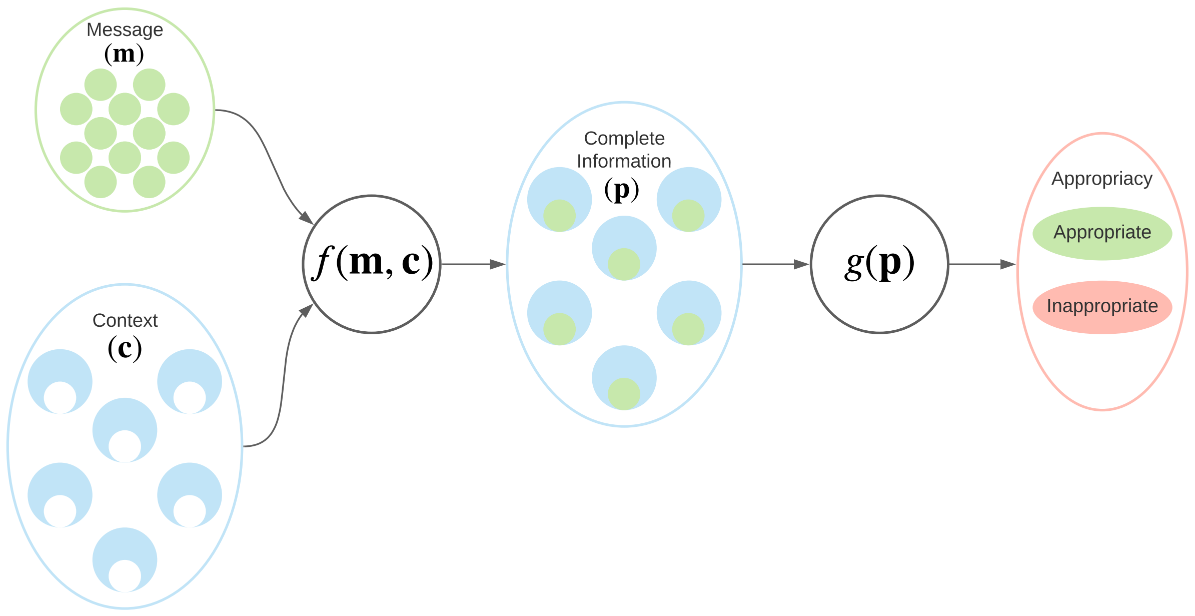


Improving Algorithmic Text Moderation via Context-Based Representations of Word Semantics

Identifying Semantic Nuances Within the Domain of Short and Conversational Chat

Master's Thesis in Data Science and AI

Felix Nordén

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021

Master's Thesis, 2021

Improving Algorithmic Text Moderation via Context-Based Representations of Word Semantics

Identifying Semantic Nuances Within the Domain of Short and Conversational Chat

Felix Nordén

Department of Mathematical Sciences
Division of Data Science and AI
Chalmers University of Technology
Gothenburg, Sweden 2021

Improving Algorithmic Text Moderation via Context-Based Representations of Word Semantics
Identifying Semantic Nuances Within the Domain of Short and Conversational Chat
Felix Nordén

Supervisor: Fredrik Johansson, Department of Computer Science and Engineering
Examiner: Johan Jonasson, Department of Mathematical Sciences

Master's Thesis 2021
Department of Mathematical Sciences
Division of Data Science and AI
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 32 50

Cover: Functional mapping model of how a message and a context can be combined to identify the semantics and determine the appropriacy of the given message. More in-depth information and a more concrete illustration is given in chapter 4.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Improving Algorithmic Text Moderation via Context-Based Representations of Word Semantics
Identifying Semantic Nuances Within the Domain of Short and Conversational Chat
Felix Nordén
Department of Mathematical Sciences
Division of Data Science and AI
Chalmers University of Technology

Abstract

Reliable text moderation requires proper domain knowledge. With scaling requirements increasing as platforms of the Internet grow larger and larger, the prevalence of algorithmic text moderation has increased with the intention to alleviate, or even replace, its manual counterpart. Nonetheless, these algorithm-based solutions are harder to interpret, evaluate, and risk being biased in their decision making, resulting in more rigid and error-prone behavior when changes in context end up shifting the semantics of the text itself. To solve these shortcomings, this thesis presents an approach that learns semantic nuances within shorter pieces of text when given a related context represented by various layers of information. For this purpose, the sentence transformer architecture is employed which jointly learns embeddings of the short-form text and its context. The embeddings are used as input to a Log-loss optimized, fully-connected network to classify the appropriacy of the text. Furthermore, the thesis investigates the tradeoff between gained performance and added time- and implementation complexity for each additional layer of information. The approach is evaluated on chat data from *Twitch* – a live-streaming service – where the related context for each message is built up incrementally; first by introducing a layer of stream metadata and then augmenting the stream metadata by introducing a layer of related game metadata provided by *IGDB* – the Internet Game Database. From the results, the approach demonstrates that representing a context using both stream- and game metadata has a significant impact on the performance; yielding an F1 score of 0.37 compared to 0.18 and an AUROC score of 0.63 compared to 0.45 of the best-performing baseline. Furthermore, a linear time complexity dependence is identified on the number of sentences to embed per datapoint, causing a forward pass to take at worst ∼78 ms. per datapoint. With this, it is concluded that contextual information is able to improve predictive performance for algorithmic text moderation on shorter pieces of text. Additionally, exploring contextual relevance of data is easy when using sentence transformers, albeit with a linear growth in time complexity.

Keywords: Natural Language Processing, Machine Learning, Deep Learning, Algorithmic Text Moderation, Sentence Transformers, Master's Thesis.

Acknowledgements

I would like to thank the following people, without whom I would not have been able to complete this research.
First, my supervisor Fredrik Johansson at Chalmers, whose knowledge and insights about the research process steered me in the right direction and helped me step into territories which were previously unknown to me. Furthermore, I would like to thank him for the counseling hours he has put in, the laughter we have shared during these hours, and for always answering my questions, no matter how ambiguous they may be!
I also want to thank Saad Ali, my mentor and one of my supervisors at Twitch, whose knowledge in applied science regarding machine learning and artificial intelligence has helped me investigate interesting aspects of my research and who has taught me fundamental lessons regarding working with data at Twitch. Also, I want to thank him once more for the times he helped me troubleshoot my database queries and helping me get data dump after data dump until the data was correct.
Next, my other supervisor and also very creative mind, Sanjay Kairam, whom without his genuine interest and the million ideas for topics, this thesis would likely never even have been thought about. I also want to thank him for his guidance, hospitality, and help throughout this thesis and my time at Twitch so far – without it, my work would have been so much harder!
To Julia Tavarez at Twitch for her knowledge about Sentence Transformers and being willing to take her time to help me troubleshoot my model when it did not show any good results. Without her, these results would likely not have been found in this project.
A thank you to Gerardo Mendez at Twitch for taking your time and explaining how AutoMod works under the hood and for showing a genuine interest in my work – it gave me a boost when I really needed it!
To the Safety Proactive team at Twitch for showing a genuine interest, asking me tons of questions, and for giving me a guiding hand with regards to both people and tooling at Twitch.
A huge thank you to Christian Frithiof – my mentor, dear friend, and one of my pillars of support during the year of 2020. Without his engagement and support, I would not have gotten to where I am today in this short amount of time.
Last, but certainly not least, I want to thank my friends, family, and loved ones for being there for me with unconditional support. Without you being there in my stressful moments with hugs, laughs, and activities, I would likely have been overworked before finishing this work.
Once again, thank you everyone! This is the end of this journey, but I sure hope that it is only the beginning of something bigger!

Felix Nordén, Gothenburg, May, 2021

Contents

List of Figures

List of Tables

List of Code

1 Introduction

Languages are ever-evolving and vary greatly in terms of both syntax and semantics. A variation in one of these terms does not necessarily imply a variation in the other. For example, different languages have their own set of grammar rules that define the syntax of the language. These rules are then used to construct sentences meant to mediate the intended semantic value. In the opposite case, the semantics of a phrase, sentence, or any other word construct can vary greatly depending on the context in which it occurs. Communities, as an example of context, may have specific ways of communication which affect the perception of the communicated language to a quantifiable degree [1], even though the syntax remains unchanged. Within the field of text moderation – the practice of identifying and modifying inappropriate pieces of text – these variations give rise to a difficult problem as the moderator is required to have enough domain knowledge to correctly interpret the text and to not make mistakes.

In general, variations in the semantics of a language can cause issues, not only for moderators, since misinterpretation is likely to happen if reader does not have knowledge of the semantic nuances contained associated with the specific context. Furthermore, with the evolution of large platforms on the Internet, the growing influx of content has reached a point where purely human text moderation has become unmanageable. To make this influx more manageable, algorithmic text moderation – among other algorithmic moderation systems – has been developed to process user-generated text in significantly shorter timeframes and at scale [2]; albeit, by introducing additional ambiguity to non-trivial subjects regarding interpretation [2] and potential bias in the algorithmic decision making process [3].

1.1 An Overview of Manual- and Algorithmic Text Moderation

In simple terms, two kinds of errors can be done when moderating text: *moderating appropriate text* and *not moderating inappropriate text*. “Moderation” in this context means any form of modification of the original text, up to the point of removing it completely. Incorrectly identifying a piece of text as appropriate (false positive) is problematic as it may negatively affect the receiving end, e.g., hurting or insulting the recipient of the text. However, incorrectly identifying a piece of text as inappropriate (false negative) is also problematic as it might instead negatively affect the writer, e.g., by evoking a feeling of mistreatment. Therefore, well-performed text moderation is a balance act, where being too focused on one side causes you to fall off the other.

When considering algorithmic text moderation with the intention of mitigating the issues mentioned in [2, 3], a new dimension of complexity is added to the problem; not only is the required domain knowledge necessary, but the domain knowledge also needs to be learned by the moderation model. Furthermore, since this type of model is difficult to interpret, verifying that the model is appropriately performing the balance act that is text moderation at a level deeper than simple metrics, e.g., Accuracy or F1 Score, becomes intractable. There has been successful work done related to identifying changes within word semantics both over time and between communities, where the proposed solution yields intuitive and interpretable results [4]. Moreover, the results from [4] suggests that there are semantic nuances to be inferred from contextual information, which may act as learnable domain knowledge for a moderation model. While keeping this possibility of learnable domain knowledge in mind, [5] proves this possibility by proposing and evaluating a classification technique. This technique combines the implicit patterns occurring in textual content – comments – with contextual information – a social network of the users who wrote the comments – to both score and classify the content. Compared with more traditional methods, which were decision trees and *Support Vector Machines* (SVM), the proposed technique outperformed the traditional techniques with regards to accuracy and executed faster than the SVM model; therefore, the remaining question is "what contextual information contains the inferrable semantic nuances?".

1.2 Large-Scale Text Moderation at Twitch

At Twitch[1], a live-streaming service where users can broadcast various types of content live via *channels* for others to view, the stream chat functionality is a prime example of large quantities of text in need of moderation. On channels with many viewers, the viewers that distinguish themselves may be offered the role of human moderator[2] for that channel's chat. However, as previously mentioned, this becomes unmanageable as a channel grows. In order to make this task more manageable for both broadcaster and moderator, Twitch's current solution is the algorithmic moderation tool *AutoMod* – "a moderation tool that blocks inappropriate or harassing chat with powerful moderator control" [6].

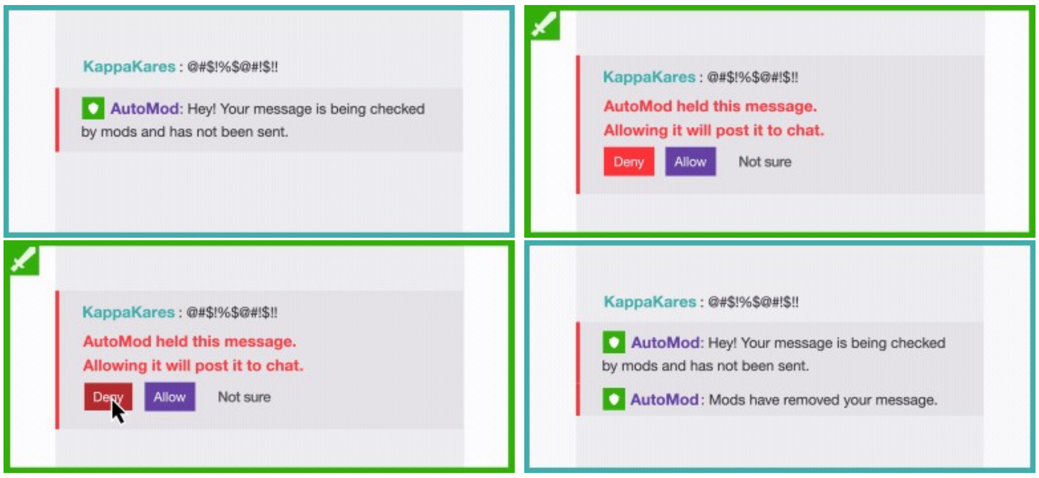
With the work performed in this thesis, the goal is to explore and find potential answers to the aforementioned question in the context of Twitch's stream chat and AutoMod – what information can improve AutoMod and allow it to understand the semantic nuances occurring in the plethora of communities residing in the channels of Twitch? Furthermore, in order to quantify the value of introducing such information, the tradeoff between potential performance gain and introduced complexity will be

[1]Twitch's About page: https://www.twitch.tv/p/en/about/
[2]"Human moderator" will from hereon also be synonymous to "moderator".

measured in order to yield results which may be applicable in a more general setting.

1.2.1 History of AutoMod and Its Current Shortcomings

In alignment with the general issue of word semantics differing depending on the context in which they occur, Twitch has identified that AutoMod – their current algorithmic moderation tool – is not granular enough in its predictions to perform well over differing contexts. AutoMod is intended to capture chat messages which the tool considers inappropriate or harassing and then holds these messages for either the broadcaster or moderators of the channel to allow or deny. If the message is allowed, it will show up in chat, if it is denied it will not. An example of this moderation flow for both the sender and the moderator is shown in figure 1.1. Furthermore, if the captured message is ignored by the broadcaster or moderator, this message will remain captured by AutoMod, meaning that the message will not show up in chat in this case either. Therefore, compared to the previous definition of moderation in section 1.1, the role of a moderator in Twitch chat is simplified to only being required to identify what messages are appropriate and inappropriate and then ensure that the inappropriate messages are not shown in chat.


KappaKares : @#$!%$@#!$!!
AutoMod: Hey! Your message is being checked by mods and has not been sent.

KappaKares : @#$!%$@#!$!!
AutoMod held this message. Allowing it will post it to chat.

KappaKares : @#$!%$@#!$!!
AutoMod held this message. Allowing it will post it to chat.

KappaKares : @#$!%$@#!$!!
AutoMod: Hey! Your message is being checked by mods and has not been sent.
AutoMod: Mods have removed your message.

Figure 1.1: Moderation flow from viewpoints of both the sender and moderator of a sent message and how AutoMod functions in conjunction with a moderator. Frames in turquoise borders and without a sword in the upper-left corner are from the sender's viewpoint and the ones with that are in green borders with the sword are from the moderator's viewpoint.

AutoMod does not consider the complete context in which a piece of text occurs, but only depends on the piece of text that constitutes a chat message to make its prediction. In the early days following AutoMod's introduction, when Twitch had a more consistent context in the form of mainly major game titles, only considering the chat message was enough. However, now that Twitch is expanding their content catalogue both horizontally (additional types of streams) and vertically (more categories in each type of stream), AutoMod is starting to show its shortcomings. As one channel can be drastically different in content and community compared to another, assuming a consistent context becomes a problem.

1.2.2 Motivating the Interest for Improvement

For example, imagine that a broadcaster is playing Assassin's Creed[3]; a game franchise in the adventure and sneak'em up genres, where killing is a frequent and non-stigmatized action. Furthermore, consider the event where the viewers are discussing what actions the broadcaster should take in order to progress. A viewer sends the message *"You just need to sit in a dark corner, wait for the them to walk past you and then walk out and kill them"*. Out of context, this may sound serious and highly inappropriate, meaning that this message could be captured by AutoMod. Conversely, given the current context, the message is meant to be helpful, has no ill intentions and is highly appropriate.

Similarly, in the case of different channel contexts, the message *"They have AK47s"* is perfectly relevant in a channel streaming a first-person shooter (FPS) game, such as *Counter Strike: Global Offensive*. However, in a channel that is in the category *"Just Chatting"* and nothing related to violence should occur, this message would likely be inappropriate.

Therefore, there is a need to investigate the possibilities of leveraging the large quantities of stream-related data they house; along with related metadata, such as games metadata (e.g., genres or summaries) or creative arts metadata (e.g., music or painting), to help AutoMod "contemplate" and become more granular in its approach when classifying text for appropriacy. More concretely, the idea is to have AutoMod learn to capture the nuances that define a channel's true moderation preferences by learning from the actions of channels' human moderators and to generalize these nuances across streams that appear similar by using various types of related data.

1.3 Aim

As previously mentioned in the end of the introduction of this chapter, the aim of this thesis is to investigate the possibilities of improving the performance of AutoMod by adding more comprehensive layers of contextual information for the analysed text. Furthermore, the performance- and complexity tradeoff brought by each layer will be measured to give results that are applicable in a more general context. The contextual information is intended to comprise stream metadata, data that relates to the stream metadata, as well as any data that can be referenced within the analysed text itself. Data that is not referred to as "stream metadata" will henceforth be synonymous with "external data" and are intended to be gathered from external sources. Examples of external data are (1) game metadata for a game that is currently being played in a stream, (2) a notable/famous personality within the community, and (3) information about applicable products/accessories and their manufacturers. An illustrative example of how these information layers relate to each other and with a chat message, as well as how all components jointly represent a context for the problem can be found in figure 1.2.

[3]Information about the franchise from the publisher Ubisoft: https://www.ubisoft.com/en-gb/game/assassins-creed

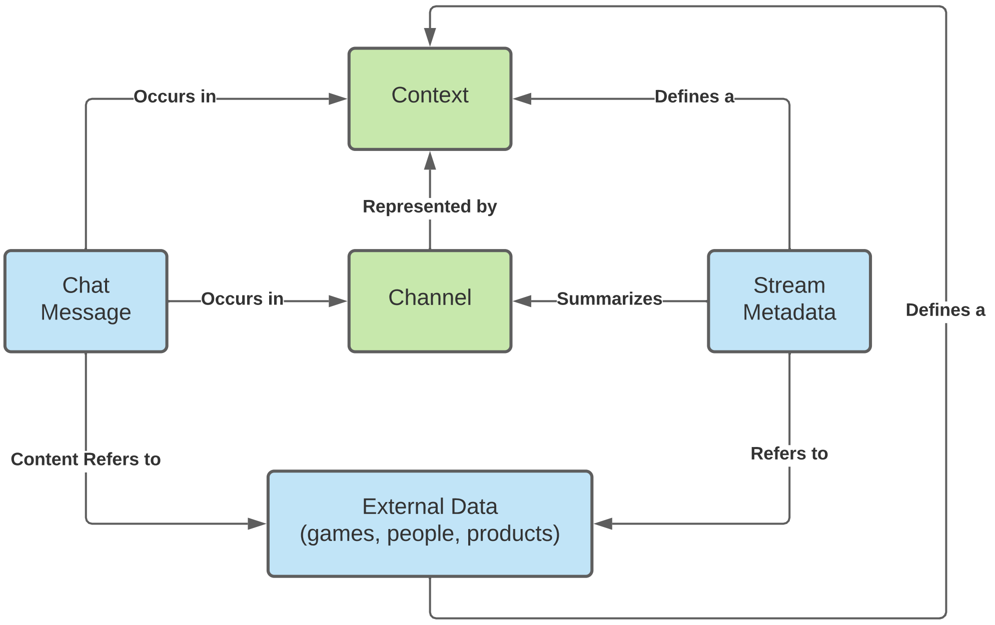


Figure 1.2: Relationship diagram of how a stream chat message, stream metadata for the channel, and data from external sources help define the appropriacy of the chat message. Note that the stream metadata summarizes the channel, but not necessarily completely represents the channel in the way that the context intends to do.

By the end of the project, the intended outcome is to have a deliverable proof of concept application that uses contextual information to improve the predictive performance of Twitch's current AutoMod solution. In conjunction with the deliverable, the measurements for the aforementioned performance- and complexity tradeoffs for each layer is intended to have been collected in order to justify whether or not a specific layer of information is worth introducing.

1.4 Delimitations

Since the complete domain that is Twitch is expansive and continuously growing, producing a deliverable that models the complete domain is not feasible within neither the timeframe nor headcount of this project. Therefore, to scope the project to a feasible scale, the following delimitations are taken into account:

1. Since AutoMod is opt-in for broadcasters, only channels that are using AutoMod will be considered
2. Only channels that are identified as English-speaking will be considered
3. Only messages that have been captured by AutoMod or manually removed will be used for training and validation
4. Only channels that have a concurrent viewer count within 100-1000 will be

considered

5. Categories will be selected to be as diverse as possible to capture various kinds of contexts
6. Data gathered from external sources will be limited to the *Internet Game Database* (IGDB) [7]

Regarding item 3, only messages captured by AutoMod or manually removed by a moderator are considered as these have inferrable labels based on moderator interaction; why only these cases are inferrable is explained when describing the inference procedure. For item 4, only a concurrent viewer count of 100-1000 is considered as the number must be high enough to have a constant influx of messages yet low enough to mitigate the risk of inferring incorrect labels. Going above 1000 concurrent viewers increases the risk of messages overflowing, making it difficult for the moderators to keep up.

