UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Karmen Kink

Classification of E-Commerce Products Based on Textual Product Descriptions

Master's Thesis (30 ECTS)

Supervisors: Kairit Sirts, Karl-Oskar Masing

Tartu 2021

Classification of E-Commerce Products Based on Textual Product Descriptions

Abstract:

Assigning Harmonized System (HS) codes to products is necessary to comply with customs regulations, gather statistics, and prevent fraud. Since the HS is a complex system with many classes, automatic HS code classification is required to speed up the process and ensure correctness. In this thesis, we explore two types of machine learning methods for HS classification – shallow neural network classifiers and deep neural network classifiers that are based on the Transformer architecture. We find that with a large dataset, shallow classifiers can be relatively easily improved to outperform Transformer-based classifiers, while tuning the latter is a more complex and time-consuming task. We also discover that the training data includes erroneously labeled entries and that this has a negative impact on the models.

Keywords:

Natural language processing, Harmonized System, multi-class classification

CERCS:

P176 Artificial Intelligence

E-kaubanduse toodete klassifitseerimine tekstiliste tootekirjelduste alusel

Lühikokkuvõte:

Harmoneeritud Süsteemi (HS) koodide määramine toodetele on vajalik tollinõuetele vastamiseks, statistika kogumiseks ja maksupettuste vältimiseks. Automaatne HS-koodide klassifitseerimine aitab aega säästa, sest HS on kompleksne ja paljude klassidega süsteem, mistõttu käsitsi õige klassi valimine on keerukas ja aeganõudev. Selles magistritöös kasutame HS-koodide klassifitseerimiseks kaht tüüpi masinõppemeetodeid: pindmistel tehisnärvivõrkudel põhinevaid klassifitseerijaid ja sügavatel tehisnärvivõrkudel, täpsemalt Transformer-arhitektuuril, põhinevaid klassifitseerijaid. Selgub, et sellise suurusega andmestiku puhul, nagu meil on võimalik kasutada, saab pindmisi klassifitseerijaid võrdlemisi kergesti arendada maani, kus nad annavad paremaid tulemusi kui sügavad närvivõrgud, viimaseid on aga oluliselt keerukam ja aeganõudvam edasi arendada. Lisaks leidsime, et kasutatav andmestik sisaldab valesti märgendatud kirjeid ning et sellel on negatiivne mõju mudelite kvaliteedile.

Võtmesõnad:

Loomuliku keele töötlus, Harmoneeritud Süsteem, mitmeklassiline klassifitseerimine

CERCS:

P176 Tehisintellekt

Table of Contents

1 Introduction

In the expanding field of international e-commerce, it is important to ensure that the movement of goods across borders is as smooth and fast as possible. It is also crucial that the goods have correct information attached to collect taxes, gather statistics and prevent fraud.

This necessity for accurate product information is now more topical than ever. As of the year 2021, some notable changes in the European Union's customs regulations will take effect. When previously, goods with a value of up to 22€ were allowed to be imported to the EU without value-added tax (VAT) and a customs declaration, then now this lower bound will be removed, and all goods will need an accompanying customs declaration[1].

Among other effects, this means that merchants need to provide customs officials with information about all products traveling between EU and non-EU countries. This information includes sufficiently detailed product descriptions and respective product codes from the Harmonized System (HS) nomenclature[2]. The HS nomenclature is meant to provide a common, internationally recognized classification system for products.

Automatically assigning an HS code or recommending a limited choice of possibly correct HS codes to a product is necessary to minimize the amount of manual labor, avoid human error, speed up the process, and avoid delays or penalties caused by mistakenly assigned HS codes. Considering that all imported products will need a customs declaration that includes the HS code and that the number of low-value consignments imported to the EU is high[3], the mentioned legislation change is expected to drastically increase the need for such automatic classification.

---

1 https://ec.europa.eu/taxation_customs/business/vat/modernising-vat-cross-border-ecommerce_en

2 https://unstats.un.org/unsd/tradekb/Knowledgebase/50018/Harmonized-Commodity-Description-and-Coding-Systems-HS

3 https://ec.europa.eu/taxation_customs/news/new-form-customs-declaration-low-value-consignments_en

In this thesis, HS code classification is performed using only product descriptions as input. In some cases, additional features could be considered, such as origin and destination country, dimensions of the package, etc. Some examples of product descriptions with corresponding HS codes are shown in Table 1. The table also presents explanatory texts, i.e., the texts that describe these classes in the nomenclature.

Table 1. Exemplary product descriptions with corresponding codes and explanatory texts from the HS nomenclature.

Product Description	**HS Code**	**Explanatory Text**
SPA CEYLON RED SANDAL & LEMONGRASS - Massage & Bath Oil(ISFTA CERTIFICATE NO: CO/ISFTA/2019/12058 DATED: 14.11.2019) (CD	330730	Perfumed bath salts and other bath and shower preparations
Bicycle Parts (BICYCLE SPARE PARTS) (grip)(1 SET) -	871499	Parts and accessories, for bicycles, n.e.s.

The classification task is difficult due to a large number of classes – there are more than 5,300 six-digit HS codes in the nomenclature. In addition to this, the datasets we have access to are imbalanced, that is, the distribution of classes is strongly skewed. Another complication is that product descriptions can be very short, include spelling errors or seemingly non-informative noise. Lastly, differences between classes can be minimal, and separating lines are sometimes drawn based on tiny details.

The task of HS classification has been tackled in previous publications using traditional machine learning methods like support vector machines and Naïve Bayes [1], as well as deep learning approaches like convolutional neural networks [2]. However, the field of Natural Language Processing (NLP) has seen rapid improvements lately with the progress of pre-trained language models such as BERT [3], which is based on the Transformer architecture [4]. To our knowledge, there are currently no publications where such pre-trained language models are applied to HS classification, and therefore, we aim to experiment with these methods.

Our objective is to build a classifier using neural networks that is able to predict the HS code based on a textual product description among a large number of classes. Based on the state-of-the-art results achieved with Transformer-based models on various NLP tasks, we apply this architecture as the basis of our classification models. Methods for handling class imbalance and taking advantage of the nomenclature's hierarchical structure are explored and experimented with.

Chapter 2 of this thesis introduces the HS nomenclature and gives an overview of publications on related topics. Chapter 3 describes the technical concepts and architectures used. Chapter 4 presents an overview of the dataset and discusses preprocessing of the data. Chapter 5 describes the experiments conducted with different types of classifiers. Chapter 6 presents the results on evaluation splits and a qualitative analysis of these results. Chapter 7 discusses the limitations of this thesis and directions for future work.

This thesis is part of the project "A Digital Infrastructure for Cross Border E-Commerce (SaaS) Applied Research," conducted by STACC OÜ and Tallinn University of Technology. The partner company that the contributing parties are developing the digital infrastructure for and with is Eurora Solutions OÜ.

2 Background

This chapter describes the nomenclature used for product classification and introduces previous publications on related topics.

2.1 The Harmonized System Nomenclature

The Harmonized Commodity Description and Coding System, which we will refer to as simply the Harmonized System or HS in this thesis, was created by the World Customs Organization [5]. The system is used by most international traders and serves to unify product classification, provides a basis for customs duties calculations, helps monitor and gather statistics of tradeable goods, and more.

The HS has a hierarchical structure and is divided into levels. The first level is known as the section and is not explicitly represented in the HS code. However, one section contains chapters of a similar topic, e.g., under the section *Vegetable products*, we can find a chapter titled *Coffee, tea, maté and spices*, as well as one titled *Cereals*.

The other levels are reflected in the HS code. As illustrated in Table 2, the first two digits of a six-digit HS code signify the chapter that a product belongs to. The first four digits signify the corresponding heading, and the full six digits signify the corresponding subheading. We will refer to these (sub)parts of the HS code as HS2, HS4, and HS6 throughout this thesis.

Table 2. Structure of the HS.

Section XI – Textiles and textile articles		
Chapter (HS2)	52	Cotton
Heading (HS4)	5210	Woven fabrics of cotton, containing predominantly, but < 85% cotton by weight, mixed principally or solely with man-made fibres and weighing <= 200 g/m^2
Subheading (HS6)	521039	Woven fabrics of cotton, containing predominantly, but < 85% cotton by weight, mixed principally or solely with man-made fibres and weighing <= 200 g/m^2, dyed (excl. those in three-thread or four-thread twill, incl. cross twill, and plain woven fabrics)

The levels of the HS are ordered from more general to more specific, meaning that chapters are broad classes of quite different types of goods, while headings under a given chapter are its more specific subclasses, and subheadings under a given heading are still more specific subclasses of that.

Due to the evolving nature of international trade and new inventions, the HS is updated periodically. In this thesis, we focus on the current state of the nomenclature, i.e., its 2017 edition [6]. In total, this edition includes 97 chapters (of which one is reserved for future use), 1222 headings, and 5387 subheadings [7]. However, not all these classes are present in the datasets that we use since some chapters and their subclasses are out of scope for our partner.

The HS is the first part of a more detailed classification system, the Integrated Tariff of the European Union (TARIC)[4], consisting of ten digits. The first six correspond to the HS code, the first eight correspond to the Combined Nomenclature code, and the full ten digits then make up a TARIC code. The scope of this thesis is limited to HS codes, that is, the first six digits.

2.2 Related Work

HS code classification has not been widely covered in previous publications. The approaches in the few topical articles are rather varied, though, ranging from background nets to convolutional neural networks.

One of the earliest publications on the subject tackles the classification problem with background nets [8]. According to the article, background nets are weighted undirected graphs where vertices represent terms and edge weights represent their co-occurrence counts in the training data. Classification with this approach works by learning a background net for each category from the training data. During inference, the input text is used to construct its own background net which is then compared to the learned nets. An acceptance criterion is used for choosing the category whose

[4] https://ec.europa.eu/taxation_customs/business/calculation-customs-duties/what-is-common-customs-tariff/taric_en

background net's acceptance to the input's net is maximal. In this paper, only two chapters and their subclasses were used, and classification was performed separately within each of the two chapters.

In a recent publication, several machine learning models were experimented with and compared [1]. Both the dataset used and its number of categories were large, and severe down-sampling was performed due to computational limitations and class imbalance. However, the impact of the sampling was not analyzed. The authors compared various models by their performance on only the heading, that is, the first four digits, and on the full HS code. The best results were reportedly achieved with linear support vector machine (SVM) and random forest models, while the worst performers were decision tree and Naïve Bayes models.

Convolutional neural networks (CNNs) have been used with only textual inputs in the form of product descriptions [2] but also with both text and image inputs in a model fusion setup [9]. In the former work, part of the focus was on pre-training domain-specific word embeddings on additional unlabeled product data scraped from the web, and part on creating a customized model architecture. In the latter publication, the images and product descriptions used for training and testing were scraped from a trade website. The authors found that using only images does not provide good performance while using only product descriptions is a much better option, and the combination of the two gives a slight improvement over the text-only model. However, in this paper, classification was performed on just four different classes.

Possibilities for using similarity measures for assessing the correctness of already assigned HS codes or predicting/recommending HS codes have also been explored [10]. To predict an HS code for a given input description, the proposed method uses a pre-trained doc2vec [11] model for finding the most similar product descriptions from training data according to cosine similarity and then finds the weighted mode from a set of most similar descriptions to predict its corresponding HS code. In order to assess the correctness of already assigned HS codes, the same cosine similarities from doc2vec are used, but additionally, the semantic similarity between the HS codes is calculated based on the taxonomy structure.

There have also been attempts to model the hierarchy explicitly. An example of this is the Deep Hierarchical Classification Network [12], developed for the task of predicting the correct category for online shop products. This model includes a flat neural network for creating the root representation of an input sequence, followed by one linear layer per hierarchy level. Each layer receives as input the root representation concatenated with the representation from the previous level. Additionally, it features a hierarchical loss where the losses from each layer and layer mismatch penalties are taken into account. The authors found that both sharing the representations between layers and the proposed loss function had a positive impact on performance, with the former contributing more than the latter.

3 Technical background

This chapter introduces the most important concepts related to or used in this thesis – word representations, neural networks, attention mechanism, and the Transformer architecture.

3.1 Word Representations

When working with textual data such as product descriptions, we need a way to represent this data in vectorized form so that it would be usable in a neural network. This subsection describes some methods for creating such representations and is based on the book *Neural Network Methods for Natural Language Processing* by Yoav Goldberg [13].

The most basic concept would be to convert our data into what are called one-hot encodings. This means that the vector for each word would be as long as our vocabulary, containing a large number of zeros and a single one at the index that corresponds to this exact word. However, this approach has two main drawbacks. Firstly, such representations do not capture the semantic relatedness between words since each word is considered independent in the representation space and distances from a given word to all other words are equal. Secondly, the dimensionality of the representations would be too high for computationally heavier models.

Distributed representations are an alternative solution and are commonly used with neural networks. Unlike one-hot encodings, they have lower dimensionality and allow semantic relatedness to be learned. In the case of distributed representations, the dimensions do not correspond to specific words anymore. Rather, the information about some aspect of meaning can be distributed across many dimensions; thus, the dimensions are difficult to interpret independently. Such representations can be compared using distance measures, e.g., cosine similarity, to identify the semantic similarity between words.

Distributed representations can be learned via unsupervised pre-training. Language modeling, an NLP task where the goal is to predict the probability of a given sentence occurring in a language, is one such task that does not require labeled data and can therefore be learned in an unsupervised way. Learning the task of language modeling

also creates vector representations for words since the model needs to predict probabilities for all words in the vocabulary. Unsupervised training means that vast amounts of unlabeled data can be used with the objective of creating similar representations to words that appear in similar contexts, relying on the distributional hypothesis [14].

While language modeling can be used to create word representations, it is computationally rather expensive due to its requirement of outputting a probability distribution over the whole vocabulary. Several algorithms such as Word2Vec [15] or GloVe [16] have been developed for obtaining similar distributed representations as language modeling allows to obtain but in a more efficient way. However, there are still limitations to these approaches – they allow learning one representation per surface form of a word. For example, the word *party* would receive one common representation, while it would often be necessary to distinguish between its senses.

