Evaluating machine learning models for text classification

A comparative study of Amazon Comprehend & Amazon SageMaker

Jonas Lilja

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6213
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2023 Jonas Lilja. All rights reserved.
Printed in Sweden by Tryckeriet i E-huset
Lund 2023

Abstract

This thesis will explore the use of AWS machine learning services that enable natural language processing (NLP). More specifically, this work will focus on sentiment analysis of product and service reviews written in Swedish.

To find the most efficient solution to this task, the ready-made sentiment analysis tool available on AWS (Amazon Comprehend) was compared to custom-made solutions built on the AWS platform Amazon SageMaker. Several model options were considered for the custom-made alternative, ranging from simple regression to state-of-the-art language models, along with different ways of deploying the best performing model to the cloud so that its insights could be accessed in the most efficient way.

For the evaluation, a use-case was put forward that is deemed realistic for how Sigma Technology Cloud would use such a service. Based on this, the thesis findings suggest that a scaled-down version of the BERT model called DistilBERT is the best alternative of the models evaluated. Furthermore, this model should be set up as an asynchronous endpoint with a policy allowing it to auto-scale down to zero when it is not invoked. Doing this on the Amazon SageMaker platform results in a solution that is both cheaper and that shows better performance than other alternatives.

Finally, this comparison between the services based on their performance, ease of use, and cost efficiency was put forward as a recommendation as to what model and configuration should be used by Sigma Technology Cloud.

Sammanfattning

Examensarbetet utforskar användningen av AWS maskininlärningstjänster som möjliggör hantering av textdata. Mer specifikt undersöker arbetet uppgiften att analysera attityder i produkt- och tjänstrecensioner skrivna på svenska.

För att hitta den mest effektiva lösningen på den här uppgiften jämfördes de färdiga verktyget för attitydsanalys som finns tillgängligt hos AWS (Amazon Comprehend) med egengjorda lösningar som tagits fram på AWS-plattformen Amazon SageMaker. För de egengjorda lösningarna övervägdes flera modellalternativ, från enkla regressioner till moderna språkmodeller. Tillsammans med olika sätt för att koppla upp den bäst presterande modellen till molnet, så att dess resultat blir tillgängligt på det mest effektiva sättet.

För utvärderingen togs ett användarscenario fram som kan anses realistiskt för hur Sigma Technology Cloud skulle använda en sådan tjänst. Baserat på det här scenariot tyder resultatet på att av de utvärderade modellerna är en nedskalad version av BERT-modellen kallad DistilBERT det bästa alternativet. Och att denna modell bör sättas upp med hjälp av en asynkron endpoint med en inbyggd användings-policy som gör att den automatiskt kan skala ner sin resursallokering till noll när den inte är aktiv.

Slutligen presenteras resultaten i en jämförelse mellan tjänsterna baserat på deras prestanda, användarvänlighet och kostnadseffektivitet. Där det slutliga förslaget läggs fram som en rekommendation till vilken modell och konfiguration som bör användas av Sigma Technology Cloud.

Acknowledgements

I want to thank my supervisor at LTH, Johan Eker, for his constant support and encouragement throughout this master's thesis. As well as to my examinator Karl-Erik Årzén for providing valuable feedback and suggestions for improvement.

At Sigma Technology Cloud I also want to show my gratitude towards William Cloarec for always being helpful with any questions I might have during my work, especially regarding AWS. Thanks also to Emmy Vartiainen and Rashin Kabodvand for ensuring that I got the support I needed and for making me feel welcome at the office.

Lastly, I thank Tim Isben for allowing me to build upon his work on Swedish sentiment analysis.

Notations and Symbols

Abbreviation

AI - Artificial Intelligence
ML - Machine Learning
NLP - Natural Language Processing
AWS - Amazon Web Services
SDK - Software Development Kit
API - Application Programming Interface
USD - US Dollar

Contents

1 Introduction

Sentiment analysis is a sub-part of natural language processing that concerns techniques capable of determining the opinion or emotion expressed in a text. Thanks to state-of-the-art language models and machine learning algorithms, the field has seen rapid growth in recent years and has become an increasingly valuable tool for businesses. One company that realizes the potential of such tools is Sigma Technology Cloud, a technology consulting firm based in Sweden that offers technical solutions to clients in various industries through competence and proficiency in the latest forms of cloud integration and deployment.

Designed to provide the text analyzing capabilities to Sigma Technology Cloud, this report will discuss possible implementations and trade-offs in developing and deploying such sentiment analysis tools and helping Sigma understand customer feedback and public opinion about their services and brand by unlocking valuable insights into customer satisfaction that supports the company in strategic decisions. The considerations put forward in this report will therefore be concerned with the effectiveness of the solution and the overall cost of the implementation.

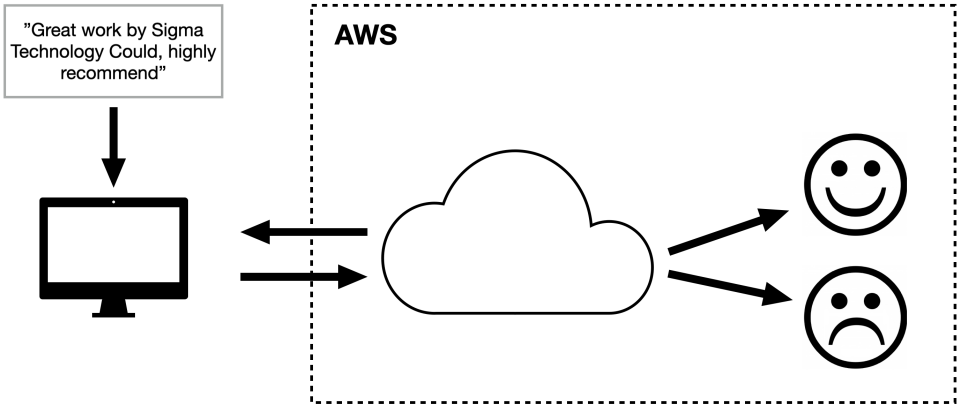

Figure 1.1: Overview of a sentiment analysis model deployed to AWS that takes text input from a user and returns the resulting analysis (positive or negative).

In the pursuit of exploring the possibilities of such a tool, the main part of the report will be composed of two parts. The first part concerns the development of a model capable of classifying the sentiments, and the second part explores the deployment of this model to the AWS cloud for scalability and accessibility.

1.1 Outline

The report will be divided into four main sections, each exploring a different part of the work process. Starting with the background section, which will provide the information

necessary to better understand the report's decisions, trade-offs, and eventual results. It starts with an overview of the vast field of artificial intelligence and moves on to focus on specific aspects of machine learning and natural language processing. The section then discusses the different cloud deployment options available on AWS. Specifically discussing the differences and capabilities of the two services explored in this thesis, namely Amazon Comprehend and Amazon SageMaker. The background section is then finished by showing alternative ways of representing text in a format that can be fed into a machine learning model, as well as a showcase of the inner workings of some of the latest language models and examples thereof.

After giving the necessary background information, the next part of the report explains how the work was carried out and thoroughly describes the methodical choices. Since the thesis is concerned with a specific implementation, the examination method will consist of experimentally trying out different options on a simulated realistic use case where a variety of reviews written in Swedish expressing different sentiments is given to the model in the hope that it will be able to classify them correctly, after which deployment of the model will be simulated to explore the pros and cons of different alternatives and understand the necessary associated costs.

The approach taken in this report is designed to be scientific and to examine the various options objectively. However, developing a scientific approach to a problem like this is a challenging task, especially when the subject matter is complex and multifaceted. The examination, therefore, includes just a subspace of the various available options, including necessary trade-offs. Nonetheless, focusing on a scientifically rigorous approach will help ensure that the sentiment analysis tool provides the company with the most accurate and reliable results.

Finally, in the result section, the experiments are shown in the format of evaluation metrics that capture the effectiveness of different models on the constructed use case. In addition, the findings also include several graphs illustrating different deployment option's divergent results, ultimately leading to a final recommendation that provides Sigma Technology Cloud with the best possible solution for sentiment analysis.

As a whole, the main contribution of the thesis will be an exploration of systematic approaches for evaluating cloud deployments and benchmarking the solutions performances. The hope is that this contribution will help enable organizations to make informed decisions about cloud utilization and adoption.

1.2 Aim

This thesis aims to improve the understanding of in which cases or under which circumstances a self-built model (so-called in-house solution) would be preferable to ready-made cloud-based services for text classification, as well as which models are most suitable for the task and their possible advantages and disadvantages, showing these findings on practical test cases that are used in a comparative study to research and understand the underlying differences.

2 Background

2.1 Artificial intelligence

Chances are that even without being especially well-versed in the technological landscape, most people have at least heard the term artificial intelligence, or its common abbreviation, *AI*. This field of study has received much attention these last years thanks to some breakthroughs that made it possible for AI to become an (often) invisible part of our everyday life, from movie recommendations, face recognition, text translation, and much more. Although often thought of as a modern invention, the very first start of the field is attributed to Alan Turing in his paper *Computing Machinery and Intelligence* published way back in 1950 [1].

Since then, the definition of what exactly defines AI has been an open debate. However, despite the exact definition, the goal is often to make programs that can perform tasks traditionally thought only to be doable by humans. John McCarthy (the same man who coined the term artificial intelligence back in 1956) offers his definition in a 2004 paper that read as follows: *"It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable"*[2].