1.5 Specification of Issue Under Investigation

To concretize the aim of the thesis further, the project is intended to investigate and either accept or reject each of the following hypotheses:

1. Adding a layer of stream metadata to an algorithmic moderator tool based on AutoMod's output will result in significant improvement of moderator performance
2. Adding a layer of domain-specific data relating to the stream metadata of item 1 to an algorithmic moderator tool based on AutoMod's output will result in significant improvement of moderator performance
3. Improved performance results of each additional layer of information outweigh the time and complexity that follows from incorporating the layers' information into an algorithmic moderator tool based on AutoMod

1.6 Thesis Outline

For the remainder of this thesis, the work towards reaching a conclusion regarding the aforementioned hypotheses are presented. In chapter 2, more in-depth information is presented regarding the domain that is Twitch chat, the domain language itself, and how moderation in the domain functions with regards to AutoMod. Following this domain information is a more in-depth description of how the system that is AutoMod is constructed. Components of the system which are brought up mainly revolve around those which are depended upon in this work. Then, a more concrete definition of what defines a "context" within Twitch chat is given, which can also be generalized to other domains within algorithmic content moderation. Going into more formal language, the mathematical notation which is used throughout the thesis is outlined as a reference. Lastly, multiprocessing and shared memory within Python is described as this is used within the project for loading and processing data in parallel, as well as performing parallel training of the models using multiple GPUs.

In chapter 3, the procurement process of the data for training- and validation sets is described along with the process of splitting the data into their respective sets. Additionally, the process of procuring both the additional stream metadata from within Twitch and the games data from the *Internet Game Database* (IGDB) is described. Lastly, the procurement process of the test set is described.

Next is chapter 4, where the architecture, implementation, and iteration of the model is described. First, the mathematical formulation of the problem is given with a mathematical definition of the involved components, such as the involved data and the loss function. Next, the baselines are defined in conjunction with the general architecture of both the baselines and the main model itself. In this part, a small ablation study of feature importance within the current AutoMod system is also performed. After that, the architecture and implementation of the main model is described, along with the iterative process of incorporating the two additional layers of information. With the implementations described, the process of tuning both the baselines and the model iterations is described along with the chosen hyperparameter configuration. Lastly, the evaluation process of all models, including baselines, is outlined along with the methods used for procuring confidence intervals.

In chapter 5, the results for both the baselines and the model iterations are presented following the process outlined at the end of chapter 4. First, general statistics about the procured training- and validation sets are given. Then, the results from evaluating all models on the validation data are presented, along with some analysis of the results. Lastly, the results from the final evaluation on the test set are presented.

Following the results, in chapter 6, the results are discussed in more detail. Furthermore, some discussion is done regarding issues related to unexpected behavior which occurred during the training of one of the models in order to try and find an answer for the unexpected behavior. Limitations with regards to this work are also brought up and discussed as to how and where the achieved results may prove valid or not. Lastly, multiple ideas for future work are presented on how to improve upon the work done here as well as potential alternative routes to take.

Lastly, in chapter 7, the final verdict regarding the hypotheses outlined in section 1.5 are given. Additionally, some closing thoughts tied to the results and discussion are given along with how this work has contributed to the domain of algorithmic text moderation.

2
Theory

In order to give some additional information to explain underlying concepts which are not directly relevant for the main thesis, this chapter is to present this information in an unobtrusive manner. The following sections describe the necessary domain language for Twitch, how AutoMod is used for text moderation of chat, and what is defined as a *context* within text moderation. Furthermore, some mathematical notation is introduced for easier reading in the following chapters. Lastly, some relevant theory regarding how shared memory in Python works and how it was used in conjunction with PyTorch in this project.

2.1 Twitch Domain Language

In chapter 1, Twitch was introduced as *"... a live-streaming platform where users can broadcast various types of content live via channels for others to view..."*. This excerpt introduces some fundamental concepts for the Twitch-domain; namely, *broadcasting users* and *types of content*. Every user at Twitch can broadcast, or usually referred to as *stream*, content via their *channel*. The content can be of any type as long as it is "stream-friendly" and not inappropriate.

Each time a user starts broadcasting, at the stage where they are a *broadcaster*, their channel goes live and other users can watch the stream entering the channel on the platform. When a channel goes live, the broadcaster is able to modify the *channel title* to describe what their stream is about in free-text form. If a user finds the channel intriguing and joins in, they become a *viewer* of the channel.

Every viewer of a channel can interact with the broadcaster and other viewers via the channel's *chat* function. In order to ensure that the language in chat is appropriate, some form of moderation needs to be introduced. At Twitch, the broadcaster can choose at least one of three options: (1) Manually moderate the chat themselves, (2) Assign one or more viewers as *moderators* for whom the broadcaster deems fit the role and then have them moderate the chat, or (3) make use of Twitch's algorithmic moderation tool AutoMod [6]. As a channel grows, the broadcaster usually chooses a combination of (2) and (3) to delegate the task and focus on delivering their content in the stream.

Regarding the content, a broadcaster can describe what the primary content of their broadcast is using *broadcast categories* which can range from game titles, e.g., *Dota*

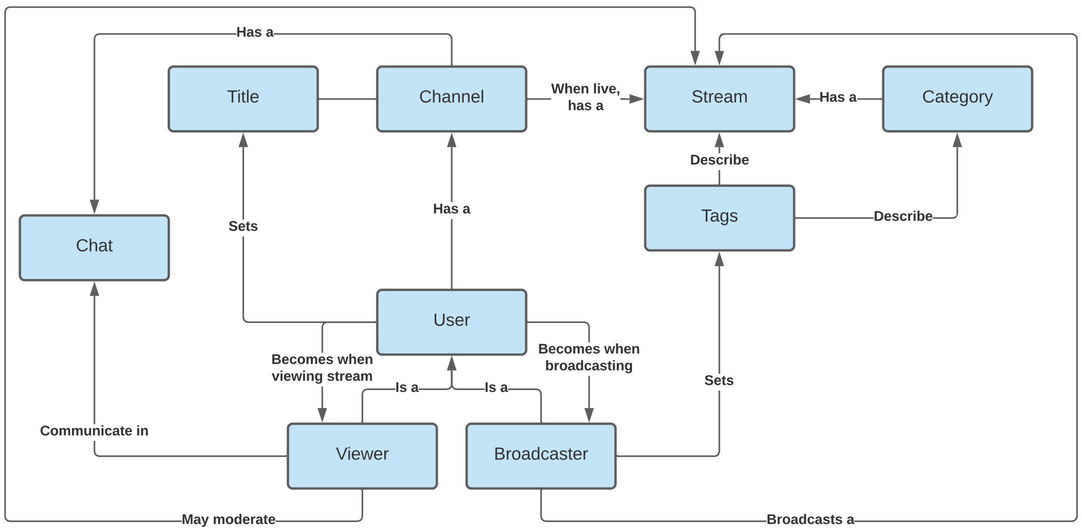


Figure 2.1: Domain model diagram of Twitch for the problem domain of chat moderation.

2 or *Assassin's Creed*, to professional and creative arts, e.g., *Music* or *Science and Technology*, or even simple casual conversations in *Just Chatting*. Each category is also associated with *category tags* which are a small subset of all tags at Twitch which describe the category itself, e.g., game genres such as *Action* or *MMO*, or what the non-game category involves, e.g., *Creative* or *IRL* (In Real Life) [8]. Furthermore, a broadcaster can choose to tag the stream itself by using *stream tags* to add additional flavor to their own channel. Tags such as *AMA* (Ask Me Anything), *Anime*, and *Singleplayer* are examples among many other [8].

All of the emphasized terms up until this point, except for the examples of different tags, are key terms within the problem domain of Twitch chat and this project. Therefore, in order to give a more clear picture of how these terms relate and interact with each other, a domain model diagram is presented of this problem domain in figure 2.1.

2.2 Text Moderation using AutoMod

In order to get a more comprehensive understanding of how AutoMod functions, this section gives a complete overview of the parts of AutoMod that are relevant within this project. AutoMod as a system does more than simply classifying pieces of chat messages as appropriate and inappropriate; it allows the broadcaster to define sets of permitted and blocked words or phrases and also to configure how aggressive the system should be in moderation on a level from 0 (no moderation) to 4 (more moderation on discrimination, sexual content and profanity, and most moderation on hostility) [6]. Furthermore, before classifying the chat message itself, the system performs other checks in order to determine whether the message is allowed to occur in the chat to begin with. These checks include, among others, checking the ban status of the sender, verifying that the message does not contain any banned phrases defined by the broadcaster, and that the sender has not been flagged as a bot.

In the case of all checks passing, the classification of the message is done before it is shown in chat. This classification makes use of an internal machine learning-model which produces a set of severity scores between [0,1] for the filters listed in the level-section of [6]. If any severity score is above the threshold for what is defined based on the broadcaster's set AutoMod level, the message is captured and shown privately to the broadcaster and their moderators for any of them to take action and either approve or deny the message. If the message is instead ignored, it remains in the queue of messages to process and will not show up in chat unless approved by a moderator, implicitly making it denied from the chat's point of view.

The internal model runs on all sent chat messages at Twitch up-front, making it the first action AutoMod takes before performing all other checks and eventually using the internal model's output to classify the appropriacy of the message. A more concrete picture of how a message flows through AutoMod is presented in figure 2.2 to further illustrate the process.

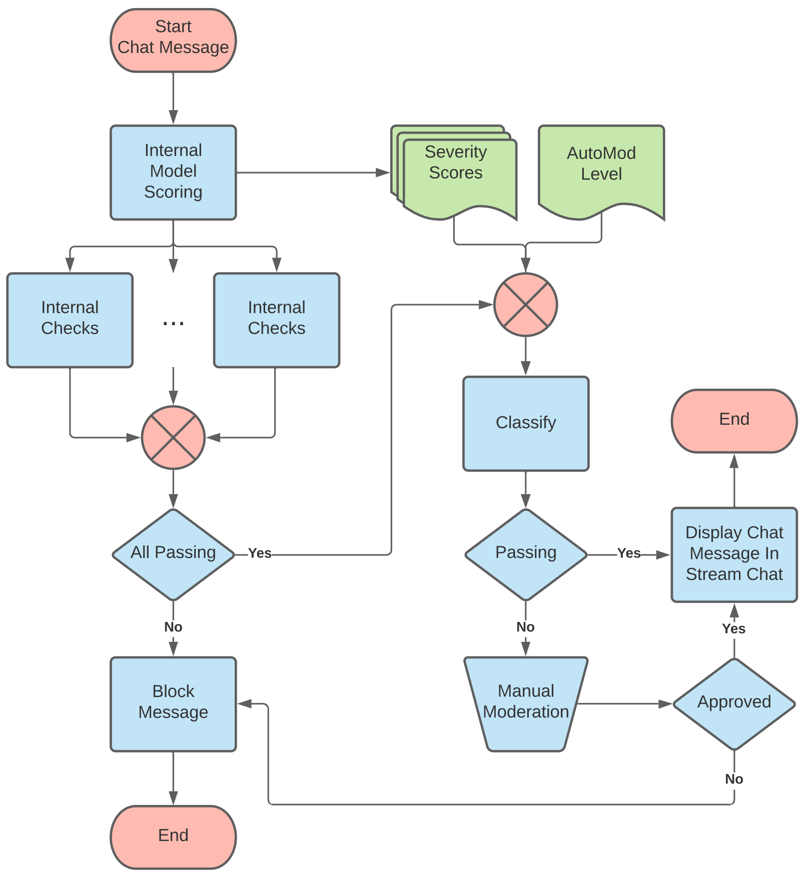


Figure 2.2: Flowchart of how AutoMod processes a message before it is either shown immediately in chat, sent for manual moderation by a moderator, or blocked directly by AutoMod.

2.3 Defining a Context Within Text Moderation

Context is a word which may describe many different things depending on the setting. The word is a noun and is defined in the Oxford Dictionary of English as "the situation in which something happens and that helps you to understand it" [9]. Within the domain of text moderation, this "something" is the piece of text that is to be moderated. The "situation" which helps one understand the text is the aforementioned knowledge that is required to correctly moderate the text.

In a sense, any information which relates to the text could be seen as a context. However, whether the information is helpful in the sense that the final verdict is correct may be highly subjective to the moderator. Therefore, here two categories of contexts are defined: (1) *informative context* and (2) *uninformative context*. Informative context is the context which is useful for the moderator to reach the correct verdict, whilst uninformative context is the context which contributes nothing, or even negatively, towards the final verdict.

With these two categories of context, the setting of this project can be interpreted and summarized as "investigating what contextual information can be categorized as *informative* and *uninformative* when moderating Twitch chat with the moderator being an algorithmic moderation tool".

2.4 Mathematical Notation

In some parts of this thesis, mathematical notation is used to concretize and further illustrate how mathematics are used and implemented inside the models. Therefore, a small notational framework is outlined here to give sufficient knowledge of how to read the mathematics in the remainder of this thesis.

Variables of one dimension are denoted using an italicized typeface, e.g., y for a target response or label. Variables of multiple dimensions, i.e., vectors, are instead denoted using a bold typeface, e.g., $\mathbf{x}$ for predictors or features of a datapoint or $\mathbf{y}$ for a vector of target responses.

Matrices of arbitrary dimensions $M \times N$ are denoted using capitalized letters in a bold typeface, e.g., $\mathbf{X}$ for the predictors of a collection of datapoints. If need be, the dimensions will be specified in conjunction with the matrix itself, e.g., "the matrix $\mathbf{X}$, where $\dim(\mathbf{X}) = M \times N$."

Sets are denoted using a calligraphic typeface, e.g., $\mathcal{S}$ for the set of all possible strings, except for the cases where sets are already well-defined in mathematics, e.g., $\mathbb{R}$ for the real numbers. Primary sets of interest in this paper are the following:

- $\mathcal{S}$ – the set of all possible strings
- $\mathbb{R}$ – the real numbers
- $\mathcal{P} = \mathcal{S}^k \times \mathbb{R}^l$ – the set of possible string- and number-based predictor pairs for string vectors of dimension k and number vectors of dimension l

- $\mathcal{B}$ – the set of messages, i.e., the predictors that do not define a context
- $\mathcal{C}$ – the set of contextual information for $\mathcal{B}$
- $\mathcal{Y}$ – the set of labels, i.e., the target responses mapping to $\mathcal{B}$

Other sets do occur, but these are defined in conjunction with them being declared.

2.4.1 Types of Variables and Functions

Apart from regular mathematical notation, types are used to give better context as to what a mathematical atom is. In general, this is usually done for functions, or *mappings*, e.g., $f : \mathcal{D} \to \mathcal{R}$. Here, $\mathcal{D}$ is the *domain* and $\mathcal{R}$ is the *range* of the function.

However, this can be generalized to arbitrary mathematical expressions or variables, e.g., $\mathbf{x} : \mathcal{B}$, which states that the variable $\mathbf{x}$ has the associated type $\mathcal{B}$. If $\mathbf{x}$ instead is a concrete value or element of $\mathcal{B}$, then is instead denoted as $\mathbf{x} \in \mathcal{B}$.

For the case of vectors and matrices, their concrete type can also be used instead of the more abstract data-related type (e.g., $\mathcal{B}$ for the set of messages). A vector of strings $\mathbf{s}$ containing k strings would then be denoted as $\mathbf{s} : \mathcal{S}^k$, much like for the aforementioned predictor type $\mathcal{P} = \mathcal{S}^k \times \mathbb{R}^l$, where both string- and number vectors are used.

For expressions containing pairs, or *tuples*, of variables, i.e., (x, y, z), regular cartesian product of all variable types are used: $(x, y, z) : \mathcal{A} \times \mathcal{B} \times \mathcal{C}$. In the case of nested tuples, e.g., $\big((x, y), z\big)$, the inner cartesian products is enclosed within parentheses to emphasize the nested structure of the tuples: $\big((x, y), z\big) : (\mathcal{A} \times \mathcal{B}) \times \mathcal{C}$.

Throughout this paper, this generalized form of typing is used in order to help disambiguate what an expression is. One place where these typings are used extensively are in code listings, where type annotations are used in documentation or as Python type hints [10] inside the code itself.

2.5 Shared Memory in Python and PyTorch

Working with machine learning models involves training and evaluation on data. In PyTorch, these data are represented using `Datasets` and `DataLoaders` [11]; Datasets act as representations of data and how to access to it, and Data Loaders manage the sampling, loading, and batching of the data.

After trial and error of loading in data efficiently, indexing and preprocessing of the raw data is performed in batches which are then stored in a cache managed by a custom implementation of the `Dataset` -class. By utilizing the cache, the expensive preprocessing of the data is only necessary during the first epoch, after which the cache is filled and is used instead of the raw data.

Data loading is often a bottle-neck during training of models, which is somewhat remedied by using multiple worker processes for loading. The number of worker processes to use is easily set in the `DataLoader`-class' constructor [11]. However, by using multiple worker processes, the required memory is copied to new separate processes to achieve true parallelism as the *global interpreter lock* (GIL), the global lock which enforces only one thread to have control of a Python interpreter, does not allow parallelism via simple threads [12]. In the case of data loading, the `Dataset`-instance is copied across each worker process, cloning the current state of the original `Dataset`-instance each time they are instantiated. This instantiation occurs at the start of each sampling iteration of the `DataLoader`-instance, e.g., at the beginning of each epoch. In this case, the processes are then killed at the end of each epoch.

With the worker processes cloning the `Dataset` each time they are started and sampling different indices of the data [11] to ensure each datapoint occurring only once per epoch, using a simple cache is not feasible since each worker process will cache their own equally-sized subset of the data in their own `Dataset`-instance. Therefore, in order to use both the cache and parallel data loading, the custom `Dataset`-class uses shared memory to store the data in memory space shared by all Python processes [12]. Native Python shared memory was introduced in version 3.8, allowing multiple processes to use shared memory defined using any form of valid Python value [13]. However, as this project is implemented to be run in the cloud, the Python version is locked to version 3.6 and the only form of shared memory is done using `sharedctypes` and the `ctypes`-module [12, 14].

Using shared memory via `sharedctypes` involves constructing a C-type structure using the C-bindings in the `ctypes`-module and then instantiating a shared array-object of the correct size, i.e., `len(data) * cache_type`, where the `cache_type` is the C-type structure class. To then write and read cache entries of the shared array, the Python representation needs to be transformed to and from the C-type structure. This procedure of reading back and forth is however significantly faster than performing the preprocessing repeatedly over the lifetime of the `Dataset`-instance. An example implementation for the C-type structure and the transformation functions are shown in listing 1, where the simplest cache type for this project is presented. Furthermore, instantiation of the shared memory with a dataset is shown in the function `init_dataset` at the bottom of listing 1.

Do note that some dependencies are exempt in listing 1, i.e., the classes which are derived from for the concrete types `TOKENIZED_SENTENCE` and `BASE_TYPE`. These are instead presented in listings 7 to 9 in appendix C. In order to make use of the shared array using a custom C-type in multiple processes, the classes need to be statically defined as classes themselves are not copied across processes when using the `multiprocessing`-module [12].

Going with a shared memory cache could arguably be avoided by using the main thread for data loading and let the loading take longer. However, since this project also utilizes `DistributedDataParallel`-module in PyTorch for parallel training with multiple GPUs, the work is distributed across each GPU by replicating the main process once for each GPU [15]. Therefore, even though the main thread is used for data loading, the shared memory cache is still necessary in order to utilize a cache with multiple GPUs, much like with multiple worker processes for one dataset.

Listing 1: Shared memory cache type implementation using Python's `ctypes` - and `sharedctypes` -modules. Note that the `DerivedDataset` -class is only a placeholder-class, meaning that it lacks an implementation. Additional code for the classes which are derived from can be found in listings 7 to 9.

```
''' Example snippet of the simplest CACHE_TYPE C-structure and
    how it can be used using multiprocessing as a cache
'''
import ctypes
import torch.multiprocessing as mp
from custom_dataset import DerivedDataset # Dummy class,
                                          # needs implementation
TOKEN_ARR = ctypes.c_long * 128
ATTENTION_ARR = ctypes.c_int8 * 128
LABEL = ctypes.c_float

class TOKENIZED_MESSAGE(TOKENIZED_SENTENCE):
    ''' Derived C-type data class for transformers.BatchEncoding '''
    _token_type = TOKEN_ARR * 1
    _attention_type = ATTENTION_ARR * 1
    _fields_ = [("input_ids", _token_type),
                ("attention_mask", _attention_type)]

class BASE_TYPE(CACHE_TYPE):
    ''' Derived C-type data class for torch.utils.data.Dataset
        cache using sharedctypes

        This class consists of a tokenized sentence
        from transformers.PreTrainedTokenizer,
        the severity score vector of AutoMod's
        internal model for the datapoint,
        and the output label.
    '''
    _sentence_type = TOKENIZED_MESSAGE
    _vector_type = ctypes.c_float * 10
    _label_type = LABEL
    _fields_ = [
        ("sentence_tokens", _sentence_type),
        ("vector", _vector_type),
        ("label", _label_type)
    ]

def init_dataset(data):
    # initialize C-type shared memory array
    # using multiprocessing's shared arrays.
    cache = mp.Array(BASE_TYPE, len(data))
    # initialize dataset logic here...
    dataset = DerivedDataset(data, cache, ...)
    return dataset
```

3

Procuring, Refining, and Preparing Datasets

Fulfilling the defined aim of this project will involve different procedures, methods, and tools. For the sake of consistency and coherence, the project is structured into two primary components: the *datasets* and the *moderator model.* Each of these components may have multiple variations which consist of smaller subcomponents that are implemented at different stages of the project's progression.