Advancements in this regard have been made with the introduction of contextualized word representations such as Embeddings from Language Models (ELMo) [17]. ELMo creates a representation for a given token as a function of all tokens in the input sequence. This means that the representation of a token depends on its current context and is not fixed in a learned embedding matrix, allowing to dynamically change the representation for different word senses and according to smaller changes in context. A further improvement to contextualized representations was made with Bidirectional Encoder Representations from Transformers (BERT) [3]. One of the main differences compared to ELMo is BERT's deep bidirectionality which means that context from both left and right sides can be more effectively incorporated into the representations.

3.2 Tokenization

In the previous subsection, the word *token* occurred, signifying some unit with meaning in a sequence of text. However, there are multiple ways of breaking a sequence into tokens, and the choice of tokenization method is often connected to the choice of pre-trained embeddings. To map parts of a given input sequence to entries in an existing vocabulary with corresponding vector representations, we need to use the same tokenization method as was used when the pre-trained representations were first learned.

An inherent problem when working with natural language texts is the possibly vast vocabulary. A single word can take on numerous slightly different forms, words are often compounded or abbreviated, and languages evolve in time. Therefore, using a simple approach such as separating text into tokens by whitespaces for learning a fixed size vocabulary from a given training corpus is problematic in several ways. Firstly, the size of the vocabulary can grow very large, leading to computationally expensive models. Secondly, models using such a vocabulary are bound to encounter unseen word forms in new data. These forms do not have a corresponding representation and must be represented by a generic "unknown token" representation.

Byte Pair Encoding (BPE) [18] is a tokenization method proposed to alleviate this problem. Its core idea is to build a fixed-size vocabulary of character sequences that may be either full words, subwords, or single characters. The method is based on a data compression technique with the same name. It works by iteratively merging frequent character combinations into one vocabulary unit until reaching a predefined vocabulary size limit. As a result, frequent character sequences will be represented as a whole in the vocabulary, while rarer sequences are represented by their parts.

An extension to regular, character-based BPE is byte-level BPE [19]. When the training corpora are large and heterogeneous, as is often the case when pre-training large language models, all the various single Unicode characters can make up a significant part of the vocabulary. This variant makes the base vocabulary much smaller since it considers a byte as a basic unit instead of a character.

3.3 Neural Networks

In this thesis, we use model architectures based on neural networks to tackle HS code classification. This subsection is based on chapter 6 of the book *Deep Learning* by Ian Goodfellow, Yoshua Bengio, and Aaron Courville [20].

Feed-forward neural networks consist of an input layer, one or more hidden layers, and an output layer. Such networks are called feed-forward because of how information flows through them – from input to output, in one direction only. Hidden layers, the basic building blocks of neural networks, usually consist of a determined number of hidden units. Each hidden unit has its corresponding vector of weights and a bias value, which make up the trainable parameters of the neural network.

Since computations in neural networks are performed in batches, the weight vectors of all units within a hidden layer can be denoted as a single matrix W and the corresponding biases as a single vector b. For a given input vector x, a hidden layer performs a linear transformation followed by a non-linear activation function g, and outputs a hidden representation vector h:

$$h = g(W^T x + b).$$

The objective of a neural network is to approximate a function that maps inputs to respective outputs, e.g., text to classes in the case of HS classification. To learn suitable values for a neural network's parameters for approximating a function, we need to define a loss function and an optimization method. The loss function determines how exactly the mismatch between the model's outputs and actual labels is calculated. This, in turn, affects how the parameters are tuned since neural networks use gradient-based learning where each parameter is tweaked during back-propagation according to how it contributes to the total loss, using the chosen optimization method.

Recurrent neural networks (RNNs) are a type of neural networks that are able to model the connections between different positions in the input. Thus, they are suitable for sequential inputs such as text or time series data. RNNs treat the input as a sequence of timesteps and allow cyclical connections, as opposed to feed-forward networks where information flows only in one direction. The hidden representations for each position in the input are calculated sequentially, with the output of a given timestep contributing to the input of the next timesteps. In applications where context from both left and right sides of the input is available, bidirectional RNNs [21] are used, meaning that one forward and one backward RNN are applied to an input, and the representations from both are combined.

A drawback of regular RNNs is their inability to capture long dependencies, i.e., connections between input positions far away from each other, due to vanishing or exploding gradients during back-propagation. This problem has been addressed with architectures such as Long Short-Term Memory (LSTM) [22] which consists of memory blocks with dedicated non-linear activation gates. The mentioned gates help the model to decide when to overwrite the information from previous timesteps with new inputs and when to preserve it.

3.4 Attention Mechanism

Although RNNs can capture dependencies between different input positions, which is highly beneficial when working with textual data, some limitations remain. Firstly, even further developments such as LSTMs still suffer from forgetting very distant yet relevant information. Secondly, RNNs are sequential, and therefore one training example cannot be processed in parallel, which becomes a computational bottleneck for longer sequences.

The attention mechanism was proposed in 2014 as a novel method for neural machine translation [23] to address these limitations. In the introducing paper, attention was described as building a context vector by creating a weighted sum of hidden representations. More weight is assigned to representations of tokens that are more relevant or important for the previous hidden state created by the decoder. The scores for each token (that are later converted into probabilities for calculating the weighted sum) are computed using an *alignment model*.

More specifically, the encoder creates a hidden representation h for each token using a bidirectional RNN. Then, the decoder generates an output sequence, at each step using the hidden state and output from its previous timestep, and a context vector c. The attention mechanism allows having a different context vector c_i for each timestep i, since for each step the weighted sum of the encoder's representations is computed again, considering the last hidden state of the decoder:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j.$$

In this equation, α_{ij} is the weight applied to the hidden representation h_j at the i-th timestep in the decoder, and T_x is the last timestep as x represents the length of the input sequence. The weight α_{ij} is computed using the softmax function, which maps its inputs to range 0…1 such that the sum of all outputs equals 1:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}.$$

Here, e_{ij} represents the output of the alignment model, which is based on the previous hidden state of the decoder s_{i-1} and the hidden representation h_j of the token at position j, produced by the encoder:

$$e_{ij} = a(s_{i-1}, h_j).$$

Here, a is the alignment model, for which several options have been proposed. In the previously mentioned article by Bahdanau [23], the alignment model was a single-layer feed-forward network:

$$a(s_{i-1}, h_j) = v_a^T \tanh\,(W_a s_{i-1} + U_a h_j),$$

where v_a is a trainable weight vector and W_a and U_a are trainable weight matrices.

The paper demonstrated that not only is the proposed architecture better at translating long sequences than traditional encoder-decoder models without attention, but it also brought a significant improvement in performance for shorter sequences. It was emphasized that building a new alignment for each decoder step had a strong positive impact.

In another article on attention-based neural machine translation [24], Luong et al. described other types of alignment models. One of the types was the *general* alignment score function:

$$score(s_i, h_j) = s_i^T W_a h_j$$

where W_a is a trainable weight matrix, s_i is the current hidden state of the decoder and h_j, as previously, the hidden representation of the token at position j from the encoder. This variant has also been referred to as *bilinear* attention [25], which is the term we will use in this thesis as well.

3.5 Transformers

Based on the idea of attention and parallel processing, a neural network architecture named the Transformer was developed [4]. It was originally intended for sequence-to-sequence problems such as machine translation and comprised of an encoder and a decoder. A core component of the architecture is *multi-head attention* which eliminates the need for using any recurrence in the network, making the training process more parallelizable.

Multi-head attention means that an attention function is applied not only once for a given input, but h times where h indicates the number of heads in the multi-head at-

tention block (illustrated in the right part of Figure 1). The dimensionality of each attention head is reduced to ensure that the dimensionality of the input is maintained and to limit computational costs. To do so, the inputs are passed through linear layers, which project them to lower dimensionality before applying the attention function of each head. Finally, the results are concatenated into a single output of the whole multi-head attention block. This output is passed through another linear layer.

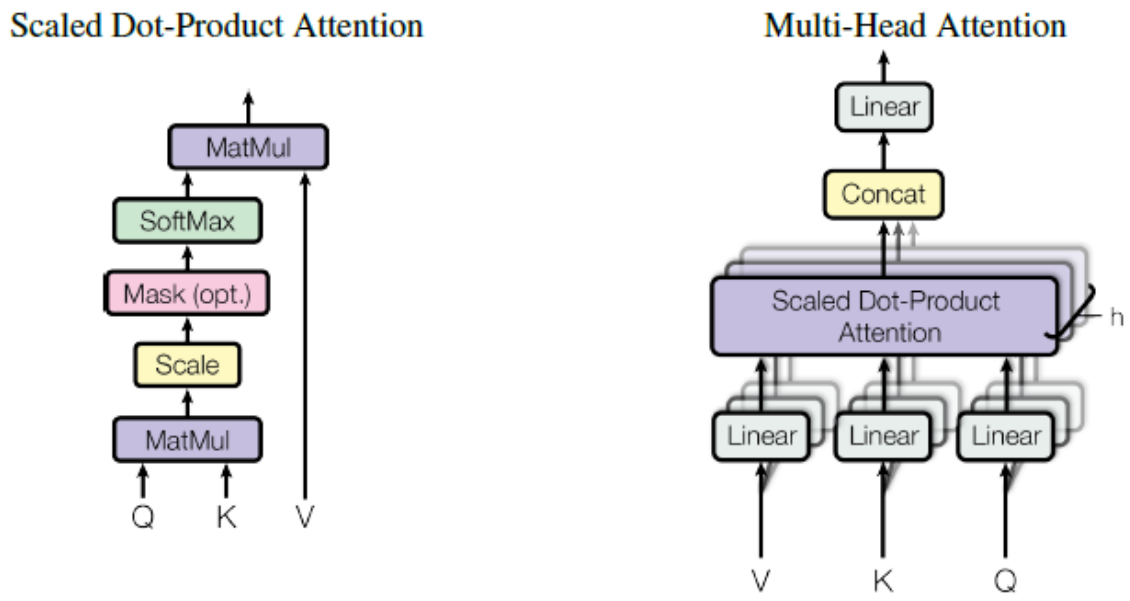

Figure 1. Scaled dot-product attention mechanism and multi-head attention as implemented in the Transformer architecture. [4]

While there are several types of attention functions, the Transformer uses a variant called scaled dot-product attention, illustrated in the left part of Figure 1. The inputs to the attention function are called *queries*, *keys*, and *values*. The attention mechanism aims to determine the level of compatibility between the queries and the keys and calculate a weighted sum of the *values* where the weights describe this compatibility.

After projecting inputs down through linear layers, we have a matrix of queries Q, a matrix of keys K, and a matrix of values V (a vector for each input, combined into matrices for batch-processing). As shown in the left part of Figure 1, a dot product between Q and K is then calculated. The result is scaled by $\frac{1}{\sqrt{d_k}}$ where d_k denotes the dimensionality of the query and key matrices. This results in a vector of scores for each

entry in the batch. These scores are turned into weights using the softmax function, and the output of the attention function is a weighted sum of the value vectors in V:

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

Which elements are used as queries and which as keys and values depends on where the attention function is applied. In sequence-to-sequence models for which the Transformer was originally proposed, multi-head attention is used between the encoder and the decoder. In this case, query inputs are taken from the previous decoder layer, while key and value inputs come from the last layer of the encoder.

Elsewhere in the Transformer, multi-head attention is used in *self-attention* layers in both the encoder and the decoder. The name self-attention comes from the fact that all inputs to the attention function come from the same place. More specifically, for self-attention, both the query and the key-value pairs are taken from the previous layer's output. In the self-attention layers of the decoder, each token can only "attend" to tokens to the left of it; therefore, the rest are masked out and are not given any weight (as denoted by "optional mask" in Figure 1).

The encoder and the decoder of the Transformer model are both stacks of identical layers where the output of one layer is the input to the next. Although the structure of the layers is the same, the parameters are not shared. Each layer in the encoder includes a multi-head attention block and a feed-forward network, both followed by layer normalization. The decoder layers additionally contain the previously mentioned encoder-decoder attention blocks.

3.6 Transformer-based Language Models

The idea of the Transformer has been transferred to language modeling and representation learning. Unlike the original encoder-decoder structure, language models use only the encoder part of the Transformer. Due to the success of such language modeling approaches on NLP benchmark tasks, many works have been published to further improve these architectures or make them more efficient.

BERT was presented in a 2019 paper as a language representation model based on Transformers [3]. The article described two variants of BERT, *base* and *large*, where the former consists of 12 and the latter of 24 layers. The dimensionalities of hidden

layers and the number of attention heads also differ. To learn general representations, BERT models are pre-trained on unlabeled data using two pre-training objectives – masked language modeling (MLM) and next sentence prediction (NSP).

The MLM task means that, given an input sequence of tokens, a randomly chosen fraction of the tokens is masked out, and the model is asked to "fill the gaps" by predicting the missing tokens. The NSP task includes presenting the model with two input sequences and asking it to predict whether one sentence follows the other in the source text or not.

The authors of BERT argue that using the MLM task instead of traditional forward or backward language models allows the model to achieve deep bidirectionality instead of a shallow one that results from a concatenation of forward and backward contexts. The reason for adding the NSP task to pre-training was that many tasks where word representations are used require modeling connections between a pair of sentences.

Soon after the publication of BERT, an optimized version of it was proposed, named RoBERTa, which stands for Robustly Optimized BERT Pre-Training Approach [26]. Its authors used the same model architecture but applied several changes to the pre-training process. The main modifications were as follows:

1. changing the masking strategy from static to dynamic for the MLM task (instead of masking tokens once during preprocessing, a new masking pattern is created every time the model sees an input sequence);
2. removing the NSP task;
3. increasing the batch size;
4. using byte-level instead of character-level BPE;
5. training on more data for a longer time.

Both BERT and RoBERTa models have been made available to the community, making it possible for us to use them as a basis for our classification models. As the authors of RoBERTa showed that their design choices improved performance on benchmark tasks significantly, the pre-trained RoBERTa model is the base model of choice in this thesis, to which we add new, task-specific layers.

4 Data

This chapter describes the dataset used in this thesis and the methods of preprocessing applied.

4.1 Splits

The dataset, provided to us by our partner, includes a total of 19,680,566 product descriptions labeled with respective HS codes. We split the dataset into training, development, and two test sets with sizes making up 80%, 10%, 5%, and 5% of the total dataset. The size of each split is shown in Table 3. We separate two test sets instead of one due to the nature of the project, which includes constant development and comparisons of models. The second test set is not used for evaluations in this thesis, as it will only be used later in the project.

