the Appendix (5.2).

To leverage the *POLIANNA* dataset for *ML* applications, appropriate preprocessing steps are required. The following subsection details the preprocessing procedures applied to ensure data consistency, enhance interpretability, and prepare the dataset for further analysis.

POLIANNA Dataset Workflow

The workflow for creating the *POLIANNA* dataset involves three main stages: corpus assembly, preprocessing, and annotating. In the corpus assembly stage, policy documents are sourced from the *EUR-Lex* database, prioritized based on relevance, and refined into a corpus of 18 policies. During the preprocessing stage, these documents are split into 448 articles, with irrelevant sections such as preambles and recitals removed to focus on substantive content. In the annotating phase, the articles are imported into the *INCEpTION* tool, where a coding scheme is developed and applied to identify key components, such as *Layers*, *Features*, and *Tags*. Two annotators independently annotate the data, ensuring high IAA and quality [13].

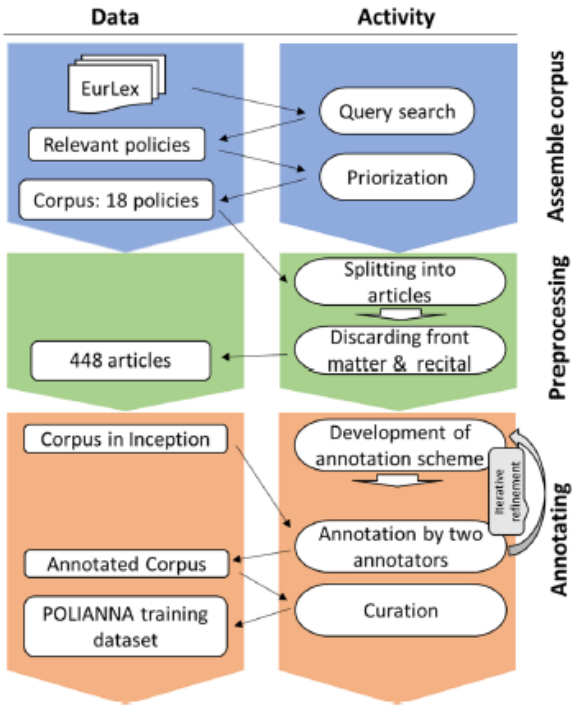

Figure 2.1: Workflow for creating the *POLIANNA* dataset. The process includes corpus assembly from *EUR-Lex*, preprocessing to refine the content into articles, and annotation using *INCEpTION* with a dual-annotator approach to ensure quality [13].

2.2 Data Preprocessing

The preprocessing of the *POLIANNA* dataset is a crucial step to ensure consistency, enhance data quality, and prepare the dataset for *ML* applications. This section outlines the specific procedures applied to refine the dataset and make it suitable for analysis.

Loading the Preprocessed Dataset

The preprocessed *POLIANNA* dataset is loaded from a pickle file to initialize the analysis environment.

```
import pandas as pd
df = pd.read_pickle('/Users/svenlutz/Documents/Studium/UBT/DSP/POLIANNA/data/02
    _processed_to_dataframe/POLIANNA_v1_12/02_processed_to_dataframe/
    preprocessed_dataframe.pkl')
```

Listing 2.1: Loading the Dataset

Data Cleaning

Data cleaning is a crucial step in ensuring the quality and consistency of the dataset. By removing irrelevant or statistically insignificant entries, the dataset becomes more streamlined and reliable for

downstream analysis.

Filtering Out Unfinished Curations

The *Article_State* column contained two rare states: *ANNOTATION_IN_PROGRESS* and *CURATION_IN_PROGRESS*, each occurring only once in the dataset. These entries were removed due to their negligible occurrence and lack of statistical significance, ensuring a streamlined and consistent dataset.

```
df.drop(df[df['Article_State'].isin(['ANNOTATION_IN_PROGRESS', 'CURATION_IN_PROGRESS'])
    ].index, inplace=True)
```

Listing 2.2: Removing Unfinished Curations

Removing Inactive Annotator Columns

The *Finished_Annotators* column was analyzed to identify active contributors.

```
df['Finished_Annotators'].explode().value_counts()
```

Listing 2.3: Analyzing Annotator Contributions

Only annotators A, B, C, and F were active, while E, G, and D did not appear in the dataset. These inactive annotator columns were removed.

```
df.drop(columns=['E', 'G', 'D'], inplace=True)
```

Listing 2.4: Removing Inactive Annotator Columns

Text Data Cleanup

The text data contained various control characters, such as $\backslash r$ (carriage return), $\backslash n$ (line break), and $\backslash xa0$ (non-breaking space), as well as extra spaces. These artifacts were introduced during the data extraction process and can disrupt downstream analysis by affecting text parsing, tokenization, and alignment. Removing these control characters ensures consistent formatting of the text, which is essential for accurate text processing. Additionally, this step enhances readability, simplifies preprocessing workflows, and minimizes the risk of errors during *ML* model training.

```
annotations_df['Text'] = (
    annotations_df['Text']
    .str.replace('\r\n', ' ', regex=False)
    .str.replace('\xa0', ' ', regex=False)
    .str.replace('\n', ' ', regex=False)
    .str.replace(r'\s+', ' ', regex=True)
    .str.strip()
)
```

Listing 2.5: Cleaning Text Data

Annotation Extraction and Structuring

To enable structured analysis, the annotations in the *POLIANNA* dataset were systematically extracted and categorised using regular expressions. This process identifies key annotation elements - *Layers*, *Features* and *Tags* - that serve as target variables for further analysis. The extraction method uses predefined regex patterns to search for annotation markers in the *Curation* column. Each entry is processed by converting it to a string and applying regex matching to extract relevant components. The identified elements are then stored in separate lists for *Layers*, *Features* and *Tags*. To quantify the occurrence of each annotation type, the extracted components are aggregated using Python's *Counter* class, which computes frequency distributions for *Layers*, *Features* and *Tags*. This transformation transforms unstructured text annotations into analysable categorical data, allowing statistical evaluation and visualisation.

```
import re
from collections import Counter

layer_pattern = r"layer:([A-Za-z0-9_]+)"
```

```
feature_pattern = r"feature:([A-Za-z0-9_]+)"
tag_pattern = r"tag:([A-Za-z0-9_]+)"

sum_layers, sum_features, sum_tags = [], [], []

for curation_list in df['Curation']:
    curation_str = " ".join(map(str, curation_list))
    sum_layers.extend(re.findall(layer_pattern, curation_str))
    sum_features.extend(re.findall(feature_pattern, curation_str))
    sum_tags.extend(re.findall(tag_pattern, curation_str))

layer_distribution = Counter(sum_layers)
feature_distribution = Counter(sum_features)
tag_distribution = Counter(sum_tags)
```

Listing 2.6: Extracting Annotation Components

Visualization of Annotations

To better understand the structure and composition of the *POLIANNA* dataset, we present several visualizations that illustrate key characteristics. The dataset classifies policies using three annotation types: *Layers*, *Features*, and *Tags*.

Layers represent the highest level of classification, dividing policy documents into broad analytical categories. *Features* provide more specific thematic elements within these categories, capturing key aspects such as actors, compliance mechanisms, and technology-related factors. Finally, *Tags* serve as the most granular level of classification, offering detailed insights into specific policy attributes, such as regulatory instruments or energy targets.

Layer Distribution

The *Layer Distribution* in the *POLIANNA* dataset highlights the structured approach of the dataset to analysing climate policy documents. The annotations are grouped into three primary *Layers*: *Policy Design Characteristics*, *Technology Specificity*, and *Instrument Types*, each focusing on different aspects of policy design. The largest proportion of annotations, 54.6% (11,380 entries), fall under the *Policy Design Characteristics Layer*, reflecting the dataset's strong emphasis on capturing structural and design-related elements of policies. These include components such as *Compliance Mechanisms*, *Actor* involvement and timelines, which are critical to understanding how policies are formulated and implemented. The second *Layer*, *Technology Specificity*, accounts for 28.7% (5,989 annotations) and demonstrates the dataset's focus on policies aimed at promoting low-carbon technologies and renewable energy solutions. This *Layer* underlines the relevance of technological considerations in modern climate policy design. Finally, *Instrument Types* represents 16.7% (3,487 records) and includes specific policy mechanisms such as tax incentives, subsidies or regulatory measures. This *Layer* provides insights into the practical tools and strategies used to achieve policy objectives.

Feature Distribution

The *Feature Distribution* in the *POLIANNA* dataset highlights the frequency with which policy design components are categorised under different *Features*. Figure 2.3 illustrates the prominence of key *Features*, providing insight into the composition of the dataset and its focus. The most common *Feature* is *Actor*, with a total of 6,021 annotations, representing 28.9% of the dataset. This indicates a strong emphasis on identifying the *Actors* involved in policy implementation. *InstrumentType* is the second most common *Feature*, with 3,388 annotations, representing 16.7 percent of the dataset, reflecting the dataset's focus on policy mechanisms such as tax incentives and subsidies. *TechnologySpecificity* ranks third, with 2,526 annotations or 12.1 percent of the dataset, reflecting the dataset's focus on the promotion of low-carbon technologies. Other *Features*, such as *EnergySpecificity* with 1,908 annotations and Compliance with 1,829 annotations, capture more nuanced aspects of policy design.

Tag Distribution

The *Tag* distribution in the *POLIANNA* dataset provides an overview of the frequency of certain *Tags*, which represent different elements of policy design. Figure 2.4 illustrates the relative prominence of these

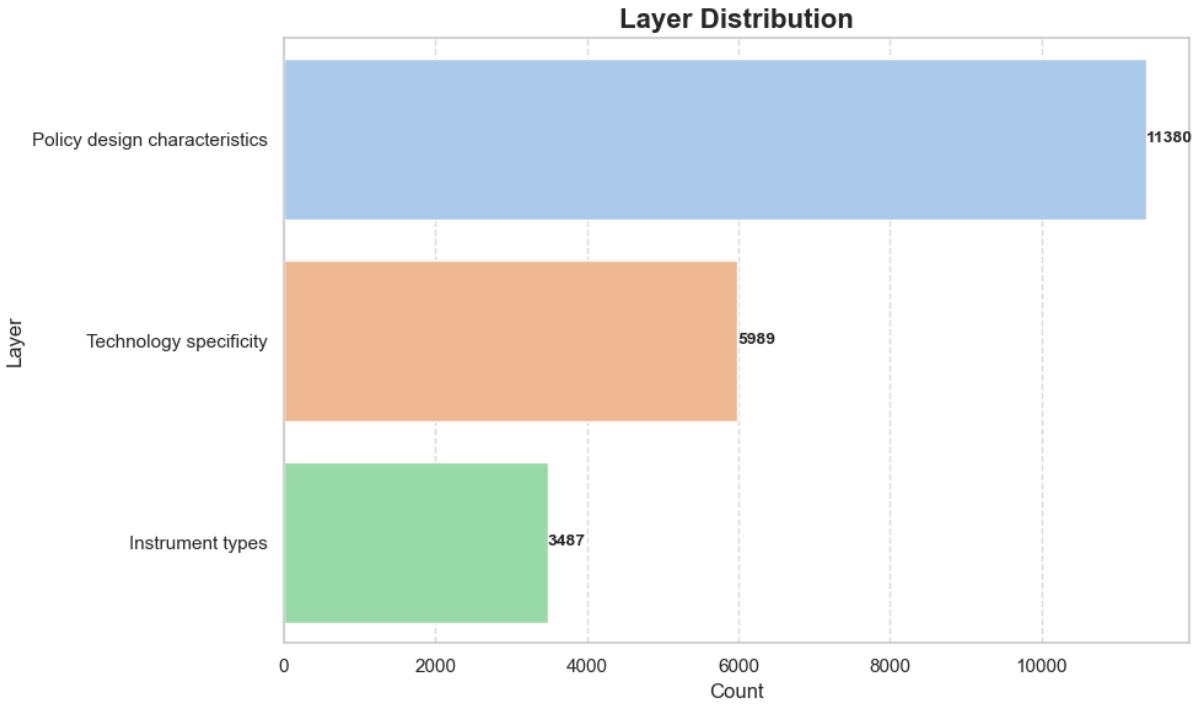

Figure 2.2: *Layer Distribution* in the *POLIANNA* dataset. The dataset is structured into three hierarchical *Layers*: *Policy Design Characteristics* (54.6%, 11,380 entries), *Technology Specificity* (28.7%, 5,989 entries), and *Instrument Types* (16.7%, 3,487 entries). The colour coding highlights distinct *Layers*, with *Policy Design Characteristics* in blue, *Technology Specificity* in orange, and *Instrument Types* in green, emphasizing their different roles in climate policy analysis.

Tags, highlighting key areas of focus within the dataset. The most common *Tag* is *Form_ monitoring* with 1,781 occurrences, reflecting the importance of policy monitoring in the dataset. *Tech_ LowCarbon* and *Energy_ LowCarbon* are the second and third most common *Tags* with 1,468 and 1,358 occurrences respectively, highlighting the focus of the dataset on the promotion of low carbon technologies and energy solutions. *RegulatoryInstr* follows closely with 1,231 annotations, representing the role of regulatory instruments in policy implementation. Other significant *Tags* include *Tech_ Other* (1,058 annotations) and *Authority_ default* (939 annotations), which capture additional technological aspects and administrative structures respectively. Less common *Tags*, such as *Form_ sanctioning* (48 annotations) and *VoluntaryAgmt* (72 annotations), represent more niche components of policy design.