2.1.1 Machine Learning

Today, AI can be used in many different ways. However, by far, one of the most popular branches of AI is what is called machine learning (ML), defined in 1955 by AI pioneer Arthur Samuel as *"The field of study that gives computers the ability to learn without explicitly being programmed."* [3]. Meaning that, unlike traditional computer programs where the programmer has to type the code defining what will be executed explicitly, these are ways for the program to learn by itself what action should be taken. Tom Mitchell made a modern take on the definition in his 1997 book "Machine Learning"[4] where he defines the subject as follows: *"A computer program is said to learn from experience* $\boldsymbol{E}$ *with respect to some class of tasks* $\boldsymbol{T}$ *and performance measure* $\boldsymbol{P}$*, if its performance at tasks in* $\boldsymbol{T}$*, as measured by* $\boldsymbol{P}$*, improves with experience* $\boldsymbol{E}$*."*. As stated in the quote, machine learning can be applied to solve different kinds of problems (tasks). Usually, these can be divided into either *classification* or *prediction tasks*. Classification mean that given a certain input (such as a line of text or an image), the program should define what category this specific input belongs to.

On the other hand, a prediction task take some input and estimate the likelihood of a particular outcome. For example, given a specific day of the year, the program could predict the likely temperature in Stockholm. In order to perform these feats

however, the program must first learn from previous data and adapt to the task at hand. These learning steps are usually divided into three main categories, namely *supervised*, *reinforcement*, and *unsupervised learning*.

2.1.2 Learning Approaches

By far, the supervised approach is the most common of the two machine learning techniques. Here, the algorithm (the function used in the machine learning program) is learning to take some input and produce the desired output by first looking at a *training data set*. This data set consists of inputs (called training examples) similar to those that we would like the function to be able to classify/predict, along with its corresponding correct output (labels). The goal is then that the algorithm will be familiar enough with the type of input it might encounter to make correct assumptions about the output, even on previously unfamiliar data.

Expressed differently, given a data set D consisting of n number of examples and their corresponding label on the form $\{(x_0, y_0)...(x_n, y_n)\}$ (where x_i is often a vector of numeric values, called a *feature vector*) the goal of the supervised learning is to find a function f such that $f : X \rightarrow Y$, where X is the input space, and Y is the corresponding output space. [5]

In contrast, unsupervised learning does not make use of these labeled training examples but is instead a technique where the algorithms have to learn these by themselves. This is also true for reinforcement learning that uses a reward function in order to make the model behave according to a certain favourable pattern.

2.1.3 Artificial Neural Networks

Taking inspiration from the human brain, one subset of machine learning that can apply the supervised learning technique described above is through so-called *Artificial Neural Networks*. These networks consist of several layers built up by individual components called *nodes*. The first layer is often called the input layer, and the last is the output layer. In between those are (usually) one or more so-called hidden layers. The layers are connected by their nodes, enabling information to flow through the network.

Each node can take inputs from other nodes. After processing them (usually by multiplying each input feature with weights and applying an activation function), it can send its output onward. In the input layer, inputs are not taken from another node but rather from the feature vector, for example, a text or the pixel values of an image. [6]

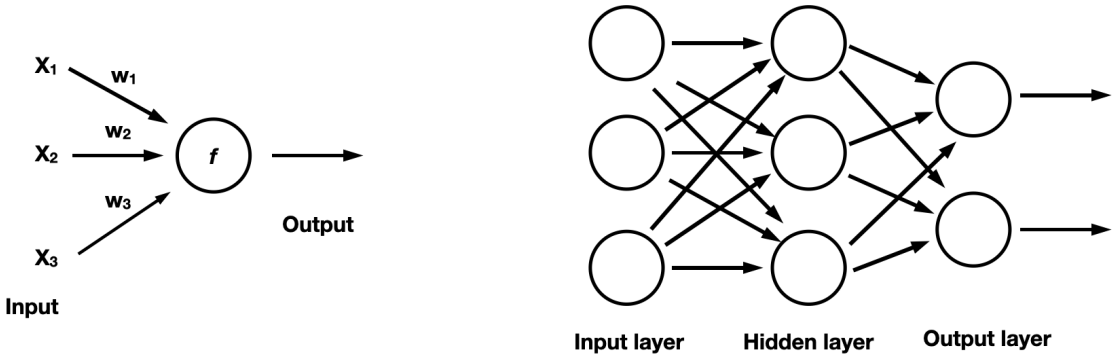

Figure 2.1: To the left is a single neuron with inputs x, weights w and activation function f. The right figure shows an artificial neural network with input, output, and a single hidden layer.

Weights

As depicted in Figure 2.1, the general function for a single neuron can be described as:

$$o = f(w_0x_0 + w_1x_1...w_nx_n) = f(\sum_{i=1}^{n} w_ix_i) = f(\boldsymbol{w}^T\boldsymbol{x})$$

To yield the correct output from a neuron and, by extension, the correct output of the entire network, the weights are added as parameters that can be changed during training. Thus, tuning the expression given as input to the activation function to produce the most favorable output. Features that are very important to obtain the final (correct) result can be given higher priority by increasing their weight. And features that do not contribute to a better final result can be decreased to have less influence.

In addition, as input to each neuron there is also a so-called *bias* term, which is a unique weight that is also changeable but not associated with any of the input features. More precisely, it is an input feature with a constant value of 1 and the bias being the weight of that constant feature. Thus, this bias gives the network more opportunities to achieve the correct output. [6]

Activation function

The activation function f takes all the input values (weights and features) to produce the output from a neuron, where the result is a single value mapping the input to the output. The function is typically a non-linear function producing a result between 0 and 1 or between -1 and 1. A common approach for a classification task is thus to use the *Sigmoid* (or *Logistic*) function in the final output layer, which maps the input to a number between 0 and 1. This value can then be interpreted as the probability of an input belonging to a certain class.

Another popular option for classification tasks is a modification of the Sigmoid function called the *SoftMax* function, where the output values are normalized. This is usually helpful if the task is to classify the input into multiple different classes, as the normalization turns the output vector into a probability distribution over the likelihood of different classes that adds to one (as shown in Figure 2.2). [7]

Figure 2.2: To the left is an input vector where the Sigmoid function is applied to create the corresponding output, here the resulting vector does not become a probability distribution. To the right is the same input vector, but here the SoftMax function is used instead. The resulting output vector now becomes a probability distribution where the sum of the vector adds to one.

The network can also have varying activation functions for different layers. For example, a common approach is to use Softmax in the output layer and other functions in the hidden layers. A popular option for these inner activation functions is the *ReLu* function, which is probably the most common activation function used in deep learning today. The main idea behind this function is quite simple. All negative values are mapped to zero, while positive values are returned as is. This may prove beneficial for a deep learning approach since it decreases the number of computations. However, it also reduces the model's ability to fit the data properly since a lot of information will be lost in the process. A solution to this so-called "dying ReLU problem" is a modified approach called the *Leaky ReLu* function. Here, negative values are not mapped to zero but rather to a linear function with a slight slope (usually 0.01), increasing the range of the original ReLu function [8]. The two difference activation functions are shown in Figure 2.3.

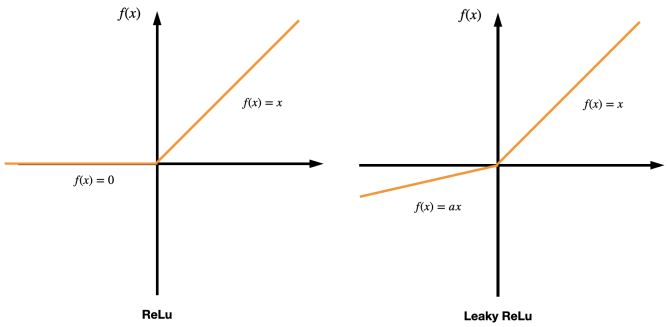

Figure 2.3: To the left is the original ReLU with all negative input being mapped to zero. To the right is the modified Leaky ReLu function, which maps negative inputs to a non-zero value.

Loss function

One of the most critical aspects that guide a network in how to update its parameters to make good predictions is the use of a so-called *loss function*. When the input is fed into the network, it creates an output by traversing all layers from beginning to end, so-called *forward propagation*. The resulting output is then compared to the target values in a reversed process called *backpropagation*. The difference between the predictions and the targets can be measured over all training examples by putting them in a loss function that delivers an average difference, the loss. The parameters in the model should then be changed in such a way that this loss is minimized, meaning the predictions are closer to the actual targets.

Like the activation function, the choice of the loss function can differ, and some functions will be more appropriate for different circumstances. Different approaches are usually used to create the loss of regression and classification models. Some of the most popular options for a model trying to predict categories are the *binary cross-entropy loss* (for binary classification) and the *categorical cross-entropy loss* (for multi-class classification). The binary cross-entropy loss function with predictions p and target values y can be described as:

$$L_{BCE} = -\frac{1}{n}\sum_{i=1}^{n}((y_i \log(p_i) + (1 - y_i) \log(1 - p_i)))$$

2.1.4 Deep learning

Just like machine learning is a subset of AI, *deep learning* is an even smaller subset of machine learning. Nonetheless, it has captured much attention in later years thanks to its successful track record of producing state-of-the-art solutions across several industries. From the seamless machine translation of languages, voice control, and the emergence of the driverless car, deep learning has been the key technology enabling these results. Essentially, it is a powered-up version of the traditional network model, where *deep* refers to the increased number of hidden layers in the network architecture.

As traditional neural networks may have 1 or 2 hidden layers, deep networks, on the other hand, can have hidden layers up to the hundreds.