As the classes are highly imbalanced, we ensure that their distribution in each split is similar, using the *stratified* option in the *train_test_split* function from the scikit-learn library [27]. Since some classes are represented by very few examples in the full dataset, not all classes that exist in the training set also exist in the development and/or test set.

Table 3. Number of examples in each split of the dataset.

train	**dev**	**test1**	**test2**	**total**
15,744,451	1,968,057	984,029	984,029	19,680,566

Table 4 presents a selection of statistics about the distribution of classes in each split and in total for each level in the HS nomenclature. For the HS2 level, a more detailed overview of all chapters and their supports (i.e., number of examples in a given chapter) is provided in appendix II on page 62.

Table 4. The number of classes and the mean, minimum, median, and maximum supports in each split and in total.

		train	dev	test1	test2	total
HS2	**classes**	55	55	55	55	55
	mean	286,263	35,783	17,891	17,891	357,828
	min	1439	178	87	86	1790
	median	104,669	13,084	6539	6541	130,833
	max	2,981,122	372,643	186,329	186,338	3,726,432
HS4	**classes**	752	749	742	742	752
	mean	20,937	2,628	1,326	1,326	26,171
	min	2	1	1	1	2
	median	3874	489	257	257	4843
	max	417,038	52,130	26,068	26,068	521,304
HS6	**classes**	3226	3181	3127	3111	3226
	mean	4880	619	315	316	6101
	min	2	1	1	1	2
	median	839	109	57	58	1048
	max	289,329	36,168	18,083	18,083	361,663

On the HS2 level, the dataset contains 55 classes. The largest class is chapter 84 (*Nuclear reactors, boilers, machinery and mechanical appliances; parts thereof*) with 3,726,432 examples in the dataset, 18.9% of all examples. On the HS4 level, the dataset contains 752 classes. The largest class is heading 7318 (*Screws, bolts, nuts, coach screws, screw hooks, rivets, cotters, cotter pins, washers, incl. spring washers, and similar articles, of iron or steel (excl. lag screws, stoppers, plugs and the like, threaded)*) with 521,304 examples making up 2.65% of all examples. On the HS6 level, the dataset contains 3226 classes. The majority class is the subheading 392690 (*Articles of plastics and articles of other materials of heading 3901 to 3914, n.e.s.*) with 361,663 examples, 1.84% of all examples.

Table 4 illustrates that the difference between the mean and median support per class is large, which indicates that the distribution is skewed – there are many classes with rather few examples and few classes with very many examples. It also shows that already on the HS4 level, there are classes represented by only a few examples. The distributions are visualized in appendix III on page 65.

Due to the relatively large size of the development set, we further split it into two non-overlapping parts. We reduce the sizes of the two resulting datasets by randomly undersampling large classes and name these subsets *dev_small* and *dev_large*. Undersampling enables us to preserve small classes while reducing the total size to speed up the development process. In dev_small, we keep a maximum of 150 examples per HS6 class, making the total size 230,876 examples. In dev_large, we keep a maximum of 1000 examples per HS6 class, making the total size 611,262 examples. The sizes and class counts for both subsets are presented in Table 5.

Table 5. Number of classes and total size of dev_small and dev_large.

	dev_small	**dev_large**
HS2	55	55
HS4	740	749
HS6	3081	3181
size	230,876	611,262

We use dev_small for evaluations during training and hyperparameter tuning, and dev_large for first comparisons between trained models and error analyses.

4.2 Preprocessing

There are cases in the dataset where the same product description occurs with multiple labels (HS codes). The labels can be completely different or match up to some level (e.g., 090111 vs. 090220). We remove all such ambiguous duplicates from the training split before training our models, as we do not have the resources to verify which, if any, of the multiple assigned labels is correct. This removal policy decreases the size of the training split by 491,971 examples. All classes are still preserved, meaning that all those with only a few examples had some descriptions unique to a particular class.

Although several previous works on HS classification perform strict preprocessing on product descriptions, such as removing all non-alphabetic characters [1], [8], [10], punctuation [1], [2], predefined stop words [1], [8]–[10] or contents within brackets [8], we choose a different approach. Our intuition is that numbers might be helpful for classification since they can be the primary or only distinguishing element between several classes. Some examples are the composition of fibers in textiles, as in *Sewing thread, containing >= 85% cotton by weight (excl. that put up for retail sale)* versus *Sewing thread, containing predominantly, but < 85% cotton by weight (excl. that put up for retail sale)* or measurements of furniture, such as *Wooden furniture for offices, of <= 80 cm in height (excl. desks and seats)* versus *Wooden furniture for offices, of > 80 cm in height (excl. cupboards)*. In such cases, removing the numbers during preprocessing creates more ambiguous duplicates, as the only differing part is removed from the text while labels remain the same. Furthermore, sequences of mixed alphanumeric symbols might indicate model names, also possibly useful for classification.

5 Experiments

This chapter describes the setup of experiments, the architecture of proposed and baseline models, and the methods used for evaluation.

5.1 Setup

The classification models developed in this work are built on pre-trained Transformer models from the Hugging Face Transformers library [28]. The library provides tools for loading and fine-tuning various pre-trained models that can be extended to add custom layers or other functionality. We use the PyTorch library [29] for model customization. Additionally, we use the Weights and Biases library [30] to keep track of our experiments, create intermediate reports, manage hyperparameter tuning runs, and log metadata related to each run.

Due to the size of our datasets, we need an efficient way to process the data. We use the Hugging Face Datasets library[5], which provides two significant benefits. Firstly, it performs memory mapping, meaning that large datasets do not have to fit into RAM at once. Secondly, it performs caching, meaning that the same processing on a certain dataset is never done more than once.

Experiments are carried out in the Rocket Cluster of the High Performance Computing Center of the University of Tartu [31]. The more resource-demanding runs are performed on the GPU nodes with NVIDIA Tesla V100 GPUs. Other runs are performed on CPU nodes.

5.2 Transformer-based Classification Models

We experiment with two types of classification models based on the Transformer architecture for HS classification. The first type is a flat model with one classification head trained to predict the full HS6 directly. The second type is a hierarchical model with three classification heads, one for each HS level. The second type further branches into two subtypes with different attention layers.

[5] https://huggingface.co/docs/datasets/

5.2.1 Types of Classifiers

In a survey of hierarchical classification [32], Silla and Freitas have described three approaches to classification where the classes form a tree-structured hierarchy. First, there are *flat* classifiers that predict classes in the leaf nodes and ignore the hierarchical structure (illustrated in Figure 2). Our flat classifiers, described in the following sections, belong to this type.

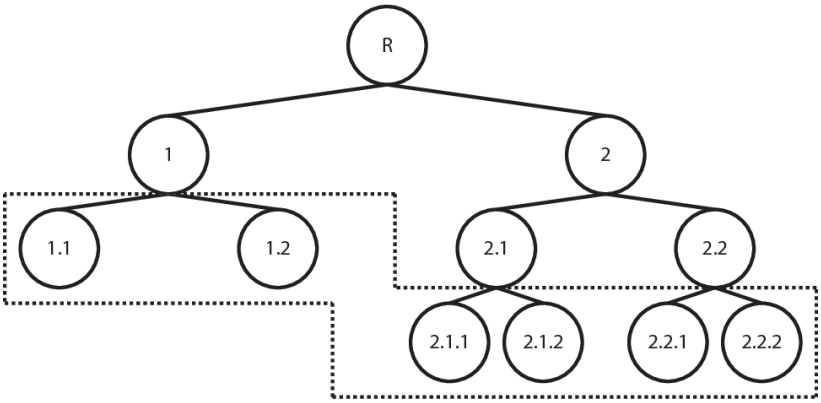

Figure 2. Structure of the flat classifier approach [32]. It must be noted that unlike in this example, in our task of HS classification, all leaf nodes are on the same level (i.e., at the same distance from the root node). Even if an HS4 class does not branch further into several HS6 codes, it is represented as HS4 + '00' on the HS6 level.

Second, there are *local* classifiers where a set of classifiers is used – either one binary classifier per node, one multi-class classifier per parent node, or one multi-class classifier per level (illustrated in Figure 3). Third, there are *global* classifiers where similarly to the flat approach, a single classifier is trained, but in this case, it learns the hierarchical structure and can also predict classes from intermediate levels, not only leaf nodes.

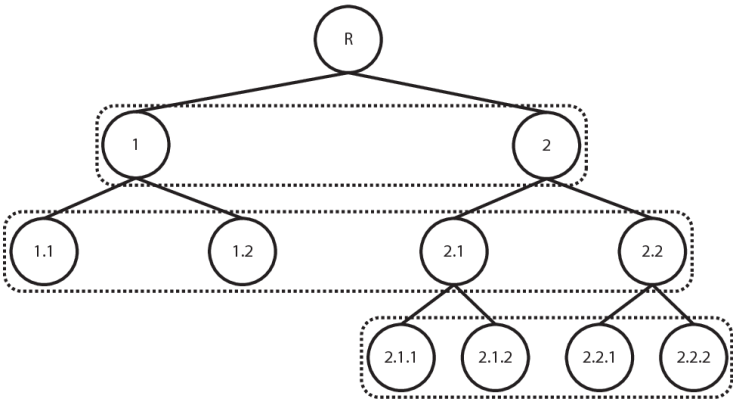

Figure 3. Structure of the local classifier per level approach [32].

According to the definitions from Silla and Freitas, our hierarchical approach can be considered a fusion of local classifier per level and global classifier methods. It is a local classifier in the sense that a classification decision is made on each level. However, we do not train three independent models with each their own training datasets; instead, we train a single model that includes one decision point on each level and passes through the levels in a top-down manner.

5.2.2 Flat Classifier

Our flat classification model consists of a pre-trained RoBERTa-*base* encoder and one classification head. The suffix *base* refers to the size of the encoder, as the authors of RoBERTa proposed two variants, *base* with 12 layers in the encoder and *large* with 24 layers in the encoder. Although the *large* variant was shown to produce better results on benchmark tasks, it is also much slower to train due to having more parameters (~125 million in *base* vs. ~355 million in *large*). Because of this, we use the *base* variant in our experiments.

The authors of BERT have described two approaches for using their proposed models for downstream tasks – the fine-tuning approach and the feature-based approach [3]. With additional layers on top of the pre-trained encoder, the two differ in how the model is trained. In the fine-tuning approach, all parameters, including those of the encoder, are jointly trained when training on the labeled dataset of the downstream task. The feature-based approach, on the other hand, means that the encoder representations are used as fixed features. In this approach, only the parameters in the additional layers are trained during supervised training. Since fine-tuning was shown to

produce somewhat better results and the feature-based approach requires additional experimentation with which encoder layers to use for extracting the features, we decide to use only the fine-tuning approach in this work.

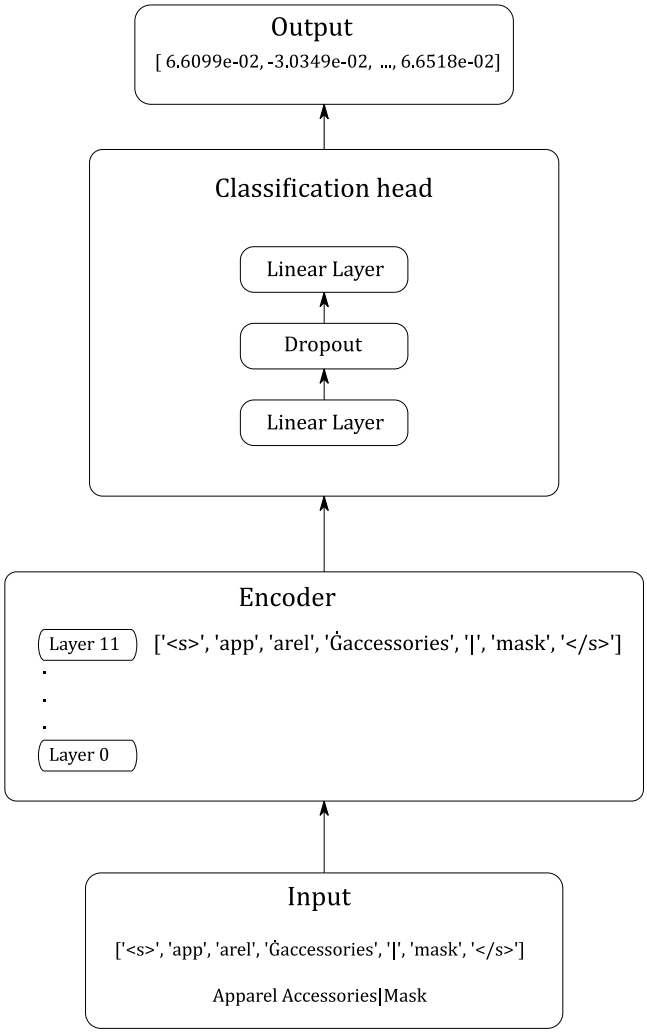

Figure 4. Architecture of the flat Transformer-based classifier. The tokenization of an example input description is shown. The outputs are scores for each HS6 class.

We use the RobertaForSequenceClassification class from the Hugging Face Transformers library[6], as it has already been implemented as a generic class for sequence classi-

[6] https://huggingface.co/transformers/v4.3.3/index.html

fication. It includes the RoBERTa encoder and a classification head consisting of a linear layer, a dropout layer, and another linear layer that projects the hidden representations to vectors with as many elements as classes in the dataset. These vectors are passed as input to the loss function, which calculates the cross-entropy loss between the predictions and the true labels. The structure of the model is illustrated in Figure 4.

The input to the classification head is the hidden representation of the special <s> token from the last layer of the encoder. The <s> token is prepended to each example during tokenization. It has been used as an aggregate representation of the input in previous publications, including the original papers introducing BERT and RoBERTa [3], [26].

5.2.3 Hierarchical Classifiers

As an alternative to the flat model, which directly predicts the complete HS code, we explore possibilities for modeling the hierarchical structure of the nomenclature.

From the local classifier types described by Silla and Freitas [32], we prefer the classifier per level approach. This approach helps us limit the number of models to be trained and avoid the overhead of managing and storing many models. As we have 55 and 752 classes on HS2 and HS4 levels, respectively, using the classifier per parent node approach would mean training 808 models. Instead of this, we build a single model that includes three classification heads and passes the predictions from previous layers to the next.