For the datasets, these are defined as the sets of predictor-response pairs $(\mathbf{x}, y) : \mathcal{P} \times \{0, 1\}$ that are to be used for training, validation, and testing of each variation of the model. Predictors $\mathbf{X}$ are the messages sent in the chat of any Twitch channel which fulfill all criteria listed in section 1.4, along with potential string-based- or real number-based metadata about the content which form a context. The responses $\mathbf{y}$ are binary encodings of boolean values representing whether each message is appropriate ($\mathbf{True} = 1$) or not ($\mathbf{False} = 0$).

3.1 Procurement of Data from Twitch

In order to begin investigating the impact on performance from layering additional information onto Twitch chat, the first step is to produce the necessary dataset in which all potential models can be trained and validated against. What is considered as *necessary* data are the predictors $\mathbf{M} : \mathcal{S}$, i.e., the chat messages from chat, and the labels $\mathbf{y}$, which are inferred from the predictions made by AutoMod in its current state and the interaction of moderators.

To be more precise, let $A, C, D : \mathcal{S} \to \{\mathbf{True}, \mathbf{False}\}$ be predicates working on messages defined as

$$\begin{aligned}
A(\mathbf{m}) &:= \text{“}\mathbf{m}\text{ is allowed by a moderator}''\text{,} \\
C(\mathbf{m}) &:= \text{“}\mathbf{m}\text{ is captured by AutoMod}''\text{,} \\
D(\mathbf{m}) &:= \text{“}\mathbf{m}\text{ is manually deleted by moderator}''\text{.}
\end{aligned}$$

Then, given the message $\mathbf{m}$, the corresponding label y is inferred as

$$y(\mathbf{x}) \coloneqq \begin{cases} 1, & if\ A(\mathbf{m}) \land C(\mathbf{m}), \\ 0, & if\ \neg A(\mathbf{m}) \land C(\mathbf{m}) \lor D(\mathbf{m}), \\ -1, & otherwise. \end{cases} \tag{3.1}$$

Do note that the third return value (-1) of equation (3.1) is defined for the set of messages which are either captured by AutoMod and ignored by moderators (i.e., the appropriacy of the message is ambiguous) or not captured by AutoMod and not manually deleted by a moderator (i.e., the message is correctly identified as appropriate). In both cases, it is not certain that the inferred label should be zero or one, since the prior case could be a false negative from AutoMod and the latter a false positive; without any interaction from the moderators, it is not possible to distinguish such cases from true negatives and true positives, respectively.

Furthermore, with the first additional layer of information being the metadata of the broadcast at the time of each message being sent, procuring these data in conjunction with the necessary data described above is a sound idea. Therefore, the following broadcast metadata is fetched along with the necessary data:

1. Severity scores of the categories of inappropriate words from AutoMod's internal model
2. Current channel title of the broadcast at the time of message (on a per-minute basis)
3. Channel category
4. Broadcast tags
5. Broadcast category tags
6. User category

For more information regarding items 2 to 6, refer to section 2.1.

Each category's severity score (item 1) is defined to be within the range [0,1], with 0 being no severity and 1 being utmost severity. Currently, the categories which are considered by AutoMod are defined as follows:

1. *Aggression* – Threatening, inciting, or promoting violence or other harm
2. – Name-calling, insults, or antagonization
3. *Disability* – Demonstrating hatred or prejudice based on perceived or actual mental or physical abilities
4. *Sexuality, sex, or gender* – Demonstrating hatred or prejudice based on sexual identity, sexual orientation, gender identity, or gender expression
5. *Misogyny* – Demonstrating hatred or prejudice against women, including sexual objectification

6. *Race, ethnicity, or religion* – Demonstrating hatred or prejudice based on race, ethnicity, or religion
7. *Sex-based terms* – Sexual acts, anatomy
8. *Swearing*

All these data, both necessary data and broadcast metadata, are gathered from Twitch's data warehouse, housed inside *Amazon Redshift*[1] and unloaded in the *Apache Parquet*[2] tabular format.

In order to get as much data as possible under the delimitations defined in section 1.4, the timeframe of which the data are gathered from the data warehouse is April 15th, 2020 to November 1st, 2020 since April 15th, 2020 is the date of which the current internal model of AutoMod was introduced. There have been some configuration changes to the model over this timeframe; however, the assumption is made that these changes should not have a significant impact on the overall outcome of the investigation. Since the same datasets will be used across the implemented model variations during training and validation, the aim of investigating the impact of additional context should not be affected. Furthermore, since each datapoint in the datasets are subject to the eyes of a moderator, the labels in the dataset should still be correct w.r.t. appropriacy, assuming that the moderator is doing their task correctly.

The unloaded data is then stored in a storage bucket within *Amazon S3*[3], where the data is then accessible for the next step of splitting it into training- and validation sets, followed by the process of separating the necessary data from the broadcast metadata.

3.1.1 Constructing the Training- and Validation Sets

With the necessary data for training and validation having been gathered, along with the broadcast metadata, the next step is to split these data into reusable training- and validation sets. Depending on the data quantity, the considered options are either, in the case of larger data quantities, to go with hold-out cross-validation of the data or, in the case of smaller data quantities, to go with k-fold cross validation [16]. In this instance, where the data is of large quantity (2,647,048 datapoints), going with hold-out validation should suffice to get representative results. Furthermore, a single split is preferred on this scale, since it will reduce the training times by a factor of k compared to k-fold cross validation [17]. The training-validation ratio is chosen to be $9 : 1$, meaning that approximately 90% of the data will be used for training of the models and the remaining 10% will be used for validation.

Another important factor to consider is how the data is split/sampled to reduce both the bias and variance in the model's performance [16]. Having a validation set which mimics the test set is also important since the validation set is intended to act as the

[1]Amazon Redshift Overview Page: https://aws.amazon.com/redshift/
[2]Apache Parquet Documentation: https://parquet.apache.org/documentation/latest/
[3]Amazon S3 Overview Page: https://aws.amazon.com/s3/

basis of performance indication during training and tuning of a model [18]. Since the problem at hand involves real-time moderation of chat, it has the goal of correctly classifying appropriacy of chat messages sent in the future and, hence, involves time series forecasting. Therefore, the choice of sampling method is the time-dependent technique *convenience sampling* for its efficiency and determinism [16], which is frequently used when working with time series [19, 20].

More concretely, the necessary data is ordered with regards to the chat messages' timestamp and then segmented on a fixed date where the approximate $9:1$ ratio is maintained between the training- and validation sets. Segmentation is performed such that all messages within the validation set occurred the day after the last message occurred in the training set; this is to mimic the behavior of the test data, where the messages will occur at a later time than for the messages in both the training- and validation sets.

With the hold-out cross-validation being done using convenience sampling and a training-validation ratio of $9:1$, the resulting sets are saved as separate files in Apache Parquet format in the same Amazon S3 bucket as the original data which was fetched from Twitch's data warehouse; ready for consumption by the implemented model variations. In addition, in order to keep the necessary data separate from the broadcast metadata, the latter is sampled in the same manner as the former and then stored alongside the datasets in separate metadata-datasets: one for the training dataset and one for the validation dataset.

3.1.2 Investigating Robustness and Generalization Against Unseen Data

Apart from simply evaluating each model on the validation set, the models are also evaluated on specific subsets of the validation dataset. These subsets are defined such that different aspects of interest are considered; namely, (1) channel ID – the channel itself, (2) broadcast category – the content of the stream, and (3) message body – the content of the sent message. For each of these aspects, the validation dataset is split into two subsets; one where the value of the aspect is shared with the training set and one where the value is unique to the validation set, i.e., it has not before been seen by a model during training. More formally, let a be an aspect of interest, $\mathcal{V}$ and $\mathcal{T}$ the validation- and training sets, and let d be a datapoint. Then, the subsets can be defined as follows:

$$\mathcal{V}_a^u = \{d \mid d \in \mathcal{V}, d^{(a)} \in \mathcal{V}^{(a)} \setminus \mathcal{T}^{(a)}\}, \tag{3.2}$$

$$\mathcal{V}_a^c = \{d \mid d \in \mathcal{V}, d^{(a)} \in \mathcal{V}^{(a)} \cap \mathcal{T}^{(a)}\}, \tag{3.3}$$

The parenthesized superscripts of the two sets and datapoint represents the value w.r.t. the aspect (feature), e.g., for $a :=$ "channel ID", $d^{(a)}$ is the datapoint's channel ID and $\mathcal{V}^{(a)}$ is the set of all channel IDs within the validation set. Equation (3.2) is the equation for validation data with unique (or unseen) values of a and equation (3.3) is the equation for the validation data with values in common with the training data.

3.2 Procurement of External Data

In one of the models to be produced, the intention is to incorporate information from external sources to investigate how external information may provide more contextual value. More precisely, as listed in section 1.4, this project is scoped to only depend on the data provided by IGDB. IGDB's data is open to the public and available via their API [21]. What the data provided by IGDB consists of are metadata about games. With games being the primary source of content at Twitch, the reasoning is that the model will improve in understanding the appropriacy of a viewer's chat message when the message is paired with additional information about the game content of the stream.

As IGDB is a part of Twitch, there exists a mapping between the category IDs at Twitch with the game IDs at IGDB, making it easy to join the two data sources together. Of all the metadata, the metadata of interest which are also in a relatively concise representation are the following:

- Summary – A short paragraph describing the main points of the game title.
- Genres – A list of genres that apply to the game, e.g., *FPS* or *RPG*.
- Themes – A list of themes that apply to the game, e.g., *Adventure* or *Open world*.
- Franchises – A list of franchises that the game title belongs to, e.g., *Assassin's Creed* for any Assassin's Creed game.

An important aspect to note, however, is that IGDB only requires there to exist a game title for the game to be recognized as an entity in the database. In this instance, that means that the game data are far from complete and there are large amounts of data gaps, especially for less known titles. To handle this issue, each "data hole" is filled with a default value, `"[]"` – an empty list as a string – for multi-entity data and `""` – the empty string – for single-entity data, in order to avoid unexpected behavior when manipulating the data during the preprocessing steps. Each default value is a string since each well-defined value of the raw data is of this type before they are parsed using Python's `ast`-module [22].

3.3 Procurement of the Test Dataset

After having procured all necessary data for training and validating the models and having trained them all, procuring the test data is the next and final step regarding data. In order to ensure that the test set is large enough to show significant results, the minimum required sample size is estimated using the formula for standard error of a proportion, i.e., the standard error $SE(p)$ for a sample proportion p is

$$SE(p) = \sqrt{\frac{p(1-p)}{n}}.$$

To get an adequate measure of significance, the SE is chosen such that a result for a metric is significant with a 95% confidence interval. The greatest SE occurs when $p = 0.5$, meaning that if the metric which falls closest to 0.5 in general among the models is chosen and optimized for, the other metrics should also be safe as long as the SE is chosen to be small enough for all metrics to not have an overlap in the confidence interval. In this instance, based on the validation results, the metric closest to 0.5 happened to be Precision (about 0.62), with F1 Score being the metric with the narrowest difference in scores (about 0.03).

By letting $SE(p = 0.62) = 0.03/1.96$, where 1.96 is the approximate value of the 97.5 percentile of the normal distribution, and then solve for the sample size n, we get an estimated sample size of

$$\begin{aligned} n &> \frac{p(1-p)}{SE(p)^2} \\ &> \frac{0.62(1-0.62)}{(\frac{0.03}{1.96})^2} \\ &> 1005.6455 \\ &\geq 1006. \end{aligned} \tag{3.4}$$

With the estimated sample size calculated in equation (3.4), a test set of data is gathered from November 2nd until January 22nd with a subset of channels where there are moderators who have shown interest in helping Twitch with data tasks, e.g., surveys and data labeling. By following the constraints for the data listed in section 1.4, with some relaxed constraints, a total of 3,532 entries were gathered within the aforementioned timeframe. The constraint which has to be relaxed is the range of allowed concurrent viewers, which is changed from [100, 1000] to [30, 2000]. Without this relaxation, the total sample size is less than the estimated sample size of 1,006.

By having gathered the data, the moderators who belong to the channels where the data constraints are fulfilled were contacted. These are a total of 11, with varying quantities of data for their respective channels. The goal is for each of them to fill in a minimum of 50 samples (if they have that many) up to as many as they like, with the hopes of those with more than 50 entries filling in about 100 on average. In the end, this is (optimistically) expected to yield about 1,200 manually labeled datapoints, which is done by the domain experts themselves.

4

Architecturing, Implementing, and Iterating on the Model

With regards to the second primary component of the project – the moderator model, which is the predictive component of the project – the model will act as the probability function $g : \mathcal{B} \times \mathcal{C} \to [0, 1]$, where $\mathcal{C}$ will vary depending on the use of contextual information. For example, in the case of a AutoMod, $\mathcal{C} = \varnothing$ as the model does not use any additional contextual information. Conversely, the "complete model" with regards to the project aim will have a $\mathcal{C} = \mathcal{M} \times \mathcal{E}$, where $\mathcal{M}$ and $\mathcal{E}$ are the sets of stream metadata and external data, respectively. Furthermore, since the external data are gathered from the source IGDB, $\mathcal{E} \in \{\varnothing, \mathcal{I}\}$ where $\mathcal{I}$ is the set of datapoints from IGDB. Note that $\varnothing$ is part of the set of external datasets since incorporating only $\mathcal{M}$ to the training process is the first layer to be tested after the baseline is defined.

As for the loss function, logistic loss [23] will be used since the task is a binary classification task. That is, the loss function $\mathcal{L}(\mathbf{x}, y, \mathbf{c}) : \mathcal{B} \times \mathcal{Y} \times \mathcal{C} \to \mathbb{R}^+$ is defined as

$$\mathcal{L}(\mathbf{x}, y, \mathbf{c}) \coloneqq -\Big(y \log \big[g(\mathbf{x}, \mathbf{c})\big] + (1 - y) \log \big[1 - g(\mathbf{x}, \mathbf{c})\big]\Big), \tag{4.1}$$

where g is an instance of the moderator model, $(\mathbf{x}, y)$ is a pair of the predictor and response belonging to the dataset and $\mathbf{c}$ is the contextual data related to the pair and of the shape aligning with the requirements of g. Do note that the chat message $\mathbf{x} : \mathcal{B}$ and the corresponding context $\mathbf{c} : \mathcal{C}$ are separated here, compared to how they are jointly defined as predictors $\mathbf{x} : \mathcal{P}$ in chapter 3. These two notations are used to highlight different aspects; $\mathcal{P}$ being directed towards the predictors in the form of data and $\mathcal{B}, \mathcal{C}$ depicting more how the predictors are used by the moderator model g.

4.1 Defining a Baseline

At this point, the training- and validation sets have been defined and stored for reuse, which concludes the preliminary work that is necessary to start producing a model. Therefore, the next step is to produce a baseline model to compare the other variations against. In order to also compare against the current AutoMod, two types of baselines are constructed: one baseline is inferred from the labels produced by

AutoMod, and the other is defined as a Bag of Words (BoW) model [24], where the BoW-model is applied only on the chat messages **X**. Do note that the first baseline is based on calculations of AutoMod's output and not a trained model whilst the BoW-model is trained on the BoW-representation of each chat message.

Furthermore, two additional baseline models are constructed where the severity score vector of AutoMod's internal model (see list two of section 3.1) is introduced as a separate channel. One model only depends on this channel (SSV) and the other is an extension of the BoW-model (EBoW) where the severity vector-channel is combined with the BoW-transformed chat messages; the motivation is to investigate the effect of the internal model's classification on performance and to see if the implemented model is able to learn useful information from these severity scores. All three models BoW, EBoW, and SSV are implemented using Keras' functional API [25] within the library TensorFlow [26]. In figure 4.1, a legend describing the meaning behind the coloring and shape of each component in the following architecture diagrams. The general component architecture for all models is shown in figure 4.2a and the concrete baseline architecture of the models is shown in figure 4.2b.

All baselines, including both the SSV and EBoW models, are also compared to the performance of always selecting the majority label (a "Dummy classifier") in order to determine whether or not they are able to learn anything of value from the input data as to improve their predictions.

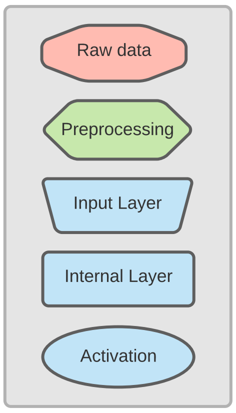
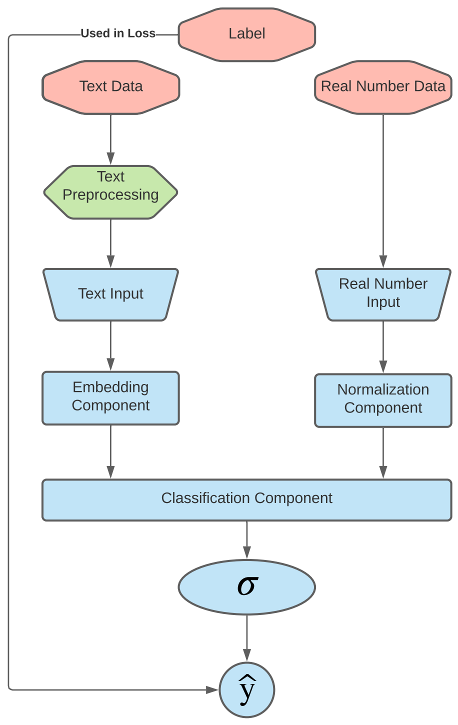
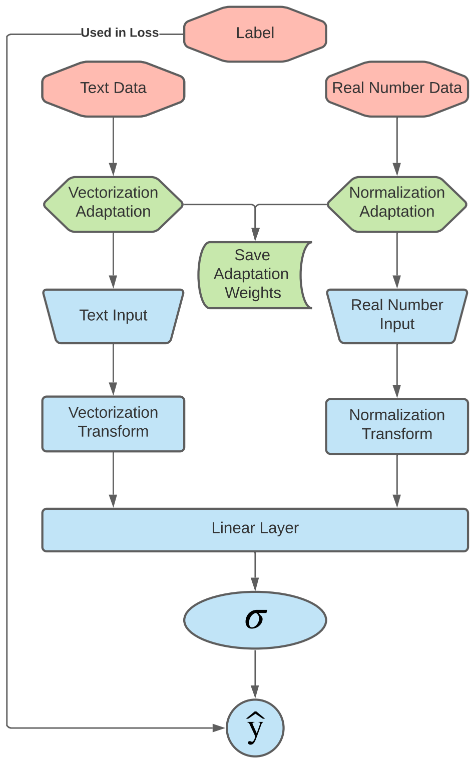


Figure 4.1: Legend description of what the color and shape encodes for each architecture diagram. Red is for raw data, green is for preprocessing procedures, and blue is for a model itself. Blue trapezoids are the inputs of a model, blue rectangles are intermediary layers, and ellipses are the activation functions used on a layer's output.



(a) General architecture of all models on a component level. There are three main components of each model; (1) the Embedding component for text input, (2) the Normalization component for real number input, and (3) the Prediction component for interpreting the internally-processed inputs to produce and output.



(b) Architecture diagram of the BoW, EBoW, and SSV models. Do note that the BoW model only uses the left (text) channel, the SSV model only uses the right (number) channel, and the EBoW model uses both channels.

Figure 4.2: General architecture- and baseline architecture diagrams. Note that each component of the general architecture may be defined by different implementations and have additional layers placed before/after depending on what the actual model is, as shown in the baseline model.

4.1.1 A Small Ablation Study of Chat Messages and Severity Scores

In order to investigate the importance and effect of the chat messages and the severity scores produced by AutoMod's internal model, a small ablation study is performed by training the baseline BoW, EBoW, and SSV models and then evaluating them on the validation dataset. With the validation results and trained weights of the linear layer for each baseline, some interpretation of the effect of the two data types can be done.

For example, if any of the weights for the severity score vector from the internal model of AutoMod is positive within either the EBoW or SSV models, then this could be an indication of there being a positive correlation between a message being appropriate and any identified severity from the internal model. More concretely, as the classification component of each baseline model being simple logistic regression, by letting $\mathbf{w}$ represent the weights of the linear layer and $\mathbf{x}$ be the preprocessed predictors, the output probability $\widehat{y}$ is given by

$$\widehat{y} = \sigma(\mathbf{w}^\intercal \mathbf{x})$$

Since each component of the severity vector is bound within [0, 1], the sign of each component k in the dot product is determined by the sign of $\mathbf{w}_k$. Therefore, if $\text{sign}(\mathbf{w}_k) = 1$, then

$$\mathbf{w}_k \cdot \mathbf{x}_k \geq 0$$

will indicate that the higher the severity is of type k (see the second list in section 3.1), the more likely $\widehat{y}$ will tend towards one since

$$\lim_{\mathbf{w}^\intercal \mathbf{x} \to \infty} \sigma(\mathbf{w}^\intercal \mathbf{x}) = 1.$$

This could answer why AutoMod misinterprets some of the data, as a high severity score should be a sign of *inappropriate* language and not the inverse.

4.2 Defining the Main Model

With baselines having been defined for comparison, the next step is to define the model that is to be used for investigating the hypotheses defined in section 1.5. Much like the EBoW baseline, the architecture of this model, called *AutoBERT*, is designed to consist of two separate channels due to the shape of the data; one for string inputs and one for numerical inputs. What differentiates AutoBERT from the baselines are the *Embedding*- and *Classification* components of the general architecture shown in figure 4.2a.