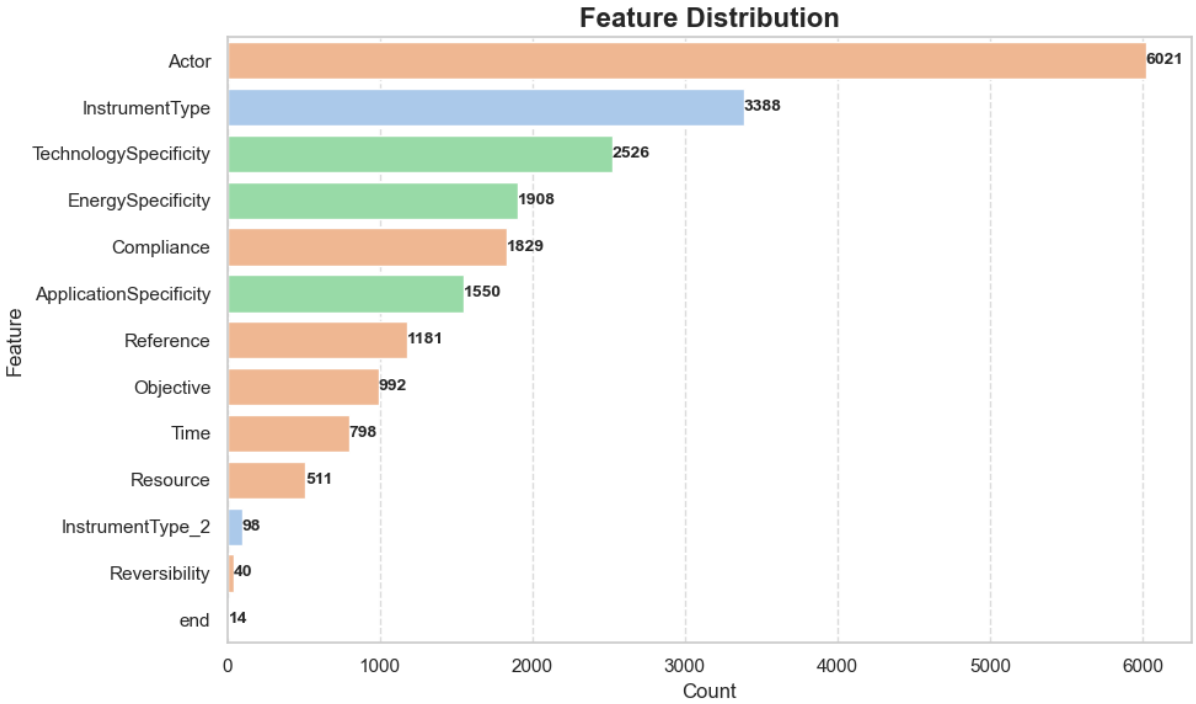

Figure 2.3: *Feature Distribution* in the *POLIANNA* dataset. The most frequent *Feature* is *Actor* (6,021 annotations, 28.9%), followed by *InstrumentType* (3,388 annotations, 16.7%) and *TechnologySpecificity* (2,526 annotations, 12.1%). The colour scheme categorizes *Features* into thematic groups: governance-related elements in orange, technology-focused aspects in green, and regulatory mechanisms in blue. This visualization highlights the dataset's emphasis on stakeholders, policy mechanisms, and technological scope.

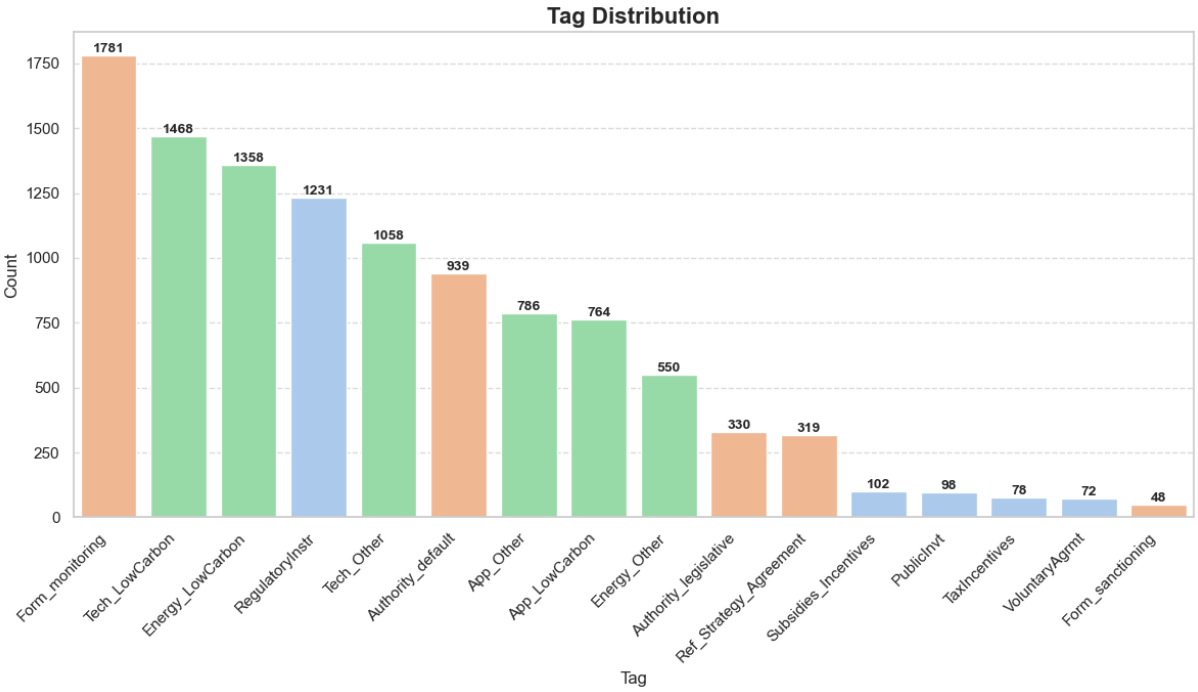

Figure 2.4: *Tag* distribution in the *POLIANNA* dataset. The most common *Tag* is *Form_ monitoring* (1,781 annotations), followed by *Tech_ LowCarbon* (1,468 annotations) and *Energy_ LowCarbon* (1,358 annotations). The colour scheme differentiates policy mechanisms (orange), technological elements (green), and regulatory aspects (blue), visually distinguishing key themes in policy classification.

Chapter 3

Approaches

Since the challenge of this work lies in dealing with unbalanced class distributions, it requires a targeted strategy for data processing and model adaptation. By combining sampling methods, hyperparameter optimisation and the specialised language model *ClimateBERT*, the aim is to achieve the most accurate classification possible. This chapter describes the approaches used and their contribution to answering the *Research Questions*.

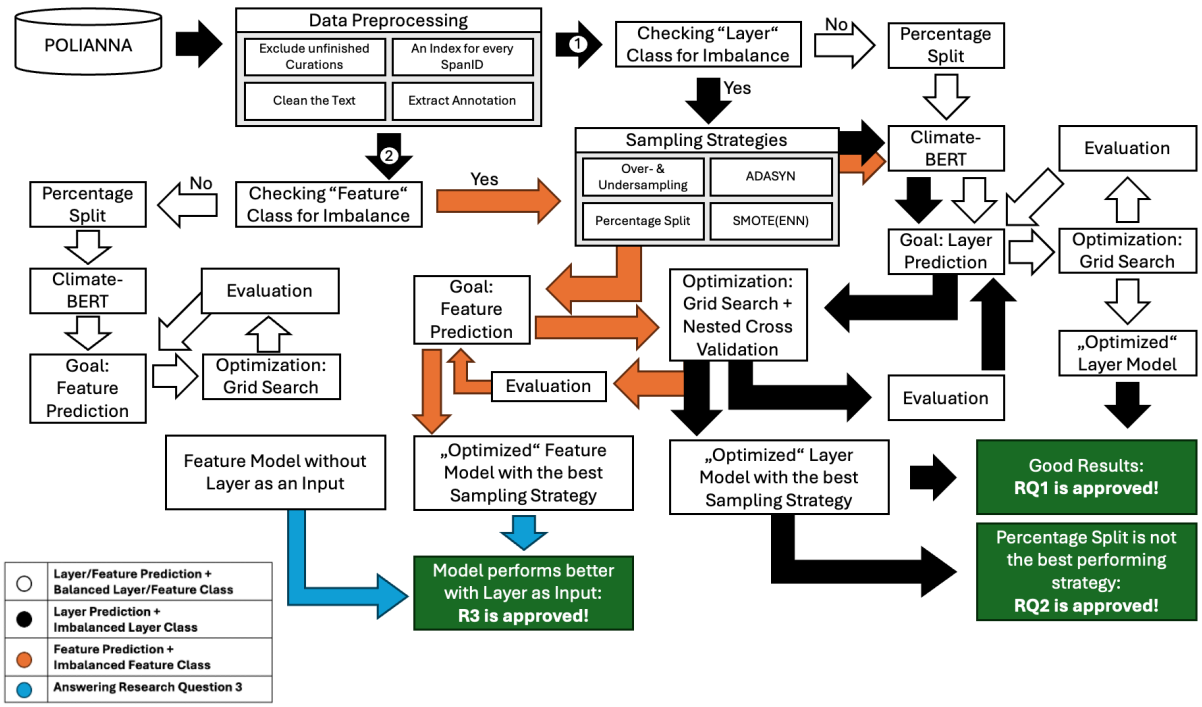

Figure 3.1: Process Pipeline

The figure 3.1 shows the pipeline that will be used in this work, leading to the answers to the three *Research Questions*. First, the *POLIANNA* dataset is transferred to an IDE and read in, followed by data pre-processing steps. The incomplete curations are removed, a row (index) is created for each *SpanID* to allow further processing. Finally, the text is cleaned up and the annotations are extracted.

After the preprocessing steps, the *Layer* class is analysed in step one. Is it balanced? This is then immediately followed by a *Percentage Split*, which is transferred to the *ClimateBERT* transformer model, which in combination with a *Grid Search* should reflect an initial optimised model and its results. If the class is unbalanced, different sampling strategies are tested and the best model with the best hyperparameters is output in combination with *ClimateBERT*, *Grid Search* and *NCV*. If we get good results, the first *Research Question* is answered. If we get better results with a better sampling strategy than with a *Percentage Split*, then the second *Research Question* is also answered.

In the second step, the *Feature Class* is analysed and checked again for balance. This is followed by the same steps as in step 1, except that in order to answer the third *Research Question*, we now check

when better results are achieved. If we pass the predicted *Layer* as input variables or if we just use the text to infer the *Features*. If we get better results, or if both results are very poor in terms of *Accuracy*, we can also answer the third question.

3.1 Introduction to ClimateBERT

NLP has become an essential tool for analysing climate-related discourse, assessing public opinion and supporting effective communication strategies. General-purpose language models such as Bidirectional Encoder Representations from Transformers (BERT) have transformed *NLP* tasks, but their effectiveness in specialised domains such as climate-related texts is often limited due to unique linguistic *Features* and domain-specific vocabulary [17]. To address these limitations, *ClimateBERT* has been developed as a domain-specific language model tailored to the nuances of climate-related texts [17]. Building on pre-trained models such as BERT, *ClimateBERT* undergoes an additional domain-adaptive pre-training phase using a corpus of over 2 million climate-related paragraphs from news articles, corporate disclosures and scientific publications. This phase significantly improves the model's ability to understand the specialised language of climate science, policy and communication, while retaining the bidirectional transformer architecture of BERT. The result is a model that is better equipped to deal with domain-specific challenges such as terminological variation and contextual ambiguity.

The development of *ClimateBERT* follows a structured three-phase training approach, as illustrated in Figure 3.2:

- **General Pretraining**: The model is initially trained on a broad, generic corpus (e.g., Common Crawl) to establish a foundational understanding of language patterns.
- **Domain-Adaptive Pretraining**: The model is further trained using climate-specific corpora to refine its understanding of domain-specific language.
- **Fine-Tuning**: The model is optimized for specific downstream tasks such as text classification, sentiment analysis, and fact-checking [19].

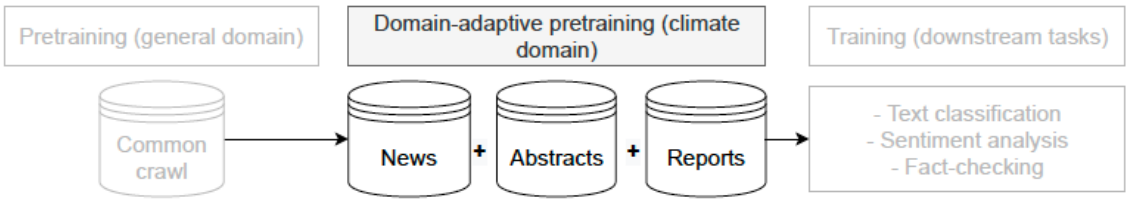

Figure 3.2: Workflow of *ClimateBERT*: From general pretraining to domain-adaptive pretraining on climate-related data, followed by fine-tuning for downstream tasks such as text classification, sentiment analysis, and fact-checking [17].

A crucial component of this workflow is the domain-adaptive pretraining phase, where *ClimateBERT* is exposed to specialized datasets that enhance its ability to process climate-related texts. As depicted in Figure 3.2, this phase integrates three primary sources of climate information:

- **News**: Articles covering climate policies, actions, and natural disasters, providing real-world context for climate discussions.
- **Scientific Abstracts**: Summaries of peer-reviewed research papers, contributing technical depth and specialized terminology.
- **Corporate Reports**: Sustainability disclosures and climate-related statements from companies, offering insights into corporate climate communication and risk management.

By using these datasets, *ClimateBERT* effectively captures the distinctive linguistic patterns and specialised terminology that are essential for understanding climate-related discourse. This enhanced understanding enables the model to excel in several downstream applications. A key task is text classification, which allows the model to determine whether a given text is climate-related. In addition, sentiment analysis helps to assess whether climate discussions convey a positive, neutral or negative perspective. Another key application is fact-checking, where the model assesses the validity of climate-related claims, supporting efforts to combat misinformation and ensure accurate climate communication.