The main structure of the hierarchical classifiers is illustrated in Figure 5. As mentioned before, the model includes a classification head for each level of the HS. The inner architecture of the classification heads is identical to that in the flat model.

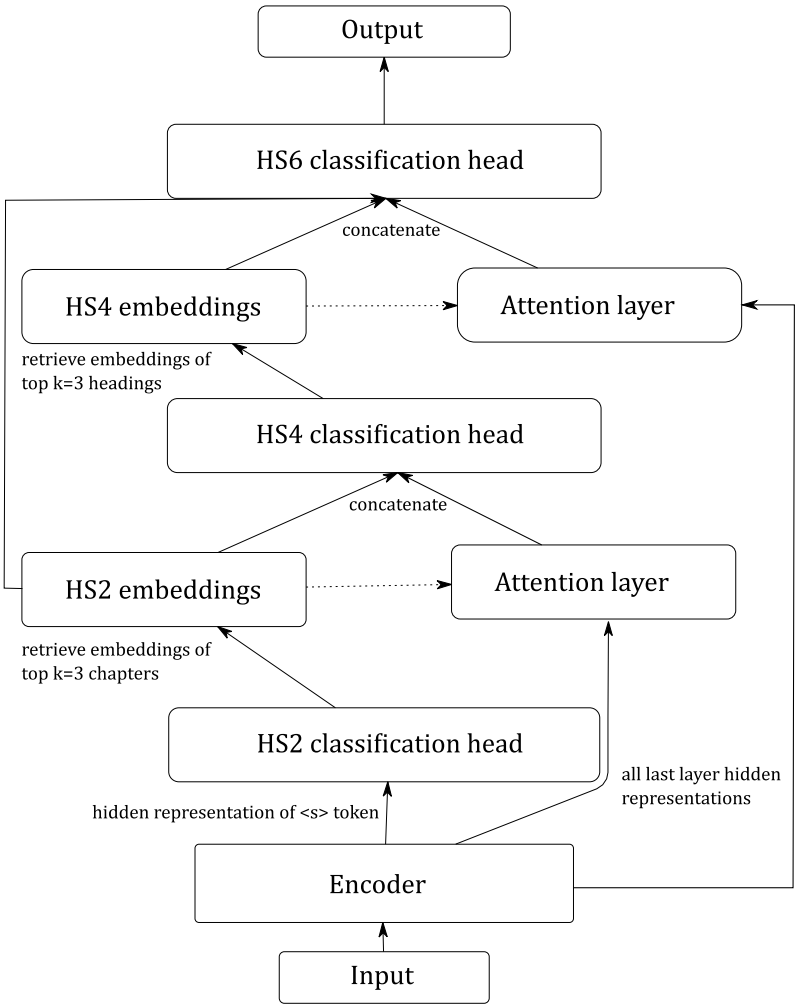

Figure 5. Architecture of the hierarchical Transformer-based classifier. The dotted lines indicate that in the version using self-attention, the inputs to attention layers are only the encoder representations, not the class embeddings.

Additionally, the hierarchical classifiers include attention layers between HS2-HS4 and HS4-HS6 classification heads. The type of attention function is the discriminative part between the two subtypes of our hierarchical classifiers – one subtype has bilinear attention layers while the other includes self-attention layers. The self-attention layers are identical with those in the RoBERTa encoder, i.e., the queries, keys, and values are all computed by multiplying the same input (in this case, the hidden representations from the last layer of the encoder) with respective weight matrices.

The intuition behind extra attention layers is that we expect the most important parts of product descriptions to be different depending on the level. For example, making an HS2-level prediction does not usually require precise details about a product, and we expect that the more general tokens contribute the most on that level. However, on the HS6 level, minor details can be critical as the only distinguishing elements between neighboring classes. A similar approach has been used by Yang et al. [33] for document classification, where separate attention layers were built for the word and sentence levels.

Like in the flat model, the input to the first classification head is the hidden representation of the <s> token from the last layer of the encoder. Then, if the model is in training mode, the embedding for the *correct* HS2 class is retrieved from the respective embedding layer. On the other hand, if the model is performing inference and the true label is unknown, then the normalized probability-weighted sum of the embeddings of the top three predictions from the HS2 classification head is used instead.

In the model type using bilinear attention layers, the resulting embedding (or weighted sum of embeddings) is passed into the first attention layer. The embedding is used as a query and the hidden representations as keys. In the model using self-attention layers, the hidden representations are used as queries, keys, and values, and the output is the reweighted representation of the <s> token. The purpose of the attention layer is to create a newly weighted aggregation of all hidden representations from the encoder's last layer.

The output of the attention layer is concatenated with the HS2 embedding, and the result is fed into the HS4 classification head. The same process is repeated, except that in the bilinear version, the next attention layer uses the HS4 embedding as a query. The input to the final classification head is a concatenation of three elements: the newly weighted input representation, the embedding from the HS2 level, and the embedding from the HS4 level.

During training, local losses are computed on each level, using the outputs of the classification heads and the true labels of the corresponding levels. The total loss is a sum of all three local losses, allowing the weights in each classification head to be modified both according to their output and how it affects the following layers.

5.3 Baseline Models

To compare the Transformer-based models to other approaches, we use the fastText classifier[7] as a baseline. Its authors have shown that their classification approach produces results comparable with deep learning models while being faster to train and evaluate [34].

As shown in Figure 6, fastText classifiers have a shallow architecture, consisting of an input layer, one hidden layer, and an output layer. The input text is transformed into a bag-of-features representation by taking an average of all features. These features include the embeddings for all words and optionally for all word n-grams, which help capture the local order of words. The embeddings are obtained from a trainable lookup matrix, where the key to being fast and efficient lies in hashing. Using the so-called hashing trick, the matrix is not required to store embeddings for each possible n-gram explicitly. Instead, fastText uses an embedding matrix of size *b*, a hyperparameter that denotes the number of buckets, and each n-gram is assigned to a bucket in the matrix.

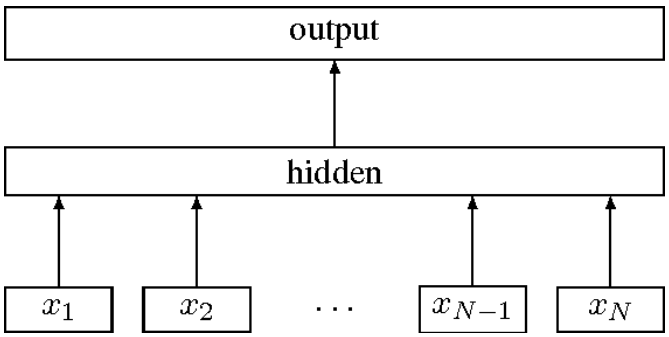

Figure 6. FastText model architecture. The input sequence includes N features, x1, ..., xN [34].

Similarly to the proposed models, we train and evaluate fastText in two versions: flat and hierarchical. However, in this case, the hierarchical version is structured according to the local classifier per parent node approach (illustrated in Figure 7) as the training speed allows us to do so. In this approach, one model is trained to predict the headings under chapter 90, another to predict those under chapter 91, and so on.

7 https://github.com/facebookresearch/fastText/tree/master/python

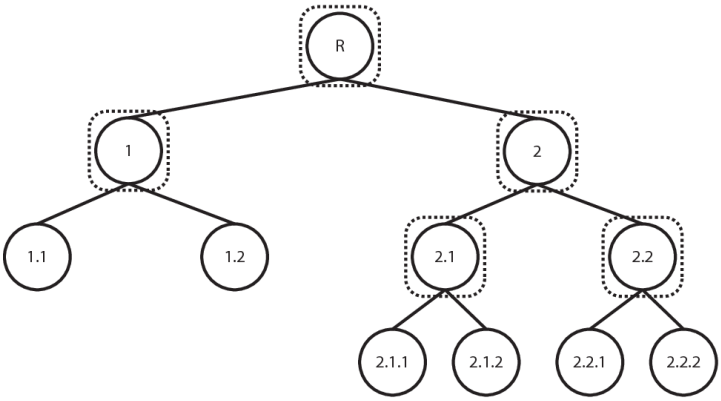

Figure 7. Structure of the local classifier per parent node approach [32].

During inference, the prediction from the current level determines which model will be used on the next level. The predictions from all levels are collected in a top-down manner.

5.4 Metrics

We use accuracy as the metric to decide which classification model performs best on our held-out evaluation sets. Accuracy is defined as

$$accuracy(y, \hat{y}) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} 1(\hat{y}_i = y_i)$$

where $\hat{y}_i$ is the predicted label of the i-th example and y_i is the true label of the i-th example[8]. Accuracy measures the fraction of correctly predicted elements of all elements.

Although we base our comparisons and decisions on the accuracy score, we additionally track the macro-averaged f1-scores of the models. The f1-scores help us understand the models' average performances across all classes, regardless of the support of each class in the dataset, as macro-averaging considers all classes equally important. The macro-averaged f1-score is calculated by finding the f1-score for each class and

[8] https://scikit-learn.org/stable/modules/model_evaluation.html#accuracy-score

taking their unweighted mean. The f1-score is defined as the harmonic mean of precision and recall[9]:

$$f1 = \frac{2(precision*recall)}{(precision+recall)}.$$

Precision and recall are themselves indicators of a model's ability to, firstly, avoid false positives (i.e., mistakenly assigning a label to examples from other classes) and secondly, to avoid false negatives (i.e., mistakenly not assigning a label to examples from that class). The f1-score is a metric that considers both as important.

5.5 Experiments

This section describes the comparative experiments performed to study the impact of preprocessing, dataset sampling, and hyperparameter tuning on HS classification results.

5.5.1 Preprocessing

Section 4.2 discussed possible ways of preprocessing the data. To determine whether our intuition about the benefits of less preprocessing is grounded, we run an experiment by training models with three variants of preprocessing. In the first approach, we do not apply any preprocessing, i.e., we train and evaluate the model on raw product descriptions. In the second approach, we perform only lowercasing. In the third approach, we apply lowercasing plus removing a set of stop words, single-character tokens, punctuation, and numbers longer than three digits.

All three models are trained using the flat approach – a pre-trained RoBERTa encoder and a classification head with two linear layers. We train the models for ten epochs on a reduced training set with a maximum of 200 examples per HS6 class. All models are trained using the AdamW optimizer [35] and with the same default hyperparameters, listed in appendix IV on page 67. We use dev_small for monitoring progress during training and for possible early stopping when the HS6-level accuracy has not improved for two epochs.

[9] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html#sklearn.metrics.f1_score

The results from evaluation on dev_large are presented in Table 6. The table illustrates that the differences in accuracy are more visible on the more specific levels (HS4 and HS6) than on the HS2 level. The first approach with no preprocessing gives the lowest scores. This is possibly due to the tokenizer's case sensitivity as the dataset includes many descriptions fully in uppercase. The second approach with only lowercasing ranks first. The ranking indicates that although lowercasing helps, stronger cleaning can erase important information and negatively affect performance. However, it must be noted that the third approach still performs better than no preprocessing, which also confirms the case sensitivity of the tokenizer.

Table 6. Accuracy scores on dev_large. Since the trained models predict the entire HS6 directly, the accuracy scores for HS2 and HS4 levels are calculated using the respective parts extracted from the predicted HS6 code.

Preprocessing method	**HS2**	**HS4**	**HS6**
None	0.8409	0.7044	0.4915
Only lowercasing	**0.8491**	**0.7202**	**0.5021**
Lowercasing and removals	0.8449	0.7112	0.4923

We acknowledge that the preprocessing performed here is only one option out of many. There might very well exist combinations of preprocessing steps that help distill the informative parts from the meaningless noise. One of the benefits of that would be decreasing the number of tokens to be processed, speeding up training and inference. However, we do not aim to go in-depth to such experiments here. As a conclusion of this comparison, we use the lowercasing approach in all further experiments.

5.5.2 Dataset Sampling

Considering the class imbalance in our dataset, we pose two questions related to managing the total size of the dataset and making the class distributions more balanced. Our questions are as follows:

1. How does using a subset of the majority classes (and therefore decreasing the gap between the supports of minority and majority classes) affect model performance?
2. Does oversampling the smallest classes while preserving the entirety of the majority classes have a positive effect on model performance?

The intuition behind the first question is that, as we have a large number of examples from a few classes and a small number of examples from most classes, then perhaps reducing the size of the former helps to perform better on the latter, speeding up training at the same time. An important aspect to consider is whether decreasing the variance of the majority classes (by ignoring a part of the examples) is worth it. In other words, if the largest classes tend to prevail over the smaller ones strongly, then undersampling them would be a reasonable approach. However, if this is not a significant problem or it turns out that using fewer examples significantly decreases accuracy due to performing worse on these large classes, then undersampling should be avoided.

To answer the first question, we train two flat models on differently sampled data. The first model (with sampling method *None*) is trained on the original training set. The second (sampling method *Max 800*) is trained on a randomly undersampled version of the training set. We use the Imbalanced-learn library [36] to perform dataset resampling in this and also further experiments.

The undersampled version includes a maximum of 800 examples per HS6 class, making its size ~11% of the full training set. The limit of 800 examples was chosen according to the median of HS6 class supports in the training set. The results on dev_large are illustrated in Table 7.

Table 7. Accuracy and f1-scores on dev_large for flat models trained on differently sampled datasets.

Sampling method	**HS6 accuracy**	**HS6 f1-score**
None	0.7518	0.4877
Max 800	0.677	0.5756
Min 1000	**0.7555**	**0.6731**

The results show that although the average performance per class (as indicated by the macro f1-score) did significantly improve with the *Max 800* method, there was also a strong decline in accuracy. We conclude that decreasing the variance of large classes to such an extent is harmful.

This raises the second question, as we are now interested in making the model better at predicting the minority classes while not reducing the majority classes. We create another version of the training set by randomly oversampling the minority classes such that there are at least 1000 examples per class in the training set (sampling method *Min 1000*). The majority classes are kept in their original sizes. This method increases the size of the training set by ~8%.

When comparing the results of *None* and *Min 1000* sampling methods in Table 7, we can see that the latter shows a somewhat better accuracy score and a significantly better f1-score on the HS6 level. Furthermore, the f1-score is ~10% better than with the *Max 800* approach, which was already better than *None*. In conclusion, we will use the partly oversampled (*Min 1000*) training set in further experiments.