Instead of using a BoW-representation, the embeddings will be generated by *sentence transformers* – an extension of the BERT model called Sentence-BERT [27]. The authors of [27] provide state-of-the-art pretrained models for various tasks within natural language processing (NLP). These models can be applied on sentences and are available from the *Transformers*-library[1]. For this project, where the problem involves sequence classification through semantic textual similarity (STS), the pretrained models fine-tuned for STS-tasks are perfect candidates for the task of embedding the text-based inputs. In this instance, the choice of model is the *RoBERTa Base* model, a more robust version of its predecessor BERT [28], after being fine-tuned for STS [29], as it performed in the top while also having a high speed performance for transforming tokenized sentences into embeddings compared to other models[2]. Nonetheless, introducing a sentence transformer involves more work than simply changing out the embedding component. First off, the sentence

[1]Transformers library: https://huggingface.co/

[2]STS Transformers benchmarks: https://docs.google.com/spreadsheets/d/14QplCdT[...]

transformers are recommended to be used with PyTorch instead of Tensorflow as the implementation itself is based on PyTorch [27, 30]. Documentation of the library and the examples of the transformers are also written using PyTorch [29]. Therefore, in order to have the easiest time incorporating the transformers, the data loading and preprocessing pipeline needs to be revised and defined in PyTorch as well.

Furthermore, in order to use a transformer on a GPU for accelerated computation in PyTorch, the strings need to be tokenized to an encoding representations before they are transferred onto the GPU as PyTorch does not support strings to be stored in tensors [31] – the multi-dimensional matrices used for computations in PyTorch. What this means is that tokenization will be part of the data loading and preprocessing step, which will be performed by the *Fast Tokenizer*, a word tokenizer which uses byte-pair encoding [32], which is paired with the sentence transformer of choice. Before tokenization, each piece of raw text is transformed to sentence form, which is simply a mapping $m : \mathcal{S} \to \mathcal{S}$. An example of these mappings is the one for Twitch categories, which takes a category $C : \mathcal{S}$, e.g., “Assassin’s Creed Valhalla” and returns the string “with the content of the stream being ‘Assassin’s Creed Valhalla’ ”. These transformed pieces of text are then concatenated together with the *delimit-token* `<s/>` interspersed inbetween, as per the documentation of RoBERTa [32]. By interspersing the delimit-token, the tokenizer can reason about what defines a sentence and tokenize the text fragments while considering a sentence structure. Doing this procedure results in each text input being constructed as a template string, which should help the sentence transformer reason about the inputs easier as it is trained on fluent natural languages.

With tokenization being performed in the data loading step and since data loading within PyTorch uses multiprocessing [11], the dataset definition within PyTorch is a custom class which is defined to make use of a shared memory cache in order to avoid unnecessary duplicate work during data loading. Details on how this custom class is implemented is further described in section 2.5. From using the shared memory cache, each data loader will work on a subset of the complete dataset to perform all preprocessing, which is then stored in the dataset cache during the first epoch. From thereon out, the shared memory cache is used by all data loaders, meaning that the preprocessing is only performed once and never duplicated across any worker processes, independent of the worker process’ lifecycle. A complete architecture diagram of the AutoBERT model and the surrounding data loading is shown in figure 4.3 to further concretize how the data flows on a per-batch level.

After the tokenized strings have been transformed by the sentence transformer, the output is first mean pooled, i.e., the collection of word embedding $\mathbf{W} : \mathbb{R}^{768\times k}$ of the sentence is averaged over k to produce a “sentence embedding” $\mathbf{s} : \mathbb{R}^{7}68$, as per the recommendations in [27], and then normalized to ensure that the magnitude between the fixed vector embeddings and normalized vector inputs are not large enough to skew the importance of the respective channels. After the sentence/paragraph has been normalized, the normalized embeddings are fed into the classification component along with the normalized vector input to produce the final prediction score.

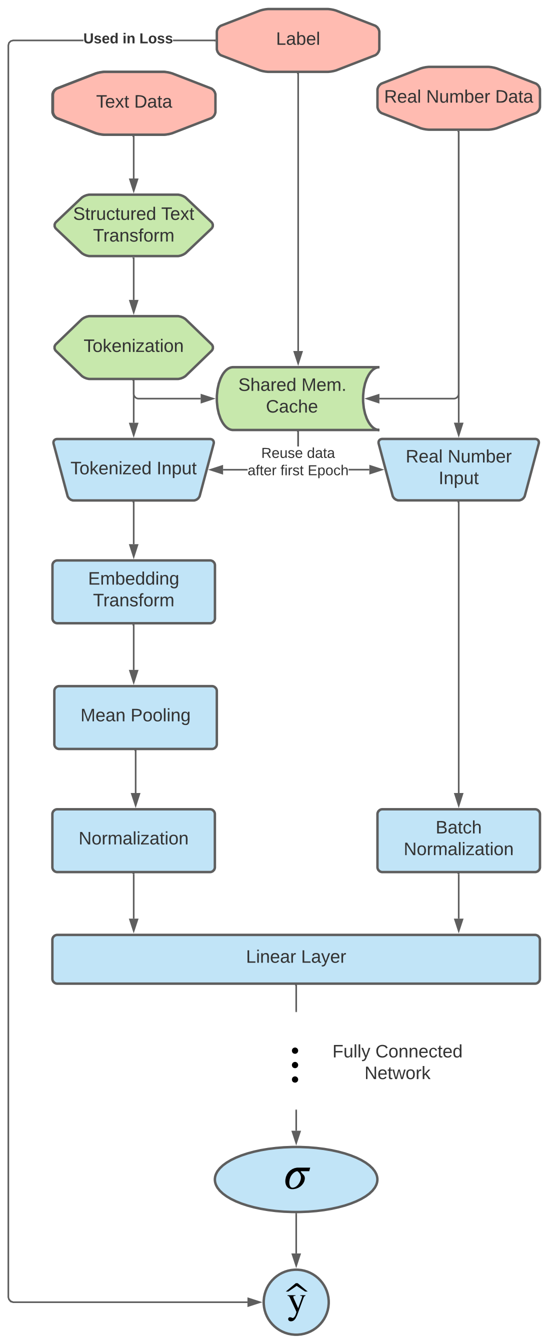


Figure 4.3: Model and data loading architecture of the AutoBERT model. During the first epoch, the data is loaded from the raw table data within a dataset. The string data is then tokenized, followed by being combined with the number data and target label for storing in the dataset cache. Once stored in the dataset cache, the cache is then queried instead of the raw table data the next time it is queried for. The label is also stored withing the cache as the data loader within PyTorch requires both predictors and targets to be returned in the loaded batches.

As shown in figure 4.3, the classification component is a fully connected network instead of a simple linear layer. The complete fully connected network architecture is shown in figure A.1. Apart from the linear layer, which is seen as the input layer of the network, four hidden linear layers with interspersed *Leaky ReLU* (LReLU) activations constitute the classification component, with the last activation being the Sigmoid activation function, much like for the baselines. LReLU is similar to the regular *ReLU* activation function, but with the contrast of having a negative slope k for the case of $x < 0$ instead of simply zero [33]. More concretely, the LReLU- and ReLU activations are defined as

$$\text{LReLU}(x; k) = \begin{cases} x, & \text{if } x \geq 0 \\ k \cdot x, & \text{otherwise,} \end{cases} \tag{4.2}$$

$$\text{ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise.} \end{cases} \tag{4.3}$$

Each of the dimensions for the hidden layers; as well as many other hyperparameters, such as learning rate, batch size, etc.; are defined using configurations.

4.2.1 AutoBERT Configurations of each Variation

With each AutoBERT variation depending on additional data, the input dimensions to each model will vary and the model instances need to be configured accordingly. Therefore, each model is instantiated from a run-time configuration that has been predefined to incorporate the correct data into the inputs and how these data should be segmented into text-based- and number-based inputs. The three model variations are defined as follows:

1. Base – Depending on both chat messages and the severity score vectors from AutoMod's internal model which are listed in the second list in section 3.1
2. Stream – Depending on the same data as Base, but also stream metadata defined in the first list of section 3.1
3. IGDB – Depending on the same data as Stream, but also games metadata from IGDB, listed in the first list of section 3.2

Since the data of each new variation is a superset of the prior, the configuration for the IGDB variation will also include the relevant information for the Base- and Stream variations. The configuration for the IGDB variation is listed in listing 6.

In the `mixins` -field in listing 6, the dedicated text transformations are mapped to their corresponding data types. Note that all are specific to each data type, except for the `MESSAGE_BODY` - and `SUMMARY` fields, which have the identity function $id : a \to a$. These two fields are free-text data, meaning that their lengths are arbitrary. Therefore, the identity function is used in order to minimize the risk of having the tokenizer truncate the tokenized input to its maximum size of 128 tokens. The other transformations are simply functions which put the data into sentence form, e.g., the aforementioned category mapper described in section 4.2.

4.3 Training the Models

In order to have the models learn to solve the problem, it is trivial that they need to be trained. Therefore, each model (both baselines and AutoBERT variations) is trained and validated on the training- and validation sets described in section 3.1.1. Since the baseline models are implemented in Keras/TensorFlow and the AutoBERT models are implemented in PyTorch, the training procedures are not identical for the two types of models. However, both adhere to the typical training- and validation flow, where the model is trained on shuffled training data which is sampled without replacement until no more training samples remain in the training set. Once there are no more training samples, i.e., one epoch has finished, the model is evaluated on the validation set.

Furthermore, during a training session, i.e., over the time of multiple epochs, the best-performing weights are tracked on a per-epoch basis by tracking where the lowest validation loss, described in equation (4.1), occurs based on the full validation set. Each time there is improvement, a checkpoint is saved for that model and reused later on during evaluation.

Once the parameters of the models themselves are tuned in accordance to the validation set and their best-performing checkpoints are saved, the decision threshold for each model is tuned by selecting the threshold which yields the highest F1 Score in the validation set, where the threshold is from the range [0,1] with 0.1 step increments. These thresholds are then used for the final evaluation on the validation- and test sets, as well as in the investigation of generalization and robustness described in section 3.1.2.

In the following subsections, the differing factors of the training procedure are outlined for both the baseline- and AutoBERT models. Such factors include hyperparameters, data loading, and data preprocessing.

4.3.1 Baseline Model Training and Hyperparameters

When training the baseline models, the data loading and data preprocessing is identical, except for what data type each model depends on, i.e., what channels to depend on in figure 4.2b. In fact, they reuse the same preprocessing layer instances. After being adapted to the training data once before training the first model, the preprocessing layer instances are saved and then loaded in depending on if the data type each layer manages (chat messages or severity scores) is needed by the model to be trained. Before persisting the Text Vectorization layer (see figure 4.2b), however, the hyperparemeters of the layer had to be tuned. Keras' regular Normalization layer does not have any real hyperparameters, except if one knows a preset mean and variance to normalize from. In this instance, these parameters are unknown and have to be determined via the layer's `adapt` -method before being saved.

Tuning of the hyperparameters of the Text Vectorization layer involved the parameters:

1. `max_tokens` (vocabulary size)

2. `standardize`
3. `split`
4. `n_grams`
5. `output_mode`

The tuning procedure involved first instantiating the layer with the parameters to tune and then calling that instance's `adapt`-method on the training dataset. Once the layer had adapted to the training data, the BoW baseline was trained on the training set and then evaluated on the validation set. The configuration of the Text Vectorization layer for the best-performing BoW model after 10 epochs was chosen as the tuned parameter configuration. When having tuned these parameters, the best configuration of the layer was found to be the configuration shown in listing 2.

Listing 2: Keras' Text Vectorization layer configuration after hyperparameter tuning. Note that the parameters which are exempt are left to default and that the parameters which are occurring without the mention of being tuned are necessary for the model to work.

```
TextVectorization(
    max_tokens=10000,
    standardize=LOWER_AND_STRIP_PUNCTUATION,
    split=SPLIT_ON_WHITESPACE,
    ngrams=2,
    output_mode='tf-idf'
)
```

Apart from tuning the Text Vectorization layer, not much else of the model could be tuned. The linear layer of the baselines are dependent on the `max_tokens`-parameter and the dimensions of the severity score vectors, since the dimensionality of these layers are equivalent to $\dim(BoW) + \dim(SSV)$, where $\dim(BoW) = 10000$ and $\dim(SSV) = 10$ in the case of the parameters tuned for in listing 2. Do note that for the BoW baseline $\dim(SSV) = 0$ and for the SSV baseline, $\dim(BoW) = 0$, as these channels are not used in the respective models.

4.3.1.1 Hyperparameters for Loading and Processing of Data

Regarding hyperparameters of the data loading and data processing, the training data loader was configured to shuffle data in buffers of size 1024 and had a batch size of 256. What the shuffle buffer size results in is a 1024 size buffer of elements which the batches are sampled from. Each time an element in the buffer is selected, it is replaced with the first element in the remainder of the dataset. For example, initially, the buffer will contain the first 1024 rows of the training set. When an element i is selected, it is replaced with the 1025th element of the training set. This is not true shuffling, as is stated in the TensorFlow documentation [34]. However, by setting the parameter `reshuffle_each_iteration=True`, the dataset is pseudorandomly reshuffled each epoch, making the shuffling more stochastic and likely stochastic enough given a batch size of 256 and the size of the dataset.

Do note that the shuffle buffer size was not tuned, but the batch size was tuned for performance in both speed and loss. In the end, the dataset configuration was defined as shown in listing 3. The `baseline_mapper` is simply a model-specific mapping function from the dataset to the predictors and labels, i.e.,

$$\texttt{baseline\_mapper} : R \to (\mathbf{x}, y),$$

where R is a dataset row, $(\mathbf{x}, y)$ is the predictor-response pair as described in the second paragraph of chapter 3.

Listing 3: Dataset configuration for the three baseline models. Note that for the validation set, the `shuffle`-call is exempt. Furthermore, the argument `baseline_mapper` is a mapping function from a complete row of data to only the necessary predictors and the label, e.g., the chat message and label for the BoW model. The argument `tf.data.experimental.AUTOTUNE` is a variable set by TensorFlow's engine which allows the system to dynamically optimize for parallel workers when this is possible.

```
dataset = data_source.map(
            baseline_mapper,
            num_parallel_calls=tf.data.experimental.AUTOTUNE
        ).shuffle(1024) \
         .batch(256) \
         .prefetch(tf.data.experimental.AUTOTUNE)
```

As for training duration, each baseline model was trained for 10 epochs each as this was found to be enough time for the best checkpoint to occur before the last epoch and without having performance increases at the end of the duration, i.e., the model's learning had stagnated or it had started to overfit.

4.3.2 AutoBERT Model Training and Hyperparameters

When training the AutoBERT models, much like for the baselines, the data loading and data processing is generalized such that the same procedures are used, independent of the data types that each model depends on. However, nothing more can be persisted between runs except for the model checkpoints themselves, as the heavy lifting is done by the internal RoBERTa model when transforming the tokenized sentences into embeddings and the tokenized sentences will differ between each model variation due to the difference in transformed string data before tokenization.

Do note, however, that all tokenized sentences are stored in a shared memory cache such that the raw data of each dataset is only traversed in the first epoch, as shown in figure 4.3. After the first epoch has finished, the shared memory cache is used instead in order to save time when prefetching- and preprocessing data during training. More on how the shared memory cache and how it interacts with the dataset and data loading can be found in section 2.5.

As for hyperparameters of AutoBERT, the involved parameters are: (1) `hidden_dims`, (2) `learning_rate`, (3) `max_learning_rate`, and (4) `epochs`. Parameter

Listing 4: Hyperparameter configuration of the AutoBERT models.

```
autobert_conf = {
    "hidden_dims": (300, 300, 72, 72),
    "learning_rate": 0.001,
    "max_learning_rate": 0.01,
    "epochs": 20
}
```

`hidden_dims` is a tuple of four positive integers defining the number of neurons in each hidden layer of the classification component, shown in figure A.1.

Both `learning_rate` and `max_learning_rate` define the range in which the learning rate will be cycled, using cyclical learning rate with a triangular2 configuration [35]. If `max_learning_rate` is omitted, then the learning rate will be static across all epochs of a run. Per the recommendations of [35], the stepsize, i.e., the number of iterations between minimum and maximum learning rates, is set to be $2 \cdot |epoch|$ where $|epoch|$ is the number of batches in an epoch. In [35], the recommended stepsize is somewhere in $[2 \cdot |epoch|, 10 \cdot |epoch|]$

Regarding `epochs`, this parameter defines the number of epochs for a complete run. Tuning of this parameter was not performed very rigorously, since the best-performing model checkpoint was always saved in either case. Therefore, the goal of this parameter was for it to be set high enough for the model to converge before the last epoch, much like the motivation for setting the epochs of the baselines in section 4.3.1.

One last thing to note is that the tuning was performed for both the Base- and Stream variations, but not for the IGDB variation due to lack of time. Instead, the same hyperparameters of the Stream variation was used in the IGDB variation, which also ended up being the same for the Base variation. The decided upon hyperparameters for all three model variations are listed in listing 4.

4.3.2.1 Hyperparameters for Loading and Processing Data

When it instead involves hyperparameters for the loading and processing of data, the involved parameters are: (1) `batch_size`, (2) `sample_batch_size`, and (3) `data_workers`. `batch_size` and `sample_batch_size` combined define the total batch size for a run. The former parameter defines how many "sample batches" of size `sample_batch_size` which should be joined together as a batch. The latter parameter instead defines how many string data entries should be tokenized in a batch, as the tokenizer also works faster when working on batched data.

As for `data_workers`, this parameter defines how many parallel processes should work with loading data and preparing the batches for the model, i.e., either loading raw data, preprocess it, and store it in the cache; or fetch the already preprocessed data entries from the cache. The indices for each sample batch comes fully shuffled using a Random Sampler from PyTorch, and the batch indices corresponding to `batch_size` instead come from PyTorch's Batch Sampler; resulting in fully shuffled

Table 4.1: Tuned hyperparameters for each AutoBERT model with regards to loading and processing data before it is ingested into AutoBERT.

Model	`batch_size`	`sample_batch_size`	`data_workers`
Base	4	256	4
Stream	4	128	3
IGDB	3	128	1

batches every epoch, as compared with the solution for the baselines.

In order to leverage the computation resources as much as possible, each model variation has been tuned to focus on filling up the GPU memory in order to have RoBERTa embed as many paragraphs as possible in order to minimize the lead time during the embedding transformation. Furthermore, with the use of four GPUs, the models were trained in a parallel and distributed manner using PyTorch's Data Distributed Parallel- and Process Group modules. This meant that the total batch size was quadrupled as compared to only running on a single GPU.

All data-related tuned parameters for each model are listed in table 4.1. Do note the drop in all variables as more data are included for tokenization and embedding transformation. For the Base model, i.e., only chat messages in the embedding data, the GPU memory filled up at around a total batch size of 1024 entries per GPU. However, in order to have the data loading keep up with the fast tokenizer, the size had to be dropped in order to have a higher `data_workers` value set. As more data were introduced, `data_workers` also had to be reduced in tandem with the other parameters; not due to the GPU memory filling up, but due to the shared memory limit of 64MB when running SageMaker (Jupyter) in local mode. This caused many workers to fail and become zombie processes and locking up memory in the GPUs, eventually causing the training to fail due to memory leakage.

4.4 Evaluating the Models

With the models being trained, the next and final step is the evaluation of each model. Evaluation is done by first selecting the best-performing epoch of each model w.r.t. validation loss via their saved checkpoint, as described in section 4.3. After selecting the best epoch for a model, the model is re-evaluated on the validation set to find the best decision threshold by varying the threshold and picking the case where the F1 Score is the greatest.

When a decision threshold is made, the primary evaluation is performed; first by re-evaluating the model on the full validation set, second by performing the robustness and generalization of each model using the validation subsets described in section 3.1.2, and lastly by evaluating the model on the test set described in section 3.3. In all three cases, with true positives = TP, false positives = FP, true negatives = TN, and false negatives = FN, the used metrics are a subset of:

1. Accuracy: $\dfrac{TP + TN}{TP + FP + TN + FN}$
2. Precision: $\dfrac{TP}{TP + FP}$
3. Recall[3]: $\dfrac{TP}{TP + FN}$
4. F1 Score: $2 \cdot \dfrac{Precision \cdot Recall}{Precision + Recall}$
5. Specificity: $\dfrac{TN}{TN + FP}$
6. Area under ROC curve (AUROC) – Exempt for the investigation of robustness and generalization
7. Area under Precision-Recall curve (AUPR) – Exempt for the investigation of robustness and generalization

These metrics are selected in order to interpret each model's predictive abilities on both a dependent and independent level w.r.t. the decision threshold. Furthermore, in each part of the evaluation, confidence bounds are estimated for each metrics using the found best-performing algorithm in [36]. In the following subsection, this algorithm is described in more detail.

4.4.1 Estimation of Confidence Bounds

With the distribution of each metric being unknown, parametric confidence intervals are not certain to be representative. Therefore, the iterative bootstrap-sampling approach described in [36] is used as it is both straightforward and was found to be one of the best-performing methods for the case of estimating confidence intervals with small confidence interval widths. Based on the described method in [36], the interpreted and implemented algorithm is described in listing 5. Using this algorithm, the confidence interval for each of the metrics listed in section 4.4 are estimated.

4.5 Summarizing the Baselines and AutoBERT Models

Up until this point, much information has been given regarding the different baselines, the AutoBERT models, and what each model is intended to do. In the following sections, the two types of models and their respective instances are briefly summarized.

4.5.1 Baseline Model Summary

To reiterate, the baseline models are as follows:

[3]Recall is also referred to as *Sensitivity* when looked at along with Specificity

1. Bag of Words (BoW) – Depending on only the chat messages themselves
2. Severity Score Vectors (SSV) – Depending only on AutoMod's model's severity scores which are listed in the second list in section 3.1
3. Extended Bag of Words (EBoW) – Depending on both chat messages and the severity score vectors from AutoMod's internal model which are listed in the second list in section 3.1

Each of the three baseline models are used for two purposes: first, to act as baseline benchmarks for the AutoBERT models to beat in the evaluation process described in section 4.4; second, to be used for the small ablation study described in section 4.1.1. The baselines follow the architecture found in figure 4.2b and their hyperparameters are defined in section 4.3.1.