As seem to be common in the field of AI, the subject of deep learning has been theoretically discussed for a long time (leading back to the 1970's [9]) but has only recently gained its recognition in practical applications. The reason for this is mainly two-fold. Firstly, deep learning requires **high amounts of labeled data** to fit the models. Today, the most proficient deep language models are trained on several million examples of text to learn semantic encodings [10]. Secondly, because of the large size of the model and the large number of tune-able parameters, it requires **considerable amounts of computational resources** to train. Therefore, training the model on a traditional CPU is usually not proficient; instead, a combination of GPUs (a specialized processing unit with enhanced mathematical computation capability) and cloud computing is often used to enable a deep model. [11]

Over- and underfitting

When training a machine learning model, we want it to incorporate a proper understanding of the training data (often called *fitting*) while also being able to generalize to make valid predictions on new unseen data.

Underfitting is when the model cannot capture the relationship between input examples and target values and therefore does not perform well on either the training or the new data. *Overfitting* is the opposite problem, where the model fits the training data really well but fails to achieve generalization. A sign that the model is overfitting is often that the validation metrics on the training data show excellent results, while the metrics on the new data become increasingly poorer the more the model trains. [12]

2.1.5 Natural Language Processing

Natural Language Processing (NLP) is the branch of AI that concerns understanding and processing language in the form of text or spoken words. It is sufficient to say that most of us interact with some form of NLP in our everyday life in the form of the voice assistant on our phones, chatbots, spam detection, and much more. NLP spans several AI fields, from traditional methods to deep learning, and can combine these methods with statistics and linguistics to gain insights from a text in much the same way as a literate human would.

However, making a computer program insightfully understand language is more complex than it might seem. In addition to the non-trivial task of just understanding the meaning of a text, human language is also full of linguistic ambiguities that make texts open to multiple interpretations and can obscure the intended meaning. Sarcasm, metaphors, and lexical ambiguity (several meanings to the same text) are just

a few examples that complicate the task. In this light, it becomes obvious why AI is the right tool for tackling these problem sets, as handling explicitly programmed software would soon be overwhelming. [13]

2.1.6 Sentiment analysis

With sentiment analysis, the NLP model aims to extract the sentiment present in some input of natural language text. The sentiment categories can either be pre-determined or automatically decided by the classifier. The sentiments can be broadly defined, such as *positive* or *negative* or more specific categories, such as *friendly* or *hostile*. [14] An example of this is illustrated in Figure 2.4 below.

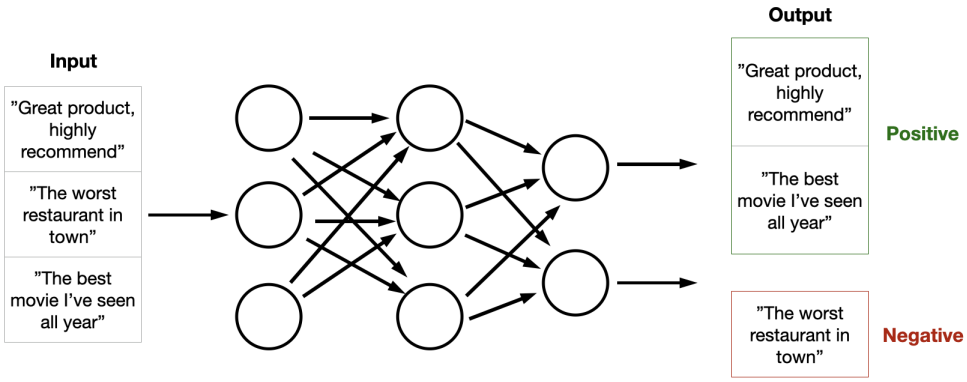

Figure 2.4: A neural network model performing binary sentiment analysis on written reviews. The results produced in the output layer are categorised as either *positive* or *negative*

2.2 Cloud computing

Today, most people use some cloud service or have at least heard the term *the cloud*, mainly when using storage for pictures or other electronic material. However, the expression of using a cloud service might create misleading associations to what the term cloud actually refers to. Although the premise of this setup is relatively simple, in essence, it is the ability to use IT resources offsite and rent resources that are not one's own. For storage or for the processing and analytics of data (to name a few), it usually proves to be more cost-efficient to refrain from investing in the hardware and software required oneself and instead use services that let the user rent different computational resources (referred to as the cloud) managed by a cloud provider allowing the resources to be accessed through the internet.

Besides the upturn of computational power available, moving resources to the cloud is often associated with increased scalability and flexibility, since the user only has to pay for the resources being used and can decide to up- or downscale it depending on the current need. Usually, this is done according to a schedule or depending on some metric deciding the threshold for the scaling policy. [15]

2.2.1 Amazon web services

At the time of writing, Amazon web services (commonly abbreviated as AWS) is the single most popular cloud provider. As of Q2 2022, it held a 34% market share of the total cloud service market, putting it way ahead of other top contenders such as Microsofts Azure (21%) and Google Cloud (10%) [16]. The business model of AWS is centered around the idea of offering cloud computing to everyone, and today it can be reached from most populated areas thanks to its many globally distributed data centers, making it the most extensive global cloud infrastructure to date.

The platform offers over 200 available cloud services that can be utilized depending on the intended use case, spanning several industries and technologies, besides the most common services that generally offer storage, databases, and computation power, there are also many specialized services that target specific use cases such as security, content delivery, or machine learning. As for most other cloud providers, the overall payment model for AWS is a pay-as-you-go approach, avoiding high up-front costs. However, the factual pricing for each service is uniquely defined. [17]

2.2.2 The AWS machine learning stack

Of the services available at AWS, there is a special section for using machine learning tools called the *AWS machine learning stack* [18]. The purpose of the stack is to offer several services directed toward users with varying knowledge of machine learning. In order to achieve this, it is built up in a layered approach where each layer requires a certain degree of proficiency. The first layer consists of the so-called AI services, which are meant to be used by practitioners unfamiliar with machine learning practices and that are willing to use pre-made solutions to solve various problems.

The second layer is the ML services, consisting of Amazon Sagemaker [19], an end-to-end integrated developer environment (IDE). This means that it is possible to control the entire ML workflow, from building and training to managing and deploying one's own models. This requires a better understanding of creating a solution and what solution might suit the task.

Finally, the third layer is the most customizable, consisting of popular frameworks and interfaces to build a solution from the ground up [20]. To use this layer in a solution meant for production, it takes deep knowledge of how to build, maintain, and manage a fully fleshed machine learning pipeline. In this thesis, it is the first and second levels that will be explored. An overview of the three different layers are shown in Figure 2.5 below.

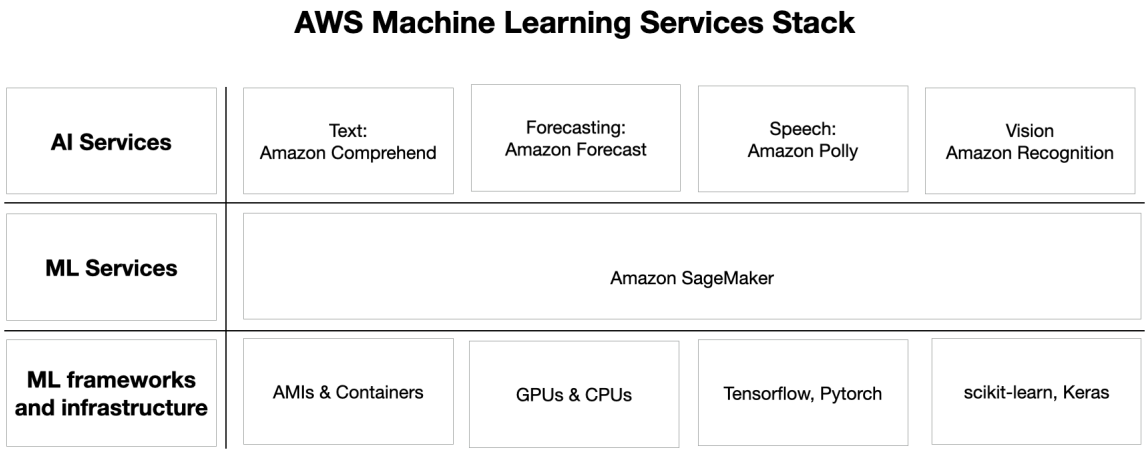

Figure 2.5: The figure shows an overview of the AWS machine learning stack's layered structure, all the services included in the stack are not represented in the image. This thesis will not examine the third layer (ML infrastructure and frameworks) [18].

2.2.3 S3 buckets

One of AWS's most popular services is simple storage, also called S3 [21]. For storing data S3 uses a public cloud storage container called *S3 bucket*, which will be what most users come in contact with when using S3. The purpose of the S3 bucket is in a way similar to the use of file folders for different objects, where each object stored in the bucket has to consist of three main parts. Namely the object content itself (the data a user would like to store), the object's metadata (including size, name, date of storage, etc.), and finally, a unique identifier connected to the object. Once the object is stored in a bucket, it can be accessed from anywhere in the AWS environment and by anyone that is allowed to retrieve it.

In order to use it, a user must first create the S3 bucket. Since the bucket, like other cloud services, is part of a data center maintained by AWS, it is necessary to specify the region in which the bucket should be created. In order to reduce latency (and, by extension, the cost of use), the region should be chosen as close as possible to the intended end-user. Before deploying the bucket, it is also possible to specify the access policies and privileges of the bucket, although this can also be modified after creation. Lastly, the bucket has to be given a globally unique name.