5.5.3 Hyperparameter Tuning

For both the Transformer-based models and the baseline models, we perform limited hyperparameter tuning. The choice of hyperparameters that are experimented with is narrow due to time constraints; thus, we select those we expect to be more relevant.

When running the tuning experiments for Transformer-based models, we use the same reduced training set as in the preprocessing experiment to limit training time. This set includes a maximum of 200 examples per HS6 class, meaning that classes larger than that are undersampled. When running the experiments for the baseline, we can use the full training set. For evaluations, we use the dev_small set.

Table 8 and Table 9 present the hyperparameters and corresponding search spaces used in the experiments for flat and hierarchical Transformer-based classifiers. With a few exceptions, noted in appendix IV on page 67, the remaining hyperparameters are set to default values as described in the Hugging Face documentation[10].

[10] https://huggingface.co/transformers/v4.3.3/main_classes/trainer.html#transformers.TrainingArguments

Table 8. Values used for tuning the hyperparameters of the flat Transformer-based classifier.

Hyperparameter	**Values**
Classifier hidden size	256, 512, 768
Learning rate	1e-5, 2e-5, 3e-5, 5e-5
Learning rate scheduler	Constant with warmup, linear

Table 9. Values used for tuning the hyperparameters of the hierarchical Transformer-based classifier.

Hyperparameter	**Values**
Classifier hidden size	256, 512, 768
Learning rate	1e-5, 3e-5, 5e-5
Embedding size	128, 256

From the experiment with the flat classifier, we conclude that the hidden size of the classifier does have an impact on performance, with larger sizes leading to better results. We also find that it is beneficial to increase the learning rate from the initial value 1e-5. Lastly, the experiment shows that when the learning rate is higher, it is better to use a linear scheduler that includes not only warm-up but also decay after reaching the peak.

The experiment with the hierarchical classifier also confirms the contribution of learning rate and the hidden size of the classifier, showing that larger values produce better results. However, increasing the size of the embeddings of HS2 and HS4 classes did not bring significant improvement. A graphical overview of the tuning runs and scores for both model types is presented in appendix V on page 69.

With the fastText baseline models, we can work with more hyperparameters in a somewhat broader range as training takes much less time. For the hierarchical version,

we first tune only the HS2-level classifier with values presented in Table 10 to determine a reasonable scale for the learning rate and find out whether using character n-grams is beneficial.

Table 10. Values used for tuning the fastText HS2-level model.

Hyperparameter	**Values**
Learning rate	0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5
Minimum length of character n-grams	0, 3
Maximum length of character n-grams	0, 3, 5

As a result, we find that using character n-grams brings benefit, and the best performance comes from including n-grams that are three to five characters long. We also conclude that learning rate values from the higher end of the scale are too high to train the models successfully and that the lowest values are not as competitive.

We use this information to limit the search space when tuning all models in the hierarchy. We fix character n-grams to length between three and five characters. Additionally, we tune the number of epochs, as this can depend on the amount of available training data which is different for each model. The selected hyperparameters and corresponding search spaces are presented in Table 11. Other hyperparameters are specified in appendix IV on page 67 or otherwise set to default values described in the fastText documentation[11].

Table 11. Values used for tuning the hyperparameters of hierarchical fastText.

Hyperparameter	**Values**
Word n-grams	1, 2, 3
Learning rate	0.05, 0.1, 0.2, 0.5, 0.7
Epochs	10, 25, 50

[11] https://fasttext.cc/docs/en/options.html

Since the hierarchical baseline model consists of many models, we cannot draw conclusions about which values are generally the best. In the version used for final evaluations, each model has its hyperparameters set to the values that produced the best results on dev_small.

Due to time constraints, we do not perform a comprehensive tuning experiment for the flat fastText model. Instead, we apply our insights from previous experiments. Based on our findings, we add word n-grams of length up to three, and character n-grams of length between three and five. The hyperparameter values for all models selected based on the tuning experiments are presented in appendix VI on page 70.

6 Results and Error Analysis

This chapter describes the results on the dev_large and test1 sets, and additionally on a gold-labeled test set. The performance of selected models is further qualitatively analyzed.

6.1 Results on Development Set

We first compare all models with both default and tuned hyperparameters on the dev_large set for model selection. We also perform error analysis on dev_large as the test set might be needed for adding more models into comparison later.

Table 12. Model performances (accuracy scores) on dev_large. The scores of the best baseline and the best Transformer-based model are marked in bold.

Model Group		**Type**	**Training Steps**	**HS2**	**HS4**	**HS6**
FastText (baseline)	Flat	Default	-	0.8947	0.8210	0.7281
		With n-grams	-	0.9284	0.8749	0.8024
	Hierarchical	Default	-	0.8841	0.8156	0.7358
		Tuned	-	0.9257	0.8766	**0.8107**
Flat		Default	1.78M	0.9160	0.8478	0.7471
		Tuned	1.75M	0.9254	0.8650	**0.7757**
Hierarchical	Bilinear attention	Default	2.35M	0.9194	0.8500	0.7457
		Tuned	2.31M	0.9082	0.8307	0.7185
	Self-attention[12]	Default	1.3M	0.9161	0.8450	0.7286

[12] The hierarchical self-attention model was trained for 8 days, but the number of training steps is significantly smaller than for other hierarchical models.

Table 12 illustrates performances on dev_large. All Transformer-based models were trained for ~8 days on two GPU-s. Furthermore, within one model subgroup, it was ensured that the number of training steps was similar. This is because some computing nodes have more powerful GPU-s than others, leading to a significantly different number of steps with the same training time. An exception is the hierarchical self-attention model, which was neither trained until equal steps nor tuned due to time constraints.

Based on these results, we select the hierarchical tuned fastText model and the flat tuned Transformer-based model for further comparisons with error analysis methods. We also analyze the predictions of the hierarchical Transformer-based model using bilinear attention in an attempt to understand why the hierarchical approach does not outperform the flat approach.

6.2 Error Analysis

In this section, we look at the models' predictions on dev_large in more detail to understand where and why the models make mistakes. We will refer to the selected baseline model as FAST, the hierarchical bilinear attention model as HIER, and the flat Transformer-based model as FLAT.

6.2.1 Hierarchical Approach

Regarding HIER with classifier per level structure, we first look at how the predictions from previous layers affect the next layers. We hypothesized that although the structure of the model theoretically requires the classification heads to predict the correct class among *all* classes on a given HS level, then in practice, the information from previous layers should "mask out" classes that do not correspond to the prediction so far.

When analyzing predictions with the highest probabilities from each classification head, it appears that there are relatively few mismatches (e.g., a mismatch is when the HS2 classification head predicts chapter 90 and the HS4 classification head predicts a heading from some other chapter, such as 8410). Between the first two layers, only

1.02% of examples get inconsistent predictions, and between the last two layers, the corresponding figure is 1.53%.[13]

This kind of information transfer has two sides: it can help decision-making in further layers by pruning the search space, but it can also propagate errors from the first layers to the next. We find that in ∼7.4% of the cases, a wrong prediction from the HS2 layer persists in the HS4 prediction. In ∼13.8% of the cases, a wrong prediction from the HS4 layer is propagated to the predicted HS6. However, propagated errors only account for roughly half of the incorrect predictions on HS4 and HS6 levels. Knowing that the percentage of mismatches is small, most of the other mistakes must come from the inability to choose between classes in the correct subtree in the hierarchy (e.g., when the HS2 classification head predicts chapter 90 and the HS4 classification head predicts heading 9010, while the correct heading is 9020).

6.2.2 Most Difficult Classes

When looking at per-class performances, we find that two non-minority HS6 classes have a notably low precision score, i.e., many false positives. One of these classes is 392690 (*Articles of plastics and articles of other materials of heading 3901 to 3914, n.e.s*) with a precision of 0.223 for FAST, 0.16 for HIER, and even less, 0.146 for FLAT. It appears that among its siblings, that is, other classes under heading 3926, this class is the most represented in the training set (around five times more than the second most common class under this heading). We also find that a large percentage of its siblings (5-11% for FAST, 14–22% for HIER, 11–26% for FLAT) are mistakenly assigned this class, and by frequency, the top classes that are mistakenly assigned this class are either from the same heading or at least the same chapter.

The second low-precision HS6 class is 732690 (*Articles of iron or steel, n.e.s. (excl. cast articles or articles of iron or steel wire)*), with a precision score of 0.316 for FAST, 0.29 for HIER, and 0.278 for FLAT. We again find that the classes with most false positives are from the same heading or at least from the same chapter.

[13] We used a setup where the top k=3 predictions are passed on to next layers as a weighted sum. We also experimented with k=1, which almost completely eliminated mismatches but did not improve scores.

Such classes are inherently difficult given that, by definition, class 392690 includes anything of plastics or a list of other materials *not elsewhere specified*. Likewise, class 732690 is similar, only with a different list of materials. This means that a wide variety of products can belong to these classes, which might also be the reason for being relatively overrepresented in the dataset.

Although the training set we used was oversampled to at least 1000 examples per HS6 class, some of these minority classes still received f1-scores of 0.0, i.e., no true positives. When analyzing examples from such classes in dev_large, it appears that the descriptions are often not detailed enough to choose between neighboring classes. In such cases, if there is a sibling class more represented in the training set, it can be prioritized by the models. It must be noted that when details are missing, even human annotators cannot tell which of the options is correct without making additional assumptions.

However, it is not the case that all minority classes have low f1-scores. When comparing class supports to average f1-scores, we find that for the least represented HS6 classes (less than 5000 examples in the training set), the average f1-scores are 0.68 for FLAT, 0.61 for HIER, and 0.70 for FAST. Since ~81% of HS6 classes fall into this bin due to the class imbalance, we look at these small classes in more detail. As expected, the average recall is somewhat lower than the average precision for these classes. We further divide these classes into bins according to their supports. As a result, we find that although the average recall for the smallest classes (1000–1099 examples in the training set) is among the lowest, the difference between recall in this bin and the total average recall is not very large for FLAT (3.2%,) and HIER (0.7%). For FAST, it is somewhat larger (6.3%). Also, there are small classes the models do notably well on. For example, class 950720 (*Fish-hooks, whether or not snelled*) has 299 unique examples in the training set and still receives an f1-score as high as 0.97 from all models.

6.2.3 Quality of Product Descriptions

As noted above, our dataset includes product descriptions that are too short and lacking in detail to be assigned an HS6-level code. Having such entries in the training set means that the models can learn to extrapolate from insufficient data (either correctly

or incorrectly). Having such examples in the evaluation sets means that we can analyze what the models predicted and whether the predictions are reasonable.

Figure 8 illustrates the distribution of description lengths across all dataset splits, counted in tokens (the result of the BPE tokenization as performed for Transformer-based classifiers). The most common length is between 10 and 25 tokens. It must be noted that since this tokenization method works on the level of subwords, ten tokens do not necessarily equal ten full words; instead, they can also include single-symbol and other short tokens.

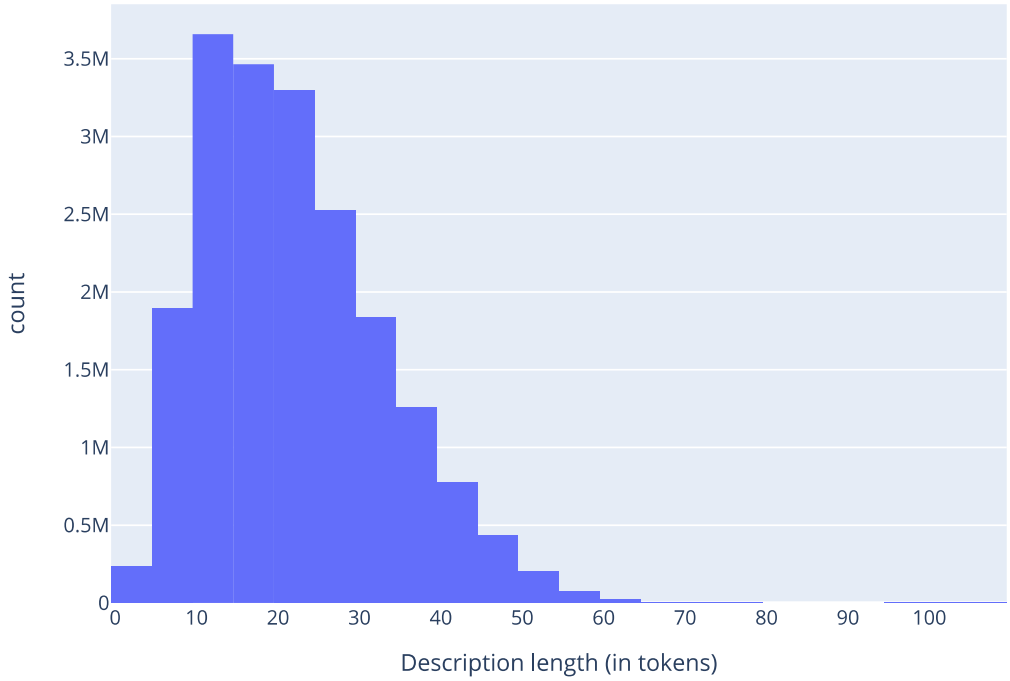

Figure 8. Histogram of description lengths of the full dataset.

Some examples of descriptions with very few tokens are presented in Table 13. Although many such short descriptions are indeed not informative, some do provide sufficient information for HS6-level classification. For example, while *Show pcs* does not say much about the product, a description as short as *Toothbrush* would be enough to assign the HS6-level code 960321 (*Tooth brushes, incl. dental-plate brushes*).

Table 13. Examples of short product descriptions (less than five tokens long).

Description	**HS Code**
BATH TOWEL MAROON	630260
PVC SHEET	392043
GOLDSMITH TOOLS Dies	820559
SHOW PCS	830629
GRATER RENA	732399

When looking at correctly classified short descriptions from, e.g., class 392690, we notice descriptions such as *hanger (25116 pcs)* and *hanger [1926 pcs].* The models must have assumed that, when not specified, the hangers are made of plastic. If they were made of wood, they could be assigned the class 442110 (*Clothes hangers of wood*). The training set contains more examples including the word *hanger* with the code 392690 than with 442110, which is probably why the models are biased towards plastic hangers.