Applications

Understanding climate-related texts requires advanced *NLP* models capable of processing highly technical language, domain-specific terminology, and evolving discourse patterns. *ClimateBERT* addresses these challenges by leveraging its domain-adaptive training, allowing it to excel in various climate-specific tasks. These capabilities make it particularly suitable for applications where precise classification, sentiment detection, and fact verification are crucial.

ClimateBERT has demonstrated substantial advancements in addressing climate-specific *NLP* tasks, including:

- **Sentiment Analysis**: Assessing public sentiment on climate issues in social media and news [20].
- **Text Classification**: Categorizing corporate disclosures and scientific texts based on climate-related topics [21].
- **Topic Modelling**: Identifying trends in climate change narratives across time and regions [20].
- **Fact-Checking**: Verifying the *Accuracy* of climate-related claims in media and research [17].

By enabling accurate analysis of specialized texts, *ClimateBERT* represents a significant step forward in the use of *AI* for climate research and policy analysis. Its ability to process complex policy documents, detect sentiment shifts, and classify text based on climate-related content makes it an invaluable tool for structured policy assessments. Given its specialized training and adaptability to regulatory texts, *ClimateBERT* is the perfect model for the tasks in this work.

3.2 Tackling Class Imbalance with Diverse Sampling Strategies

Class imbalance is a persistent challenge in *ML* [22], occurring when certain classes are significantly underrepresented [23]. As clearly observed in our dataset, this issue is evident in the *Layers* (Figure 2.2) and *Features* (Figure 2.3), where certain categories appear far less frequently than others. This imbalance often skews model performance, leading to a bias toward the majority class while overlooking critical insights from the minority class. In policy text analysis, this issue is particularly problematic, as underrepresented policy elements may contain crucial information necessary for comprehensive evaluations. In the context of this work, class imbalance poses an even greater challenge, as misclassified *Layers* can cascade into errors in *Feature* and *Tag Classification*. Since *Features* and *Tags* depend on the correct identification of *Layers*, errors at the *Layer* level propagate, leading to compounded misclassifications and reducing the overall reliability of the model.

Ensuring a balanced representation of classes is therefore essential to maintaining structural integrity in policy text analysis. A simple *Percentage Split* serves as a baseline for evaluation, but it does not sufficiently address class imbalance, as models inherently favor more frequent patterns. To potentially mitigate this issue, sampling strategies can be applied to rebalance datasets by adjusting sample distributions or generating synthetic examples [23]. These techniques are hypothesized to enhance the model's ability to learn from minority classes, improve robustness, and ensure more accurate policy classifications. However, their actual effectiveness in this specific context remains an open *Research Question*.

In the *POLIANNA* framework, policy texts are annotated with diverse policy design elements, some of which appear infrequently. Without addressing this imbalance, the model risks underrepresenting these critical *Features*, leading to low recall and incomplete policy analysis. We therefore explore the impact of *Oversampling*, *Undersampling*, and synthetic data generation techniques (*ADASYN, SMOTE, SMOTEENN*), aiming to determine whether they can improve the model's ability to capture rare but crucial policy *Features*. The effectiveness of these strategies in enhancing model fairness, improving recall for underrepresented policy categories, and increasing overall robustness will be systematically evaluated as part of this work.

For visualisation, we used t-Distributed Stochastic Neighbor Embedding (t-SNE), a non-linear dimensionality reduction technique that projects high-dimensional data into a lower-dimensional space while preserving pairwise similarities between data points. It is particularly effective for visualising complex datasets because it captures local structure and clusters similar data points together. Each point in the t-SNE plot represents a data sample, with its position indicating its similarity to other samples, making it easier to interpret patterns and relationships within the data. The colour coding in the visualisation highlights different *Layers*: *Policy Design Characteristics* in blue, *Technology Specificity* in orange, and

Instrument Types in green. This differentiation helps to identify distinct patterns and relationships within the dataset, making the structure of the data more interpretable.

These methods, which are based on mathematical principles, are described in detail below:

Oversampling

Oversampling balances class distributions by duplicating samples from the minority class [24]. For a dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$, where $y_i \in \{0, 1\}$ are the class labels, the minority class C_1 is expanded to match the size of the majority class C_0:

$$D' = D \cup \{(x_j, y_j) \mid (x_j, y_j) \in C_1\}, \quad \text{for } |C_1| < |C_0|.$$

While *Oversampling* effectively addresses class imbalance, it can lead to overfitting if the duplicated samples dominate the training process.

```
from imblearn.over_sampling import RandomOverSampler

oversampler = RandomOverSampler(random_state=42)
X_train_resampled, y_train_resampled = oversampler.fit_resample(X_train, y_train)
```

Listing 3.1: Applying Oversampling

Figure 3.3 illustrates the oversampled dataset visualized in a two-dimensional space using t-SNE dimensionality reduction. *Oversampling* preserves the inherent structure of each class while ensuring a balanced representation, reducing bias towards the majority class and improving performance on underrepresented classes.

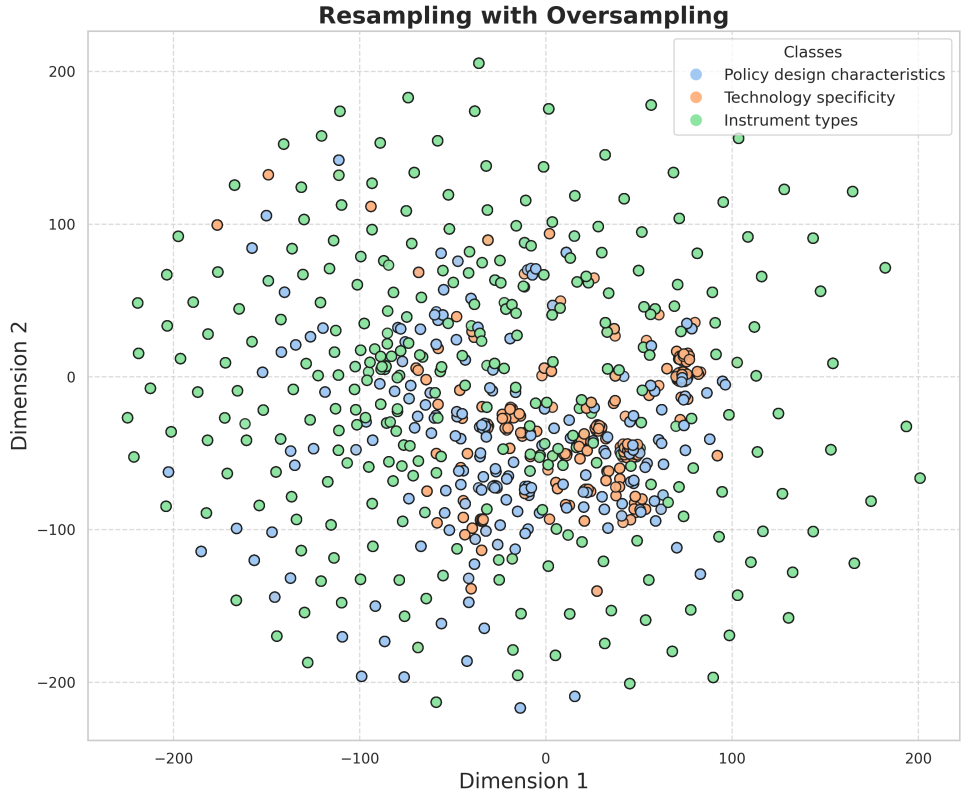

Figure 3.3: t-SNE visualization of the oversampled dataset. The colour coding highlights different *Layers*, with *Policy Design Characteristics* in blue, *Technology Specificity* in orange and *Instrument Types* in green.

Undersampling

Undersampling addresses class imbalance by removing samples from the majority class [24]. The modified dataset is defined as:

$$D' = D \setminus \{(x_k, y_k) \mid (x_k, y_k) \in C_0,\ |C_0| > |C_1|\}.$$

Although *Undersampling* reduces bias, it risks discarding valuable information, potentially limiting the model's generalizability.

```
from imblearn.under_sampling import RandomUnderSampler

undersampler = RandomUnderSampler(random_state=42)
X_train_resampled, y_train_resampled = undersampler.fit_resample(X_train, y_train)
```

Listing 3.2: Applying Undersampling

Figure 3.4 visualizes the dataset after *Undersampling* in a t-SNE-reduced space. Each point represents a data sample, and its position reflects its similarity to others. By removing majority class samples, the distribution is balanced across classes. However, the reduction in majority class data may lead to the loss of useful information, which could impact model performance.

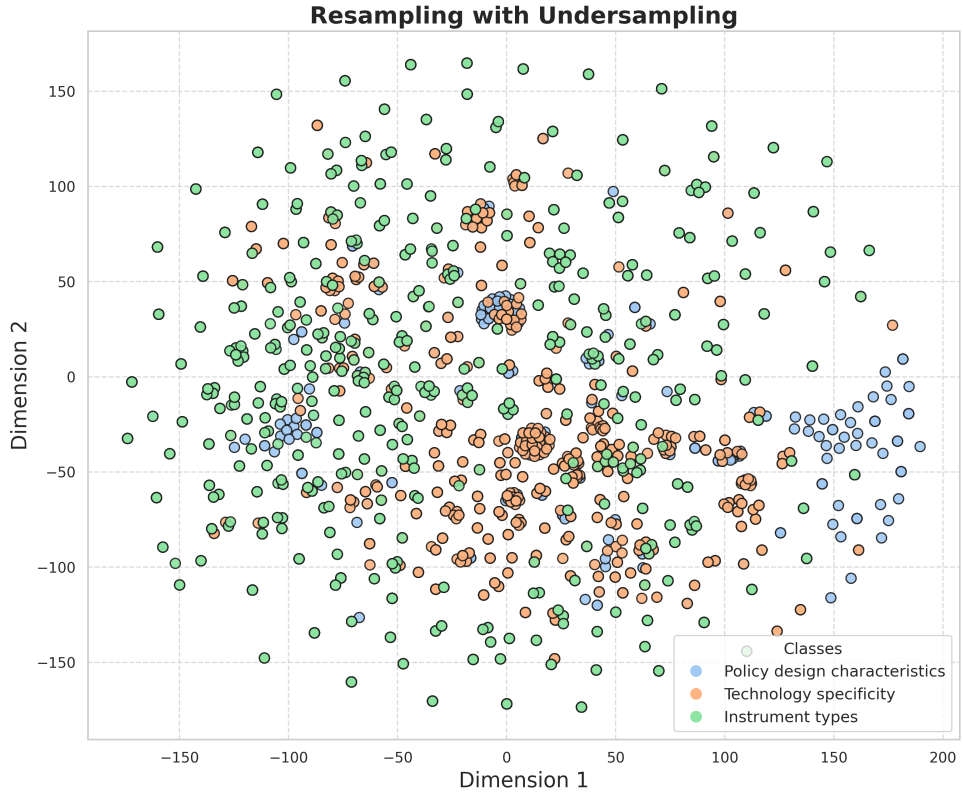

Figure 3.4: t-SNE visualization of the undersampled dataset. The colour coding highlights different *Layers*, with *Policy Design Characteristics* in blue, *Technology Specificity* in orange and *Instrument Types* in green.

SMOTE (Synthetic Minority Over-sampling Technique)

SMOTE generates synthetic samples for the minority class by interpolating between existing samples [25]. For a minority sample x_i, a synthetic sample is generated as:

$$x_{\text{new}} = x_i + \lambda(x_i - x_{\text{nearest}}), \quad \lambda \sim U(0,1),$$

where x_{nearest} is one of the k nearest neighbors of x_i in the *Feature* space. This method increases diversity in the dataset, reducing overfitting risks.

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
```

Listing 3.3: Applying SMOTE

Figure 3.5 shows the result of *SMOTE*, with the dataset visualized in a t-SNE-reduced space. The interpolated samples form new clusters within the minority classes, preserving the dataset's structural integrity while addressing class imbalance.

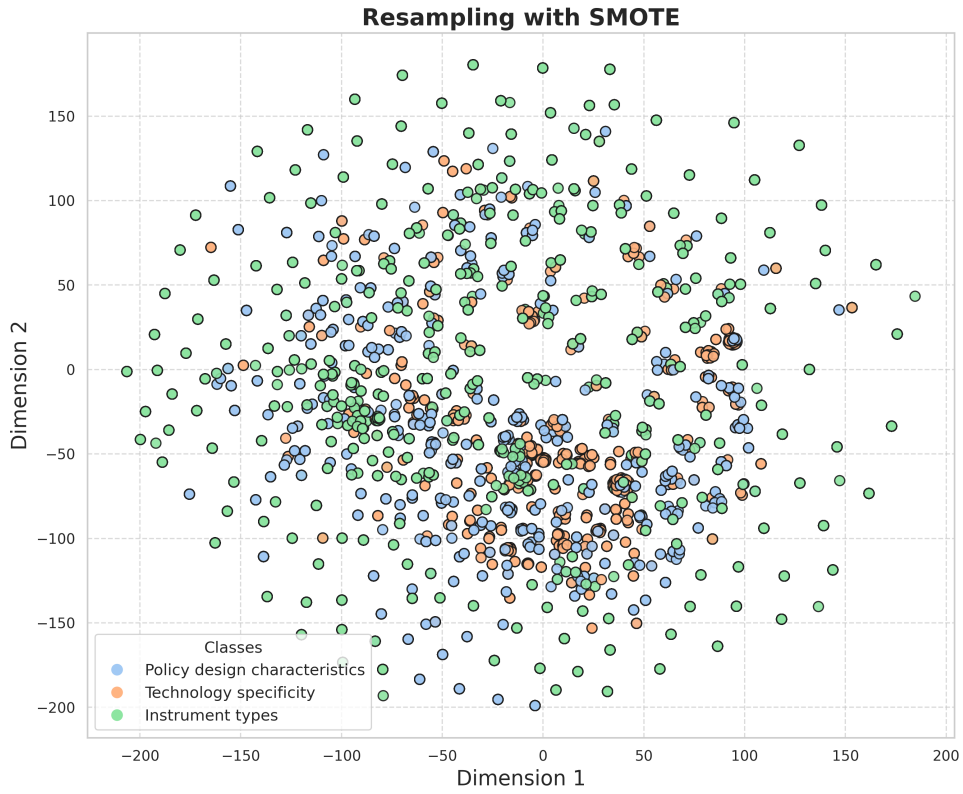

Figure 3.5: t-SNE visualization of the dataset after applying *SMOTE*. The colour coding highlights different *Layers*, with *Policy Design Characteristics* in blue, *Technology Specificity* in orange and *Instrument Types* in green.

SMOTEENN (SMOTE with Edited Nearest Neighbors)

SMOTEENN combines *SMOTE* with *Edited Nearest Neighbors (ENN)*, which removes noisy samples after synthetic generation [26]. A sample is retained only if its class matches the majority of its k nearest neighbors:

$$\text{Retain } (x_i, y_i) \text{ if } \sum_{j=1}^{k} I(y_i = y_j) > \frac{k}{2}.$$