Creating, interacting with or deleting the bucket can be done easily through the console on the AWS website. Alternatively, the bucket can be reached programmatically (for example, from another AWS service) or through a suitable API or command-line interface. [22]

2.2.4 API's & endpoints

API stands for *Application Programming Interface* and is the part of one system that allows it to interact with other systems. Each API specifies how the systems communicate with one another through contracts and documentation, declaring how the

communication will be handled. An *endpoint* is the other outer part of the communication channel and the first contact point for communication. When a request is sent to an API, it will reach the endpoint first, allowing it to access the API's resources. [23]

2.2.5 Amazon Comprehend

As part of the first (AI) layer of the machine learning stack, Amazon Comprehend [24] is a natural-language processing service provided by AWS. As such, Comprehend uses machine learning to analyze text and present insights across several domains of NLP, Figure 2.6 shows an overview of the intended use of the service. Use cases include automatically producing summaries of a longer input text, finding common entities across several documents, and identifying sentiments, which will be the topic explored in this thesis. The tools for sentiment analysis use a pre-trained built-in machine learning model that can be accessed in one of two ways. Either by using it as is, the so-called *built-in* version, or by first training the model on training data, thereby fine-tuning it to a specific use case, called the *custom* version [25]. In neither of these options is the structure of the actual underlying model available to the public. When reaching out to the AWS technical support team about whether or not there is any way of knowing how the model is built, they left the following comment: *"No, this is not information that is shared publicly. For what it's worth, this information is not even shared internally outside the Comprehend team"* [26].

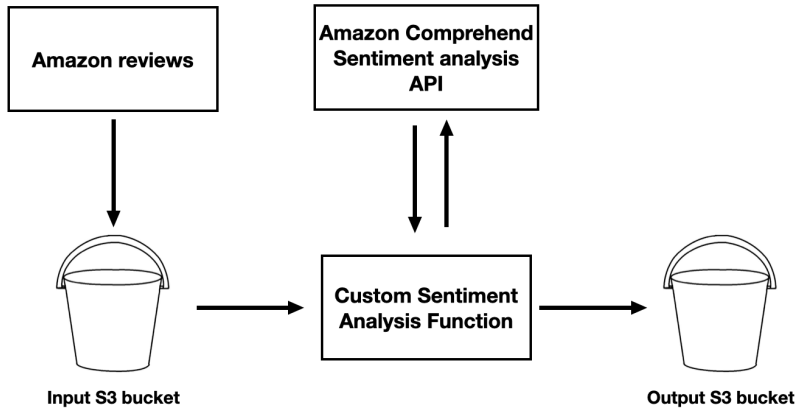

Figure 2.6: High-level overview of using the Amazon Comprehend service for sentiment analysis. The text is taken from Amazon reviews, stored in an S3 bucket, and sent to Comprehend via a custom function that reaches the Sentiment Analysis API. The result is then stored in an S3 bucket, which could be the same as the one containing the input. [27]

Like other AWS services, Comprehend can be used through several options. A user-friendly interface in the AWS console allows users to decide what to process, what tools to use, and how to store the results. Another popular option is to use AWS Lambda, a compute service by AWS. Alternatively, it can be reached by some other suitable SDKs (software development kits) which support some of the most popular programming languages [28]. This includes the AWS SDK for Python called Boto3 [29].

When uploading text to the service, there is no need to pre-process it in any way, regardless of using the built-in or custom version. Like all services on AWS, the pricing of the Comprehend service is measured in US dollars (USD, $) and is charged on a monthly basis, where different overall cost structures depend on how much the service is being used. For the work of this thesis, the pricing is based on the cost table of a monthly process of 0 up to 10 million so-called *units* (1 unit corresponding to 100 characters of text, including white spaces), which is considered reasonable for the intended usage of the service. [30]

Built-in version

At the moment of writing, there is no option to create one's own endpoint to reach the built-in versions API. However, it is possible to use the service directly via the AWS console or to reach it via the API (using, for example, Boto3). There are two different methods of text processing when using the built-in version. Firstly, the text can be analyzed in real-time for a single text with a maximum of 5 KB (about 5000 characters when using UTF-8 encoding). For real-time sentiment analysis, Comprehend only analyze the first 500 characters of the input text and ignores any additional text in the input. After analyzing the text, the resulting sentiment will be returned directly. Secondly, it can be called by scheduling an asynchronous processing job where the input comes from a file saved in an S3 bucket. When using this option, the service is no longer bounded by a maximum size on the input, which makes it more suitable for situations that handle large amounts of text for non-latency-dependent tasks.

The built-in version does not support training the model or fine-tuning it as it is meant to be used solely as an "off-the-shelf"-solution. One is meant to use the service as a sentiment analysis tool simply by uploading the input text with no extra work. When using the service, depending on the size of the text and the purpose of the analysis, the intended use must first be chosen (sentiment analysis, entity extraction, etc.). Then, in the asynchronous case, it must be defined how the text should be interpreted, either as one coherent text or several ones separated by a new line. Lastly, for both the real-time and the asynchronous options, it is also necessary to specify what language the text will be in. At the moment of writing, the service supports 12 different languages, including *English*, *French*, *German*, *Italian*, *Hindi*, *Chinese* and more.

After this, the sentiment analysis job can be initiated, and when the job is completed, a report will be produced that explains what the model finds the sentiment for each text to be. This report will be saved directly in an S3 bucket in the asynchronous case. In the real-time case, the results are returned immediately to the user without being automatically saved [31]. In the built-in version, each text is given one of the labels *positive*, *negative*, *neutral*, or *mixed*, where mixed means that the text contains both positive and negative sentiments [32]. It is currently not possible to customize these labels to fit the purpose of the analysis.

Pricing for built-in version

The cost associated with the built-in version is driven only by the number of units that are being classified. This is true both for the real-time and the asynchronous way of processing data, meaning that from a cost perspective, the time required for the prediction is not relevant, only the size of the input predicted. At the time of writing, the cost for predicting each unit is 0.0001 USD [30].

Cost examples for built-in version

We want to classify 500 texts, each being 350 characters long, using the built-in model with asynchronous processing:

Number of texts: 500
Units per text: 4 (units are rounded up)
Price per unit: 0.0001 USD

Total cost: $500 \cdot 4 \cdot 0.0001 = 0.2$ USD

Custom version

In Amazon Comprehend, it is also possible to train a custom model and use it for classification. How this model is structured or what happens during the training is not publicly available. When using this custom alternative, a piece of text with its corresponding correct label will be fed into the model in advance to train it on a particular use case. For this purpose, the training text must be representative of the text that will later be analyzed in order to get good results. The training data must also consist of several examples. The minimum number for each label is ten labeled texts (called *documents* in the AWS documentation). However, AWS recommends at least 1000 examples per label, each with a corresponding sentiment label. [33] The maximum is a total size of 5 GB for all the files used when training the model. [34] It must also be specified which language the text is in. At the moment of writing, there are only six supported languages for training the custom model, and they are *English*, *French*, *German*, *Italian*, *Portuguese*, and *Spanish*.

Later, when the model is trained, it can classify text with the labels present in the training data. Meaning this option allows for the use of custom labels, unlike the built-in model. After the training is completed, the custom model analysis can be performed on any input text [35]. However, as is the case for all supervised learning tasks, it can only be expected to work well on a text that is similar to the data that the model has been trained on.

Similar to the built-in version, when classifying text, the user is given two options: either processing it through an asynchronous job, called *batch analysis* or (unlike the

built-in version) deploying an endpoint that can be used for real-time synchronous analysis. The endpoint can be deployed, reached, and deleted via the AWS Console or an SDK. Since the endpoint is meant only for real-time analysis, it can only classify one text at a time and will return the result immediately to the user. For the asynchronous option, several texts can be classified at once, automatically saving the result in an S3 bucket [34].

Pricing for custom version

The cost model for the different analysis methods differs, although the cost for analyzing the text (as shown below) effectively becomes the same. However, the cost of training the model in the custom case is additional to analyzing it. The training cost is currently 3 USD per hour, and a monthly cost of 0.5 USD for storing the model.

In the asynchronous (batch analysis) case, the user is charged for every unit being analyzed, just like the built-in version, but at the rate of 0.0005 USD per unit. In the case of setting up an endpoint, the user will be charged for every second the endpoint is up, even if it is not being actively used. The charging rate depends on the amount of throughput that the endpoint is set to be able to handle, with a minimum of being able to handle 1 unit per second for a cost of 0.0005 USD, increasing the cost linearly with every additional unit throughput [30]. Thus, ultimately achieving the exact cost per unit as the asynchronous option if not considering the idle time when the endpoint is not actively used. Although in a realistic scenario, it would not be possible not to have some extra time when analyzing several inputs, which would add to the total cost.