The question is whether making such assumptions is desired or not. An alternative would be to learn to detect cases when the provided information is insufficient and not assign any HS code or assign a code up to some point, e.g., only the chapter. However, since we currently have such examples with HS6-level labels in the training set, the models can learn to make assumptions with high confidence (assigned probability).

While short descriptions intuitively seem problematic due to missing information, long descriptions do not always mean high quality either. Longer texts can include unnecessary information such as invoice numbers and dates or repeat the same information several times. A large number of tokens might indicate that there are long sequences of symbols that do not contain in-vocabulary tokens and must be broken down into single-symbol tokens. Table 14 presents a sample of descriptions where the number of tokens is large.

Table 14. Examples of long product descriptions (more than 55 tokens long).

Description	**HS Code**
INSU.FITT OF PLSTIC HEAT SHRIN.PWR/TELE COMM CBLE ACCE. EC/014/04/040 (3M CRB 10 -25 67-29-403)(B.NO.200053)P.SAP ID700	854720
USED AIRCRAFT TYRE FOR REPAIR & RETURN S IZE 30X8.8 R15 S/N-(96)8360S093(97)8349S 198(98)8346S119(99)9015S182(100)6221S118	401213
E010A0103601, CAPACITOR CAPAL, HVM_16V100F60,HF,16V,100UF,105?,2000H,- 55? to +105?,F6.3?5.7,JIANGHAI (FOR CAPTIVE CONS	853224
TELEVISIONS 55 INCH(55K3A)(ODF NO.AJB144669E)(BIS NO.CRS2018-1157/R-41089680/11.07.2019)(ETA NO.ETA-SD-20190601691/18.06	852872
867959-B21 HSTNS-2154 SER,HPE 360G10_4114X2_32GB_1TBX4_300GBX2 (STORAGE SERVER) (BIS - R-41000698 DT.04/09/2019)	847150

When analyzing how the models perform on long descriptions, we notice that accuracy drops for very long descriptions, but the models mostly do well on longer-than-average descriptions. For example, all models correctly classify *led tv, ua32n4003ar, 32, india, uar60/u32nn1 (model no. ua32 - n4003arxxl) (asean cert.no. vn-in 19/02/06816 dt.15.07.2019)* with the HS code 852872. We can hypothesize that in cases where the description contains the required information, even if surrounded by unnecessary tokens, the models can extract the useful elements. However, to determine which parts of such long and noisy descriptions are actually considered more important by the models, we would need to apply model explainability methods.

6.2.4 Correctness of Labels

Analyzing model confidence and average accuracy scores for various confidence ranges, we find that the two correlate well, especially for FLAT. In the case of FLAT, ~39% of examples have been classified with confidence higher than 97.5%. The average accuracy in this range is as high as 98.9%. FAST predicts almost 60% of examples

with this confidence, with an average accuracy of 95.4%. This leads us to inspect cases where the confidence is high but the prediction incorrect.

For FLAT, the most common confidently misclassified pair is 711719 (true label) vs. 711790 (predicted label). According to the nomenclature, both classes include imitation jewelry. The difference is in the material – the former class includes jewelry of base metal, while the latter anything else but base metal. By looking at examples in the training set, both appear to contain, e.g., imitation jewelry made of brass, which is a base metal and should therefore belong to class 711719.

For FAST, the most common confidently misclassified pair is 490199 (true label) vs. 490110 (predicted label). According to the nomenclature, both classes include printed books, brochures, etc., but the difference is whether they are in single sheets or not. When looking at the incorrectly predicted cases in dev_large and instances from these two classes in the training set, we see that this aspect is very rarely mentioned in product descriptions. Thus, it is not possible to distinguish between the two classes based on the given information, and the decisions seem to have been mostly arbitrary.

This problem of inconsistent labels becomes evident when analyzing erroneous but confident predictions. Since verifying the correctness of labels is a complicated task, we do not attempt to do this ourselves at scale. Instead, we rely on professionals from the partner company who verify the labels of a subset of our dataset. We will later use this verified data to understand whether the inconsistent labels in the training set negatively affected the models.

6.3 Results on Test Set

Table 15 presents the accuracy scores of the selected Transformer-based and fastText models on the held-out test1 set. From here on forward, we will distinguish between the flat and hierarchical baseline versions by referring to them as FAST_FLAT and FAST_HIER.

Table 15. Accuracy scores of selected models on test1.

Model	**HS2**	**HS4**	**HS6**
FLAT	0.9292	0.8759	0.8044
FAST_HIER	0.9284	0.8837	**0.8281**
FAST_FLAT	0.9323	0.8847	0.8255

The results illustrate that Transformer-based models do not outperform the tuned baseline models. Considering the complexity of training and tuning Transformer-based models and the computational resources required, it seems reasonable to rather focus further development efforts on fastText models that can be experimented with more easily. With some tweaks, fastText classifiers can be improved to produce results that exceed those from Transformer-based models.

6.4 Results on Gold-Labeled Data

As noted above, we received a dataset with verified labels from our partner company. The verification was performed by content analysts who are familiar with the HS nomenclature and have experience with assigning and correcting HS codes.

Since the entries chosen for verification might have originated from any split in our original dataset, we remove all entries with such product description and HS6 code pairs that also existed in our training or development splits before running evaluations. The resulting dataset, which we will refer to as *test_gold*, is described in Table 16. The table illustrates that this subset contains a significantly smaller number of classes than the total dataset.

Table 16. Number of classes per level and total size of the test_gold set.

	test_gold
HS2	30
HS4	238
HS6	747
size	1,055,785

Table 17 presents the performances of the models on test_gold. The results differ significantly from those on the test1 set. Accuracy scores on the HS2 level are higher than for test1, but the situation is the opposite on HS4 and HS6 levels. On the HS4 level, accuracy has decreased by ~10 percentage points for all models, and on the HS6 level, the decrease is larger than 30 percentage points.

Table 17. Accuracy scores on test_gold.

Model	**HS2**	**HS4**	**HS6**
FLAT	0.9616	0.7806	**0.4832**
FAST_HIER	0.9643	0.7725	0.4587
FAST_FLAT	0.9656	0.7772	0.4681

When comparing test_gold with the training split, we find a large amount of partial overlap between the two. With partial overlap, we refer to cases when the models have seen the same product description during training with one HS code, but in test_gold, the same product description appears with an either partly or entirely different HS code. Such situations result from label corrections, and the performance drop on lower levels indicates that the mislabeled descriptions were more often mislabeled on HS4 and HS6 levels.

More specifically, ~17.7% of test_gold is made up of descriptions seen during training with an HS code that matches up to HS4 but does not match on the HS6 level. Approximately 29.3% of the dataset has been seen during training with an HS code that matches up to HS2 or up to HS4. However, only ~2.2% of test_gold consists of product descriptions seen during training with completely different (not matching to any extent) labels, which partly explains the high HS2 scores. Another factor is that there are fewer HS2 classes in test_gold, and those not present were mostly also less represented in the training set and thus not as well learned.

Interestingly, although all three models are strongly affected by the erroneous labels in the training set, the performance of FLAT has decreased the least. While it was outperformed by both baseline models on test1, it performs best among the three on

test_gold. Since we find that the training accuracy of FLAT is lower than FAST_FLAT and FAST_HIER, we can speculate that FLAT did not learn the patterns in the training data as strongly, i.e., overfitted less to the erroneously labeled training data.

7 Discussion

This chapter brings attention to some limitations related to the methods used in this thesis and presents directions for future work on the topic.

7.1 Limitations

In terms of computational complexity, some of the models developed in this thesis have downsides that must be considered. For example, FLAT requires a long training cycle on GPUs and is therefore expensive to train; it also performs inference much slower when run on a CPU than when run on a GPU.

Furthermore, in our experiments, we limited the training time of Transformer-based models to eight days (although, as mentioned, in some cases, training was resumed to reach a similar number of training steps as other models in the comparison). We found that the models do not converge within this time when using the full training set, as training always continued until the time limit, and the early stopping mechanism did not run out of patience. Therefore, this type of models could possibly achieve better results when trained for longer.

Another limitation regarding Transformer-based models is that we did not manage to improve HIER with hyperparameter tuning, again partly due to the slow training process. Since we used a subset of the training data for the tuning runs, the conclusions drawn from the limited-size and limited-time runs were not transferable to a longer training process with full data. FLAT, on the other hand, achieved notably better scores with tuned hyperparameter values. The difference might have come from the learning rate schedulers used. For FLAT, we increased the learning rate and changed the scheduler from constant to linear, while for HIER, we increased the learning rate but did not change the scheduler type. For HIER, this entailed a constantly high learning rate after the initial warm-up phase, which we presume negatively affected the learning process.

Among the two baseline models, FAST_HIER takes less time to train than FAST_FLAT (~1.5h vs. ~8h), but due to using word- and character-level n-grams in many models in the hierarchy, the total size grows very large (> 600 GB), making it less feasible for use in the industry. We compressed the models using the *quantize* method from the fastText library. The compression brought the space requirement down to ~80 GB

which can still be restrictive. In addition to this, quantization takes time – for models with less training data, even more time than the training itself – making the process much slower (~2 days and 10 hours to train and quantize all models).

FAST_FLAT stands in the middle ground in terms of time and space requirements. It is slower to train than FAST_HIER but is still significantly faster than FLAT and faster than FAST_HIER with quantization. It consists of only one model and does not require much space even when word- and character-level n-grams are used. The sizes of FLAT and FAST_FLAT are similar, between 1 and 2 GB.

7.2 Future Work

In our experiments, we found that the dataset used for training the models contained many low-quality entries. This entailed low accuracy scores of all models on a dataset with verified labels, as the mismatch between the labels in the training set and the evaluation set was remarkable. Based on this finding, a direction for future work with HS classification is to focus on collecting data with verified quality.

A source of information that was not utilized in this thesis but could prove beneficial for classification is the body of explanatory texts in the HS nomenclature. Furthermore, the nomenclature is supplemented by explanatory notes that are more detailed and often list examples of products for each class. Although the structure of these explanatory notes is different from product descriptions, they contain useful examples and keywords which could be used, e.g., for building a knowledge base to be used as an external source for aiding models with classification decisions.

Another possibility would be to use additional features that often complement product descriptions as input to classification models. Such features might include the dimensions or weight of a package, which could help infer the correct class when these details are relevant, even if this information is not stated explicitly in the product description.

Looking at examples of product descriptions, we have seen that they often include grammatical errors, misplaced whitespaces, and other typing errors (e.g., *plstic* instead of *plastic*; *s ize* instead of *size*). Since our models use subwords, we can expect the effect of such errors to be somewhat less pronounced than for methods that assume whole words to be in vocabulary in the exact same form. Nevertheless, correcting errors

could be beneficial, as according to a recent publication [37] on the sensitivity of the Transformer-based BERT to misspellings, mistakes in informative words can significantly affect performance. We did not analyze the effect of spelling errors on different models in this thesis.

Possible directions for continued work with Transformer-based models would be to either resume pre-training or pre-train the encoder from scratch on in-domain data instead of directly fine-tuning an encoder pre-trained on large general corpora. As the pre-training process is unsupervised, only product descriptions are needed, and whether they also have correct labels or any labels at all would not matter for this purpose. Resumed pre-training is an alternative option, and it has been shown that using in-domain data for this purpose is beneficial [38].

Regarding product classification for customs declarations more generally, a logical next step is to move forward from the six-digit HS codes and explore possibilities for ten-digit TARIC code classification. When importing goods into the EU, the HS code is sufficient only for low-value consignments. For other cases, the ten-digit code is required, which also entails an increase in the number of classes and an even more fine-grained nomenclature. Consequently, it is expected that collecting high-quality data for each of the classes is more difficult.

8 Conclusion

In this thesis, we explored HS classification with different model architectures. As a baseline, we used fastText, a shallow classifier with one hidden layer. For comparison, we employed publicly available pre-trained language models to develop custom classifiers with flat and hierarchical structures.

In the default configuration where fastText did not use subwords or word n-grams as features, Transformer-based models performed better. However, after some modifications, both flat and hierarchical fastText classifiers outperformed Transformer-based classifiers. These results suggest that fastText models can be improved more easily while tuning Transformer-based models is more time-consuming and complicated.

Comparing the various flat and hierarchical approaches, we found that hierarchical fastText achieved better HS6-level accuracy than flat fastText but did so at the cost of a much larger model, which requires more time for training and compressing. The classifier per level approach used in the hierarchical Transformer-based classifier did not perform better than the flat Transformer-based classifier. When analyzing the possible reasons, we concluded that the main complexity lies in choosing between classes in the same subtree of the nomenclature, not selecting the correct subtree, which should be easier with the hierarchical model.

For our task and dataset, using an encoder pre-trained on large corpora did not produce better results than the baseline that did not use pre-trained embeddings. This might indicate that with a dataset of this size, the benefit gained from knowledge transferred from the general to the task-specific domain was not large enough to compensate for other disadvantages of Transformer-based models, such as the necessity of more training steps to reach convergence.

From a qualitative analysis of the performances of selected models, we found that our dataset includes mislabeled entries and insufficiently detailed product descriptions. When evaluating the same models on a verified dataset, we discovered that the mislabeled training examples had strongly affected the models, and their performance on the gold-labeled set was significantly lower. This finding emphasizes the importance of having access to high-quality training data.

References

[1] F. Altaheri and K. Shaalan, "Exploring Machine Learning Models to Predict Harmonized System Code," in *Information Systems*, vol. 381, M. Themistocleous and M. Papadaki, Eds. Cham: Springer International Publishing, 2020, pp. 291–303. doi: 10.1007/978-3-030-44322-1_22.

[2] J. Luppes, "Classifying Short Text for the Harmonized System with Convolutional Neural Networks," *Masters Thesis*, p. 58, Aug. 2019.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *ArXiv181004805 Cs*, May 2019, [Online]. Available: http://arxiv.org/abs/1810.04805

[4] A. Vaswani *et al.*, "Attention Is All You Need," *ArXiv170603762 Cs*, Dec. 2017, [Online]. Available: http://arxiv.org/abs/1706.03762

[5] "World Customs Organization." http://www.wcoomd.org/en/topics/nomenclature/overview/what-is-the-harmonized-system.aspx

[6] "World Customs Organization." http://www.wcoomd.org/en/topics/nomenclature/instrument-and-tools/hs-nomenclature-2017-edition/hs-nomenclature-2017-edition.aspx

[7] C. Weerth, "HS 2002–HS 2017: Notes of the tariff nomenclature and the additional notes of the EU revisited," vol. 11, no. 1, p. 20.