```
from imblearn.combine import SMOTEENN

smoteenn = SMOTEENN(random_state=42)
X_train_resampled, y_train_resampled = smoteenn.fit_resample(X_train, y_train)
```

Listing 3.4: Applying SMOTEENN

Figure 3.6 visualizes the dataset after applying *SMOTEENN*. Synthetic samples improve minority class representation, while *ENN* removes misclassified points to enhance class separability and reduce noise.

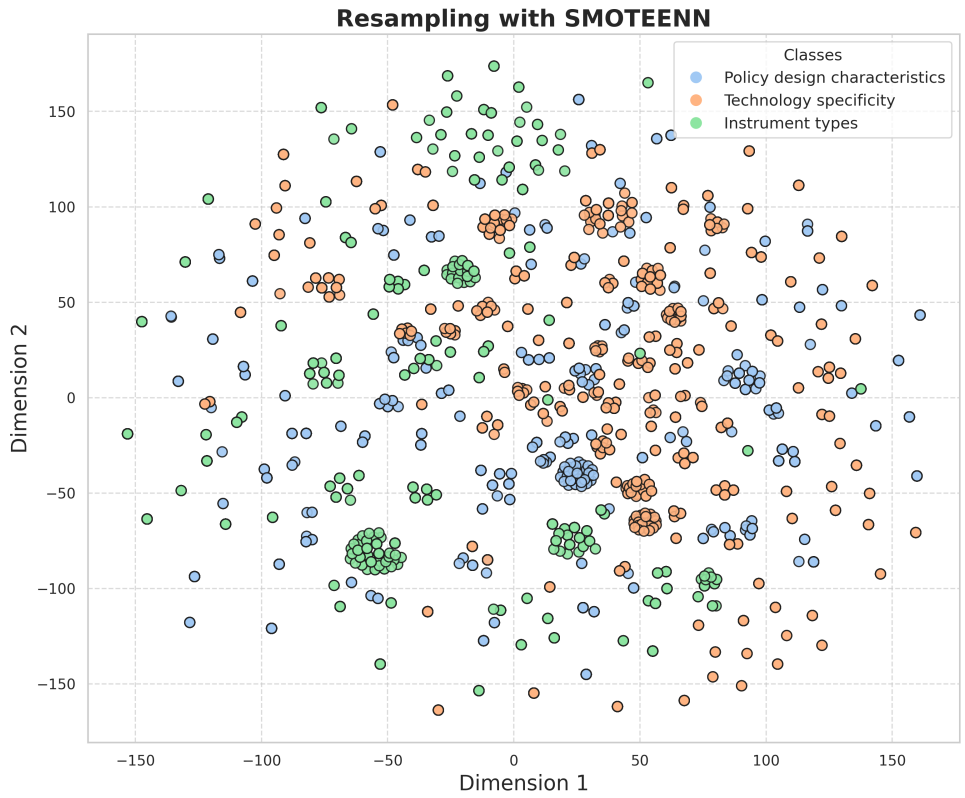

Figure 3.6: t-SNE visualization of the dataset after applying *SMOTEENN*. The colour coding highlights different *Layers*, with *Policy Design Characteristics* in blue, *Technology Specificity* in orange and *Instrument Types* in green.

ADASYN (Adaptive Synthetic Sampling)

ADASYN generates synthetic samples near difficult-to-classify instances [27]. For each minority sample x_i, the number of synthetic samples G_i is determined by the classification difficulty d_i:

$$d_i = \frac{\text{Number of majority neighbors}}{k}.$$

These samples are generated similarly to *SMOTE* but weighted by d_i, targeting challenging regions to enhance model learning.

```
from imblearn.over_sampling import ADASYN

adasyn = ADASYN(random_state=42)
X_train_resampled, y_train_resampled = adasyn.fit_resample(X_train, y_train)
```

Listing 3.5: Applying ADASYN

Figure 3.7 demonstrates how *ADASYN* focuses on regions where classification is difficult. The scatterplot, visualized in t-SNE-reduced space, reflects targeted diversity in minority classes, resulting in a more nuanced and balanced dataset.

Each resampling strategy addresses class imbalance with unique mechanisms and trade-offs. Evaluating their impact systematically is crucial to improving fairness, recall for underrepresented classes, and robustness within the *POLIANNA* framework.

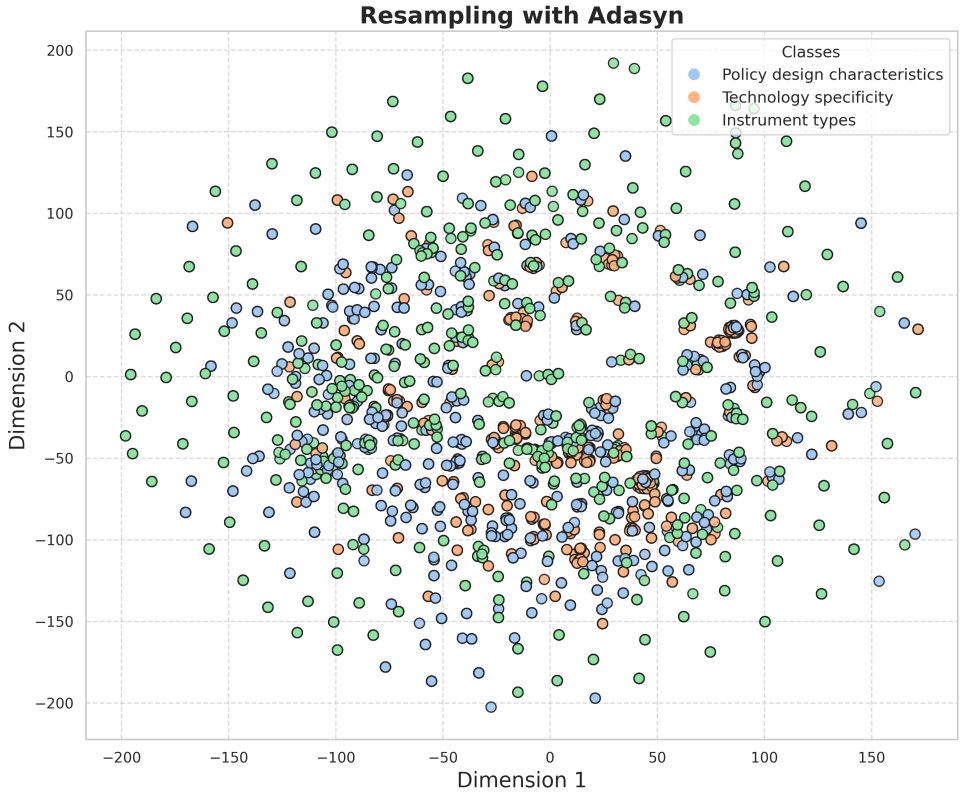

Figure 3.7: t-SNE visualization of the dataset after applying *ADASYN*. The colour coding highlights different *Layers*, with *Policy Design Characteristics* in blue, *Technology Specificity* in orange and *Instrument Types* in green.

3.3 Hyperparameter Optimization and Evaluation

HPO helps improve model performance by finding the best settings for training. In this work, we fine-tune the transformer-based *ClimateBERT* model while testing different hyperparameter configurations. To ensure fair and reliable performance evaluation, we use *NCV* together with *Grid Search*. This method allows us to systematically test different learning rates while keeping other parameters fixed, reducing the risk of overfitting and ensuring meaningful comparisons between different sampling strategies.

Nested Cross-Validation

Hyperparameter tuning and model evaluation are inherently interconnected; optimizing hyperparameters on the same dataset used for testing can lead to overoptimistic and biased results. To address this challenge, we employ *NCV*, a two-stage cross-validation framework designed to separate hyperparameter selection from final model evaluation.

NCV consists of:

- **Outer loop**: Splits the dataset into training and test sets, ensuring that the final evaluation is conducted on unseen data.
- **Inner loop**: Utilizes *Stratified K-Fold Cross-Validation* to tune hyperparameters while preserving class balance.

This nested approach ensures that hyperparameters are optimized within the inner loop while model evaluation remains independent in the outer loop. Since different sampling strategies modify the class distribution, *NCV* is particularly advantageous in this work, as it allows us to fairly compare resampling methods without introducing biases due to hyperparameter selection.

Grid Search for Hyperparameter Optimization

To explore the best hyperparameters for fine-tuning *ClimateBERT*, we employ *Grid Search* within the inner loop of *NCV*. Due to computational constraints, we focus on optimizing the learning rate while keeping the batch size fixed. This allows for efficient resource allocation while ensuring a systematic search for optimal model configurations.

The key hyperparameters explored in this work are:

- **Learning Rate (η)**: Values tested include {2e-5, 5e-5}, representing commonly used rates for fine-tuning transformer models.
- **Batch Size**: Fixed at 8 to manage memory constraints and maintain training stability.
- **Epochs**: Set to 2 to balance computational efficiency and model convergence.
- **Dropout Rate**: Fixed at 0.1, based on findings from preliminary experiments.

Grid Search systematically iterates over the predefined learning rates, training multiple model instances and selecting the best-performing configuration based on validation performance. The optimal hyperparameters are then used to train the model on the entire training set before final evaluation in the outer loop. This structured process ensures that:

- Hyperparameter tuning remains independent of the final model evaluation.
- Resampling strategies are compared under controlled and consistent conditions.
- Overfitting is minimized by preventing information leakage from the test set into parameter selection.

The *HPO* and evaluation process was implemented using a structured pipeline. The outer loop applied *StratifiedKFold* cross-validation ($N_{\text{outer}} = 2$), ensuring balanced class distributions across training and test sets. Within each fold, *Grid Search* was applied to explore different learning rates while maintaining a fixed batch size. The following code snippet provides the core setup for *Grid Search* and model evaluation:

```
import logging
import numpy as np
import pandas as pd
import torch
from pathlib import Path
from transformers import AutoModelForSequenceClassification, AutoTokenizer, Trainer,
    TrainingArguments
from sklearn.model_selection import StratifiedKFold
import datasets
import os

os.environ["TOKENIZERS_PARALLELISM"] = "false"

logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message
    )s")

LEARNING_RATES = [2e-5, 5e-5]
BATCH_SIZES = [8]
MODEL_NAME = "ClimateBERT/distilroberta-base-climate-f"
N_OUTER_SPLITS = 2
N_INNER_SPLITS = 2
N_EPOCHS = 2
```

Listing 3.6: Grid Search Setup and Execution

The function *train_and_evaluate*, shown in Listing 3.7, fine-tunes *ClimateBERT* using training and validation datasets while optimising key hyperparameters. It tokenises the input text, converts it into a dataset compatible with the *Trainer* API and initialises *ClimateBERT* with a classification head. Training parameters such as batch size, learning rate and epochs are defined using the *TrainingArguments* class. The *Trainer* API fine-tunes the model and evaluates its performance, returning key metrics for comparing training configurations and sampling strategies. This structured approach uses transfer learning and systematic hyperparameter tuning to improve classification performance.