4.5.2 AutoBERT Model Summary

To reiterate, the AutoBERT models are as follows:

1. Base – Depending on both chat messages and the severity score vectors from AutoMod's internal model which are listed in the second list in section 3.1
2. Stream – Depending on the same data as Base, but also stream metadata defined in the first list of section 3.1
3. IGDB – Depending on the same data as Stream, but also games metadata from IGDB, listed in the first list of section 3.2

Each of the three AutoBERT models are used for one purpose: to accept or reject the hypotheses listed in section 1.5 by evaluating them against the baselines using the precedures described in section 4.4. The AutoBERT models follow the architecture found in figure 4.2b, where their classification component is found in figure A.1. All hyperparameters of the AutoBERT models are defined in section 4.3.2.

Listing 5: Confidence interval sampling algorithm based on the procedure description of [36]. Note that the implementation is pseudocode using Python syntax and not the real implementation itself.

```
''' Confidence interval bootstrap sampling algorithm

    Functions:
    - sample : ([int], int, bool) -> [int]
        - Takes a list of indices, a number `n` of samples to draw,
          and whether to sample with replacement.
          Returns a list of `n` indices
    - percentile : ([[float]], [float]) -> [float]
        - Takes a list of sample vectors of same length and a list of
          percentiles to interpolate values for based on the samples.
          Returns a list of same length as the provided percentiles with
          the corresponding values

    Arguments:
    - y_true : [float]
        - True labels
    - y_pred : [float]
        - Predicted labels
    - metric : ([float], [float]) -> float
        - Metric function
    - n_trials : int
        - Number of iterations of sampling results
    - q : int
        - q-th percentile for confidence interval
    - one-tailed : bool
        - Perform a one-tailed or two-tailed interval estimation

    Returns: (float)
    One or two confidence intervals depending on `one_tailed`,
    corresponding to a confidence interval with an alpha = `q` or `q`/2
'''
def sample_confidence_intervals(y_true, y_pred,
                                metric, n_trials=1000,
                                q=5, one_tailed=False):
    n_samples =  len(y_true)
    point_range = range(n_samples)
    bootstrapped_results = []
    for _ in range(n_trials):
      # Sample a new prediction vector of indices with replacement
      indices = sample(point_range, n_samples, replace=True)
      y_true_ = y_true[indices]
      y_pred_ = y_pred[indices]
      bootstrapped_results.append(metric(y_true_, y_pred_))
    if not one_tailed:
        q = [q/2, 100-q/2]
    return percentile(bootstrapped_results, q)
```

5 Results

Statistics of the procured data and the results from evaluating each model are necessary to measure improvement with regards to the hypotheses in section 1.5. In this chapter, some relevant statistics of each dataset are presented. Following these statistics are the results from evaluating both the baselines and the AutoBERT model variations, first with regards to the validation set, then with regards to the subsets of the validation data as described in section 3.1.2, and lastly with regards to the manually labeled test set.

5.1 Data Statistics

Basic statistics for the training- and validation sets can be found in table 5.1.

Table 5.1: Basic statistics depicting some of the differences between the training- and validation sets, based on the made decisions regarding the sampling of training- and validation data in section 3.1.1. Recall that the positive labels frequency represents the proportion of *appropriate* messages in the dataset.

Statistic	Training Set	Validation Set
Date range (DD/MM/YY)	15/04/20-13/10/20	14/10/20-01/11/20
n samples	2,382,705	264,343
% of all data	90.014%	9.986%
Positive label frequency	0.2490	0.2755

Tying back to section 3.1.2, the counts of all aspects w.r.t. uniqueness of both the training- and validation sets, as well as counts of what instances are shared between the two are shown in table 5.2.

However, this does not show the complete distribution w.r.t. each feature as none of them are weighted, i.e., we assume a uniform distribution of all features when conditioned upon, which is likely incorrect.

5.1.1 Test Dataset

Looking at the test set, which was procured in accordance with section 3.3, the resulting data from the contacted moderators are shown in table 5.3. Unfortunately, the number of datapoints is significantly smaller than originally anticipated and falls below the estimated optimal lower bound from equation (3.4) ($n = 1006$). Out of the 11 contacted moderators, only two replied before the results were produced.

When comparing with the inference procedure in equation (3.1) w.r.t. Accuracy, Sensitivity (Recall), and Specificity, it becomes clear that the inferred labels are inaccurate half of the time. Most often, this is due to the lack of identifying true positives (Sensitivity) and is also indicated by the inferred label frequency, which is about 0.25% (10) of all the labels.

These results suggest that the inference procedure is not representative of the underlying label distribution and is heavily biased towards negatives. The fact that the procedure would be inaccurate is not to be unexpected, since it assumes that moderators do not make mistakes, which is unlikely to be the case. It is also important to remember that the mindset of the moderators might have changed over time, meaning that their action at the time of the stream may be different from what they labeled for the same datapoint. Furthermore, there is the fact that the context they were given in the labeling task, i.e., the broadcast category at the time, is far from the complete context the moderators had at the time of the stream itself. Lastly, the case might also be that the moderator who interacted with the message during the stream is not the same moderator as the one who labeled the data, since some channels have multiple moderators. Nonetheless, it is interesting that the procedure is inaccurate to such a high degree.

5.2 Baseline Results

After training each baseline and saving the best-performing checkpoint, the checkpoints were evaluated once more to determine the decision threshold for optimal performance. The decision threshold was determined based on the maximal F1 Score within the range [0,1] with 0.1 step increments. The results for the BoW-,

Table 5.2: Counts of unique instances of the aspects *Category*, *Channel*, and *Message Body* within each of the training- and validation sets, as well as the intersection of shared features between the two sets. Note that there is a majority of unique values within these features of the validation set compared to what is shared with the training set.

Aspect	Training Set	Validation Set	Intersection
Category	2,999	1,135	887
Channel	17,380	7,024	5,772
Message Body	1,984,884	234,725	17,114

Table 5.3: Basic statistics of the test set with labels obtained from moderators, as described in section 3.3. Note that the "Inference positive label frequency" is the frequency of the inferred labels, whilst the "Positive Label Frequency" is the frequency for the "gold standard" moderator labels. Do also note the Accuracy, Sensitivity, and Specificity metrics which compare the inference procedure's labeling performance compared to the moderator labels. All metrics are defined in section 4.4.

	Test Set
Statistic	
Date range (DD/MM/YY)	02/11/20-22/01/21
n samples	400
Positive label frequency	0.4625
Inference Pos. label freq.	0.0250
Inference Accuracy	0.5375
Inference Sensitivity	0.0270
Inference Specificity	0.9767

SSV- and EBoW models are shown in table 5.4. In table 5.4, we can see that the SSV model performed the best with threshold = 0.2 and both the BoW- and EBoW models performed the best with the threshold = 0.3.

Table 5.4: F1 Scores on the validation set depending on a varying decision threshold with 0.1 step increments for each of the three baseline models. Note that the best-performing threshold is marked in bold. Note that for the case of Threshold=1, the F1 Score is undefined since Precision will be undefined and Recall will be zero. Summaries about the models are presented in section 4.5.1.

	BoW	SSV	EBoW
Threshold			
0.0	0.4319	0.4319	0.4319
0.1	0.5264	0.4319	0.5615
0.2	0.6790	**0.7034**	0.7231
0.3	**0.6996**	0.5906	**0.7235**
0.4	0.6747	0.4366	0.6888
0.5	0.6293	0.3561	0.6383
0.6	0.5645	0.2740	0.5651
0.7	0.4702	0.1653	0.4682
0.8	0.3452	0.0555	0.3399
0.9	0.1933	0.0176	0.1860

With the best thresholds selected, each baseline was evaluated on the validation set once more to measure all metrics defined in section 4.4. This time, however, the threshold was fixed to the value where each model performed the best, which resulted in the metrics found in tables 5.5a to 5.5c. In these tables, we can see that the overall best-performing model is the EBoW baseline model, meaning that both messages and severity score vectors play a significant role for predictor performance.

Table 5.5: Accuracy, Precision, Recall, F1 Scores, and Specificity scores when evaluated on the validation set for the inferred AutoMod predictions, the dummy classifier, the three baselines SSV, BoW, and EBoW, and the three AutoBERT model variations Base, Stream, and IGDB at their respective best decision thresholds. All metrics are calculated with 95% confidence intervals. The best scoring baseline and AutoBERT model is marked in bold. Summaries about the models are presented in section 4.5 and the metrics are defined in section 4.4.

(a) Accuracy scores of the inferred AutoMod labels, dummy classifier, baseline models, and AutoBERT models.

	Accuracy
Model	
Dummy	0.7245
AutoMod	0.1104
SSV (0.2)	0.8120 (±0.0014)
BoW (0.3)	0.8292 (±0.0015)
EBoW (0.3)	**0.8430 (±0.0014)**
Base (0.3)	0.8286 (±0.0014)
Stream (0.4)	**0.8743 (±0.0013)**
IGDB (0.2)	0.8623 (±0.0013)

(b) Specificity scores of the inferred AutoMod labels, dummy classifier, baseline models, and AutoBERT models.

	Specificity
Model	
Dummy	1
AutoMod	0.1524
SSV (0.2)	0.8131 (±0.0018)
BoW (0.3)	0.8699 (±0.0016)
EBoW (0.3)	**0.8799 (±0.0015)**
Base (0.3)	0.8302 (±0.0017)
Stream (0.4)	**0.8943 (±0.0014)**
IGDB (0.2)	0.8701 (±0.0015)

(c) Precision, Recall, and F1 Scores of the inferred AutoMod labels, dummy classifier, baseline models, and AutoBERT models. Note that for the case of the dummy classifier and AutoMod, some Precision and F1 Scores are undefined due to zeros in the denominator.

	Precision	Recall	F1 Score
Model			
Dummy	–	0	–
AutoMod	0	0	–
SSV (0.2)	0.6220 (±0.0032)	**0.8091 (±0.0030)**	0.7034 (±0.0025)
BoW (0.3)	0.6784 (±0.0035)	0.7221 (±0.0031)	0.6996 (±0.0026)
EBoW (0.3)	**0.7024 (±0.0033)**	0.7458 (±0.0031)	**0.7235 (±0.0026)**
Base (0.3)	0.6486 (±0.0031)	0.8241 (±0.0027)	0.7259 (±0.0024)
Stream (0.4)	**0.7472 (±0.0031)**	0.8217 (±0.0028)	**0.7827 (±0.0021)**
IGDB (0.2)	0.7113 (±0.0032)	**0.8418 (±0.0025)**	0.7711 (±0.0024)

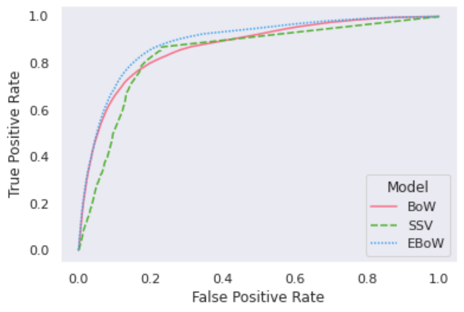
Lastly, in order to determine the performance regardless of the decision threshold, both ROC curves and Precision-Recall curves are drawn along with their respective area under curve (AUC) scores. The ROC- and Precision-Recall curves for all three

Table 5.6: AUROC and AUPR scores for the three baseline models and the three AutoBERT models with 95% confidence intervals when evaluated on the validation set. The best scoring baseline and AutoBERT model is marked in bold. Summaries about the models are presented in section 4.5.

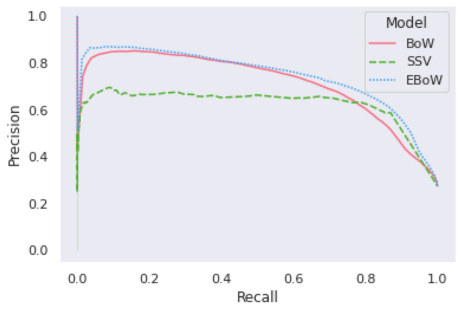
Model	AUROC	AUPR
SSV (0.2)	0.8408 (±0.0014)	0.6467 (±0.0051)
BoW (0.3)	0.8680 (±0.0016)	0.6833 (±0.0027)
EBoW (0.3)	**0.8912 (±0.0014)**	**0.7080 (±0.0033)**
Base (0.3)	0.8819 (±0.0015)	0.7434 (±0.0033)
Stream (0.4)	**0.9349 (±0.0011)**	**0.8374 (±0.0023)**
IGDB (0.2)	0.9272 (±0.0011)	0.8239 (±0.0024)

baselines can be found in figures 5.1a and 5.1b and their respective AUC values are listed in table 5.6.

Overall, from the results shown in figure 5.1 and tables 5.5 and 5.6, it becomes clear that the EBoW baseline performs the best, meaning that both the messages and severity score vectors have a significant effect on how to determine the appropriacy of message contents. Out of all metrics, only Recall is the metric where EBoW performs worse than any baseline, where it is beaten by SSV. Therefore, as the EBoW performed best overall, this baseline will be the one which the AutoBERT models are compared against, given that the EBoW model is beating the current AutoMod predictions on the data and a simple majority vote.



(a) ROC curves for the three baseline models.



(b) Precision-Recall curves for the three baseline models.

Figure 5.1: ROC- and Precision-Recall curves for the baseline models when evaluated on the validation set. Summaries about the models are presented in section 4.5.1

5.2.1 Inferred AutoMod Performance and Baseline Performance

In order to evaluate whether or not the baselines are performing well enough to be compared against, their non-threshold dependent metrics were compared against the inferred AutoMod label metrics as well as the majority vote of labels (dummy classifier) in the training set. As shown in table 5.1, the *positive label frequency* is less than 0.5, meaning that the majority vote of both sets is **False**. As before, the results are listed in tables 5.5a to 5.5c.

Comparing the inferred labels from AutoMod with the baselines, as well as the dummy classifier, it is shown that the baselines do beat the dummy classifier w.r.t. Accuracy and that the inferred AutoMod performance being the worst overall. As shown in table 5.5b, the dummy classifier is best w.r.t. Specificity. However, since the dummy classifier classifies all datapoints as inappropriate, this metric is not relevant. Furthermore, with both the inferred AutoMod- and dummy classifier performance in table 5.5c being zero or undefined, there is stronger indication that the baselines are representative for comparing the AutoBERT models against. Do also note, however, that the poor performance of the inferred AutoMod scores is due to the fact that all datapoints are from cases where AutoMod has been determined to act poorly, as described in section 3.1.

With the baselines having been confirmed to be representative for comparing the AutoBERT models against, the following step is the evaluation of the AutoBERT model variations.

5.3 AutoBERT Comparison to Baseline

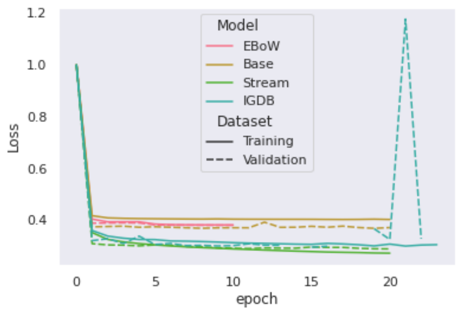
When evaluating the AutoBERT model variations, the evaluation should be done with regards to the hypotheses specified in section 1.5. In order to highlight the tradeoff between gained performance and introduced time and complexity from each layer (hypothesis 3), loss curves over both epochs and hours are shown in figures 5.2a and 5.2b.

From these curves it is shown that the Base variation performs similarly w.r.t. loss as the EBow baseline, indicating that not much more information could be derived from the messages and severity score vectors alone even though the model is more complex. With stream metadata introduced, the lowest loss on the validation set is reached in the Stream variation, indicating the best performance in terms of loss. Similar performance, but not as great, is reached with the combined stream metadata and IGDB data in the IGDB variation.

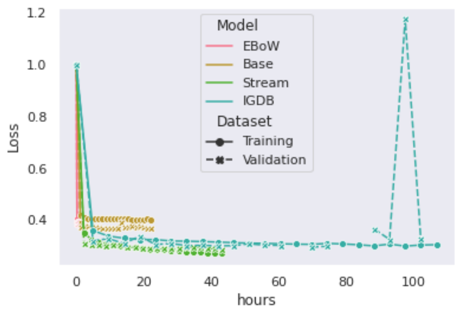
Convergence also takes the same number of epochs for all models, reaching a somewhat steady loss after one epoch. This is reasonable, however, since the training dataset is quite large. The loss then tends to stagnate and fluctuate around the same point for the remainder of the training time, except for the Stream and IGDB variations. Instead, both of these tend to slowly but surely decrease their training loss further, with the validation loss fluctuating somewhat during the time.

Furthermore, the training times themselves increase drastically for each additional layer of information. Do also recall that the AutoBERT models are trained on four GPUs in parallel, whilst the baseline is trained on a single GPU. Therefore, given the same resources as the baselines, the training times for the AutoBERT models should be about four times as long, with some potential decrease due to reduced synchronization overhead from parallel computation.

Lastly, note that the validation loss curve for the IGDB variation lacks some points around epoch 14 and onwards. This is due to the validation loss suddenly hitting ∞ at epoch 14 and then *undefined* and $-\infty$ in the 17th and 18th epochs. Due to the 18th epoch hitting $-\infty$, this checkpoint was saved as the best model for the IGDB variation. With the hopes of getting back to around the best found loss up until then, the best validation loss was reset and the training was continued for five more epochs to see if the model would recuperate, hence the additional three epochs from the specified number in listing 4. Therefore, the IGDB model checkpoint is unfortunately not the best one found during training, but rather the one from epoch 20.



(a) Loss curves over epochs. Do note the gaps of validation loss for the IGDB variation between epochs 14 and 18 and the lack of loss for epoch 23. This is due to the validation loss becoming either *undefined* or $\pm\infty$. Also note that the EBoW loss curve is included for comparison (ending at 10 epochs). Non-dashed lines are for loss on the training set and the dashed lines are for loss on the validation set.



(b) Loss curves over hours. Note that the EBoW loss curve is also included for comparison in training times. Do also note the markers which indicate the start and end of adjacent epochs. Non-dashed lines are for loss on the training set and the dashed lines are for loss on the validation set.

Figure 5.2: Loss curves of the best baseline model (EBoW) and the AutoBERT model variations with (1) the same input as EBoW (Base), (2) the added stream metadata information (Stream), and (3) the combined stream metadata and IGDB games metadata (IGDB). Each AutoBERT variation was trained for 20 epochs, except for the IGDB variation which was trained for 23 epochs. Do also recall that each AutoBERT instance is trained on four GPUs in parallel, whilst the baselines only trained on a single GPU. Summaries about the models are presented in section 4.5.

5.3.1 Evaluating Decision Thresholds

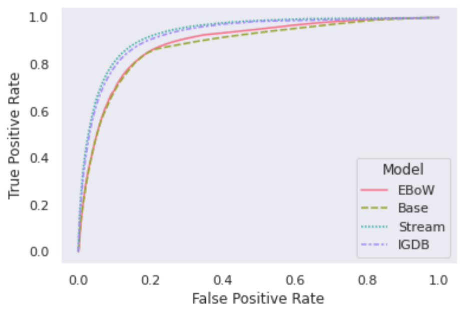
Regarding the remaining hypotheses in section 1.5 (hypothesis 1 and 2), the AutoBERT model variations need to be evaluated. As with the the baseline models, the best decision threshold is decided on in the threshold range [0,1] with 0.1 step increments and selecting the threshold which yields the highest F1 Score. In table 5.7, the results are listed along with the EBoW results from table 5.4. Interestingly, the Base model has both the same decision threshold and a similar F1 Score (with a difference of 0.0024).

Furthermore, the Stream model has the highest F1 Score overall and the decision threshold which lies closest to 0.5, indicating that the decision regarding appropriacy is more balanced when also depending on stream metadata. Conversely, the decision threshold for the IGDB model is the lowest, along with the SSV baseline in table 5.4. However, the IGDB model also has the second highest F1 Score with a 0.0116 lower score compared to the Stream model.

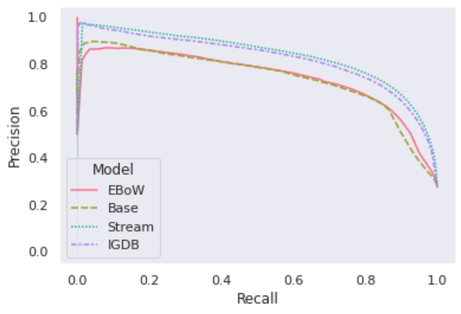
Table 5.7: F1 Scores on the validation set depending on a varying decision threshold with 0.1 step increments for each of the three AutoBERT model variations and a comparison with the best-performing baseline (EBoW) model. Note that the best-performing threshold is marked in bold. Also note that for the case of Threshold=1, the F1 Score is undefined since Precision will be undefined and Recall will be zero. Summaries about the models are presented in section 4.5 and the metric is defined in section 4.4.