[8] L. Ding, Z. Fan, and D. Chen, "Auto-Categorization of HS Code Using Background Net Approach," *Procedia Comput. Sci.*, vol. 60, pp. 1462–1471, 2015, doi: 10.1016/j.procs.2015.08.224.

[9] G. Li and N. Li, "Customs classification for cross-border e-commerce based on text-image adaptive convolutional neural network," *Electron. Commer. Res.*, vol. 19, no. 4, pp. 779–800, Dec. 2019, doi: 10.1007/s10660-019-09334-x.

[10] M. Spichakova and H.-M. Haav, "Using Machine Learning for Automated Assessment of Misclassification of Goods for Fraud Detection," in *Databases and Information Systems*, vol. 1243, T. Robal, H.-M. Haav, J. Penjam, and R. Matulevičius, Eds. Cham: Springer International Publishing, 2020, pp. 144–158. doi: 10.1007/978-3-030-57672-1_12.

[11] Q. Le and T. Mikolov, “Distributed Representations of Sentences and Documents,” in *International Conference on Machine Learning*, Jun. 2014, pp. 1188–1196. [Online]. Available: http://proceedings.mlr.press/v32/le14.html

[12] D. Gao, W. Yang, H. Zhou, Y. Wei, Y. Hu, and H. Wang, “Deep Hierarchical Classification for Category Prediction in E-commerce System,” *ArXiv200506692 Cs*, May 2020, [Online]. Available: http://arxiv.org/abs/2005.06692

[13] Y. Goldberg, “Neural Network Methods for Natural Language Processing,” *Synth. Lect. Hum. Lang. Technol.*, vol. 10, no. 1, pp. 1–309, Apr. 2017, doi: 10.2200/S00762ED1V01Y201703HLT037.

[14] Z. S. Harris, “Distributional Structure,” WORD, vol. 10, no. 2–3, pp. 146–162, Aug. 1954, doi: 10.1080/00437956.1954.11659520.

[15] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space,” *ArXiv13013781 Cs*, Sep. 2013, [Online]. Available: http://arxiv.org/abs/1301.3781

[16] J. Pennington, R. Socher, and C. Manning, “GloVe: Global Vectors for Word Representation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar, Oct. 2014, pp. 1532–1543. doi: 10.3115/v1/D14-1162.

[17] M. E. Peters *et al.*, “Deep contextualized word representations,” *ArXiv180205365 Cs*, Mar. 2018, [Online]. Available: http://arxiv.org/abs/1802.05365

[18] R. Sennrich, B. Haddow, and A. Birch, “Neural Machine Translation of Rare Words with Subword Units,” *ArXiv150807909 Cs*, Jun. 2016, [Online]. Available: http://arxiv.org/abs/1508.07909

[19] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language Models are Unsupervised Multitask Learners,” *OpenAI Blog 18 9*, p. 24, 2019.

[20] I. Goodfellow, Y. Bengio, and A. Courville, “Deep Learning,” 2016. https://www.deeplearningbook.org/

[21] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997, doi: 10.1109/78.650093.

[22] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/neco.1997.9.8.1735.

[23] D. Bahdanau, K. Cho, and Y. Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” *ArXiv14090473 Cs Stat*, May 2016, [Online]. Available: http://arxiv.org/abs/1409.0473

[24] T. Luong, H. Pham, and C. D. Manning, “Effective Approaches to Attention-based Neural Machine Translation,” in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, Lisbon, Portugal, 2015, pp. 1412–1421. doi: 10.18653/v1/D15-1166.

[25] S. Brarda, P. Yeres, and S. Bowman, “Sequential Attention: A Context-Aware Alignment Function for Machine Reading,” in *Proceedings of the 2nd Workshop on Representation Learning for NLP*, Vancouver, Canada, 2017, pp. 75–80. doi: 10.18653/v1/W17-2610.

[26] Y. Liu *et al.*, “RoBERTa: A Robustly Optimized BERT Pretraining Approach,” *ArXiv190711692 Cs*, Jul. 2019, [Online]. Available: http://arxiv.org/abs/1907.11692

[27] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python,” *J. Mach. Learn. Res.*, vol. 12, no. 85, pp. 2825–2830, 2011.

[28] T. Wolf *et al.*, “Transformers: State-of-the-Art Natural Language Processing,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Online, Oct. 2020, pp. 38–45. doi: 10.18653/v1/2020.emnlp-demos.6.

[29] A. Paszke *et al.*, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” *ArXiv191201703 Cs Stat*, Dec. 2019, [Online]. Available: http://arxiv.org/abs/1912.01703

[30] L. Biewald, *Experiment Tracking with Weights and Biases*. 2020. [Online]. Available: https://www.wandb.com/

[31] University of Tartu, “UT Rocket”, doi: 10.23673/PH6N-0144.

[32] C. N. Silla and A. A. Freitas, “A survey of hierarchical classification across different application domains,” *Data Min. Knowl. Discov.*, vol. 22, no. 1–2, pp. 31–72, Jan. 2011, doi: 10.1007/s10618-010-0175-9.

[33] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical Attention Networks for Document Classification," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, San Diego, California, 2016, pp. 1480–1489. doi: 10.18653/v1/N16-1174.

[34] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of Tricks for Efficient Text Classification," *ArXiv160701759 Cs*, Aug. 2016, [Online]. Available: http://arxiv.org/abs/1607.01759

[35] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," *ArXiv171105101 Cs Math*, Jan. 2019, [Online]. Available: http://arxiv.org/abs/1711.05101

[36] G. Lemaıtre and F. Nogueira, "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning," p. 5.

[37] L. Sun *et al.*, "Adv-BERT: BERT is not robust on misspellings! Generating nature adversarial samples on BERT," *ArXiv200304985 Cs*, Feb. 2020, [Online]. Available: http://arxiv.org/abs/2003.04985

[38] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to Fine-Tune BERT for Text Classification?," in *Chinese Computational Linguistics*, vol. 11856, M. Sun, X. Huang, H. Ji, Z. Liu, and Y. Liu, Eds. Cham: Springer International Publishing, 2019, pp. 194–206. doi: 10.1007/978-3-030-32381-3_16.

Appendix

I. Code

The source code is available at: https://github.com/karmenkk/hs_prediction (accessible upon request)

II. Supports of HS2 Classes

The following table includes the number of examples under each chapter in each split and in the total dataset.

HS2 class	train	dev	test1	test2	total
84	2981122	372643	186329	186338	3726432
85	2448888	306098	153059	153053	3061098
73	1301439	162682	81351	81355	1626827
90	913043	114141	57069	57068	1141321
39	880543	110085	55039	55033	1100700
62	777858	97229	48625	48627	972339
61	539111	67388	33695	33698	673892
40	533917	66741	33373	33368	667399
82	468780	58599	29303	29304	585986
94	456901	57114	28558	28555	571128
42	343140	42896	21445	21443	428924
83	315844	39481	19743	19743	394811
95	289663	36205	18108	18107	362083
33	285972	35749	17868	17869	357458
48	246137	30768	15384	15382	307671
63	241507	30187	15095	15091	301880
70	196107	24513	12253	12257	245130
32	195047	24382	12189	12193	243811
96	179108	22388	11195	11195	223886

64	176008	22003	11002	11005	220018
76	171733	21463	10734	10733	214663
57	169334	21169	10584	10584	211671
74	162567	20324	10161	10159	203211
71	155763	19467	9734	9735	194699
49	138143	17268	8634	8633	172678
54	119796	14979	7476	7481	149732
91	115940	14493	7247	7245	144925
68	104669	13084	6539	6541	130833
69	95397	11925	5963	5963	119248
58	95319	11917	5956	5955	119147
44	87964	10993	5489	5489	109935
34	80536	10064	5034	5033	100667
52	64634	8071	4028	4029	80762
09	53762	6726	3356	3357	67201
59	51854	6477	3240	3238	64809
55	42709	5334	2665	2665	53373
08	42444	5300	2659	2656	53059
56	31022	3877	1942	1940	38781
22	28207	3527	1762	1764	35260
60	26824	3352	1677	1677	33530
65	26530	3318	1659	1660	33167
92	15726	1968	981	981	19656

41	15598	1948	979	979	19504
97	13693	1711	856	857	17117
67	10338	1293	645	645	12921
75	9560	1193	597	596	11946
37	7595	947	472	472	9486
51	7567	944	465	465	9441
50	6154	768	383	384	7689
53	5774	721	362	361	7218
66	5081	636	317	319	6353
47	4690	584	294	294	5862
24	3658	460	225	226	4569
43	2296	286	144	143	2869
05	1439	178	87	86	1790

III. Label Distribution Histograms

The following figures illustrate the distribution of labels on each HS level in the full original dataset.

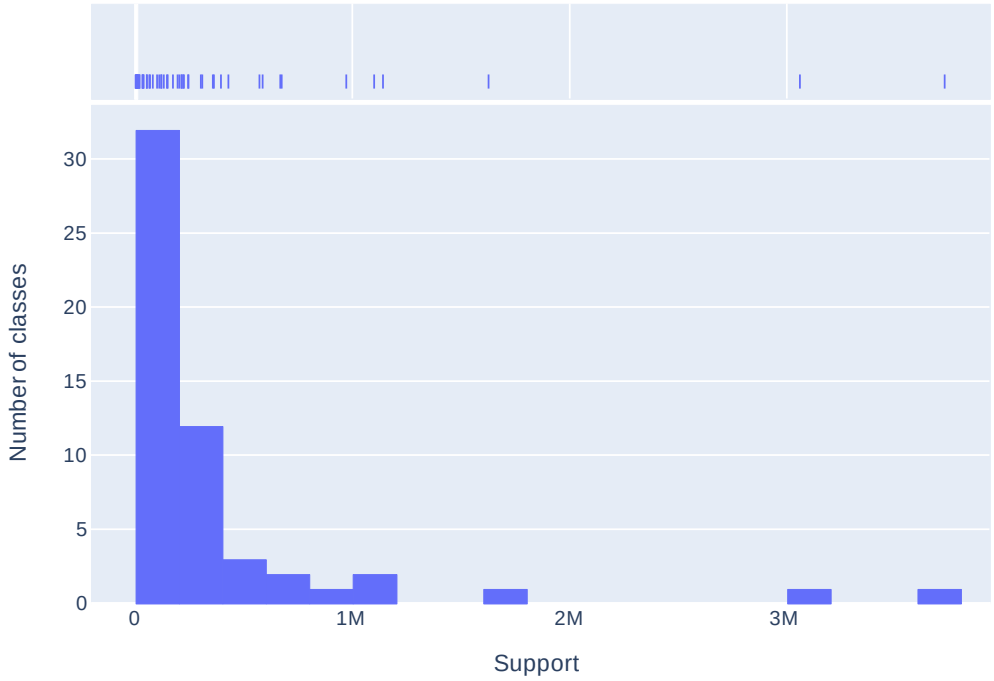

Figure 9. Histogram with a complementary rug plot of HS2 class supports.

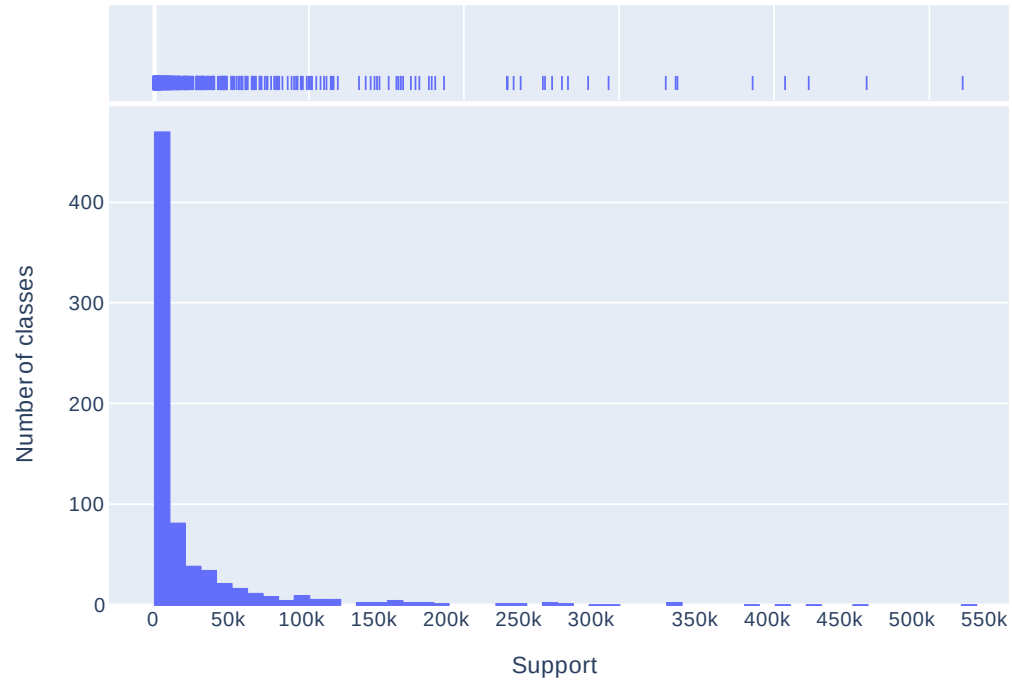

Figure 10. Histogram with a complementary rug plot of HS4 class supports.