```python
def train_and_evaluate(train_texts, train_labels, val_texts, val_labels, model_name,
    learning_rate, batch_size, n_epochs):
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    train_encodings = tokenizer(list(train_texts), truncation=True, padding=True,
        max_length=512)
    val_encodings = tokenizer(list(val_texts), truncation=True, padding=True,
        max_length=512)

    train_dataset = datasets.Dataset.from_dict({
        "input_ids": train_encodings["input_ids"],
        "attention_mask": train_encodings["attention_mask"],
        "labels": train_labels.to_numpy().tolist()
    })
    val_dataset = datasets.Dataset.from_dict({
        "input_ids": val_encodings["input_ids"],
        "attention_mask": val_encodings["attention_mask"],
        "labels": val_labels.to_numpy().tolist()
    })

    model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=
        len(set(train_labels)))

    training_args = TrainingArguments(
        output_dir="./results",
        evaluation_strategy="epoch",
        per_device_train_batch_size=batch_size,
        per_device_eval_batch_size=batch_size,
        learning_rate=learning_rate,
        num_train_epochs=n_epochs,
        logging_dir="./logs",
        logging_steps=10,
        save_strategy="epoch",
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=val_dataset,
        compute_metrics=compute_metrics,
    )
    trainer.train()
    return trainer.evaluate()
```

Listing 3.7: Core Training Function

Each sampling strategy was evaluated separately, ensuring a fair comparison between models trained with different data augmentation techniques.

Evaluation Metrics

To assess model performance, we employ standard classification metrics that provide a comprehensive evaluation of predictive *Accuracy* and reliability.

- **True Positive (TP)**: A policy text excerpt classified as an *InstrumentType* annotation correctly matches a ground truth annotation. For example, a span labeled as *Tax Incentives* in a directive discussing financial support mechanisms for renewable energy deployment.
- **True Negative (TN)**: A policy document section not related to policy design is correctly identified as irrelevant. For instance, a legal preamble discussing procedural aspects of a law (e.g., references to legislative approval) is correctly ignored by the classifier.
- **False Positive (FP)**: A classifier mistakenly assigns a *TechnologySpecificity* label to a generic statement about climate targets. For example, a sentence mentioning "reducing emissions by 30% by 2030" is labeled as *Low-Carbon Technology*, even though it does not explicitly reference technology.
- **False Negative (FN)**: A classifier fails to identify a *Compliance* annotation in a text specifying monitoring requirements. For instance, if a policy states that "member states must report compliance measures annually," but the classifier does not tag it as *Monitoring Form*, leading to a missing annotation.

The following key metrics are used to evaluate classification performance:

Accuracy: Measures the overall proportion of correctly classified instances:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

Precision (Positive Predictive Value, PPV): Represents the fraction of correctly identified positive instances among all predicted positives:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.2}$$

Recall (Sensitivity, True Positive Rate): Evaluates the model's ability to correctly identify actual positive instances:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.3}$$

F1-score: The harmonic mean of *Precision* and recall, providing a balanced assessment between *Precision* and recall:

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.4}$$

Macro-Averaged F1-score

Since this work involves multi-class classification, the standard *F1-Score* must be extended to account for multiple classes. The *Macro-Averaged F1-Score* is computed by calculating the *F1-Score* separately for each class and then averaging them equally:

$$\text{F1}_{\text{macro}} = \frac{1}{C} \sum_{i=1}^{C} \text{F1}_i \tag{3.5}$$

where C is the total number of classes, and F1_i is the *F1-Score* for class i. This approach ensures that each class is weighted equally, preventing majority-class bias and providing a balanced evaluation of model performance. The *Grid Search* used in this work optimized the *Macro-Averaged F1-Score*. This metric was chosen for the following reasons:

- **Class Imbalance:** Since some categories appear less frequently, *Accuracy* alone would be misleading. The *F1-Score* ensures that both *Precision* and recall are considered, reducing bias toward majority classes.
- **Balanced Evaluation:** The *Macro-Averaged F1-Score* computes the *F1-Score* independently for each class and then takes the average, giving equal weight to all classes, regardless of their frequency.
- **Generalization Performance:** Models optimized for *F1-Score* tend to achieve a better trade-off between missing positive instances (false negatives) and incorrectly predicting positives (false positives), making them more suitable for imbalanced datasets.

The *compute_ metrics* function implements these metrics in the evaluation pipeline:

```
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return {
        "accuracy": accuracy_score(labels, predictions),
        "f1": f1_score(labels, predictions, average="macro"),
        "precision": precision_score(labels, predictions, average="macro"),
        "recall": recall_score(labels, predictions, average="macro"),
    }
```

Listing 3.8: Evaluation Metric Computation

By optimizing for the *Macro-Averaged F1-Score*, we ensure that the model is evaluated fairly across all classes, preventing dominance by majority classes while maintaining a balanced classification performance.

This work examines the different approaches that have been tested to ensure effective forecasting for climate policy.

3.4 Approach 1: The Combination of Sampling, Hyperparameter Optimization, and ClimateBERT

This work integrates three key components—**data resampling, *HPO*, and transformer-based classification**—to improve model performance in handling class-imbalanced text data. The goal is to assess how different sampling strategies impact the fine-tuning of *ClimateBERT* while systematically optimizing hyperparameters.

- **Sampling Strategies:** To address class imbalance, we apply multiple resampling techniques, including *Oversampling, Undersampling, SMOTE, ADASYN, and SMOTEENN*. These methods modify the training distribution to mitigate model bias toward majority classes.
- ***HPO*:** A *Grid Search* within a *NCV* framework is employed to optimize key hyperparameters, focusing on the learning rate while keeping the batch size fixed due to computational constraints.
- ***ClimateBERT* Fine-Tuning:** The pre-trained *ClimateBERT* model is fine-tuned using resampled datasets and optimized hyperparameters. Model performance is evaluated using the *Macro-Averaged F1-Score*, ensuring balanced assessment across all classes.

By combining these techniques, we systematically evaluate how different sampling methods influence the classification performance of *ClimateBERT*. The structured approach ensures that models are trained under consistent conditions, allowing for a fair comparison of resampling strategies and their impact on text classification tasks.

```
    def process_strategy(strategy, output_folder, model_name):
    logging.info(f"Processing strategy: {strategy}")
    data_file = Path(output_folder) / strategy / f"{strategy}_resampled.csv"

    if not data_file.exists():
        logging.warning(f"Data file not found for strategy {strategy}")
        return None

    data = pd.read_csv(data_file)
    texts, labels = data['Text'], data['Layer']

    outer_cv = StratifiedKFold(n_splits=N_OUTER_SPLITS, shuffle=True, random_state=42)

    strategy_results = {
        "best_params": None,
        "best_results": None,
        "tested_combinations": []
    }

    best_f1 = 0

    for train_idx, test_idx in outer_cv.split(texts, labels):
        train_texts, test_texts = texts.iloc[train_idx], texts.iloc[test_idx]
        train_labels, test_labels = labels.iloc[train_idx], labels.iloc[test_idx]

        param_grid = {
            "learning_rate": LEARNING_RATES,
            "batch_size": BATCH_SIZES
        }

        for lr in LEARNING_RATES:
            for bs in BATCH_SIZES:
                inner_scores = []
                metrics_list = []

                inner_cv = StratifiedKFold(n_splits=N_INNER_SPLITS, shuffle=True,
                    random_state=42)

                for inner_train_idx, val_idx in inner_cv.split(train_texts,
                    train_labels):
                    inner_train_texts, val_texts = train_texts.iloc[inner_train_idx],
                        train_texts.iloc[val_idx]
                    inner_train_labels, val_labels = train_labels.iloc[inner_train_idx
                        ], train_labels.iloc[val_idx]

```

```
                        eval_results = train_and_evaluate(inner_train_texts,
                            inner_train_labels, val_texts, val_labels, model_name, lr, bs,
                            N_EPOCHS)
                        inner_scores.append(eval_results.get("eval_f1", 0))
                        metrics_list.append(eval_results)

                if not metrics_list:
                    logging.warning("metrics_list is empty, skipping parameter
                        evaluation.")
                    continue

                avg_metrics = {metric: np.mean([m[metric] for m in metrics_list]) for
                    metric in metrics_list[0].keys()}

                strategy_results["tested_combinations"].append({
                    "learning_rate": lr,
                    "batch_size": bs,
                    "metrics": avg_metrics
                })

                if avg_metrics.get("eval_f1", 0) > best_f1:
                    best_f1 = avg_metrics["eval_f1"]
                    strategy_results["best_params"] = {"learning_rate": lr, "batch_size
                        ": bs}
                    strategy_results["best_results"] = avg_metrics

    logging.info(f"Best parameters for {strategy}: {strategy_results['best_params']}")

    results_path = Path("results") / f"{strategy}_results.json"
    results_path.parent.mkdir(parents=True, exist_ok=True)

    pd.DataFrame(strategy_results["tested_combinations"]).to_json(results_path, orient=
        "records", indent=4)

    return strategy_results
```

Listing 3.9: Layer Prediction

The function *process_strategy*, shown in Listing 3.9, evaluates different resampling strategies for *Layer Prediction* using *NCV*. It loads the resampled data set, divides it into training and test sets using *StratifiedKFold*, and systematically explores different hyperparameter configurations. The function iterates over predefined learning rates and batch sizes, performing internal cross-validation to fine-tune the model and compute evaluation metrics. The best performing hyperparameter combination is selected based on the *Macro-Averaged F1-Score*. The results, including tested hyperparameter combinations and performance metrics, are stored in JSON format for further analysis. This structured approach ensures an unbiased evaluation of different sampling strategies in the context of *Layer Prediction*.

3.5 Approach 2: Feature Prediction without the Layer

In this approach, we evaluate the impact of excluding the *Layer* information when predicting the *Feature* category. The goal is to determine whether the model can accurately classify *Feature* based solely on textual input or if the additional *Layer* information significantly enhances performance.

- **Feature Prediction with Layer:** The model is trained using both the textual data and the predicted *Layer* as an additional input *Feature*. The hypothesis is that incorporating the *Layer* helps capture hierarchical relationships and improves classification *Accuracy*.
- **Feature Prediction without Layer:** The model is trained using only textual data to predict the *Feature*, omitting the *Layer* information. This allows for direct comparison and an assessment of how much the *Layer* contributes to classification performance.

Both models are fine-tuned using *ClimateBERT* with a fixed learning rate and batch size, ensuring consistency across experiments. Performance is compared using classification metrics such as *Macro-Averaged F1-Score*, *Precision*, and recall. To conduct this experiment, two separate training pipelines were implemented: one incorporating the predicted *Layer* as an additional *Feature*, and another using only textual input. The following code snippets illustrate both implementations.

```python
def train_and_predict_feature_with_layer(data_path, model_name, learning_rate,
    batch_size, output_csv_path):
    logging.info("Training to predict Feature using Text and Predicted_Layer...")

    data = pd.read_csv(data_path)
    texts = data["Text"]
    predicted_layer = data["Predicted_Layer"]
    target_feature = data["Feature"]

    tokenizer = AutoTokenizer.from_pretrained(model_name)
    encodings = tokenizer(
        list(texts), truncation=True, padding=True, max_length=512
    )

    input_features = {
        "input_ids": encodings["input_ids"],
        "attention_mask": encodings["attention_mask"],
        "predicted_layer": predicted_layer.tolist()
    }

    dataset = datasets.Dataset.from_dict({
        **input_features,
        "labels": target_feature.astype("category").cat.codes.tolist()
    })

    num_labels = len(data["Feature"].unique())
    model = AutoModelForSequenceClassification.from_pretrained(
        model_name, num_labels=num_labels
    )

    training_args = TrainingArguments(
        output_dir="./results_with_layer",
        evaluation_strategy="no",
        per_device_train_batch_size=batch_size,
        learning_rate=learning_rate,
        num_train_epochs=2,
        save_strategy="no",
        logging_dir="./logs_with_layer",
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=dataset,
    )

    trainer.train()

    predictions = trainer.predict(dataset)
    predicted_features = np.argmax(predictions.predictions, axis=-1)

    data["Predicted_Feature_With_Layer"] = predicted_features
    data.to_csv(output_csv_path, index=False)

    logging.info(f"Predictions with Predicted_Layer saved to {output_csv_path}")
```

Listing 3.10: Feature Prediction with Layer

The function *train_and_predict_feature_with_layer*, shown in Listing 3.10, trains a model to predict *Features* using both text and the previously predicted *Layer* information. It loads the dataset, extracts text, predicted *Layers*, and target *Features*, then tokenizes the text using the *AutoTokenizer* from the pre-trained *ClimateBERT* model. The encoded text and predicted *Layer* are combined into a dataset format compatible with the *Trainer* API, where labels are encoded as categorical values. A classification model is initialized with the appropriate number of output labels, and training is configured using the *TrainingArguments* class. The model is fine-tuned on the dataset, and predictions are made on the same data. Finally, the predicted *Feature* labels are added to the dataset and saved as a CSV file. This approach enhances *Feature Prediction* by incorporating additional *Layer* information, potentially improving classification *Accuracy*.

```python
def train_and_predict_feature_without_layer(data_path, model_name, learning_rate,
    batch_size, output_csv_path):
```

```
    logging.info("Training to predict Feature using only Text...")

    data = pd.read_csv(data_path)
    texts = data["Text"]
    target_feature = data["Feature"]

    tokenizer = AutoTokenizer.from_pretrained(model_name)
    encodings = tokenizer(
        list(texts), truncation=True, padding=True, max_length=512
    )

    dataset = datasets.Dataset.from_dict({
        "input_ids": encodings["input_ids"],
        "attention_mask": encodings["attention_mask"],
        "labels": target_feature.astype("category").cat.codes.tolist()
    })

    num_labels = len(data["Feature"].unique())
    model = AutoModelForSequenceClassification.from_pretrained(
        model_name, num_labels=num_labels
    )

    training_args = TrainingArguments(
        output_dir="./results_without_layer",
        evaluation_strategy="no",
        per_device_train_batch_size=batch_size,
        learning_rate=learning_rate,
        num_train_epochs=2,
        save_strategy="no",
        logging_dir="./logs_without_layer",
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=dataset,
    )

    trainer.train()

    predictions = trainer.predict(dataset)
    predicted_features = np.argmax(predictions.predictions, axis=-1)

    data["Predicted_Feature_Without_Layer"] = predicted_features
    data.to_csv(output_csv_path, index=False)

    logging.info(f"Predictions without Layer saved to {output_csv_path}")
```

Listing 3.11: Feature Prediction without Layer

The function *train_and_predict_feature_without_layer* is almost the same as *train_and_predict_feature_with_layer* but we don't include the layer here.

3.6 Approach 3: Feature Prediction with Sampling Strategies

In this approach, we apply the same methodology used for *Layer Prediction* (Approach 1) but now for *Feature Prediction*. The resampled results from the *Layer* classification serve as input to train models for *Feature Prediction*, evaluating different sampling strategies. To address class imbalance, various resampling techniques are applied, including *SMOTE, ADASYN, SMOTEENN, Oversampling, and Undersampling*. The resampled datasets are stored for further processing. The fine-tuning process follows a *NCV* approach, optimizing hyperparameters for *ClimateBERT*. Performance is evaluated using the *Macro-Averaged F1-Score*. The results help assess the impact of different resampling techniques on *Feature Classification*.