	EBoW	Base	Stream	IGDB
Threshold				
0.0	0.4319	0.4319	0.4319	0.4319
0.1	0.5615	0.6435	0.7106	0.7474
0.2	0.7231	0.7176	0.7601	**0.7711**
0.3	**0.7235**	**0.7259**	0.7793	0.7677
0.4	0.6888	0.7238	**0.7827**	0.7511
0.5	0.6383	0.6924	0.7741	0.7183
0.6	0.5651	0.6358	0.7410	0.6573
0.7	0.4682	0.5305	0.6806	0.5513
0.8	0.3399	0.3210	0.5677	0.3783
0.9	0.1860	0.0349	0.3432	0.1348

The results for the decided upon thresholds can be along with the baseline results in tables 5.5 and 5.6. Furthermore, the ROC- and Precision-Recall curves for each AutoMod model variation is illustrated in figure 5.3. From these results, it becomes clear that the EBoW baseline and AutoBERT Base perform similarly, with EBoW having higher Recall and Base having higher Precision, yielding about the same F1 Score. EBoW is more accurate, has higher Specificity and AUROC, but loses in AUPR. When looking at the ROC and Precision-Recall curves in figure A.1, both the EBoW and Base curves are close to each other in both cases. Overall, this shows that with only messages and severity score vectors, the EBoW model is enough to



(a) ROC curves for the three AutoBERT models.



(b) Precision-Recall curves for the three AutoBERT models.

Figure 5.3: ROC- and Precision-Recall curves for the AutoBERT models when evaluated on the validation set. Summaries about the models are presented in section 4.5.2.

recognize the relevant information within these features.

When looking at the Stream and IGDB variations of the AutoBERT model, the story is different. The best-performing model overall is the Stream variation, with a validation accuracy higher than all other models to a significant degree; the IGDB variation being a close second with about 1.2 percent units less accuracy. This pattern seems to repeat itself across all metrics, except for Recall, where the IGDB model performs the best. However, this is compensated for by the Stream model with its higher Precision, leading to a higher F1 Score in the end. As for AUROC and AUPR, Stream beats IGDB with about 1 percent unit in both cases, AUROC being a bit less and AUPR a bit more. This is also highlighted in figure 5.3, where the Stream curves are approximately above the IGDB curve at all points. All in all, this shows that the introduction of more contextual information proves to yield statistically significant performance improvements across the board.

5.4 Investigation of Robustness and Generalization Regarding Unseen Data

When also looking at investigating each model's robustness and ability to generalize in terms of unseen values for different aspects, as described in section 3.1.2, we can get another view on how well a model performs in general in conjunction with a test set. In the following subsections, the results for the three aspects (1) channel ID, (2) broadcast category, and (3) message body are presented; both for the baselines and the AutoBERT model variations. Similarly to how the performance has been measured using the metrics defined in section 4.4, the results of this investigation will be represented by a subset; namely the metrics Precision, Recall, F1 Score, and Specificity. These are chosen as they are more indicative of how well the models' will perform in practice with regards to unseen data. Results for all of these metrics are instead presented in appendix B.

Table 5.8: Precision, Recall, F1 Scores, and Specificity of the baseline- and AutoBERT models w.r.t. common and unique channels of the validation set with 95% confidence intervals. Summaries about the models are presented in section 4.5 and the metrics are defined in section 4.4.

(a) Common channels of the validation set, i.e., channels present in the training set.

	Precision	Recall	F1 Score	Specificity
Model				
SSV	0.6192 (±0.0033)	0.8077 (±0.0030)	0.7010 (±0.0027)	0.8166 (±0.0018)
BoW	0.6770 (±0.0036)	0.7242 (±0.0035)	0.6998 (±0.0028)	0.8724 (±0.0015)
EBoW	0.7012 (±0.0033)	0.7484 (±0.0032)	0.7240 (±0.0028)	0.8823 (±0.0015)
Base	0.6464 (±0.0034)	0.8239 (±0.0029)	0.7245 (±0.0024)	0.8336 (±0.0018)
Stream	0.7477 (±0.0033)	0.8247 (±0.0030)	0.7843 (±0.0025)	0.8973 (±0.0015)
IGDB	0.7113 (±0.0033)	0.8430 (±0.0028)	0.7716 (±0.0024)	0.8737 (±0.0016)

(b) Unique channels of the validation set, i.e., channels not present in the training set.

	Precision	Recall	F1 Score	Specificity
Model				
SSV	0.6625 (±0.0126)	0.8285 (±0.0110)	0.7363 (±0.0091)	0.7318 (±0.0096)
BoW	0.6994 (±0.0144)	0.6933 (±0.0127)	0.6963 (±0.0105)	0.8106 (±0.0085)
EBoW	0.7203 (±0.0135)	0.7112 (±0.0119)	0.7157 (±0.0096)	0.8244 (±0.0084)
Base	0.6792 (±0.0114)	0.8265 (±0.0104)	0.7457 (±0.0089)	0.7519 (±0.0099)
Stream	0.7398 (±0.0125)	0.7818 (±0.0127)	0.7602 (±0.0090)	0.8252 (±0.0081)
IGDB	0.7115 (±0.0123)	0.8255 (±0.0107)	0.7643 (±0.0091)	0.7873 (±0.0095)

5.4.1 Conditioning on Channels

Investigating the models w.r.t. channels involves conditioning the training set on channel IDs and evaluating the models on the two subsets where channel IDs are unique to the validation set and where they are shared with the training set. Below, the results are presented in table 5.8 and they are analyzed in the following subsections, first for the baselines and then for the AutoBERT model variations.

5.4.1.1 Baselines

In table 5.8, with table 5.8a being the results for the channels which are shared with the training set and table 5.8b being the results for the channels which are unique to the validation set, the EBoW baseline performs best overall regarding performance changes on the two datasets. The exception, however, being for the Recall in both subsets and the F1 Score in the unique channels subset, where the SSV model performs the best. Do also note that the confidence intervals in table 5.8b is about three to four times larger than in table 5.8a, indicating that there is a difference in sample size for the two sets or that the common subset contains more varied data.

Furthermore, the SSV seems to perform better on the unique channels w.r.t. Precision, Recall and F1 Score compared to the common channels. Why this is the case is hard to to reason about. One potential reason could be that AutoMod's internal model's configuration has not changed for a time – configuration changes are mentioned in section 3.1. If the current configuration is well-aligned with what the SSV model has learned to interpret as appropriate versus inappropriate in a more generalized extent, then this could definitely be the case.

5.4.1.2 AutoBERT Models

When instead looking at the AutoBERT models, the Stream variation seems to be the best model overall. However, the IGDB variation is not far behind and beats the Stream variation in Recall on both subsets and in F1 Score on the unique subset. The F1 Score does not seem statistically significant, nonetheless.

Furthermore, the Base variation is best overall in Recall on the unique subset and even improves its Precision and thus also the F1 Score on this set. This comes at the cost of a lower Specificity compared to the other models, however.

Overall, it seems that all AutoBERT models seem to not degrade in Precision when given unseen messages from unseen channels. Furthermore, the Recall and F1 Scores tend to change similarly to how the baselines changed – some increase a bit and some instead decrease within a range of [0,2] percent units. Specificity also seems to drop significantly when going from already-seen channels to unseen ones, much like with the baselines. For AutoBERT, however, the drop is about as significant as for the SSV model, which dropped the most among the baselines.

All in all, when it comes to predicting appropriate language, the performance of all AutoBERT variations tend to adapt quite well when working on unseen channels, but with a large enough gap to seem statistically significant. This could be due to the fact that the stream metadata, e.g., stream titles, tie channels together in means that the plain messages could not, which the Stream and IGDB models then were able to learn to identify and thus generalizes better. As for predicting inappropriate language, the generalization tends to be worse, with a drop being two to four times as large as for predicting appropriate language when facing unseen channels.

5.4.2 Conditioning on Broadcast Categories

Investigating the models w.r.t. broadcast categories involves conditioning the training set on the broadcast category and evaluating the models on the two subsets where the categories are unique to the validation set and where they are shared with the training set. Below, the results are presented in table 5.9 and they are analyzed in the following subsections, first for the baselines and then for the AutoBERT model variations.

Table 5.9: Precision, Recall, F1 Scores, and Specificity of the baseline- and AutoBERT models w.r.t. common and unique broadcast categories of the validation set with 95% confidence intervals. Summaries about the models are presented in section 4.5 and the metrics are defined in section 4.4.

(a) Common broadcast categories of the validation set, i.e., broadcast categories present in the training set.

	Precision	Recall	F1 Score	Specificity
Model				
SSV	0.6182 (±0.0032)	0.8090 (±0.0029)	0.7008 (±0.0025)	0.8132 (±0.0018)
BoW	0.6753 (±0.0035)	0.7223 (±0.0036)	0.6980 (±0.0028)	0.8702 (±0.0015)
EBoW	0.6998 (±0.0035)	0.7463 (±0.0032)	0.7223 (±0.0025)	0.8803 (±0.0015)
Base	0.6439 (±0.0031)	0.8235 (±0.0029)	0.7227 (±0.0025)	0.8312 (±0.0018)
Stream	0.7450 (±0.0033)	0.8214 (±0.0030)	0.7814 (±0.0024)	0.8958 (±0.0015)
IGDB	0.7091 (±0.0030)	0.8411 (±0.0029)	0.7695 (±0.0025)	0.8721 (±0.0015)

(b) Unique broadcast categories of the validation set, i.e., broadcast categories not present in the training set.

	Precision	Recall	F1 Score	Specificity
Model				
SSV	0.7873 (±0.0194)	0.8134 (±0.0187)	0.8001 (±0.0148)	0.8184 (±0.0160)
BoW	0.8088 (±0.0205)	0.7393 (±0.0217)	0.7725 (±0.0161)	0.8556 (±0.0151)
EBoW	0.8284 (±0.0182)	0.7636 (±0.0198)	0.7947 (±0.0147)	0.8693 (±0.0149)
Base	0.8375 (±0.0207)	0.8504 (±0.0194)	0.8439 (±0.0152)	0.8355 (±0.0213)
Stream	0.8539 (±0.0180)	0.8745 (±0.0172)	0.8640 (±0.0140)	0.8508 (±0.0181)
IGDB	0.8366 (±0.0206)	0.8818 (±0.0164)	0.8586 (±0.0147)	0.8282 (±0.0189)

5.4.2.1 Baselines

In table 5.9, with table 5.9a being the results for the broadcast categories which are shared with the training set and table 5.9b being the results for the broadcast categories which are unique to the validation set, the EBoW model performs the best overall. Much like for the prior case, Recall in both subsets and F1 Scores in the common subset are the exceptions and SSV is the best-performing model once again for these metrics.

What is different here, however, is that the performance drops between the common and unique sets are smaller and all models perform better on unique categories w.r.t. Precision, Recall, and F1 Score. However, the improvements do not seem to be statistically significant in the case of Recall. In Specificity, SSV also improves on the unique categories compared to the common ones, though not enough to be statistically significant.

These results indicate a positive generalization with regards to categories, meaning that the baselines tend to be independent of what the current broadcast category when the messages occur. This can be expected, nonetheless, since the models do not depend on any such information. Conversely, the case could also be that the appropriacy of messages itself is independent of broadcast categories. If this is the case, then the performance of AutoBERT Stream and IGDB should not improve due to the category information.

5.4.2.2 AutoBERT Models

Much like for the baselines, all AutoBERT models tend to improve in Precision, Recall and F1 Score, yet also in Specificity. This seems to reaffirm the hypothesis that the appropriacy of messages should be independent of broadcast categories. If not, then the IGDB model should have a performance change that is greater than the Stream model since the added information there revolves around giving additional metadata about the broadcast categories themselves. However, this is not the case.

Furthermore, Precision, Recall, and F1 Scores tend to be significantly higher than for the baselines across the two subsets, showing once more that the AutoBERT models tend to improve in overall performance by being able to reduce the number of false negatives, i.e., correctly identifying appropriate language. However, the Specificity tends to remain similar to the Specificity of the baselines, meaning that the models' ability to identify inappropriate language tends to be unaffected when regarding old and new broadcast categories.

5.4.3 Conditioning on Messages

Investigating the models w.r.t. messages involves conditioning the training set on message bodies and evaluating the models on the two subsets where the messages are unique to the validation set and where they are shared with the training set. Below, the results are presented in table 5.10 and they are analyzed in the following subsections, first for the baselines and then for the AutoBERT model variations.

5.4.3.1 Baselines

In table 5.10, with table 5.10a being the results for the message bodies which are shared with the training set and table 5.10b being the results for the message bodies which are unique to the validation set, the EBoW model performs the best overall, once more. For the messages, SSV only outperforms the EBoW model with Recall on both subsets. Furthermore, the Precision of all baselines seem to have taken a significant hit as they are all below 0.6 in the common subset.

Interestingly, all baselines also seem to perform better on the unique subset regarding Precision and, consequently, F1 Score. The confidence intervals for these metrics are similar to the inverse case as for the channels in table 5.8, which could be an indication of more varied data in the unique dataset, i.e., that the messages are much more distinctive than in the common subsets.

Lastly, the Recall remains somewhat unchanged between the two subsets, whilst the Specificity drops significantly for both BoW and EBoW when evaluated on the unique subset. This behavior is interesting, as it shows that the baselines tend to be able to recall what is appropriate regardless of seen/unseen messages. However, both BoW and EBoW have a harder time to be specific about what is inappropriate on when faced with unseen messages.

5.4.3.2 AutoBERT Models

Looking at the performance of the AutoBERT variations, it becomes clear that additional contextual information helps the model to identify appropriate language better as the Precision and F1 Scores are significantly higher in the Stream and IGDB cases compared to the Base case. The Recall, however, is not higher than for the Base variation, which could indicate that the additional information helps more in defining what is appropriate language (Precision) rather than what messages themselves are appropriate (Recall).

Furthermore, the Specificity drops significantly for each AutoBERT variation, much

Table 5.10: Precision, Recall, F1 Scores, and Specificity of the baseline- and AutoBERT models w.r.t. common and unique messages of the validation set with 95% confidence intervals. Summaries about the models are presented in section 4.5 and the metrics are defined in section 4.4.

(a) Common messages of the validation set, i.e., messages present in the training set.

Model	Precision	Recall	F1 Score	Specificity
SSV	0.4189 (±0.0092)	0.8106 (±0.0107)	0.5524 (±0.0098)	0.8170 (±0.0041)
BoW	0.5898 (±0.0116)	0.7385 (±0.0112)	0.6558 (±0.0099)	0.9164 (±0.0029)
EBoW	0.5771 (±0.0120)	0.7495 (±0.0122)	0.6521 (±0.0090)	0.9106 (±0.0031)
Base	0.5163 (±0.0113)	0.8378 (±0.0098)	0.6389 (±0.0095)	0.8722 (±0.0035)
Stream	0.7083 (±0.0107)	0.7957 (±0.0101)	0.7495 (±0.0084)	0.9467 (±0.0024)
IGDB	0.6637 (±0.0114)	0.8051 (±0.0101)	0.7276 (±0.0090)	0.9336 (±0.0027)

(b) Unique messages of the validation set, i.e., messages not present in the training set.

Model	Precision	Recall	F1 Score	Specificity
SSV	0.6485 (±0.0031)	0.8090 (±0.0031)	0.7199 (±0.0027)	0.8122 (±0.0019)
BoW	0.6873 (±0.0035)	0.7207 (±0.0033)	0.7036 (±0.0028)	0.8596 (±0.0018)
EBoW	0.7155 (±0.0035)	0.7455 (±0.0033)	0.7302 (±0.0027)	0.8731 (±0.0017)
Base	0.6631 (±0.0033)	0.8230 (±0.0031)	0.7344 (±0.0024)	0.8210 (±0.0020)
Stream	0.7505 (±0.0034)	0.8239 (±0.0028)	0.7855 (±0.0024)	0.8827 (±0.0016)
IGDB	0.7154 (±0.0030)	0.8449 (±0.0030)	0.7748 (±0.0025)	0.8561 (±0.0018)

like in the broadcast category-case. Interestingly enough, the metrics for the unique subset in table 5.10b align nicely with the overall scores for the AutoBERT variations in tables 5.5b and 5.5c, where the scores in table 5.10a instead are significantly higher.

5.5 Final Evaluation on Test Set

From performing the final evaluation on the test set procured as described in section 3.3, the results are presented in tables 5.11 and 5.12. Interestingly, on the test set, the IGDB variation is the best-performing model overall; compared to the validation set results in tables 5.5 and 5.6, where the Stream variation was the overall best model.

Another important note is the drastic performance drop the models have compared

Table 5.11: Accuracy, Precision, Recall, F1 Scores, and Specificity scores when evaluated on the test set for the three baselines SSV, BoW, and EBoW, and the three AutoBERT model variations Base, Stream, and IGDB at their respective best decision thresholds. All metrics are calculated with 95% confidence intervals. The best scoring baseline and AutoBERT model is marked in bold. Summaries about the models are presented in section 4.5 and the metrics are defined in section 4.4.

(a) Accuracy scores of the baseline- and AutoBERT models.

Model	Accuracy
SSV	0.4850 (±0.0525)
BoW	0.5000 (±0.0500)
EBoW	**0.5175 (±0.0476)**
Base	0.4950 (±0.0500)
Stream	0.5250 (±0.0500)
IGDB	**0.5775 (±0.0525)**

(b) Specificity scores of the baseline- and AutoBERT models.

Model	Specificity
SSV	0.8651 (±0.0495)
BoW	0.8279 (±0.0517)
EBoW	**0.8744 (±0.0441)**
Base	0.8744 (±0.0452)
Stream	**0.8791 (±0.0437)**
IGDB	0.8419 (±0.0524)

(c) Precision, Recall, and F1 Scores of the baseline- and AutoBERT models.

Model	Precision	Recall	F1 Score
SSV	0.2162 (±0.1352)	0.0432 (±0.0306)	0.0721 (±0.0506)
BoW	0.3729 (±0.1271)	**0.1189 (±0.0496)**	**0.1803 (±0.0648)**
EBoW	**0.4130 (±0.1426)**	0.1027 (±0.0455)	0.1645 (±0.0667)
Base	0.2703 (±0.1415)	0.0541 (±0.0350)	0.0901 (±0.0510)
Stream	0.4468 (±0.1490)	0.1135 (±0.0469)	0.1810 (±0.0670)
IGDB	**0.5952 (±0.1153)**	**0.2703 (±0.0651)**	**0.3717 (±0.0783)**

to the validation set, apart from the Specificity which remains relatively unaffected. As mentioned in section 5.1.1, however, the inference procedure tends to be inaccurate and favors negatives over positives, which is shown in both table 5.3 by the imbalanced inference positive label frequency compared to the true (moderator) positive label frequency.

Furthermore, the Stream variation does not appear to be significantly better than the EBoW baseline. The BoW baseline also appears to be evenly matched with the EBoW when looking at the confidence intervals, which could indicate that the severity scores have little to no significant effect on classifying the appropriacy of chat messages.

Table 5.12: AUROC and AUPR scores for the three baseline models and the three AutoBERT models with 95% confidence intervals when evaluated on the test set. The best scoring baseline and AutoBERT model is marked in bold. Summaries about the models are presented in section 4.5.

	AUROC	AUPR
Model		
SSV	0.4384 (±0.0337)	0.3420 (±0.0754)
BoW	**0.4454 (±0.0696)**	0.4233 (±0.0681)
EBoW	0.4450 (±0.0647)	**0.4329 (±0.0681)**
Base	0.4799 (±0.0688)	0.4313 (±0.0693)
Stream	0.5237 (±0.0522)	0.4733 (±0.0654)
IGDB	**0.6288 (±0.0567)**	**0.5490 (±0.0832)**

In the case of AUROC and AUPR, both the Stream- and IGDB variations are significantly better than the baselines w.r.t. AUROC, as shown in table 5.12. However, none of the two are significantly better than the EBoW- or BoW models w.r.t. AUPR.

Nonetheless, as mentioned in section 5.1.1, the test data was obtained from only two moderators, meaning that the messages come from only two channels. By only having data from two channels, there is a risk of the messages being skewed and are likely not representative of the whole distribution of messages which are encompassed by the specified delimitations in section 1.4.

6 Discussion

Using the results presented in the previous chapter, this chapter further reflects on the meaning of the results and further tries to explain some of the underlying causes of them. It then goes into talking about the limiting factors of the work done in this thesis. Lastly, this chapter is rounded off with a set of propositions for future work branching off of the findings in this work.

6.1 Performance Differences Between AutoBERT Stream- and IGDB Configurations

When looking at the validation results of all models in tables 5.5 and 5.6, the performance of the Stream model is slightly better than the IGDB model in all cases, and to a significant degree. This shows that more information is not always good information when it comes to having the model learn.

Nonetheless, plainly based on these results, the benefits of an additional external data layer cannot be ruled out. For example, when looking at the final evaluation results on the test data in tables 5.11 and 5.12, the IGDB variation is instead the model which performs significantly better compared to the baselines in the metrics where there is a statistical significance at all. As previously mentioned in section 5.1.1, only two moderators out of the contacted 11 responded before the final evaluation on the test set was executed; therefore, the test results may be skewed as the test set is likely not representative of the distribution of messages coming from all channels where the delimitations in section 1.4 are fulfilled.

One potential explanation of the IGDB model performing similarly to or worse than the Stream model could be due to data gaps in IGDB data, causing either tokenization to yield empty or static token strings for all entries which have the same gaps. What the vector will be depends on how the tokenizer and sentence transformer interpret the data gaps. If the tokenizer does not recognize anything in the gap, e.g., due to no words in the vocabulary being in the default string, then a null encoding is returned. Furthermore, depending on how the sentence transformer interprets each encoding, it will either return a null- or static vector for the same encoding since it is a deterministic mapping. In turn, this means that null- or static vectors are created for the sentences that are empty or static. In the case of static vectors, this may cause the mean pooling of the embedded sentences to skew each batch during normalization as the data gaps are encoded to a specific point in the embedding space rather than to origin.

Therefore, in hindsight, having good default strings could more or less be confusing to the model rather than helping after these are embedded. Instead, ensuring that the data gaps are always encoded as null vectors could at least help in more stable embeddings based on the information that is available rather than the information that is not.