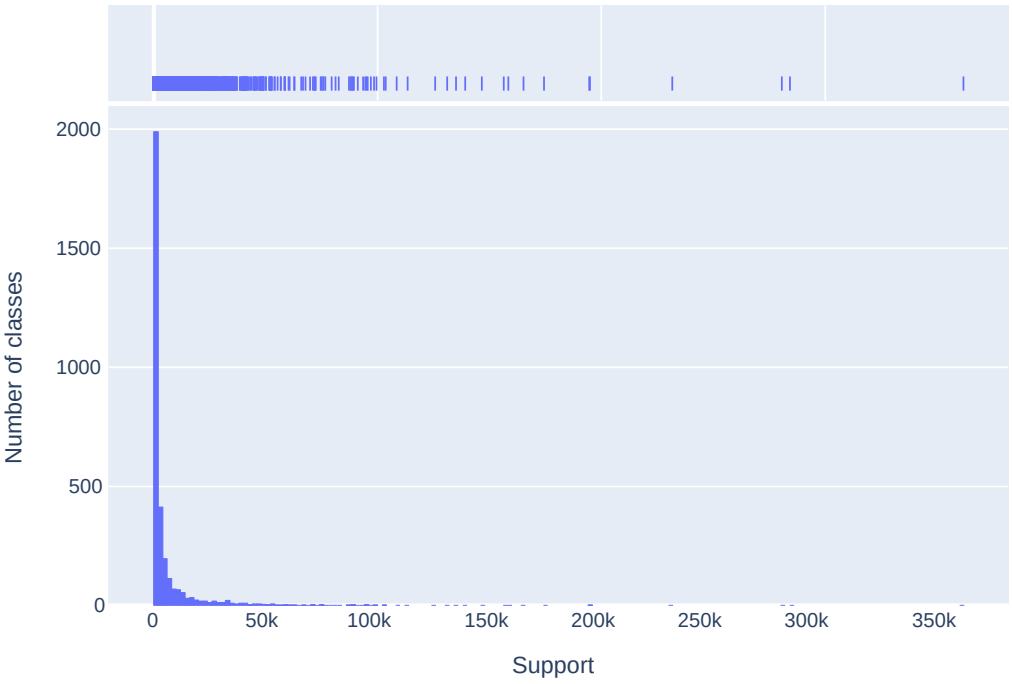

Figure 11. Histogram with a complementary rug plot of HS6 class supports.

IV. Default Hyperparameter Values

Table 18 includes the default hyperparameter values used for training Transformer-based models in all experiments where not stated otherwise. The learning rate scheduler value "constant with warm-up" means that initially (during a percentage of all training steps as defined by warm-up ratio), the learning rate increases from 0 to the defined value and then remains constant. These fixed values are inspired by the values used for fine-tuning on classification tasks in the paper introducing RoBERTa [26].

Table 18. Default hyperparameter values for Transformer-based models.

Hyperparameter	**Value**
Learning rate	1e-5
Learning rate scheduler	Constant with warm-up
Warm-up ratio	6%
Batch size	32
Dropout	0.1
Weight decay	0.1
Epochs	10
Classifier hidden size	256
Embedding size[14]	128

Table 19 presents the default hyperparameters of the flat fastText model. MinCount refers to the minimal number of word occurrences required in the training set to use a word as a feature. Both minimum and maximum lengths of character n-grams being set to 0 means that character n-grams are not used.

[14] Not applicable to flat models.

Table 19. Default hyperparameter values for flat fastText classifier.

Hyperparameter	**Value**
Learning rate	0.1
MinCount	5
Epochs	20
Min length of character n-grams	0
Max length of character n-grams	0
Max length of word n-grams	1

Table 20 presents the default hyperparameter values for the hierarchical version of fastText. In cases where the values differ depending on the level, the difference is shown. Otherwise, all models use the same values, regardless of their position in the hierarchy.

Table 20. Default hyperparameters for hierarchical fastText classifier.

Hyperparameter	**HS2 level**	**HS4 level**	**HS6 level**
Learning rate	0.1		
MinCount	5	2	1
Epochs	50	25	10
Min length of character n-grams	0		
Max length of character n-grams	0		
Max length of word n-grams	1		

V. Comparisons of Hyperparameter Tuning Runs

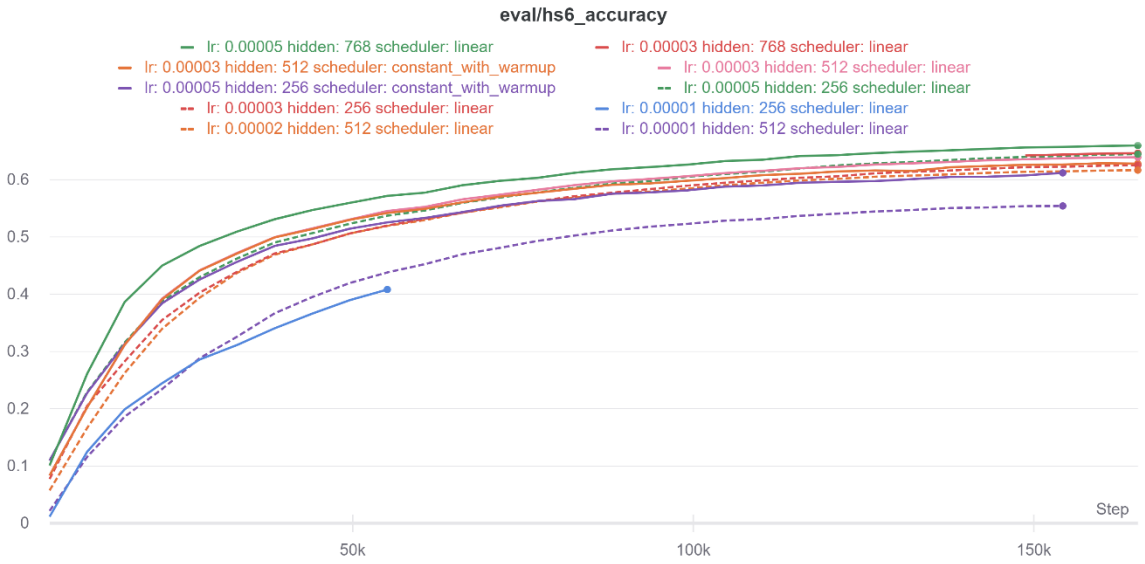

Figure 12. HS6 accuracy scores on dev_small during the hyperparameter tuning runs with flat Transformer-based classifier. The legend shows the learning rate (lr) values, classifier hidden size (hidden) values, and learning rate scheduler type (scheduler).

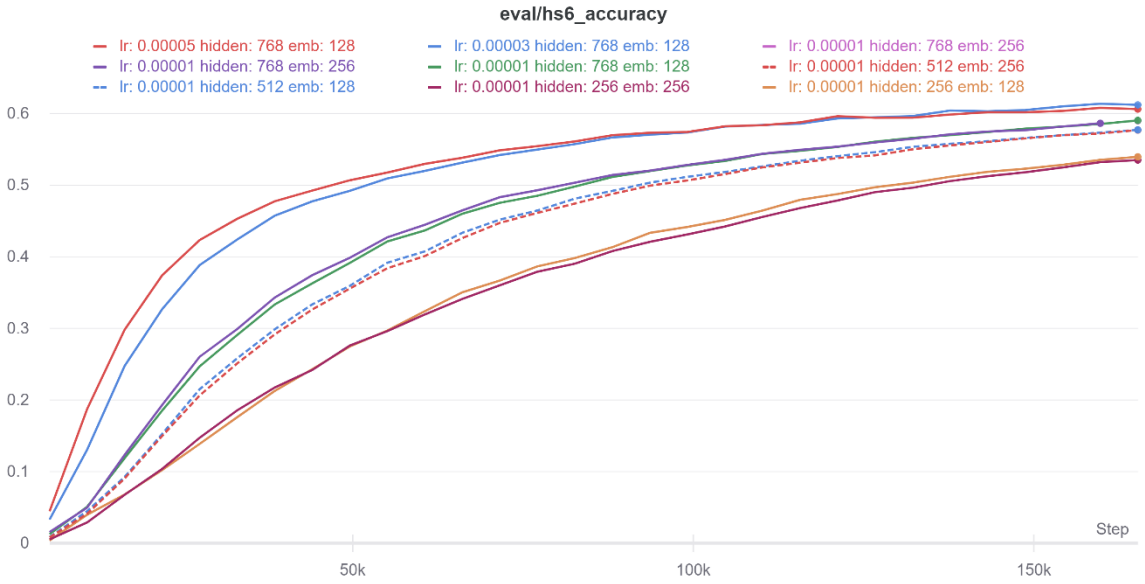

Figure 13. HS6 accuracy scores on dev_small during the hyperparameter tuning runs with hierarchical Transformer-based classifier. The legend shows the learning rate (lr) values, classifier hidden size (hidden) values, and embedding size (emb) values.

VI. Hyperparameter Values After Tuning

The following tables present the hyperparameter values used for training the final models. The values were either selected according to results from hyperparameter tuning runs or fixed as such from the beginning.

Table 21. Hyperparameter values for the flat Transformer-based classifier.

Hyperparameter	**Value**
Learning rate	5e-5
Learning rate scheduler	Linear
Warm-up ratio	6%
Batch size	32
Dropout	0.1
Weight decay	0.1
Classifier hidden size	768

Table 22. Hyperparameter values for the hierarchical Transformer-based classifier.

Hyperparameter	**Value**
Learning rate	3e-5
Learning rate scheduler	Constant with warm-up
Warm-up ratio	6%
Batch size	32
Dropout	0.1
Weight decay	0.1
Classifier hidden size	768
Embedding size	128

Table 23. Hyperparameter values for the flat fastText classifier.

Hyperparameter	**Value**
Learning rate	0.1
MinCount	5
Epochs	20
Min length of character n-grams	3
Max length of character n-grams	5
Max length of word n-grams	3

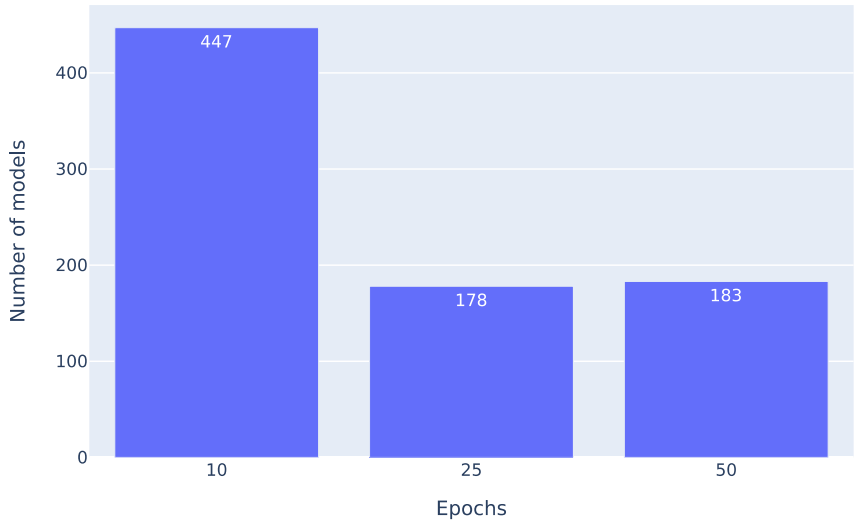

Figure 14. Number of models in the fastText hierarchical structure that selected each "number of epochs" value as the best.

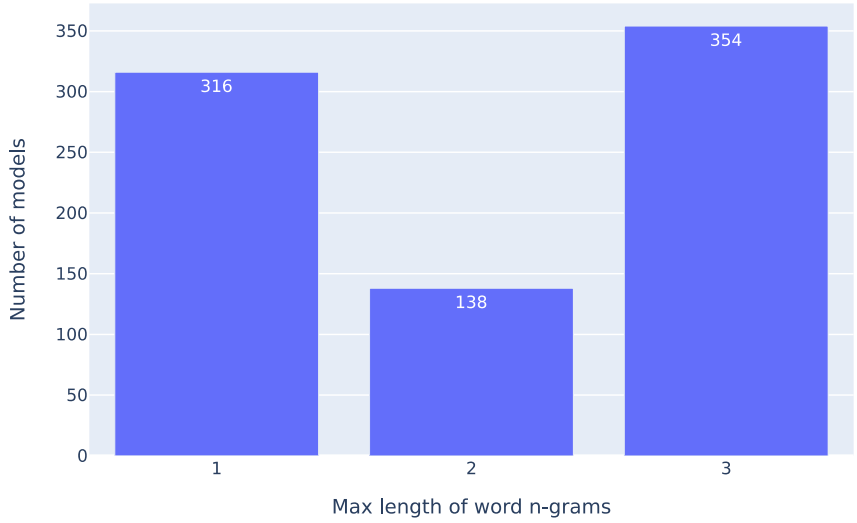

Figure 15. Number of models in the fastText hierarchical structure that selected each "word n-gram" value as the best.

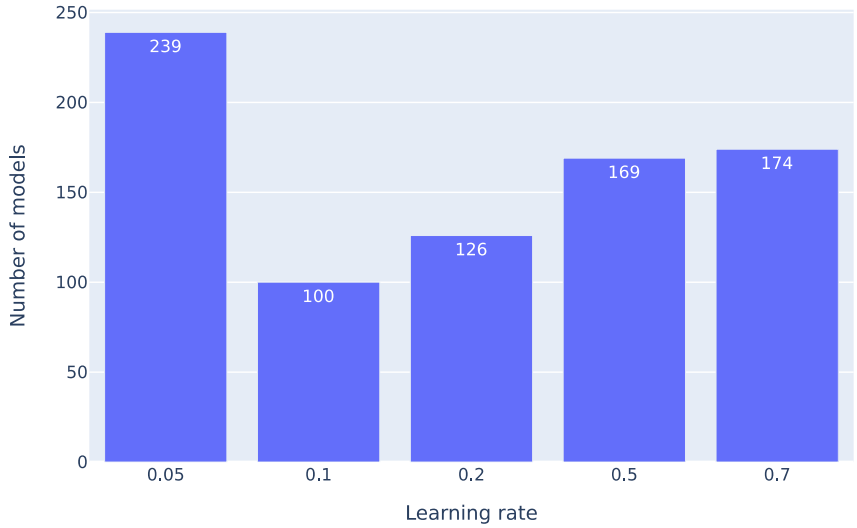

Figure 16. Number of models in the fastText hierarchical structure that selected each "learning rate" value as the best.

VII. License

Non-exclusive licence to reproduce thesis and make thesis public

I, Karmen Kink,

herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work from ***14/05/2022*** until the expiry of the term of copyright,

Classification of E-Commerce Products Based on Textual Product Descriptions,

supervised by Kairit Sirts and Karl-Oskar Masing,

2. I am aware of the fact that the author retains the rights specified in p. 1.

3. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Karmen Kink

14/05/2021