```
X_text = data["Text"]
X_layer = data["Predicted_Layer"]
y_feature = data["Feature"]

feature_encoder = LabelEncoder()
y_feature_encoded = feature_encoder.fit_transform(y_feature)
```

```
7
8  vectorizer = TfidfVectorizer()
9  X_tfidf = vectorizer.fit_transform(X_text)
10
11 X_combined = np.hstack([X_tfidf.toarray(), X_layer.values.reshape(-1, 1)])
12
13 X_train, X_test, y_train_feature, y_test_feature, X_layer_train, X_layer_test =
       train_test_split(
14     X_tfidf, y_feature_encoded, X_layer, test_size=0.2,
15     random_state=42, stratify=y_feature_encoded
16 )
17
18 output_dir = "resampled_datasets_feature_prediction"
19 os.makedirs(output_dir, exist_ok=True)
20
21 SAMPLERS = {
22     "Adasyn": ADASYN(random_state=42),
23     "SMOTE": SMOTE(random_state=42),
24     "SMOTEENN": SMOTEENN(random_state=42),
25     "Oversampling": RandomOverSampler(random_state=42),
26     "Undersampling": RandomUnderSampler(random_state=42),
27 }
28
29 resampled_distribution = {}
30
31 for strategy, sampler in SAMPLERS.items():
32
33     X_train_resampled, y_train_feature_resampled = sampler.fit_resample(
34         X_train, y_train_feature
35     )
36
37     X_train_resampled_text = vectorizer.inverse_transform(X_train_resampled)
38
39     resampled_df = pd.DataFrame({
40         "Text": [" ".join(words) for words in X_train_resampled_text],
41         "Predicted_Layer": np.tile(X_layer_train.values, len(y_train_feature_resampled)
               // len(y_train_feature) + 1)[:len(y_train_feature_resampled)],
42         "Feature": feature_encoder.inverse_transform(y_train_feature_resampled)
43     })
44
45     strategy_folder = os.path.join(output_dir, strategy)
46     os.makedirs(strategy_folder, exist_ok=True)
47     file_path = os.path.join(strategy_folder, f"{strategy}_feature_resampled.csv")
48     resampled_df.to_csv(file_path, index=False)
49
50     resampled_distribution[strategy] = np.bincount(y_train_feature_resampled)
51
52 resampled_distribution_df = pd.DataFrame(resampled_distribution)
53
54 resampled_distribution_df.index = feature_encoder.classes_
55 resampled_distribution_path = os.path.join(output_dir, "
       resampled_class_distributions_feature.csv")
56 resampled_distribution_df.to_csv(resampled_distribution_path)
57
58 train_counts = np.bincount(y_train_feature)
59 test_counts = np.bincount(y_test_feature)
60
61 original_distribution_df = pd.DataFrame(
62     {"Train": train_counts, "Test": test_counts},
63     index=feature_encoder.classes_
64 )
65 original_distribution_path = os.path.join(output_dir, "
       original_train_test_distribution_feature.csv")
66 original_distribution_df.to_csv(original_distribution_path)
```

Listing 3.12: Sampling for the Features

The function in Listing 3.12 implements various resampling strategies to balance the *Feature* class distribution before training. It begins by loading the dataset, extracting text, predicted *Layer* information, and *Feature* labels. The text data is transformed using *TF-IDF vectorization*, and labels are encoded using *LabelEncoder*. The dataset is then split into training and test sets while preserving the class distribution through stratification. A set of resampling techniques, including *SMOTE, ADASYN,*

SMOTEENN, Oversampling, and Undersampling, is applied to the training data to mitigate class imbalance. This approach ensures a more balanced representation of *Feature* classes, improving model performance in downstream classification tasks.

```python
from sklearn.preprocessing import LabelEncoder

feature_label_encoder = LabelEncoder()

def train_and_evaluate(train_texts, train_layers, train_labels, val_texts, val_layers, val_labels, model_name, learning_rate, batch_size, n_epochs):
    logging.info(f"Training with LR: {learning_rate}, Batch Size: {batch_size}")

    tokenizer = AutoTokenizer.from_pretrained(model_name)

    train_encodings = tokenizer(list(train_texts), truncation=True, padding=True, max_length=512)
    val_encodings = tokenizer(list(val_texts), truncation=True, padding=True, max_length=512)

    train_labels = feature_label_encoder.fit_transform(train_labels)
    val_labels = feature_label_encoder.transform(val_labels)

    train_labels = torch.tensor(train_labels, dtype=torch.long)
    val_labels = torch.tensor(val_labels, dtype=torch.long)

    train_dataset = datasets.Dataset.from_dict({
        "input_ids": train_encodings["input_ids"],
        "attention_mask": train_encodings["attention_mask"],
        "labels": train_labels.tolist()
    })

    val_dataset = datasets.Dataset.from_dict({
        "input_ids": val_encodings["input_ids"],
        "attention_mask": val_encodings["attention_mask"],
        "labels": val_labels.tolist()
    })

    num_labels = len(feature_label_encoder.classes_)
    model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=num_labels)

    training_args = TrainingArguments(
        output_dir="./results",
        evaluation_strategy="epoch",
        per_device_train_batch_size=batch_size,
        per_device_eval_batch_size=batch_size,
        learning_rate=learning_rate,
        num_train_epochs=n_epochs,
        logging_dir="./logs",
        logging_steps=10,
        save_strategy="epoch",
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=val_dataset,
        compute_metrics=compute_metrics,
    )

    trainer.train()
    eval_results = trainer.evaluate()

    return eval_results

def process_strategy(strategy, output_folder, model_name):
    logging.info(f"Processing strategy: {strategy}")
    data_file = Path(output_folder) / strategy / f"{strategy}_feature_resampled.csv"

    if not data_file.exists():
        logging.warning(f"Data file not found for strategy {strategy}")
        return None
```

```
66
67      data = pd.read_csv(data_file)
68      texts, layers, labels = data['Text'], data['Predicted_Layer'], data['Feature']
69
70      outer_cv = StratifiedKFold(n_splits=N_OUTER_SPLITS, shuffle=True, random_state=42)
71
72      strategy_results = {
73          "best_params": None,
74          "best_results": None,
75          "tested_combinations": []
76      }
77
78      best_f1 = 0
79
80      for train_idx, test_idx in outer_cv.split(texts, labels):
81          train_texts, test_texts = texts.iloc[train_idx], texts.iloc[test_idx]
82          train_layers, test_layers = layers.iloc[train_idx], layers.iloc[test_idx]
83          train_labels, test_labels = labels.iloc[train_idx], labels.iloc[test_idx]
84
85          param_grid = {
86              "learning_rate": LEARNING_RATES,
87              "batch_size": BATCH_SIZES
88          }
89
90          for lr in LEARNING_RATES:
91              for bs in BATCH_SIZES:
92                  inner_scores = []
93                  metrics_list = []
94
95                  inner_cv = StratifiedKFold(n_splits=N_INNER_SPLITS, shuffle=True,
                        random_state=42)
96
97                  for inner_train_idx, val_idx in inner_cv.split(train_texts,
                        train_labels):
98                      inner_train_texts, val_texts = train_texts.iloc[inner_train_idx],
                            train_texts.iloc[val_idx]
99                      inner_train_layers, val_layers = train_layers.iloc[inner_train_idx
                            ], train_layers.iloc[val_idx]
100                     inner_train_labels, val_labels = train_labels.iloc[inner_train_idx
                            ], train_labels.iloc[val_idx]
101
102                     eval_results = train_and_evaluate(inner_train_texts,
                            inner_train_layers, inner_train_labels,
103                                                       val_texts, val_layers, val_labels
                                ,
104                                                       model_name, lr, bs, N_EPOCHS)
105                     inner_scores.append(eval_results.get("eval_f1", 0))
106                     metrics_list.append(eval_results)
107
108                 if not metrics_list:
109                     logging.warning("metrics_list is empty, skipping parameter
                            evaluation.")
110                     continue
111
112                 avg_metrics = {metric: np.mean([m[metric] for m in metrics_list]) for
                        metric in metrics_list[0].keys()}
113
114                 strategy_results["tested_combinations"].append({
115                     "learning_rate": lr,
116                     "batch_size": bs,
117                     "metrics": avg_metrics
118                 })
119
120                 if avg_metrics.get("eval_f1", 0) > best_f1:
121                     best_f1 = avg_metrics["eval_f1"]
122                     strategy_results["best_params"] = {"learning_rate": lr, "batch_size
                            ": bs}
123                     strategy_results["best_results"] = avg_metrics
124
125     logging.info(f"Best parameters for {strategy}: {strategy_results['best_params']}")
126
127     results_path = Path("results_feature_prediction") / f"{strategy}_results.json"
128     results_path.parent.mkdir(parents=True, exist_ok=True)
```