Another potential improvement would be introducing some form of data imputation. If such a process could be performed on the data before tokenization, the gaps themselves would be excluded from the tokenized strings and this potential issue could be avoided altogether. Nonetheless, good imputation methods require multiple estimates to yield good standard errors and confidence intervals [37] and simpler methods such as *overall mean imputation* and the *missing-indicator method* tend to always yield biased estimates [37], which might result in similar performance as the good default strings used in this project.

One other reason for the IGDB variation not performing better than the Stream model could be due to the classification component not containing enough neurons to understand and learn from the added information. Therefore, adding either additional layers or more neurons in the existing ones could help. This was the case initially, where only a linear layer was used on the RoBERTa embeddings, much like with the baselines. In order see further improvements, the four hidden layers shown in figure A.1 had to be added before the Stream variation began learning more. Do note however, that layers were added two at a time with the two higher dimensional layers first, followed by the lower dimensional layers after. Therefore, the configuration of $(300, 300, 72, 72)$ is likely not the optimal configuration.

6.1.1 Complexity- and Performance Tradeoffs Between Layers of Information

Tying back to hypothesis 3 in section 1.5, it becomes clear from figure 5.2b that each added layer of information significantly increases the training time of AutoBERT. Looking at the trends and with the implementation details in mind, the number of hours it took to train the each AutoBERT model tended to be strongly correlated with the number of tokenized sentences had to be constructed for each datapoint. When also looking at the prediction times for each AutoBERT variation, measured from the average forward pass time using the first batch of the test set for 20 iterations, the times for each AutoBERT model are as follows:

- Base – 2.565 seconds per batch (51.3 seconds over 20 iterations)
- Stream – 5.95 seconds per batch (119 seconds over 20 iterations)
- IGDB – 9.55 seconds per batch (191 seconds over 20 iterations)

With the Base variation, only messages were tokenized in a single 128 elements long token sentence. As additional stream metadata were introduced, both messages and the metadata itself were tokenized as two separate token sentences; one for the arbitrarily long chat messages and one for the metadata listed in section 3.1. As IGDB data were introduced as well, the inputs grew to four tokenized sentences per datapoint instead; the two added token sentences being one for the arbitrarily long game summaries and the other for the remaining metadata listed in section 3.2. The two arbitrarily long data types were decided to be given their separate token sentences in order to mitigate the risk of truncating information, since the used tokenizer truncates sentences longer than 128 tokens.

With each added token sentence per datapoint, about one additional hour of computation time was added per epoch, showing a linear growth in time complexity with regards to token sentences. This also indicates that the bulk of the time spent training is spent on RoBERTa embedding the inputs. From prediction times above, the linear time complexity of the number of token sentences is also indicated, since the Base variation is about twice as fast per batch as the Stream variation, which in turn is about twice as fast as the IGDB variation. Since the architecture of the classification component for each AutoBERT model has remained the same, this further strengthens the argument that RoBERTa is the bottleneck.

One could argue that the tokenized sentences could be pre-tokenized and use the tokenized output as the string-based inputs instead of the raw strings. Nonetheless, as shown in figure 4.3, all tokenized sentences are stored in a shared memory cache such that the raw data of each dataset is only traversed in the first epoch. After that, the shared memory cache is used instead in order to save time loading in- and preprocessing data during training, which allows the complete dataset to be loaded in less than three minutes with multiple data loaders. More on how the shared memory cache works and how it interacts with the dataset and data loading can be found in section 2.5.

Another alternative is to strip out the RoBERTa model from AutoBERT and embed all sentences upfront. This would result in AutoBERT taking the embeddings as the direct input instead of the tokenized strings, which will likely speed up the model significantly since the embedding transformations are the most expensive operation. With all sentences embedded beforehand, each model which depend on multiple embedded sentences would then start with the mean pooling step shown in figure 4.2b to merge the embeddings into a single embedding instead.

However, the embeddings take up much space; about 600 embedded sentences for the AutoBERT instance which depends on additional stream metadata occupied about 1.5GB of memory on disk, meaning that embedding all sentences of the data would take up approximately

$$1.5 \cdot \frac{2,647,048}{600} = 6,617.62 \approx 6,617\text{GB},$$

i.e., 6.6TB of data. Mixing in the embeddings for all IGDB data as well would mean twice the memory, i.e., 12.2TB of data, since two more sentences are produced for each datapoint. Keeping all embeddings in memory will practically infeasible and prefetching from disk adds on additional layers of complexity with regards to implementation, which is preferrably avoided.

Nonetheless, it would be interesting to see how an increase in both space- and implementation complexity would decrease the time complexity. If training times would be reduced significantly, then the forward pass would likely be faster as well, meaning that the model could potentially be productionized for real-time purposes, such as Twitch chat. The main issue remains, however – the chat messages along with the contextual information needs to be embedded in real-time as well.

6.2 Sudden Undefined and Positive/Negative Infinite Loss of AutoBERT IGDB Configuration

During the validation period of epoch 14 when training AutoBERT IGDB, the validation loss suddenly hit ∞ due to multiple batches being computed as such. Then, in epochs 17 and 23, the validation loss instead hit NaN, first due to two batches failing and second due to multiple batches failing. Lastly, in the following epoch, the validation loss instead hit $-\infty$, this time again due to two batches failing. Due to the fact that the loss is calculated on a batch basis, it is impossible to say whether or not the issue was caused by singular predictions or if there were multiple failing predictions in each batch. Since the used loss is PyTorch's *binary cross entropy with logits*-loss (BCELog), which is supposed to be more stable than regular sigmoid activation followed by binary cross entropy loss (BCE loss) thanks to the log-sum-exp trick [38], the likelihood of this behavior occurring should be mitigated.

The primary factor that could cause these erroneous predictions is a combination of severe misclassification of the model, e.g., the model predicting close to zero for a problem with label=1, and the fact that the model uses 16-bit floating point precision. By having 16-bit floating point precision, the memory usage of the GPU is reduced by half compared to 32-bit precision. However, this comes at the cost of fewer and less accurate decimal places. In turn, this could mean that when a prediction is misclassified and close to either zero or one, the machine lacks enough precision to not interpret it as the respective integers, causing the BCE loss to become ∞. For example, in the aforementioned case where $y = 1$ we get a BCE of

$$\begin{aligned} \lim_{\sigma(x)\to 0} \mathcal{L}(x, y=1) &= \lim_{\sigma(x)\to 0} -\Big(y \cdot \log(\sigma(x)) + (1-y)\cdot \log(1-\sigma(x))\Big) \quad (6.1) \\ &= \lim_{\sigma(x)\to 0} -\Big(1 \cdot \log(\sigma(x)) + 0 \cdot \log(1-\sigma(x))\Big) \\ &= -1 \cdot \log(0) \quad (6.2) \\ &= \infty. \end{aligned}$$

One potential cause for this severe misclassification could be some exploding gradient causing the output of the model $\mathbf{x} = \infty$ and thus $\sigma(x) = 0$ (or 1 for $\mathbf{x} = -\infty$), which results in the total loss being ∞ as well. Since both the input vectors and embeddings are normalized, the risk of exploding gradients should be mitigated [39] to some extent. However, normalization is only performed on the inputs to the classification component itself, which is not the case in [39, 40], where *intermediary* batch normalization is used as it helps reduce *internal covariate shift* [41] – which is defined by the authors as "... the change in the distribution of network activations due to the change in network parameters during training." Therefore, adding in intermediary batch normalization between the hidden layers of the classification component of AutoBERT could both resolve the undefined/infinity loss issue and also help the model converge faster [41]. In the case for epoch 18, however, a valid explanation is hard to reason out.

Interestingly, the troublesome loss behavior only occurred in the IGDB model variation, where the highest amount of contextual information was introduced. This raises the question whether or not the added information is more confusing to the network rather than informative. Since the Stream variation reached better performance and converged in fewer epochs, as shown in figure 5.2a, the answer leans towards this added information being confusing. Furthermore, as more information is embedded by the RoBERTa transformer, having greater magnitudes of internal covariate shift in AutoBERT is not unreasonable as an effect. However, as mentioned section 6.1, there could also be other factors which explain the difference in performance.

6.3 Where Contextual Information Improves Predictions

As noted in sections 5.4.1 and 5.4.3, adding additional contextual information tends to lead to an improved ability in defining what is appropriate language, which is also highlighted in table 5.5c by the significant difference in Precision between the models which depend on contextual information (severity score vectors are exempt), and those which do not. Practically speaking, this should result in fewer false positives, i.e., incorrectly identifying of *inappropriate* language as appropriate. Furthermore, from the test results, both Precision and Recall gains a significant performance boost in the IGDB variation when compared to the other models. These results are indicating that the IGDB variation is able to better identify appropriate language (Precision) and to reduce erroneous classification of messages that are appropriate (Recall).

Looking at the inverse case of false positives, a positive improvement does exist for the Stream variation when looking at Specificity in table 5.5b. However, it is not as pronounced as in the prior case and the results in section 5.4 yield mixed results overall, making it hard to tell whether or not there is a significant improvement. Likewise, from the test results in table 5.11b, the Specificity does not change significantly for neither the Stream- nor IGDB variations, although the IGDB variation has a decreased observed Specificity compared to both the EBoW baseline and the Stream variation by about 3.5 percent units.

To further highlight where the AutoBERT models improve over the baselines and where they instead fall short, table 6.1 shows a set of messages from the validation set where the Stream model, EBoW baseline, and the true labels are listed next to it. These are procured by looking at the most extreme cases of failure of the Stream model and EBoW baselines, as they are likely the datapoints which lie on the boundary between being appropriate and inappropriate.

Table 6.1: Messages and predictions of best baseline and AutoBERT model compared to the true label from the validation set. Output labels for each model are listed in parentheses. Do recall that the decision thresholds for the EBoW and Stream models are 0.3 and 0.4, as shown in table 5.7. Note that some messages are not in English, even though that was the only allowed language based on the database query. The entries above the line are based on the greatest absolute distance of the EBoW model compared to the actual label and the entries below are instead based on the greatest absolute distance between the AutoBERT Stream model and the actual label. Do note that **the language here may be inappropriate**.

Message	EBoW	Stream	Label
BatChest	0.0015 (0)	0.0161 (0)	1
₤7 for a second marble strip Keepo	0.0016 (0)	0.0471 (0)	1
\PogChamp/ i never knew you can do licky lic...	0.0021 (0)	0.8090 (1)	1
that's a nice butt tho.	0.0069 (0)	0.3601 (0)	1
C00chie Police. Ma'am, I'm going to need s...	0.0074 (0)	0.0445 (0)	1
chaattuurbate de yayın açsa paranın anasını si...	0.0342 (0)	0.0000 (0)	1
@l*****r но увы альтернативы телеги ща нет, чт...	0.8658 (0)	0.0000 (0)	1
Yo, your areola be showing......just a bit	0.3214 (1)	0.0003 (0)	1
Firm ass	0.3696 (1)	0.0005 (0)	1
little asseater	0.0818 (0)	0.0006 (0)	1

In table 6.1, the first thing to note is that some labels are deemed appropriate even though the language may not be, e.g., the last row which clearly is usage of *Sex-based terms* and potentially *Aggression* or *Bullying*, which are categories considered by AutoMod as presented in section 3.1. This indicates that the inferred labels are not fully accurate, which can be expected since the labels depend on the job of the moderators themselves. Moderators whom cannot be taken for granted to do a flawless job, especially when the activity in chat is high. It is also important to note that some messages may be deemed appropriate depending on stream's configuration of AutoMod, as described insection 2.2, meaning that the leniency of the moderator might still be valid. Therefore, without the knowledge of the configuration of AutoMod, it is hard to tell whether or not a message is appropriate.

Another thing to note is that the Stream model tended to have issues primarily with foreign languages according to the labels. However, as the language is not English, it is hard to tell whether or not the model is correct. The RoBERTa model is trained to work on multiple languages, meaning that the Stream model may very well be correct in that the language is inappropriate whilst the actual label is appropriate. To begin with, since the chat is supposed to be English-only, one might argue that these languages are inappropriate. However, since the moderators have allowed the message by marking it as appropriate, this argument may fall flat.

6.4 Limitations

Considering the results that have been presented up to now, there are some limitations to consider. Firstly, the dataset itself does not contain any fully accepted messages, i.e., neither captured by AutoMod nor deleted by moderators, meaning that the dataset itself is imbalanced to a degree. Therefore, it is best to interpret AutoBERT as a model which is trained to improve, or “boost”, the residual error of AutoMod since it has only been trained on data where some human intervention is needed. It would be interesting, however, to see how AutoBERT fairs against AutoMod on the original message distribution, i.e., including the fully accepted messages as [42] had success in content moderation by training their model on partially labeled data which consisted of only negatives and unlabeled data.

Secondly, the data that was procured is not evenly distributed either, since there are a significant majority of negative labels in both the training and validation sets, as shown in table 5.1. This explains why many of the models have a higher Specificity than Recall, as shown in table 5.5. However, the Stream and IGDB variations of AutoBERT seem to be able to balance out these metrics by increasing the Recall remarkably and maintaining a Specificity similar to the baselines. Nonetheless, some form of data augmentation and/or balancing sampling methods would be appropriate to ensure that the distribution of labels is closer to 50/50 and train the models on such a set instead to mitigate the risk of bias.

Thirdly, the choice of convenience sampling, described in section 3.1.1, may prove to be problematic due to non-periodicity and/or long-time trends [16], which is reasonable as it is difficult to tell what the trends of message bodies look like. It is also reasonable to think that the periodicity of message bodies is long with the constant influence of new trends, e.g., memes and other viralities, which help shape the language used on the Internet. Furthermore, with this uncertainty in periodicity and trends, it becomes hard to decide on a useful temporal cut-off point where the messages prior to this date are superfluous or outdated. Nonetheless, this factor is likely more related to improving performance of a model rather than verifying whether or not it functions. In this project, however, the choice was to use as much data that was available in order to see if the hypotheses could be accepted.

Fourthly, as mentioned previously in both section 5.1.1 and section 6.1, the test set contains datapoints which only refer to two specific channels. Furthermore, the datapoints are labeled by only two moderators, each labeling approximately half of the datapoints each. With this in mind, it is questionable whether the test set is representative of the same distribution of messages that has been trained and validated against. If this is the case, then the inference procedure outlined in equation (3.1) is instead severely lacking since it tends to be heavily biased towards classifying messages as inappropriate when compared to the moderator labels in table 5.3. Nonetheless, in order to further ensure that the observed performance of the models w.r.t. the test set, additional datapoints should be added from other channels as well. Preferably, all 11 of the contacted moderators, or potentially other once in the future, would respond and label at least 50 datapoints each as this would (1) result in more diversified samples in the test set, increasing the likelihood of being representative of the complete distribution of messages which adhere to the delimitations in section 1.4; (2) yield more datapoints than the estimated lower bound in equation (3.4) and thus yield greater confidence in the results.

Lastly, the robustness investigation described in section 3.1.2 does not take the number of datapoints into consideration of each subset, which partially explains the difference in confidence intervals presented in section 5.4. The increase in performance on the unique subsets are unlikely going to be higher compared to the common subsets if they were to be of similar size, since the model should theoretically perform better on data that it has seen before rather than what it has not.

6.5 Future Work

As for the continuation of this work, there are plenty of paths to take in the future. A first proposal would be to try and improve the performance of AutoBERT Stream model. This could be done by fine-tuning a sentence transformer, e.g., RoBERTa, to the specific domain, in this case Twitch chat. Fine-tuning the weights to such a domain would likely improve the quality of the embeddings themselves and help build stronger semantic relationships.

As a second proposal, the computational speed of the producing embeddings could be investigated. One option would be to work with distilled sentence transformers, e.g., DistilRoBERTa, to scale the embedding performance on a vertical axis. To instead scale horizontally, one could investigate the possibilities of separating out the embedding model and have multiple instances embed information in parallel, as discussed in section 6.1.1.

The third proposal would be to work with investigating what the underlying cause of the IGDB variation not improving, as discussed discussed in section 6.1. This could be done by working with imputation methods to fill in the data gaps or further increasing the complexity of the classification component of AutoBERT. Other methods could also be helpful, e.g., by changing activation functions in the hidden layers. Furthermore, introducing intermediary normalization layers could help with the stability of the model and cause faster convergence, as discussed in section 6.2.

Lastly, similar to the small study done on the messages versus severity score vectors, described in section 4.1.1, a larger ablation study could be done to investigate the feature importance of each introduced data type of each added layer of information. If not all features are relevant to the model, then its complexity could be reduced and potentially reduce the training time complexity since the number of token sentences could be reduced.

7 Conclusion

Rounding of the discussion made in the previous chapter, this chapter aims to answer the main research questions stated at the beginning of this thesis. After that, it summarizes and reflects upon the work that was done and how it could have been executed differently. To finish off, this chapter then brings up what this work contributes to the field of algorithmic text moderation.

7.1 Answering the Questions Under Investigation

To reiterate the goal of this work, the aim has been to investigate the possibility to improve algorithmic text moderation by introducing additional layers of contextual information. In this case, the setting has been in Twitch chat with the hypotheses being that (1) introducing a layer of stream metadata along with the chat messages will have a significant impact on the moderator model's performance, (2) introducing a layer of external domain-specific data, here games metadata from IGDB, will result in significant improvement on the moderator model's performance, and (3) the improved performance resulting from each additional layer of information outweighs the time and complexity that follows from incorporating the layers' information into the moderator model.

With the results of the Stream variation of AutoBERT, the following conclusions can be made:

1. Adding a layer of stream metadata to an algorithmic moderator tool based on AutoMod's output will result in significant improvement of moderator performance. Adding stream metadata as contextual information can potentially be concluded as accepted as this was the model which was significantly better when compared to all baselines and the other model variations w.r.t. the validation data. As the Stream variation also had the second best performance on the test set, though not to a significant degree compared to the EBoW baseline, adding additional stream metadata as context is indicative of yielding improved predictive performance. However, further investigation is needed, both w.r.t. a better labeling procedure to lessen the gap between validation- and test sets and w.r.t. obtaining additional test data to narrow down the confidence bounds.

2. Adding a layer of domain-specific data relating to the stream metadata of (1) to an algorithmic moderator tool based on AutoMod's output will result in significant improvement of moderator performance. As for the IGDB variation of AutoBERT, since it did not perform better than the Stream variation on the validation set, it is not possible to conclude (2) as accepted with certainty. However, as it did perform the best on the test set, albeit with significant performance drops compared to the validation set, more work needs to be done before (2) can be either accepted or rejected with certainty. Furthermore, the cause behind the validation performance not improving, when the test performance did, has yet to be determined.

3. Improved performance results of each additional layer of information outweigh the time and complexity that follows from incorporating the layers' information into an algorithmic moderator tool based on AutoMod. With neither (1) nor (2) being accepted for certain, (3) cannot be fully accepted either. Nonetheless, depending on what is decided to be complex and time-consuming for the specific setting, e.g., the speed of forward passes (see section 6.1.1), (3) may be partially accepted since AutoBERT Stream performed better than the best baseline within the range of 2-6 percent units across all metrics on the validation data. Furthermore, AutoBERT IGDB was significantly better than all baselines in all metrics except for Specificity, and only lost to the Stream variation in Specificity, as shown in table 5.11. If the aforementioned investigation of (1) is performed and if (1) is accepted with confidence; and if the time complexity for the IGDB variation prediction can be reduced, potentially using the methods mentioned in section 6.1.1; then (3) can be fully accepted. This leaves most of the hypotheses potentially accepted; however, to come to a solid conclusion.

7.2 Summarizing and Reflecting on the Work

Regarding the work done in this thesis, it has involved data procuration, model implementation, and model evaluation. First, the process of procuring training- and validation data was done by accessing a large data lake in section 3.1. In this step, the stream metadata for the Stream layer was also procured. Then, baselines were produced in section 4.1 along with an ablation study to investigate some feature importance of severity scores from AutoMod's internal model and chat messages themselves, described in section 4.1.1. Following the implementation of the baselines, AutoBERT was designed for the Base variation, which was then followed by the Stream variation in section 4.2. Afterwards, the games metadata from IGDB was procured and joined with the already-existing datasets, described in section 3.2. With the IGDB data procured, the last variation of AutoBERT was trained. As the IGDB variation of AutoBERT was training, the test set data was procured and sent out for manual labeling by moderators at Twitch in their respective domains, described in section 3.3. Lastly, all models, both baselines and AutoBERT variations, were evaluated following the procedure outlined in section 4.4.

In hindsight, there are multiple actions that could have been taken differently in order to save time and produce more results. First of all, using the baselines, learning curves of the training data could be procured in order to see how much data was needed before performance improvements stagnated. At first, using all data seemed to be the right way to go as it did not take long to train the baselines. However, as the training of AutoBERT commenced, it became apparent how fast the models tended to converge; every model reached a high performance after about 1 epoch, as shown in figure 5.2a. Had the training set been smaller but still potentially yielded as good results, then training could have been quicker and additional efforts could have been done with troubleshooting the IGDB variation's unexpected behavior and lack of performance increase.

Second, further preparatory research could have been made regarding sentence transformers and the recommendation of using PyTorch instead of TensorFlow, which would have saved time and duplicate work related to loading and processing of data. Also, if a distilled transformer model had been used, e.g., DistilRoBERTa, then the training could have gone faster since these models are smaller and tend to have significantly higher embedding speeds.

Third, with more time being saved from the two prior actions, more time could have been spent on investigating the procured data itself. With a deeper understanding of the data under investigation, e.g., via further ablation studies, the produced models would not only be useful from a performance perspective, but also in the perspective of finding mechanistic explanations to model the behavior of Twitch chat. Such findings could help broaden the spectrum of what can be researched and raise new and interesting questions regarding Twitch chat from perspectives other then moderation as well.