Cost example for custom batch analysis

We want to classify 500 texts, each being 350 characters long, using the custom model with asynchronous processing. The model takes half an hour to train and is stored for one month:

Number of texts: 500
Units per text: 4 (units are rounded up)
Price per unit: 0.0005 USD
Cost for training: $3 \cdot 0.5 = 1.5$ USD
Cost for storage : $0.5 \cdot 1 = 0.5$ USD

Total cost : $500 \cdot 4 \cdot 0.0005 + 1.5 + 0.5 = 3$ USD

Cost example for custom real-time endpoint

The endpoint should be available for a month (30 days), where it should be able to handle 1 unit per second. The training of the model takes half an hour:

Price per second: 0.0005 USD
Number of seconds: 2592000
Cost for training: $3 \cdot 0.5 = 1.5$ USD
Cost for storage : $0.5 \cdot 1 = 0.5$ USD

Total cost : $2592000 \cdot 0.0005 + 1.5 + 0.5 = 1298$ USD

2.2.6 Instances

When accessing the resources available on AWS from one's own computer (for example, when training or deploying an endpoint for a model), the user must first set up a server that will be used as a virtual machine on the AWS cloud. In order to do this setup, the user will use a service offered by AWS called *EC2* (Elastic Compute Cloud) [36], which allows the user to access different types of virtual machines (instances). There are several virtual machines to choose from. In addition to choosing where the server should be located, it is also possible to choose different configurations combining CPUs, memory, RAM, and more, in ways that will fit the task at hand. There are five main categories of instance types to choose from. The first is for general purposes, and the other categories specialize in different aspects, namely memory-, compute-, storage- and speed optimization. In each category, it is also possible to choose from numerous configurations depending on the workload. Naturally, the cost of using different instance types differs; generally, the more resources allocated, the higher the cost. [37]

2.2.7 Amazon SageMaker

Amazon SageMaker is another layer in the AWS stack. It is a so-called *fully managed* machine learning service, which means that a lot of the setup and management is already in place, and there is no need for things such as setting up any machines, patching, or maintenance. Instead, SageMaker focuses on building, deploying, and storing machine learning models that can easily be integrated with the rest of the AWS environment. To do this, the service provides its users with a few different ways to build a model on the service depending on the purpose and the associated cost. In this thesis, the models will be built on a jupyter notebook using instances spun up on SageMaker, where it will also be trained using training data. When creating the notebook, the instance type used for training must first be chosen, which can be different from the instance used when running an endpoint. The resulting trained model will be stored in an S3 bucket.

The saved model from SageMaker can then be used to make predictions, while using a model that has been deployed these predictions are called *inferences*. There are four

main ways of deploying the model to an endpoint so that it is possible to get the model's inferences. Each way has its advantages/disadvantages and can be used in different scenarios depending on the intended use case. When invoking the endpoint to get the inference, there are a couple of options besides invoking it directly through an SDK. [38] For example, there are other services, mainly AWS Lambda [39] or AWS API gateways [40], that can be used to customize the input before taking it to the endpoint, as shown in Figure 2.7 below.

In order to monitor the models both during training and deployment, the Amazon CloudWatch tool can be used to produce logs containing metrics of cost, latency, time deployed, and more. With this tool, it is also possible to set up an alarm system if some metric is out of a specific boundary (for example, too many inaccurate predictions) and then notify the developer. This metric tool is associated with a small cost, 0.14 USD per (lower tier) instance per month. [41] As it is not strictly necessary for the usage of the service, it will not be included in the cost evaluation. However, it will be used to perform the measurements in this thesis. Also, there is a small charge for data processing for all the different inference types. However, this charge is sufficiently small in relation to the other associated costs, so it will not be considered essential for the cost evaluation and comparison.

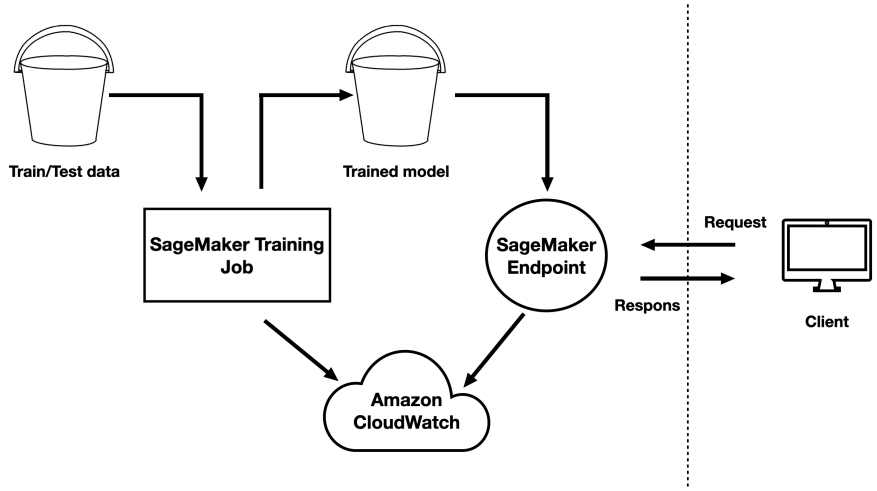

Figure 2.7: High-level overview of the SageMaker infrastructure for an endpoint deploying a model stored in an S3 bucket. The CloudWatch tool takes logs from both the training of the model and the deployment of the endpoint.

Real-Time Inference

As the name suggests, this kind of inference is intended for use in real-time, meaning the endpoint should be deployed from the time it is put online, for the entirety of the time it is being used, until it is taken down. It is best suited for tasks where it is required to handle low latency and high throughput; for example, a chatbot would be a suitable case that could experience these types of user patterns. Figure 2.8 shows an overview of how the endpoint is intended to be setup.

For this to work as efficiently as possible, there are some constraints on how this endpoint can be used. First, the model can only handle a single input at a time and has to return an inference for this input within 60 seconds. Also, the size of the single input (called the payload) can be no larger than 6 MB. If either of these constraints is violated, the endpoint will return an error and terminate the request [42].

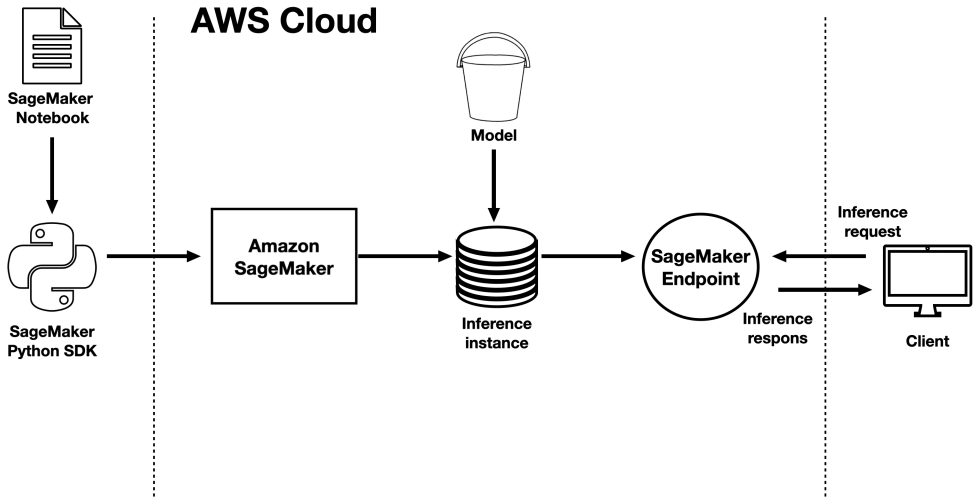

Figure 2.8: Real-time endpoint using one inference instance for model hosting. The model is set up on Sagemaker by a jupyter notebook, and the endpoint is deployed using Boto3 that communicates with the AWS cloud resources.

Pricing for Real-Time Inference

A real-time endpoint is charged for the full period that it is put online, meaning that it does not matter to what extent the endpoint is actually being used. If it is never used or called several times a second, the final cost will still be the exact same. The cost is written as an hourly rate, but the customer will be charged by the second. There is a wide range of costs, and the exact pricing depends on the instance used to deploy the endpoint. For example, the lowest resource utilization in the general purpose category has an hourly rate of 0.0056 USD. In the same category, the most expensive instance instead has an hourly rate of 6.506 USD [43].

Cost example for Real-Time Inference

We use a lower-tier instance optimized for accelerated computing to train our model. The cost for this instance (ml.p3.2xlarge) is 3.825 USD per hour, and we use it for half an hour to train the model. Then the model is deployed using a real-time endpoint on a lower-tier instance of the general-purpose category. This instance (ml.t2.medium) costs 0.0056 USD an hour and will be deployed for the entirety of one month.

Cost for training: $3.825 \cdot 0.5 = 1.913$ USD
Cost for storage : $0.5 \cdot 1 = 0.5$ USD
Hours per month : $24 \cdot 31 = 744$ hours
Cost for endpoint : $0.0056 \cdot 744 = 4.167$ USD

Total cost : $1.913 + 0.5 + 4.167 = 6.58$ USD

Asynchronous Inference

AWS introduced the asynchronous inference type in August of 2021. Unlike the real-time endpoint, an asynchronous inference puts the incoming request in an internal queue and handles them asynchronously, as can be seen in Figure 2.9 below. In order to do so, the payload must be placed in an S3 bucket, and upon invocation of the endpoint, a pointer to this payload must be given as a parameter. The results are not returned directly as in the case of a real-time endpoint but are instead stored in an S3 bucket, which can be accessed using an SDK. This can be very helpful if the constraints on the real-time endpoint are not suitable for the intended use case and there is not the same requirement of low latency. The constraint on this endpoint is a prediction time-out of up to 15 minutes, and it can also handle larger payloads, up to 1 GB for each input, making it well-suited e.g. for image analysis or text analysis that uses larger and more complex models [44].