```
129
130     pd.DataFrame(strategy_results["tested_combinations"]).to_json(results_path, orient=
            "records", indent=4)
131
132     return strategy_results
133
134 output_folder = Path("resampled_datasets_feature_prediction")
```

Listing 3.13: Feature Prediction with Sampling

The function in Listing 3.13 performs *Feature Prediction* using different resampling strategies combined with *ClimateBERT*. It begins by loading and tokenizing text data while encoding categorical *Feature* labels. The dataset is processed using *StratifiedKFold* to ensure balanced training and evaluation splits. A hyperparameter search is conducted within a *NCV* framework, where different learning rates and batch sizes are evaluated to optimize classification performance. The *Trainer* API fine-tunes *ClimateBERT* on resampled datasets and computes key evaluation metrics, including *Accuracy*, *F1-Score*, *Precision*, and recall. The best-performing model configuration is selected based on the highest *Macro-Averaged F1-Score*, and the results are stored for further analysis. This approach systematically assesses the impact of different sampling strategies on *Feature Classification*, improving model robustness in imbalanced datasets.

Chapter 4

Results and Discussion

4.1 Results

This chapter presents and critically discusses the results of the different approaches presented in 3. In section 4.1 we assess whether the first and second *Research Questions* can be answered. This will be the case if we obtain satisfactory prediction results and subsequently improve them using sampling strategies. Specifically, these *Research Questions* investigate whether prediction is possible at all and, if so, whether it can be further improved through the application of sampling techniques.

In section 4.1 we evaluate *Feature Prediction* using a *Percentage Split* approach, both with and without including the predicted *Layer* as an additional *Feature*. Furthermore, in section 4.1 we extend our analysis by applying sampling strategies at the *Feature* level. This allows us to address the third *Research Question*, which examines whether the sampling strategy has a positive impact on prediction performance. In addition, we investigate whether direct prediction of text at the *Feature* level, with or without the inclusion of *Layers*, is the optimal approach.

Results for Layer Prediction

The following table shows the results of the *Grid Search* for *Layer Prediction*. The evaluation was conducted using different sampling strategies in combination with two different learning rates to assess their impact on classification performance. The performance metrics considered include *F1-Score*, *Accuracy*, *Precision* and *Recall*, providing a holistic view of model effectiveness. The best performing configurations for each sampling strategy are highlighted in **bold**.

Table 4.1: *Grid Search* Results for *Layer Prediction*

Sampling Strategy	Learning Rate	Batch Size	F1-Score	Accuracy	Precision	Recall
Percentage Split	2×10^{-5}	**8**	**0.4604**	**0.6079**	**0.5563**	**0.4776**
Percentage Split	5×10^{-5}	8	0.4471	0.6070	0.5301	0.4713
Oversampling	2×10^{-5}	8	0.3601	0.3931	0.3695	0.3931
Oversampling	5×10^{-5}	**8**	**0.3672**	**0.4024**	**0.3777**	**0.4024**
Undersampling	2×10^{-5}	8	**0.5094**	**0.5211**	**0.5264**	**0.5211**
Undersampling	5×10^{-5}	8	0.4967	0.5111	0.5076	0.5111
SMOTE	2×10^{-5}	**8**	**0.6488**	**0.6533**	**0.6621**	**0.6533**
SMOTE	5×10^{-5}	8	0.6405	0.6482	0.6592	0.6482
SMOTEENN	2×10^{-5}	8	0.9597	0.9609	0.9632	0.9572
SMOTEENN	5×10^{-5}	**8**	**0.9835**	**0.9849**	**0.9893**	**0.9833**
ADASYN	2×10^{-5}	8	0.6212	0.6316	0.6376	0.6300
ADASYN	5×10^{-5}	**8**	**0.6266**	**0.6339**	**0.6395**	**0.6323**

The results clearly show that the choice of sampling strategy has a significant impact on the classification performance. Of all the methods tested, the **SMOTEENN** approach combined with a learning rate of 5×10^{-5} gave the best performance, achieving a ***F1-Score* of 0.9835**, a ***Accuracy* of 0.9849**, and the highest values for ***Precision* (0.9893)** and ***Recall* (0.9833)**.

A comparison of traditional sampling strategies without oversampling or synthetic data generation shows that **undersampling** with a learning rate of 2×10^{-5} performed relatively well, achieving a ***F1-Score* of 0.5094**. This suggests that while undersampling can be a viable approach in some cases, it cannot match the performance of more advanced resampling techniques.

Overall, the analysis highlights the importance of selecting an appropriate sampling strategy. **SMOTE-based techniques, particularly SMOTEENN, significantly improve model performance** by addressing class imbalances. However, the **optimal hyperparameter configuration remains data set dependent** and the selection of the best combination should be tailored to the specific distribution and classification objectives.

Results for Feature Prediction

Table 4.2 compares the performance of *Feature Prediction* models with and without the use of predicted *Layer* information. The results show that both approaches achieve poor overall performance, with low *Accuracy* and *F1-Scores*. Including the predicted *Layer* slightly improves the *Precision* (*0.151139* vs. *0.150879*), but reduces the *Recall* (*0.160966* vs. *0.161346*). Similarly, the *Accuracy* remains very low for both models (*0.304721* with predicted *Layer* and *0.304097* without). The *F1-Scores* are almost identical, with slightly better performance without predicted *Layer* information (*0.148819* vs. *0.148726*).

These results indicate that neither approach provides satisfactory performance, suggesting that the current setup or *Features* are insufficient for reliable *Feature Prediction*. Further improvements in data preprocessing, resampling strategies, or model architecture are needed to achieve better results.

We used a *Percentage Split* approach for this experiment, as the sampling strategies are only implemented in Approach 3 4.1. Additionally, while the results here are worse than in the *Percentage Split* on *Layers*, it is important to note that the dataset contains only three *Layers* but 20 *Features*, which makes the prediction task significantly more challenging. The *Layers* were predicted using the best-performing strategy from 4.1.

Table 4.2: Comparison of *Feature Prediction* with and without Predicted *Layer*

Approach	**Accuracy**	**F1-Score**	**Precision**	**Recall**
Percentage Split with Predicted Layer	0.304721	0.148726	0.151139	0.160966
Percentage Split without Predicted Layer	0.304097	0.148819	0.150879	0.161346

Results for Feature Prediction with Sampling

To improve the classification performance, a *Grid Search* was performed, testing different sampling strategies with different learning rates. Table 4.3 shows the results obtained for different combinations of sampling strategy, learning rate and batch size. Performance is evaluated using several metrics, including *F1-Score*, *Accuracy*, *Precision* and *Recall*. The aim of this work is to find out which sampling technique in combination with which hyperparameters gives the best results.

Table 4.3: *Grid Search* Results for *Feature Prediction*

Sampling Strategy	**Learning Rate**	**Batch Size**	**F1-Score**	**Accuracy**	**Precision**	**Recall**
Oversampling	2×10^{-5}	8	0.3601	0.3931	0.3695	0.3931
Oversampling	5×10^{-5}	**8**	**0.3672**	**0.4024**	**0.3777**	**0.4024**
Undersampling	2×10^{-5}	**8**	**0.0725**	**0.1354**	**0.0764**	**0.1354**
Undersampling	5×10^{-5}	8	0.0511	0.1198	0.0480	0.1198
SMOTE	5×10^{-5}	**8**	**0.4576**	**0.4979**	**0.4733**	**0.4799**
SMOTE	2×10^{-5}	8	0.4307	0.4552	0.4476	0.4552
SMOTEENN	5×10^{-5}	**8**	**0.8245**	**0.9178**	**0.8755**	**0.8051**
SMOTEENN	2×10^{-5}	8	0.5462	0.7918	0.5568	0.5613
ADASYN	5×10^{-5}	**8**	**0.4355**	**0.4635**	**0.4531**	**0.4648**
ADASYN	2×10^{-5}	8	0.4093	0.4366	0.4250	0.4377

The results show clear differences in model performance depending on the sampling strategy used. In particular, the SMOTEENN method with a learning rate of 5×10^{-5} gives the best results. With

an *F1-Score* of 0.8245, an *Accuracy* of 0.9178, a *Precision* of 0.8755 and a *Recall* of 0.8051, it achieves the highest performance across all metrics. In comparison, undersampling shows a significantly worse performance, especially at a learning rate of 5×10^{-5} with an *F1-Score* of only 0.0511. This suggests that the large reduction in sample size removes relevant information for classification. ADASYN and SMOTE also achieve acceptable results, although SMOTE (5×10^{-5}) outperforms ADASYN with an *F1-Score* of 0.4576 and an *Accuracy* of 0.4979. The optimal hyperparameter combination consists of the SMOTEENN strategy with a learning rate of 5×10^{-5} and a batch size of 8.

4.2 Discussion

This work demonstrates that for the *POLIANNA* dataset, the most effective strategy is to first apply *SMOTEENN* to both the *Layer Prediction* and *Feature Prediction* tasks, followed by training with *ClimateBERT*. The results show that *SMOTEENN* significantly improves classification performance by effectively addressing class imbalance. For *Layer Prediction*, the model achieved exceptionally high performance, with an *F1-Score* of 0.9835 and an *Accuracy* of 0.9849. These results suggest that the preprocessing steps, especially the per-*SpanID* approach, contribute to a high number of correctly classified instances. While the near-perfect performance may raise concerns about overfitting, the generalizability of the model remains promising given the consistent improvements across multiple metrics. A direct comparison with alternative resampling strategies further confirms that *SMOTEENN* is the most effective approach in this scenario. For *Feature Prediction*, the baseline model without sampling yielded poor results, with *F1-Score*s around 0.148, indicating that the *Feature*-level classification task is considerably more challenging. Incorporating the predicted *Layer* information had a marginal effect—slightly improving *Precision* but decreasing *Recall*—suggesting that direct *Feature Prediction* without additional enhancements is insufficient. These results highlight the complexity of *Feature Prediction* at this level of granularity and point to potential limitations in the current *Feature* set or model architecture. Applying *SMOTEENN* to the *Feature Prediction* task resulted in a significant improvement, with an *F1-Score* of 0.8245 and an *Accuracy* of 0.9178, reinforcing the effectiveness of this resampling method. The large performance gap between the raw *Feature Prediction* model and the resampled model suggests that data imbalance plays a critical role in the poor baseline performance. Without proper resampling, the model struggles to generalize effectively, resulting in poor predictive ability. In summary, this work highlights the importance of two-stage resampling (first at the *Layer* level, then at the *Feature* level) before training with *ClimateBERT*. Direct *Feature Prediction* without *Layer*-level structuring leads to poor results, emphasizing the need for hierarchical processing. Future work should explore additional *Feature Engineering* techniques, alternative model architectures, or ensemble learning methods to further improve *Feature Prediction* performance. Integrating Human expertise with *AI*-driven methods can significantly improve the quality, *Accuracy*, and efficiency of policy labeling. *AI* models, such as *ClimateBERT*, can efficiently process large amounts of text and identify patterns and relationships that may be difficult for humans to detect at scale. However, Human oversight is critical to ensure contextual *Accuracy*, mitigate bias, and refine model output. By using *AI* for initial predictions and allowing Human experts to validate and refine the results, the labeling process becomes both scalable and accurate. This hybrid approach not only reduces annotation effort but also ensures that the nuanced complexities of policy texts are effectively captured. Future research should explore interactive *AI*-Human collaboration frameworks to further optimize policy annotation outcomes.

Chapter 5

Limitations and Outlook

5.1 Limitations

Despite the promising results obtained in this work, a limitation must be acknowledged. One of the main challenges was the high computational cost associated with HPO. Due to resource constraints, a *Grid Search* approach was used to optimize key parameters. While effective, this method explores the hyperparameter space in a predefined manner and may not identify the most optimal configuration as efficiently as more advanced optimization techniques. A more sophisticated approach, such as *Optuna*, could potentially yield better results; however, its higher computational requirements made it impractical within the scope of this work. The limited time frame was also a challenge. A more comprehensive investigation of different sampling and HPO strategies might have led to further improved model results. The analysis of additional resampling techniques or a deeper optimisation of the model architecture and learning parameters could therefore be a promising extension for future work.

5.2 Outlook