7.3 Contributions to the Field

To wrap things up, it is time to talk about what this work contributes to the field of algorithmic text moderation. Much like the work done in [5], where algorithmic text moderation of comments was successfully improved by producing a model that combining the underlying implicit patterns of the comments' content with a context defined by patterns hidden in the related social network of the comments, this work has successfully shown that moderation of chat messages can be improved by incorporating contextual information based on the surrounding stream data where the chat message occurs.

Furthermore, it shows that the use of sentence transformers fine-tuned for semantic textual similarity tasks can be used in ways similar to the models in [5, 43]. However, instead of comments, the available metadata surrounding chat messages allows the sentence transformers to be applied on chat with significant improvements.

Lastly, this work illustrates how sentence transformers may be leveraged to easily embed additional layers of information to further investigate what text-based information can be used to construct an informative context for a classification model. Albeit, at the cost of linear time complexity for each new token sentence that needs to be added per datapoint.

Bibliography

[1] M. D. Tredici and R. Fernández, *Semantic variation in online communities of practice*, 2018. arXiv: 1806.05847 `[cs.CL]`.

[2] R. Gorwa, R. Binns, and C. Katzenbach, "Algorithmic content moderation: Technical and political challenges in the automation of platform governance", *Big Data & Society*, vol. 7, no. 1, p. 2 053 951 719 897 945, 2020.

[3] R. Binns, M. Veale, M. Van Kleek, and N. Shadbolt, "Like trainer, like bot? inheritance of bias in algorithmic content moderation", in *International conference on social informatics*, Springer, 2017, pp. 405–415.

[4] I. Brigadir, D. Greene, and P. Cunningham, "Analyzing discourse communities with distributional semantic models", in *Proceedings of the ACM Web Science Conference*, ser. WebSci '15, Oxford, United Kingdom: Association for Computing Machinery, 2015, ISBN: 9781450336727. DOI: 10.1145/2786451.2786470. [Online]. Available: https://doi.org/10.1145/2786451.2786470.

[5] A. Veloso, W. Meira, T. Macambira, D. O. Guedes, and H. Almeida, "Automatic moderation of comments in a large on-line journalistic environment", in *ICWSM*, 2007.

[6] Twitch Staff, *How to use automod.* [Online]. Available: https://help.twitch.tv/s/article/how-to-use-automod?language=en_US (visited on 09/30/2020).

[7] IGDB Staff, *Welcome to the guiding star in your world of gaming.* [Online]. Available: https://www.igdb.com/ (visited on 01/30/2021).

[8] Twitch Staff, *List of all tags.* [Online]. Available: https://www.twitch.tv/directory/all/tags (visited on 01/30/2021).

[9] Oxford University Press, *Context.* [Online]. Available: https://www.oxfordlearnersdictionaries.com/definition/english/context (visited on 01/30/2021).

[10] Python Software Foundation, *Typing – support for type hints.* [Online]. Available: https://docs.python.org/3.6/library/typing.html (visited on 01/30/2021).

[11] Torch Contributors, *Torch.utils.data.* [Online]. Available: https://pytorch.org/docs/stable/data.html (visited on 01/30/2021).

[12] Python Software Foundation, *Multiprocesing – process-based parallelism.* [Online]. Available: https://docs.python.org/3.6/library/multiprocessing.html (visited on 01/30/2021).

[13] Python Software Foundation, *Multiprocessing.shared_memory — provides shared memory for direct access across processes.* [Online]. Available: https://docs.python.org/3/library/multiprocessing.shared_memory.html (visited on 01/30/2021).

[14] Python Software Foundation, *Ctypes – a foreign function library for python.* [Online]. Available: https://docs.python.org/3.6/library/ctypes.html (visited on 01/30/2021).

[15] Torch Contributors, *Distributeddataparallel.* [Online]. Available: https://pytorch.org/docs/master/generated/torch.nn.parallel.DistributedDataParallel.html (visited on 01/31/2021).

[16] Z. Reitermanova, "Data splitting", in *WDS*, vol. 10, 2010, pp. 31–36.

[17] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference, and prediction*, 2nd ed. Springer Science & Business Media, 2009, ch. 7, pp. 241–248, ISBN: 9780387848570.

[18] Y. Xu and R. Goodacre, "On splitting training and validation set: A comparative study of cross-validation, bootstrap and systematic sampling for estimating the generalization performance of supervised learning", *Journal of Analysis and Testing*, vol. 2, no. 3, pp. 249–262, 2018.

[19] G. P. Zhang and V. Berardi, "Time series forecasting with neural network ensembles: An application for exchange rate prediction", *Journal of the operational research society*, vol. 52, no. 6, pp. 652–664, 2001.

[20] G. J. Bowden, H. R. Maier, and G. C. Dandy, "Optimal division of data for neural network models in water resources applications", *Water Resources Research*, vol. 38, no. 2, pp. 2–1, 2002.

[21] IGDB Staff, *Welcome to the guiding star in your world of gaming.* [Online]. Available: https://api-docs.igdb.com/ (visited on 01/30/2021).

[22] Python Software Foundation, *Ast — abstract syntax trees.* [Online]. Available: https://docs.python.org/3.6/library/ast.html (visited on 01/31/2021).

[23] M. Collins, R. E. Schapire, and Y. Singer, "Logistic regression, adaboost and bregman distances", *Machine Learning*, vol. 48, no. 1, pp. 253–285, 2002.

[24] Y. Zhang, R. Jin, and Z.-H. Zhou, "Understanding bag-of-words model: A statistical framework", *International Journal of Machine Learning and Cybernetics*, vol. 1, no. 1-4, pp. 43–52, 2010.

[25] F. Chollet, *The functional api.* [Online]. Available: https://www.tensorflow.org/guide/keras/functional (visited on 10/23/2020).

[26] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[27] N. Reimers and I. Gurevych, *Sentence-bert: Sentence embeddings using siamese bert-networks*, 2019. arXiv: 1908.10084 `[cs.CL]`.

[28] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach", *arXiv preprint arXiv:1907.11692*, 2019.

[29] N. Reimers, *Sentencetransformers documentation*. [Online]. Available: https://www.sbert.net/ (visited on 02/01/2021).

[30] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[31] Torch Contributors, *Torch.tensor*. [Online]. Available: https://pytorch.org/docs/stable/tensors.html (visited on 02/09/2021).

[32] The Hugging Face Team, *Roberta*. [Online]. Available: https://huggingface.co/transformers/model_doc/roberta.html (visited on 02/08/2021).

[33] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network", *arXiv preprint arXiv:1505.00853*, 2015.

[34] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *Tf.data.dataset*. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shuffle (visited on 10/23/2020).

[35] L. N. Smith, "Cyclical learning rates for training neural networks", in *2017 IEEE winter conference on applications of computer vision (WACV)*, IEEE, 2017, pp. 464–472.

[36] B. van Zaane, Y. Vergouwe, A. R. T. Donders, and K. G. Moons, "Comparison of approaches to estimate confidence intervals of post-test probabilities of diagnostic test results in a nested case-control study", *BMC Medical Research Methodology*, vol. 12, no. 1, p. 166, 2012.

[37] A. R. T. Donders, G. J. Van Der Heijden, T. Stijnen, and K. G. Moons, "A gentle introduction to imputation of missing values", *Journal of clinical epidemiology*, vol. 59, no. 10, pp. 1087–1091, 2006.

[38] Torch Contributors, *Bcewithlogitsloss*. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html (visited on 01/30/2021).

[39] J. Bjorck, C. Gomes, B. Selman, and K. Q. Weinberger, "Understanding batch normalization", *arXiv preprint arXiv:1806.02375*, 2018.

[40] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition”, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016.

[41] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, in *International conference on machine learning*, PMLR, 2015, pp. 448–456.

[42] J.-Y. Delort, B. Arunasalam, and C. Paris, “Automatic moderation of online discussion sites”, *International Journal of Electronic Commerce*, vol. 15, no. 3, pp. 9–30, 2011.

[43] J. Pavlopoulos, P. Malakasiotis, and I. Androutsopoulos, “Deep learning for user comment moderation”, *arXiv preprint arXiv:1705.09993*, 2017.

A
AutoBERT

A.1 Classification Component

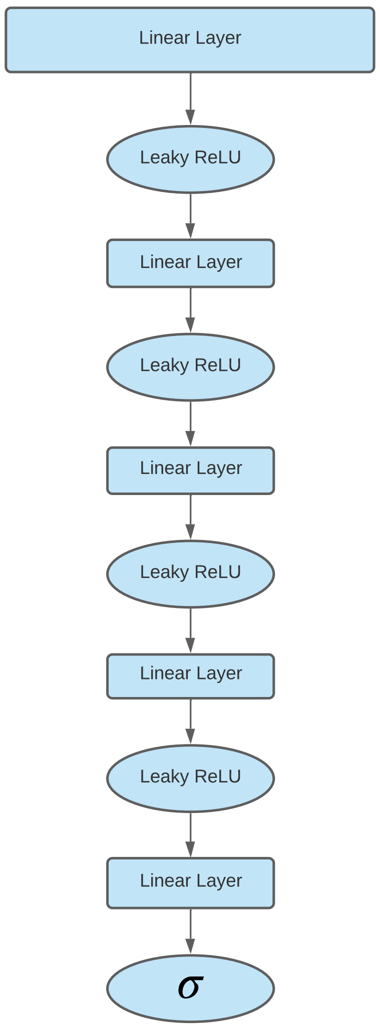


Figure A.1: Fully connected network architecture of the classification component of AutoBERT. Note that the first Linear Layer is wider than the other layers to signify the concatenated inputs from both input channels of the AutoBERT architecture shown in figure 4.3

A.2 Run-time Configuration

Listing 6: AutoBERT run-time configuration for the IGDB iteration. Do note that each field in `data_vars` defines a data type of the data and has a comment annotating at what iteration this data type was introduced. The `structure` -field defines the structure of the dataset using sets of indices; the first set of indices defines the text inputs, the second defines the number inputs, and the third defines the output label(s). The `mixins` -field defines which transformations should be applied to each text input type to be used when loading the data inside the dataset.

```
igdb_config = DatasetConfig(
    data_vars=[
        # String-based columns
        MESSAGE_BODY,                                 # Base
        SUMMARY,                                      # IGDB
        CHANNEL_TITLE, CHANNEL_CATEGORY,              # Stream
        BROADCAST_TAGS, CATEGORY_TAGS,                # Stream
        GENRES,                                       # IGDB
        THEMES, FRANCHISES,                           # IGDB
        # Number-based columns
        SEVERITY_ABLEIST, SEVERITY_AGGRESSION,        # Stream
        SEVERITY_HOMOPHOBIC, SEVERITY_MISOGYNISTIC,   # Stream
        SEVERITY_N_WORD, SEVERITY_NAMECALLING,        # Stream
        SEVERITY_NATIONALIST, SEVERITY_RACIST,        # Stream
        SEVERITY_SEXWORDS, SEVERITY_SWEARING,         # Stream
        AUTOMOD_LVL_1, AUTOMOD_LVL_2,                 # IGDB
        AUTOMOD_LVL_3, AUTOMOD_LVL_4,                 # IGDB
        # Output
        LABEL                                         # Base
    ],
    structure=[slice(0, 9), slice(9, -1), -1],
    mixins={
        MESSAGE_BODY: identity,
        SUMMARY: identity,
        CHANNEL_TITLE: channel_mapper,
        CHANNEL_CATEGORY: category_mapper,
        BROADCAST_TAGS: broadcast_tag_mapper,
        CATEGORY_TAGS: category_tag_mapper,
        GENRES: genres_mapper,
        THEMES: themes_mapper,
        FRANCHISES: franchise_mapper
    }
)
```

B

Robustness Investigation Results

B.1 Channels

Table B.1: Accuracy scores of the baseline models w.r.t. common and unique channels of the validation set.

(a) Common channels of the validation set, i.e. channels present in the training set.

Model	Accuracy
SSV	0.8142 (±0.0016)
BoW	0.8325 (±0.0015)
EBoW	0.8462 (±0.0014)

(b) Unique channels of the validation set, i.e. channels not present in the training set.

Model	Accuracy
SSV	0.7694 (±0.0077)
BoW	0.7650 (±0.0081)
EBoW	0.7804 (±0.0073)

Table B.2: Precision, Recall, and F1 scores of the baseline models w.r.t. common and unique channels of the validation set.

(a) Common channels of the validation set, i.e. channels present in the training set.

	Precision	Recall	F1 Score
Model			
SSV	0.6192 (±0.0033)	0.8077 (±0.0030)	0.7010 (±0.0027)
BoW	0.6770 (±0.0036)	0.7242 (±0.0035)	0.6998 (±0.0028)
EBoW	0.7012 (±0.0033)	0.7484 (±0.0032)	0.7240 (±0.0028)

(b) Unique channels of the validation set, i.e. channels not present in the training set.

	Precision	Recall	F1 Score
Model			
SSV	0.6625 (±0.0126)	0.8285 (±0.0110)	0.7363 (±0.0091)
BoW	0.6994 (±0.0144)	0.6933 (±0.0127)	0.6963 (±0.0105)
EBoW	0.7203 (±0.0135)	0.7112 (±0.0119)	0.7157 (±0.0096)

Table B.3: Specificity scores of the baseline models w.r.t. common and unique channels of the validation set.

(a) Common channels of the validation set, i.e. channels present in the training set.

	Specificity
Model	
SSV	0.8166 (±0.0018)
BoW	0.8724 (±0.0015)
EBoW	0.8823 (±0.0015)

(b) Unique channels of the validation set, i.e. channels not present in the training set.

	Specificity
Model	
SSV	0.7318 (±0.0096)
BoW	0.8106 (±0.0085)
EBoW	0.8244 (±0.0084)

B.2 Broadcast Categories

Table B.4: Accuracy scores of the baseline models w.r.t. common and unique broadcast categories of the validation set.

(a) Common broadcast categories of the validation set, i.e. broadcast categories present in the training set.

	Accuracy
Model	
SSV	0.8121 (±0.0016)
BoW	0.8299 (±0.0015)
EBoW	0.8439 (±0.0015)

(b) Unique broadcast categories of the validation set, i.e. broadcast categories not present in the training set.

	Accuracy
Model	
SSV	0.8161 (±0.0123)
BoW	0.8030 (±0.0123)
EBoW	0.8215 (±0.0131)

Table B.5: Precision, Recall, and F1 scores of the baseline models w.r.t. common and unique broadcast categories of the validation set.

(a) Common broadcast categories of the validation set, i.e. broadcast categories present in the training set.

	Precision	Recall	F1 Score
Model			
SSV	0.6182 (±0.0032)	0.8090 (±0.0029)	0.7008 (±0.0025)
BoW	0.6753 (±0.0035)	0.7223 (±0.0036)	0.6980 (±0.0028)
EBoW	0.6998 (±0.0035)	0.7463 (±0.0032)	0.7223 (±0.0025)

(b) Unique broadcast categories of the validation set, i.e. broadcast categories not present in the training set.

	Precision	Recall	F1 Score
Model			
SSV	0.7873 (±0.0194)	0.8134 (±0.0187)	0.8001 (±0.0148)
BoW	0.8088 (±0.0205)	0.7393 (±0.0217)	0.7725 (±0.0161)
EBoW	0.8284 (±0.0182)	0.7636 (±0.0198)	0.7947 (±0.0147)

Table B.6: Specificity scores of the baseline models w.r.t. common and unique broadcast categories of the validation set.

(a) Common broadcast categories of the validation set, i.e. broadcast categories present in the training set.

Model	Specificity
SSV	0.8132 (±0.0018)
BoW	0.8702 (±0.0015)
EBoW	0.8803 (±0.0015)

(b) Unique broadcast categories of the validation set, i.e. broadcast categories not present in the training set.

Model	Specificity
SSV	0.8184 (±0.0160)
BoW	0.8556 (±0.0151)
EBoW	0.8693 (±0.0149)

B.3 Messages

Table B.7: Accuracy scores of the baseline models w.r.t. common and unique messages of the validation set.

(a) Common messages of the validation set, i.e. messages present in the training set.

	Accuracy
Model	
SSV	0.8161 (±0.0039)
BoW	0.8915 (±0.0031)
EBoW	0.8880 (±0.0030)

(b) Unique messages of the validation set, i.e. messages not present in the training set.

	Accuracy
Model	
SSV	0.8113 (±0.0017)
BoW	0.8180 (±0.0017)
EBoW	0.8349 (±0.0016)

Table B.8: Precision, Recall, and F1 scores of the baseline models w.r.t. common and unique messages of the validation set.

(a) Common messages of the validation set, i.e. messages present in the training set.

	Precision	Recall	F1 Score
Model			
SSV	0.4189 (±0.0092)	0.8106 (±0.0107)	0.5524 (±0.0098)
BoW	0.5898 (±0.0116)	0.7385 (±0.0112)	0.6558 (±0.0099)
EBoW	0.5771 (±0.0120)	0.7495 (±0.0122)	0.6521 (±0.0090)

(b) Unique messages of the validation set, i.e. messages not present in the training set.

	Precision	Recall	F1 Score
Model			
SSV	0.6485 (±0.0031)	0.8090 (±0.0031)	0.7199 (±0.0027)
BoW	0.6873 (±0.0035)	0.7207 (±0.0033)	0.7036 (±0.0028)
EBoW	0.7155 (±0.0035)	0.7455 (±0.0033)	0.7302 (±0.0027)

Table B.9: Specificity scores of the baseline models w.r.t. common and unique messages of the validation set.

(a) Common messages of the validation set, i.e. messages present in the training set.

Model	Specificity
SSV	0.8170 (±0.0041)
BoW	0.9164 (±0.0029)
EBoW	0.9106 (±0.0031)

(b) Unique messages of the validation set, i.e. messages not present in the training set.

Model	Specificity
SSV	0.8122 (±0.0019)
BoW	0.8596 (±0.0018)
EBoW	0.8731 (±0.0017)

C

Shared Memory in Python

Listing 7: Base class implementation of the C-type structure for tokenized sentences.

```python
import ctypes
from transformers import BatchEncoding

class TOKENIZED_SENTENCE(ctypes.Structure):
    _initialized_ = False
    _shape = None

    def __init__(self, token_ids, attention_mask):
        return super().__init__(token_ids, attention_mask)

    @classmethod
    def from_torch(cls, batch_encoding):
        token_ids = batch_encoding['input_ids']
        attention_mask = batch_encoding['attention_mask']
        token_ids_ = cls._token_type(
                *[TOKEN_ARR(*ts) for ts in token_ids.tolist()]
            )
        attention_mask_ = cls._attention_type(
                *[ATTENTION_ARR(*ts) for ts in attention_mask.tolist()]
            )

        if cls._shape is None:
            cls._shape = {
                'input_ids': token_ids.size(),
                'attention_mask': attention_mask.size()
            }

        return cls(token_ids_, attention_mask_)

    def as_torch(self, shape=None):
        encoding = {}
        for k, _ in self._fields_:
            encoding[k] = torch.from_numpy(
                np.ctypeslib.as_array(self.__getattribute__(k)) \
                            .reshape(shape[k] \
                                    if shape is not None \
                                    else self._shape[k]
                                )
            )
        return BatchEncoding(encoding)
```

Listing 8: Base class implementation of the C-type structure for datapoints consisting of a message, a severity score vector, and the target response. Do note that the snippet continues in listing 9.

```python
import ctypes
from transformers import BatchEncoding

class CACHE_TYPE(ctypes.Structure):
    _initialized_ = False
    _shape = None
    _n_sentences = None
    _n_vector_dims = None

    def __init__(self, tokens, vectors, label):
        super().__init__(tokens, vectors, label)

    @classmethod
    def init(cls, n_sentences, n_vector_dims):
        if hasattr(cls, "_fields_") or cls._initialized_:
            raise AttributeError(
            """Tried to initialize CACHE_TYPE
            class when it has already been initialized"""
                )
        cls._n_sentences = n_sentences
        cls._n_vector_dims = n_vector_dims
        cls._sentence_type = TOKENIZED_SENTENCE.init(n_sentences)
        cls._vector_type = ctypes.c_float * n_vector_dims
        cls._label_type = LABEL
        cls._fields_ = [
            ("sentence_tokens", cls._sentence_type),
            ("vector", cls._vector_type),
            ("label", cls._label_type)
        ]
        cls._initialized_ = True
        return cls

''' ... continuation in next listing '''
```

Listing 9: Base class implementation of the C-type structure for datapoints consisting of a message, a severity score vector, and the target response. Do note that the snippet is a continuation of listing 8.

```python
'''Continuation of previous listing'''
    @classmethod
    def from_torch(cls, tokens, vector, label):
        tokens_ = cls._sentence_type.from_torch(tokens)
        vector_ = cls._vector_type(*vector.flatten().tolist())
        label_ = cls._label_type(label.item())

        if cls._shape is None:
            cls._shape = {
                "sentence_tokens": tokens_._shape,
                "vector": vector.size(),
                "label": label.size()
            }
        return cls(tokens_, vector_, label_)

    def as_torch(self, shape=None):
        tokens = self.sentence_tokens.as_torch(shape['sentence_tokens'] \
                     if shape is not None else self._shape['sentence_tokens'])
        vector = torch.from_numpy(
            np.ctypeslib.as_array(self.vector) \
                     .reshape(shape['vector'] \
                     if shape is not None else self._shape['vector'])
        ).half()
        label = torch.from_numpy(
            np.ctypeslib.as_array(self.label) \
                     .reshape(shape['label'] \
                     if shape is not None else self._shape['label'])
        ).half()
        return tokens, vector, label
```