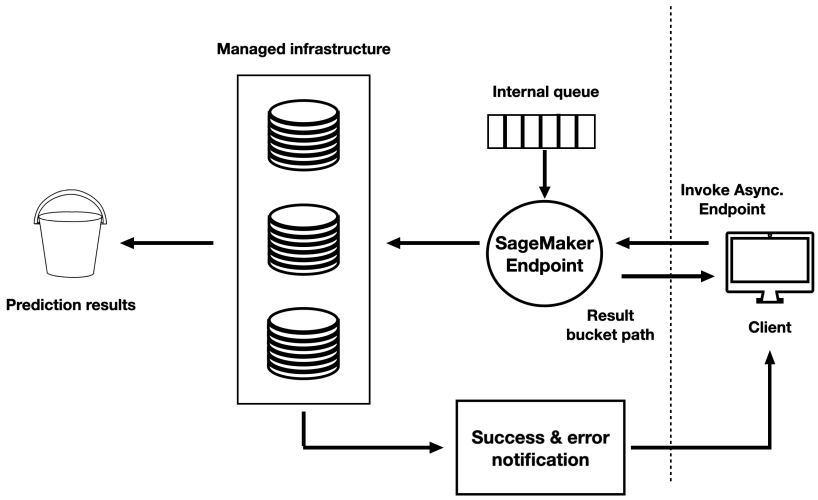

Figure 2.9: Asynchronous endpoint with an internal queue that uses several instances for deployment. Here it is the path to the S3 bucket containing the results that are returned to the user.

Pricing for Asynchronous Inference

Like the real-time endpoint, the cost for an asynchronous one depends on what instance is being used, where the cost table is the same for the two inference types. It is also being charged for the time it is up and running, although this inference type has the option of auto-scaling (described in the section below) down to zero instances. When the endpoint is at zero, it no longer drives costs. The user will, however, be

debited for the added time it takes to scale up the endpoint and the cool down time for scaling it down again. This, though to latency, means that the user will be debited for having a server put up even when it is not being actively used. [43]

Cost example for Asynchronous Inference

The same model as in the previous example is being used with the same training configurations, also taking half an hour to complete the training. It is deployed with auto-scaling so that it will go back to zero when not being used. The processing is done on an ml.t2.medium instance á 0.0056 USD per hour. For processing the payload and producing a result, the model takes 1 hour, then it has a cool down time of 30 minutes before reaching zero instances. The model will then be used once per day for one month.

Cost for training: $3.825 \cdot 0.5 = 1.913$ USD
Cost for storage: $0.5 \cdot 1 = 0.5$ USD
Total active hours per month: $1.5 \cdot 31 = 46.5$ hours
Cost for inference: $0.0056 \cdot 46.5 = 0.260$ USD

Total cost: $1.913 + 0.5 + 0.260 = 2.673$ USD

Batch Transform Inference

If the use case is not dependent on a persistent endpoint being available at all times, there is instead the option of a batch transform inference. The major difference is that it does not deploy any endpoints as in the other inference cases. Instead, we initiate a transformer object with given parameters (including the instance type) that will start a batch transformer job using the model and the input data stored in an S3 bucket. This inference type will be activated when invoked and handle the input, make predictions and then tear down the instance after the job has been completed. It is suited for even larger data sets that do not have latency requirements, since in this case there is no time-out for predictions and no upper bound on input data size. A common usage scenario is that the input data is already available and is readily stored in an S3 bucket, where it can be loaded into the prediction model, and just like the asynchronous case, the result will be a path to an S3 bucket containing the prediction results (as seen in Figure 2.10). For example, it could be suitable for evaluating our trained models on new data that has been collected to see if the model needs to be configured [42]. The different parts of the inference can be distributed over several buckets or by using the same bucket for everything.

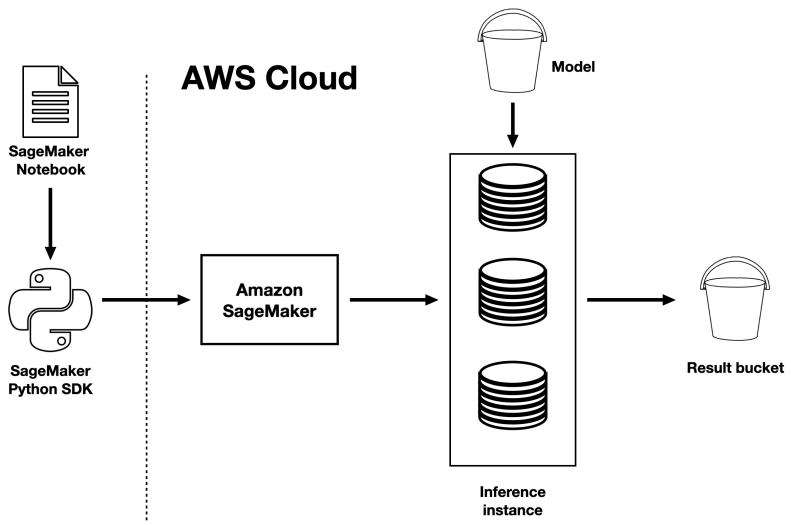

Figure 2.10: Batch transform using several instances for deploying the model. No endpoint is set up, and results are saved in an S3 bucket.

Batch Transform Inference Pricing

The price for the batch transform inference is based on the instance type used and the duration of the time a job takes to complete. Since the instance will be torn down automatically after the job has been completed, it works similarly to the asynchronous endpoint, although without requiring an auto-scaling policy. However, the customer will not be charged for any cool-down time as the cost ceases after the result has been produced. The cost for different instances is the same for the batch transform inference as for the two previous inference alternatives. Although in this case, it is not possible to use the lowest-tier instances. Instead of an ml.t2.medium instance, the lowest available instance is the ml.m5.large, with an hourly cost of 0.115 USD [43].

Cost example for Batch Transform Inference

Again, the same model as the previous examples is being used, where training takes half an hour. It is deployed on the lowest tier in the general purpose category available for the batch transform interference, ml.m5.large costing 0.115 USD per hour. The time for running an inference on the payload now takes 45 minutes to complete. The job is now allowed more resources, and it is, therefore, reasonable that the time for producing a result from the payload will take less time than in the asynchronous example. A like-sized payload will then be processed every day for a month.

Cost for training: $3.825 \cdot 0.5 = 1.913$ USD
Cost for storage: $0.5 \cdot 1 = 0.5$ USD
Total active hours per month: $0.75 \cdot 31 = 23.25$ hours
Cost for inference: $0.115 \cdot 23.25 = 2.674$ USD

Total cost: $1.913 + 0.5 + 0.2674 = 2.680$ USD

Serverless Inference

The serverless inference is the latest of the available inference types and is made for the scenario when the user does not know (or care) about the underlying computational capacity. In all the other inference types, this has to be stated explicitly in the setup and/or in the auto-scaling policy. In the serverless case, all this infrastructure management is abstracted out, and the auto-scaling is based solely on the demand from the requests. The only thing needed to state in the setup is the memory size that will be used, ranging from 1 to 6 GB. The use case for this inference could be if the incoming payload is unpredictable in time with long idle periods and sudden peaks or by shifting payload sizes.

Serverless inference allows for low latency as with the real-time endpoint and is meant to be used in much the same way. Although here, the use case should be able to handle cold starts, meaning the inference will take some time to get going before processing the requests. Like the real-time endpoint, there is an upper limit on the payload of 6 MB and a time-out of 60 seconds. At the moment of writing, the serverless inference also does not support the use of GPU:s, unlike the other inference types (used in the upper-tier instances for computing optimization), and instead is constrained to the use of CPU:s only. The result is returned after every invocation and will not be automatically stored [45].

Serverless Inference Pricing

The idea behind serverless solutions is that the user will only be charged for the time the service is actively being used. The compute charge is different depending on what memory is allocated to the endpoint and is billed by the second of its usage. The total cost also includes the time it takes to (cold) start the job and for it to be torn back down again [43].

Cost example for Serverless Inference

The same model with the same configurations is used as the previous examples. The payload consists of 1 million requests distributed over a month that each takes 100 milliseconds to complete, including the time for the up- and down-scaling. In order to do this, an instance with 1024 MB of memory is being used that costs 0.00002 USD per second.

Cost for training: $3.825 \cdot 0.5 = 1.913$ USD
Cost for storage : $0.5 \cdot 1 = 0.5$ USD
Total active seconds per month : $0.1 \cdot 1,000,000 = 100,000$ seconds
Cost for inference : $0.00002 \cdot 100,000 = 2$ USD

Total cost : $1.913 + 0.5 + 2 = 4.413$ USD

2.2.8 Auto-scaling

In order to make sure that an endpoint can handle the incoming traffic, it is possible to automatically change the instance type or the number of instances that the endpoint is using while deployed. This is done by a method called *auto-scaling*. For an endpoint, it is possible to setup an auto-scaling policy that will guide the scaling of the endpoint depending on the value of certain metrics, such as CPU utilization or the number of invocations, that can be taken from the CloudWatch logs monitoring the endpoint.

Setting up scheduled scaling to increase/decrease the resources for specific periods is also possible. In the real-time case, the number of instances can not be scaled down to zero, with a minimum of one instance at all times, unlike the asynchronous option, which can be turned to zero. In this case, the auto-scaling policy is set up the same way and can adapt to the same metrics, with the option of taking it down completely when specific criteria are being met. As stated above, serverless inference also encompasses auto-scaling that can be turned down to zero, although this process is done without any policy setup.

2.2.9 Training compiler

To reduce the training time of deep learning models, AWS introduced the SageMaker Training Compiler in December of 2021. With this capability enabled, the training of models can be reduced up to 50%. This was proven when a training compiler was used to fine-tune a GPT-2 model (a large NLP deep learning model), where the training time was reduced from 3 hours to 90 minutes [46].