The results of this work open up several promising avenues for future research. One key aspect that could be further investigated is the use of *Optuna* for HPO. While *Grid Search* is a structured but limited search strategy, *Optuna* could allow a more efficient exploration of the hyperparameter space through its adaptive optimisation. Furthermore, this work has shown that *SMOTEENN*, a combination of over- and undersampling techniques, has achieved the best results in overcoming class imbalance. This suggests that it may be worth investigating further advanced sampling strategies. In particular, hybrid methods such as *SMOTETomek*, *SVM-SMOTE*, *Borderline-SMOTE* and *KMeans-SMOTE* may offer additional improvements. *SMOTETomek* combines the synthetic sample generation of *SMOTE* with the removal of overlapping majority class instances by *Tomek Links*, resulting in a clearer separation between classes. *SVM-SMOTE* uses Support Vector Machines (SVMs) to selectively generate new synthetic samples at the decision boundaries, resulting in more precise data augmentation. *Borderline-SMOTE* focuses on generating new samples specifically at the decision boundary, where misclassifications are more common, potentially increasing the robustness of the model. *KMeans-SMOTE* integrates clustering techniques to identify relevant areas within the minority class and generate targeted new samples to ensure a more representative distribution of the data. In addition to optimising the data processing, the generalisability of the model should also be further investigated. In this work, the model was trained and evaluated exclusively on the *POLIANNA* dataset. In order to test its transferability to other datasets, it would be useful to apply the method to different policy-related datasets with different linguistic structures and annotation techniques. Such an analysis could provide information on the extent to which the model can be used reliably in other regulatory and policy contexts.

Appendix

Tables

Layer	Description
Instrument Types	Political instruments used to implement policy measures.
Policy Design Characteristics	Structural elements shaping policy design, including objectives, responsibilities, and monitoring.
Technology Specificity	The extent to which a policy targets specific technologies, such as renewable energy sources.

Table 5.1: Overview of *Layers* in the *POLIANNA* dataset

Feature	Description
Instrument Type	Classification of policy instruments, such as voluntary agreements or subsidies.
Actor	Entities involved in policy implementation, including legislative bodies and monitoring authorities.
Compliance	Mechanisms ensuring policy adherence, including sanctioning measures.
Objective	Defined policy goals, either quantitative (e.g., emission reduction targets) or qualitative.
Resource	Required resources for policy execution, such as financial funding.
Reversibility	Mechanisms allowing for policy amendments or reversals.
Time	Temporal aspects, including duration, monitoring periods, and compliance deadlines.
Technology Specificity	Policies focused on particular technologies, such as low-carbon innovations.
Energy Specificity	Energy-related policy aspects, particularly in promoting renewables.
Application Specificity	Policies targeting specific sectors or regional applications.

Table 5.2: Overview of *Features* in the *POLIANNA* dataset

Tag	Description
Voluntary Agreement	Agreements between entities to implement climate policies voluntarily.
Framework Policy	Broad policies outlining general principles and objectives.
Tradable Permit	Market-based instruments such as emission trading schemes.
Regulatory Instrument	Legal mandates and standards, including emission limits.
Tax Incentives	Fiscal measures designed to promote compliance, such as tax deductions.
Subsidies and Direct Incentives	Direct financial support for policy implementation.

Tag	Description
Research, Development & Demonstration (RD&D)	Investments in innovation and technological advancements.
Public Investment	Government expenditures on infrastructure and policy programs.
Education and Outreach	Initiatives for awareness and stakeholder engagement.
Unspecified	Instruments not explicitly classified within the dataset.
Default Authority	Standard authority responsible for policy oversight.
Legislative Authority	Legislative entities involved in policy formulation.
Newly Established Authority	Authorities created specifically for policy execution.
Monitoring Authority	Entities responsible for ensuring policy compliance.
Default Addressee	General target groups, such as citizens or industries.
Resources Addressee	Groups affected by financial or infrastructure-related measures.
Monitored Addressee	Entities subject to policy monitoring and compliance checks.
Sector Addressee	Specific sectors impacted by policy measures, such as energy or transport.
Sanctioning Form	Enforcement mechanisms, including penalties and fines.
Monitoring Form	Methods of policy monitoring, such as reporting or inspections.
Reference to Other Policy	Policies referencing existing regulations or complementary strategies.
Amendment of Policy	Changes or updates made to existing policies.
Reference to Strategy or Agreement	Policies linked to overarching strategies or international agreements.
Quantitative Target	Specific numerical objectives, such as emission reduction goals.
Qualitative Intention	Non-measurable policy aims, such as fostering innovation.
Monetary Revenues	Revenue streams generated from policy mechanisms, such as carbon taxes.
Monetary Spending	Budget allocations for policy execution.
Provision for Reversibility	Mechanisms allowing policy adaptation or cancellation.
Policy Duration Time	Period in which the policy remains active.
Monitoring Time	Timeframes for performance tracking and evaluation.
Resources Time	Period of resource allocation for policy implementation.
Compliance Time	Deadlines for meeting compliance requirements.
In-Effect Time	Time span during which a policy measure is operational.
Low-Carbon Technology	Technologies contributing to reduced CO_2 emissions.
Other Technology	Technologies not explicitly categorized within the dataset.
Low-Carbon Energy Source	Renewable energy sources with low carbon impact.
Other Energy Source	Energy sources not explicitly categorized.
Low-Carbon Application	Applications focused on emission reductions.
Other Application	Applications not explicitly defined in the dataset.

Table 5.3: Overview of Tags in the *POLIANNA* dataset

Figures

Annotator Frequency

Figure 5.1 illustrates the distribution of contributions made by individual annotators in the *POLIANNA* dataset.

- **Annotator C** has the highest number of completed annotations, with a total of 256 contributions, indicating significant involvement in the annotation process.
- **Annotators F and A** follow with 222 and 216 contributions, respectively, reflecting relatively consistent participation.
- **Annotator B** contributed 199 annotations, slightly fewer than the others, but still representing substantial engagement.

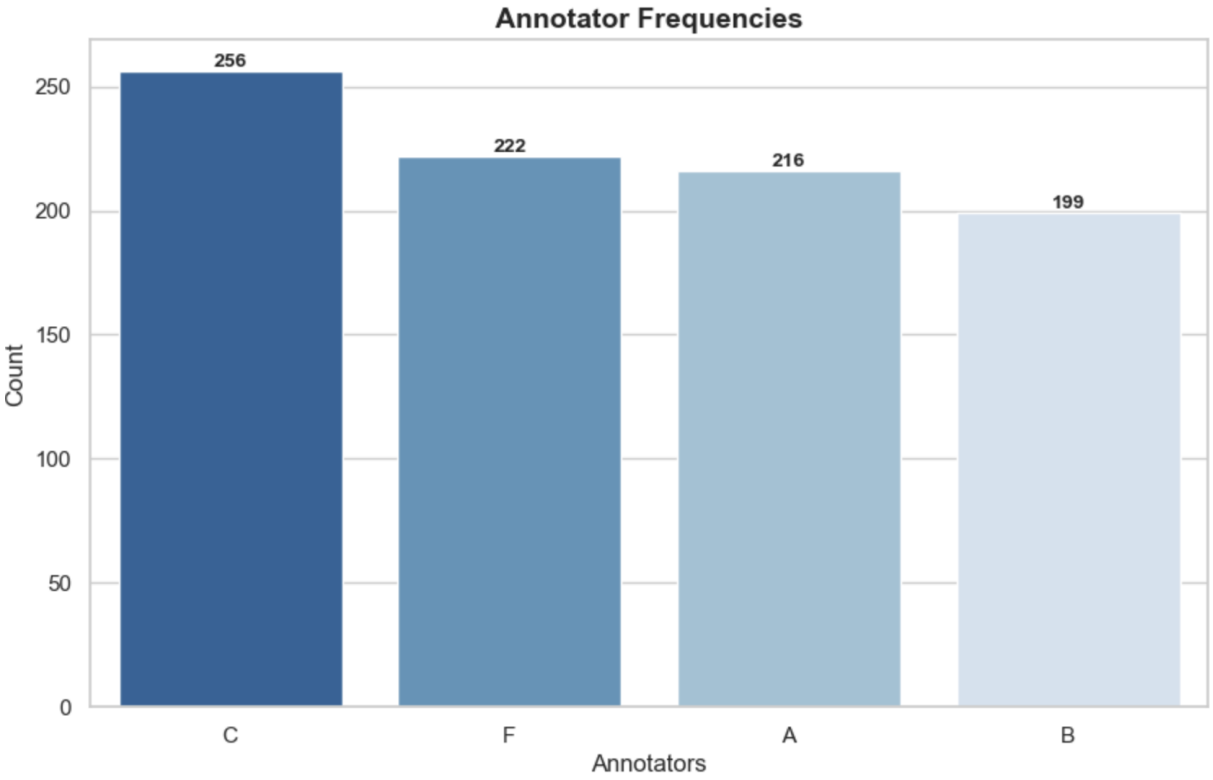

Figure 5.1: Annotator distribution in the *POLIANNA* dataset. The bar chart illustrates the contributions of individual annotators, with C contributing the most annotations (256), followed by F (222), A (216), and B (199). This distribution highlights the varying levels of engagement among annotators.

Sunburst Chart: Hierarchical Visualization of the *POLIANNA* Dataset

The sunburst figure provides a hierarchical visualisation of the structure of the *POLIANNA* dataset, showing the relationships between *Layers*, *Features* and *Tags*. The figure is divided into three concentric rings:

- The **innermost ring** represents the three main *Layers* of the dataset: *Policy Design Characteristics*, *Technology Specificity*, and *Instrument Types*.
- The **middle ring** breaks down each *Layer* into its corresponding *Features*, such as *Actor*, *Compliance*, and *Tech_ LowCarbon*.
- The **outermost ring** illustrates specific *Tags* within each *Feature*, providing detailed insights into the dataset's focus areas.

The visualisation highlights that the majority of annotations belong to the *Policy Design Characteristics Layer*, with particular emphasis on *Actor* and *Compliance*. Similarly, *Technology Specificity* emphasises tags related to low-carbon technologies, while *Instrument Types Features* includes tags such as *RegulatoryInstr* and *FrameworkPolicy*. This hierarchical representation allows a clear understanding of the organisation and focus of the dataset, providing valuable insight into the distribution and structure of policy-related annotations.

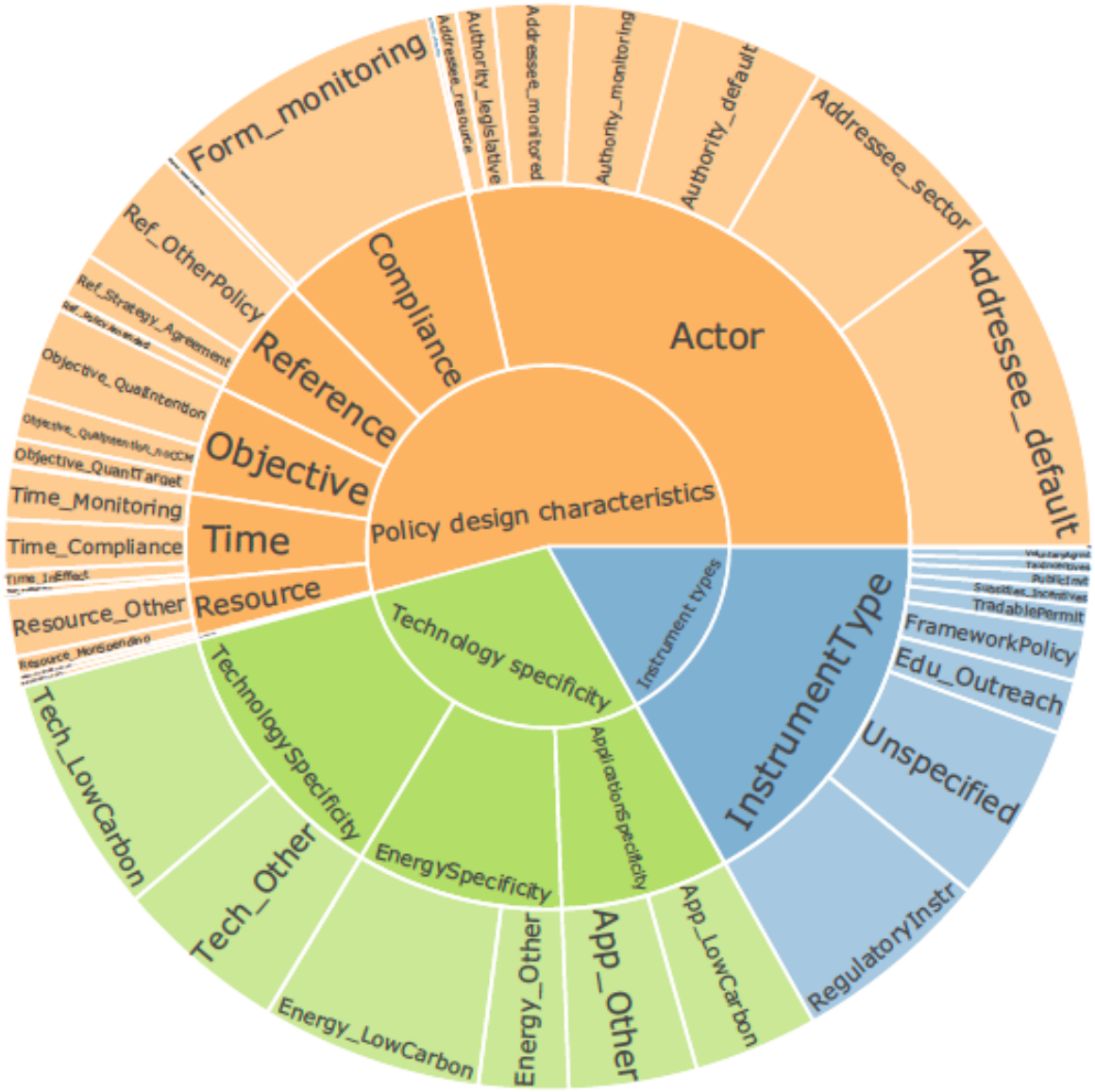

Figure 5.2: Sunburst chart visualization of the *POLIANNA* dataset. The innermost ring represents *Layers*, the middle ring depicts *Features*, and the outermost ring shows specific *Tags*. This hierarchical structure highlights the dataset's emphasis on *Policy Design Characteristics*, low-carbon technologies, and regulatory instruments [13].

Additional Information

Code and Data Availability

All code, datasets, and relevant resources used in this work are publicly available:

- **Project Code and Files:** The code and files developed for this work are available on GitHub: `https://github.com/Sven-Lutz/SA_Polianna`.
- **LaTeX Source Code:** The LaTeX source for this document is available on Overleaf: `https://de.overleaf.com/read/hwmvfwhnxmdj#b82f27`.
- ***POLIANNA* Dataset:** The *POLIANNA* dataset, along with comprehensive documentation, can be accessed at: `https://github.com/kueddelmaier/POLIANNA?tab=readme-ov-file`.
- ***ClimateBERT:*** The source code for *ClimateBERT* is hosted on GitHub: `https://github.com/ClimateBert`. The pre-trained model is available on Hugging Face: `https://huggingface.co/climatebert/distilroberta-base-climate-f`.

Bibliography

[1] C. Dupont and S. Oberthür, "Insufficient climate policy integration in eu energy policy: The importance of the long-term perspective," *Journal of Contemporary European Research*, vol. 8, no. 2, pp. 228–247, 2012.

[2] United Nations Framework Convention on Climate Change, "Unfccc timeline: Key milestones in the evolution of international climate policy," 2025.

[3] United Nations Framework Convention on Climate Change, "The kyoto protocol: A key step in combating climate change," 2025.

[4] U. N. F. C. on Climate Change (UNFCCC), "Adoption of the paris agreement," 2015.

[5] S. Oberthür and H. E. Ott, *The Kyoto Protocol: International Climate Policy for the 21st Century*. Berlin, Heidelberg: Springer, 2001.

[6] U. Nations, *United Nations Framework Convention on Climate Change (UNFCCC)*. 1992.

[7] B. Deutschland, "Bundes-klimaschutzgesetz (ksg)," 2019.

[8] Deutscher Bundestag, "Debatte zum bundes-klimaschutzgesetz," 2019.

[9] B. für Wirtschaft und Energie (BMWi), "Integriertes energie- und klimaprogramm," 2007.

[10] E. Chinchio, P. Martin-Olmedo, N. Di, and A. Franco, "Quantification of health impacts in eu chemical policy evaluations," *European Commission Publications*, 2025.

[11] B. für Wirtschaft und Klimaschutz (BMWK), "Monitoringbericht der expertenkommission zum energiewende-monitoring," tech. rep., n.d.

[12] B. für Wirtschaft und Klimaschutz (BMWK), "Monitoring-prozess zur energiewende," n.d.

[13] S. Sewerin, L. H. Kaack, J. Küttel, F. Sigurdsson, O. Martikainen, A. Esshaki, and F. Hafner, "Towards understanding policy design through text-as-data approaches: The policy design annotations (polianna) dataset," *Scientific Data*, vol. 10, p. 896, 2023.

[14] L. Chen, Z. Chen, Y. Zhang, Y. Liu, A. I. Osman, M. Farghali, J. Hua, A. Al-Fatesh, I. Ihara, D. W. Rooney, and P.-S. Yap, "Artificial intelligence-based solutions for climate change: a review," *Environmental Chemistry Letters*, vol. 21, pp. 2525–2557, 2023.

[15] N. Ahmed, A. K. Saha, M. A. Al Noman, J. R. Jim, M. Mridha, and M. M. Kabir, "Deep learning-based natural language processing in human–agent interaction: Applications, advancements, and challenges," *Natural Language Processing Journal*, vol. 9, p. 100112, 2024.

[16] J. I. Lewis, A. Toney, and X. Shi, "Climate change and artificial intelligence: assessing the global research landscape," *Discover Artificial Intelligence*, vol. 4, p. 64, 2024.

[17] N. Webersinke, M. Kraus, J. A. Bingler, and M. Leippold, "Climatebert: A pretrained language model for climate-related text," *arXiv preprint*, 2022. Presented at the Association for the Advancement of Artificial Intelligence (AAAI) conference.

[18] J.-C. Klie, M. Bugert, B. Boullosa, R. E. de Castilho, and I. Gurevych, "The inception platform: Machine-assisted and knowledge-oriented interactive annotation," in *Proceedings of the 27th International Conference on Computational Linguistics: System Demonstrations*, (Santa Fe, New Mexico, USA), pp. 5–9, August 20–26 2018. This work is licensed under a Creative Commons Attribution 4.0 International License. License details: `http://creativecommons.org/licenses/by/4.0/`.

[19] J. A. Bingler, M. Kraus, M. Leippold, and N. Webersinke, "Cheap talk and cherry-picking: What climatebert has to say on corporate climate risk disclosures," *Finance Research Letters*, vol. 47, p. 102776, 2022. This is an open-access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

[20] V. S. Anoop, T. K. A. Krishnan, A. Daud, A. Banjar, and A. Bukhari, "Climate change sentiment analysis using domain specific bidirectional encoder representations from transformers," *IEEE Access*, vol. 12, pp. 114912–114922, 2024. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.

[21] N. Hastreiter, "Can investor coalitions drive corporate climate action?," Working Paper 415, Grantham Research Institute on Climate Change and the Environment, London School of Economics and Political Science, 2024.

[22] A. Fernández, S. García, M. Galar, R. C. Prati, B. Krawczyk, and F. Herrera, *Learning from Imbalanced Data Sets*. Cham, Switzerland: Springer Nature Switzerland AG, 2018.

[23] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.

[24] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *ACM SIGKDD Explorations Newsletter*, vol. 6, no. 1, pp. 20–29, 2004.

[25] N. Chawla, K. Bowyer, L. Hall, and W. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Intell. Res. (JAIR)*, vol. 16, pp. 321–357, 06 2002.

[26] G. Husain, D. Nasef, R. Jose, J. Mayer, M. Bekbolatova, T. Devine, and M. Toma, "Smote vs. smoteenn: A study on the performance of resampling algorithms for addressing class imbalance in regression models," *Algorithms*, vol. 18, no. 1, 2025.

[27] H. He, Y. Bai, E. A. Garcia, and S. Li, "Adasyn: Adaptive synthetic sampling approach for imbalanced learning," pp. 1322–1328, 2008.