The acceleration is achieved by optimizing the hardware instructions in a way that is specifically developed for maximizing compute and memory utilization on SageMaker GPU instances. For example, this is realized by using graph-level optimization, enabling memory planning and operator fusion, along with other optimization tactics such as back-end optimizations.

These are complex operations but can be integrated very simply into the making of a model in a way that will take care of the optimization steps automatically. Firstly, importing the necessary packages from Hugging Face, and then the training compiler can be enabled by adding *compiler_config=TrainingCompilerConfig()* to the SageMaker estimator class.

2.3 Word Vectorization

To work with natural language, we need a way to represent words and sentences that is not simply in the format of text. Firstly, because the formatting of words can differ, with the capitalization of letters, diacritics, and more. Furthermore, and most

importantly, machine learning techniques require a numeric input. The task of word representation is then to take natural text and convert it into a feature vector.

2.3.1 Integer labeling

The simplest way to represent words numerically would be to replace every word in the vocabulary (collection of all the words) with a corresponding number, so-called *integer encoding*, of which an example is shown in Figure 2.11. This would, however, not be sufficient for most applications, as it would introduce accidental relationships of size and spacial correlation between words.

"I think therefore I am"
Vocabulary = [I, think, therefore, am]
Embeddings = [I=0, think=1, therefore=2, am=3]

Figure 2.11: Example of integer encoding on a small vocabulary. The embedding might introduce inaccurate relations between words. Is 'I' lesser than 'am'?

Another approach would be *one-hot-encoding*, where each word in the vocabulary is represented by a vector with the dimensions of the number of words in the text. Then every word acts as an index where the value is changed to 1 in its corresponding vector, and all the other values are zero. Now every word has the same distance from each other, solving one of the problems with integer encoding. All these vectors can then be summed up to a single sentence vector. As seen in the example, the resulting vector is the same regardless of the order of words. This approach is therefore often called a *bag-of-words*. A problem when using this technique for natural language, which often has a vocabulary of several hundred thousand words, is that the vectors become high-dimensional and very sparse. Also, since it does not retain the order of words, the context is not preserved. [47]

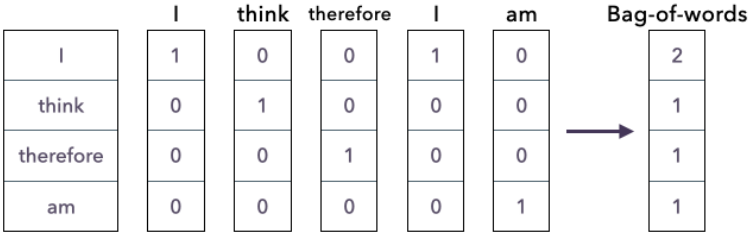

Figure 2.12: Example of the bag-of-words approach for the small vocabulary above.

2.3.2 TF-IDF

Apart from the computational burden of dealing with sparse high-dimensional vectors, the bag-of-words approach also lacks the ability to assign how important an individual word is to a text in a corpus C (collection of texts). To solve this issue, a numeric statistic called a *tf-idf score* can be created to emphasize the different words importance and assign this weight to the feature vector.

The total score is the product of two terms, the term-frequency (tf) and the inverse-document-frequency (idf). Tf is simply a measure of how frequently a word i appears in a text j. The more frequent, the higher the tf:

$$tf(i,j) = \frac{|x \in j : x = i|}{|j|} = \frac{count\ of\ i\ in\ j}{number\ of\ words\ in\ j} \tag{2.1}$$

The idf-term is a measurement of the importance of a word to the given corpus. This is represented by the number of text that contains the word i relative to all the texts j in the corpus C. The result is that a rare word gains a high idf:

$$idf(i,j,C) = log(\frac{|C|}{|j \in C : i \in j|}) = log(\frac{number\ of\ texts\ in\ C}{number\ of\ texts\ in\ C\ containing\ i}) \tag{2.2}$$

By taking the product of these two factors, we obtain the tf-idf score. The resulting vector would then be made up by letting their tf-idf score represent every word. We can understand this as giving more weight to words frequently appearing in the specific text but not common across the other text in the corpus. They are therefore interpreted as being a word that carries more importance to this particular text [48].

$$tf\text{-}idf(i,j,C) = tf(i,j) \cdot idf(i,j,C) \tag{2.3}$$

2.3.3 Word embeddings

Even though the previously mentioned classical ways of representing words can be useful for some applications, they are not very well suited for processing natural text for more complex tasks. In part because the approach of representing every word of the vocabulary as a vector leads to the problem of high dimensionality. But also, because even with the scores of the tf-idf model, it does not correctly derive any sophisticated semantic meanings from the text. These issues are what the modern approach of *word embeddings* is trying to solve by creating dense vectors that can represent words in a way that preserves the contextual meaning.

The original idea of word embeddings was outlined in a research paper from 2013 by Mikolov et al. that led up to the famous *word2vec* algorithm [49].

2.3.4 Word2Vec

The paper presenting the word2vec algorithm offers two different approaches to achieving efficient word embeddings, namely the *continuous bag-of-words model* (CBoW) and the *continuous skip-gram model*. Both models consist of two steps, first, a preproc-

essing step for creating training examples and then, feeding these examples into a shallow (few hidden layers) neural network, training it to achieve the desired vector representations.

In the CBoW model, the pre-processing consists of iterating through the text with a *context window* that incorporates some words that are next to each other. Moving the window along the text and at each step, creating pairs consisting of a word and one of its surrounding words. The first word in the pair will then be what is fed into the network with the hope of being able to predict the other word as output, like the example shown in Figure 2.13.

The continuous skip-gram model uses the same principle but operates oppositely. Instead of trying to predict a word given its context, this model tries to predict a context given the word. As the name suggests, one word is skipped in the context window, and the remaining words are used as input to the model. [50]

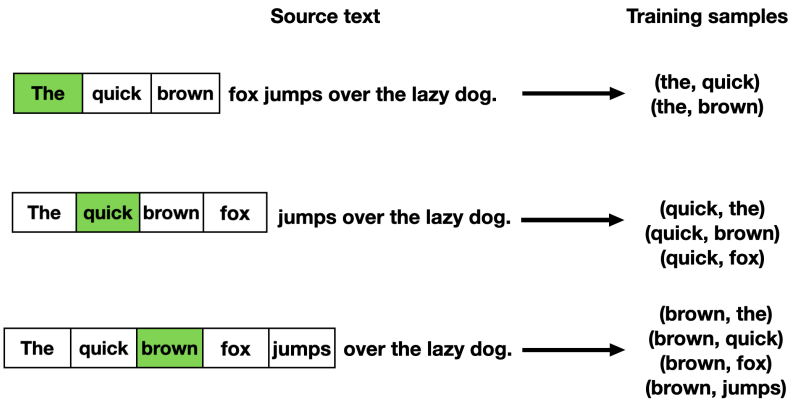

Figure 2.13: Context window of a CBoW model creating different training examples from a sentence.

The input and output layers differ depending on which of the two models is being used, but the shallow network structure is the same, consisting of only a single hidden layer. To use the words as the input they are first encoded using one-hot encoding, meaning that the input vector(s) will be of the same dimension as the number of words. The same is true for the model's output layer(s). The number of neurons in the hidden layer can vary and is given as a parameter when creating the model. [51]

The model is then trained to achieve the desired output in a supervised fashion. Interestingly though, it is not the final output that is of interest for obtaining the word embeddings. Instead, when the model has gone through the training, the model is discarded and only the hidden layers weight matrix is kept. Each row in the matrix will correspond to the word embedding vector for a single word, easily obtained by matrix multiplication with its one-hot encoding. [52] These vector values are not only dense and of lower dimensions than before, but they also represent a position in the vector space that can express semantic relations. [49]

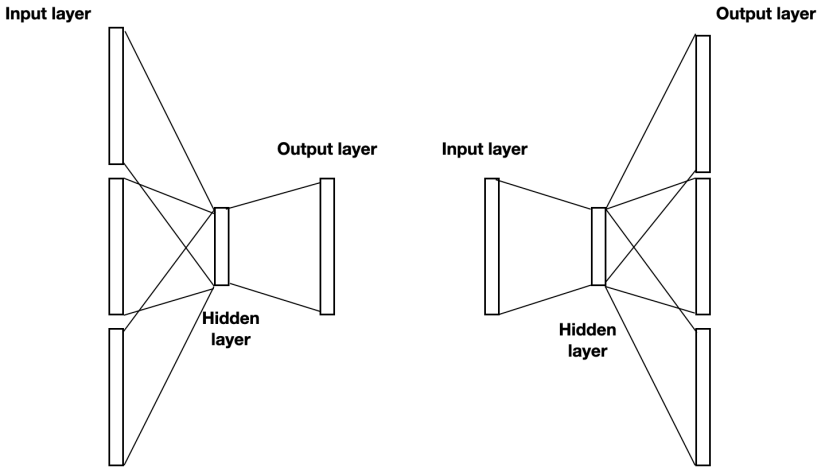

Figure 2.14: The CBoW model (left) and the continuous skip-gram model (right).

2.3.5 fastText

Word2vec solves many problems with traditional word vectorization techniques but also introduces new hurdles. The most obvious one is that the model becomes very vulnerable to words that are not present in the training data, out-of-vocabulary (OOV) words. Since the training was done using one-hot-encoding, there will be no representation of the word and now corresponding word embedding. Meaning that words will be OOV even if the word is just in a not-seen grammatical form. For example, *apples* would be OOV even if *apple* occurred in the training texts. This problem is what Facebook researchers tried to solve when publishing a 2016 paper introducing the *fastText* algorithm, which today is available as open-source. [53]

FastText builds on the word2vec skip-gram model, and the underlying structure of learning word embeddings is the same. However, fastText adds an additional step called *subword vectorazation*. In this step, the words are broken down into *n-grams* of different lengths. N-grams are the contiguous sequence of n items in a word. For example, the n-grams where $n = 3$ (called trigrams) for the word apple would be [**ap, app, ppl, ple, le**]. All the n-grams are then used to train word vectors, and the sum of all these vectors will then be used as the word embedding representing the original word.

The intuition behind this idea is that the OOV words will now share enough similarities with the n-gram representation of the vocabulary word to end up closer to it in vector space. So apart from being able to handle different inflections, the model will also represent compound words. For example, before, **plane** could not be used to predict **airplane**, but will now have enough overlapping n-grams to end up with similar word embeddings. Although storing all the sub-words will require more memory, fastText will gain a much deeper semantic understanding of the language it tries to model. [54]

fastText for classification

Presented in the original fastText paper, this algorithm can be extended to create word embeddings and classify new text into one of some pre-determined categories. The goal is to separate the texts so that their vector representations lie closer to the vector representation of the correct corresponding label rather than to the vectors of any other label.

The text to be classified is first put into a lookup layer that takes the n-gram representations of every word in the text and averages it to achieve a single vector representation for the entire text. That vector is then taken through a *linear transformation* layer to get the dimensions of the label vectors. The resulting vector is then parsed through a softmax function that creates a probability distribution over the text belonging to any of the classes. In the original paper, the loss function that is to be minimized when classifying N number of text is presented as:

$$-\frac{1}{N}\sum_{n=1}^{N} y_n log(f(BAx_n)$$

Here x_n represents the n-gram features, A is the lookup matrix, and B is the linear transformer matrix. f is the softmax function taking in the resulting vector, and y_n is the correct label vector. The training then consists of tweaking A and B to minimize the loss function.

This can be quite computationally expensive if we have a lot of possible classes since the probability has to be calculated not only for the correct label but for every other label as well. To solve this issue and reduce the time complexity, fastText instead uses the *hierarchical softmax* function based on a binary tree structure, because of this the algorithm does not have to go through every class but only through a subset of the nodes in the tree. [49]

2.4 Models

2.4.1 BlazingText

BlazingText is SageMaker's built-in algorithm used for text classification. It is an implementation of the fastText text classifier optimized for multi-core CPU and GPU architectures to accelerate the classification on AWS.

The training input to BlazingText requires a very specific pre-processing and format that has to go through specific steps (as shown in Figure 2.15). First, each label must be prefixed by the string *__label__* and be put in front of the corresponding text. Each text must consist of space-separated tokens, and each example is separated by a new line. All the examples must then be put into a single text file

that is used as input to the algorithm. When the input is fed into the algorithm, it automatically creates the word embeddings and tunes the hyperparameters to obtain the best possible result. [52]

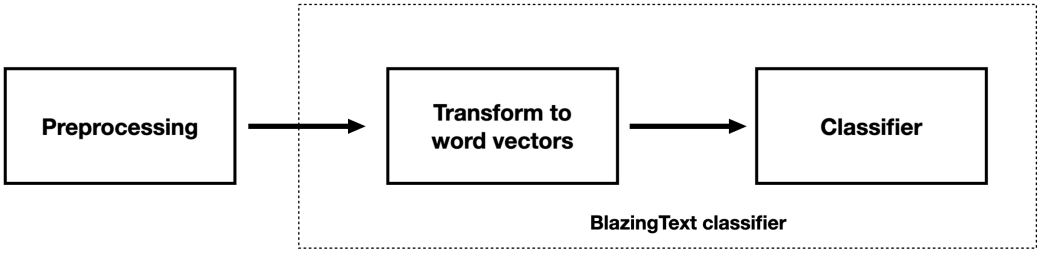

Figure 2.15: Pipeline of using the BlazingText algorithm.

Later, when using BlazingText for inference, the input must be in JSON file format with a list of all the texts wished to be classified. Here, it is not required for the text first to be separated into tokens. After the prediction is made, the algorithm will return a list of the corresponding predicted labels. [55]

2.4.2 Language models

When trying to grasp sophisticated semantic patterns, languages prove to be difficult to deal with. Both because they are huge and because they follow complex grammatical rules. Moreover, natural languages can be ambiguous, where one sentence means something else when put into a different context. Because of this, trying to model a language is always coming down to finding the probability distribution that will yield the most favorable result. One approach to make such probabilities that have become popular in recent years and are commonly thought of as state-of-the-art when trying to solve complex NLP tasks is the use of large neural network models known as *language models*. When trained on a large corpus of text, the models can be effectively applied to several NLP tasks, such as language detection, language generation, sentiment analysis, entity detection, and more. [56]

2.4.3 Transformers

Sequential data is input with some defined ordering, such as pixel values in an image, data points in a time series, or words and characters in a text. When dealing with this data, it is common to use a network architecture approach called *RNN* (Recurrent neural networks) [57] that takes in the sequence one input at a time. As the name implies, the main idea is to have an edge in the network that does not move forward toward the output but instead sends information from the previous input back to the next input in the sequence. However, there are some major drawbacks to this approach. First, the RNN takes a considerable amount of time to train, leading to very hardware-intense computations. Also, these networks are not suited for long sequences since this can lead to what is known as the *exploding/vanishing gradient problem*, where the model becomes unstable because of errors in the weight updates that are accumulating in the recursion part of the network. To solve this, the *LSTM* (Long short-term

memory) [58] network was introduced that contains a module allowing the network to retain correct updates in memory during longer sequences. Although this mostly solves the issue of unstable gradient updates, the LSTMs added modules make it a larger model and therefore are even slower to train than RNNs.

The problem with these models lies in the fact that they use recurrence as the mechanism to handle sequences. Therefore, in 2017 a paper called "Attention Is All You Need" [59] by Ashish Vaswani et al. introduced the *transformers* architecture as a new way of handling sequential input. Now, instead of relying on recurrence, transformers use *attention* and set it up in a *encoder-decoder* manner, allowing for parallelization that significantly speeds up computations while not only retaining results but making significant improvements to them.

Encoder-decoder architecture

The encoder-decoder architecture (shown in Figure 2.16) consists of two main parts: the encoder and the decoder. The idea is that the encoder will take a sequence of inputs (for example, words in a text), process it, and send it to the decoder that will output the desired result. One important note is that the input and output sequence is not constrained to a one-to-one ratio where input will have a single corresponding output. Think, for example, of machine translation of language. Here we want the translated sentence to be able to be both shorter or longer than the original input.

To solve this, the encoder has to summarize all the input features to a single vector representation (the encoder state) that can retain necessary information. In order to do this, the encoder produces vector representations called *hidden states*, generated by taking the word embedding for a word and the hidden state of the word preceding it. To produce the very first hidden state in the sequence, a unique token called *start* is used. The summation of all the hidden states are then what makes up the encoder state that is being sent to the decoder. The decoder also produces hidden states that are used for finding the most probable output where the probability distribution is the result of supervised training. Instead of the word embeddings, it is now the previous output combined with the previous state that is used when producing new states. [52]

The attention mechanism

A problem with the encoder-decoder is that every new hidden state depends on all the previous ones. This will work for shorter sequences but might become a problem when the sequences become very long. The dependencies necessary to produce the correct result must all be kept in memory at all times. For example, a reference found several passages back in a text.

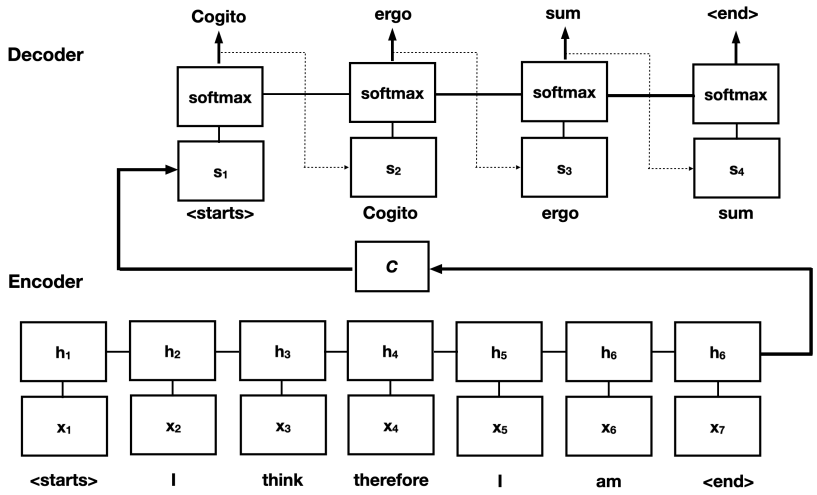

Figure 2.16: Overview of the encoder-decoder architecture, here used for translation. With encoder state C, hidden input states h_n, and hidden output states s_n.

To solve this, transformers use the attention mechanism, much like human attention, that allows the ability to focus on only the most essential states while disregarding the rest. This is done by computing an attention vector for each of the hidden decoder states that access weights to each of the hidden states in the encoder, depending on their influence on the output, meaning that here, the decoder states have access to all the hidden states of the encoder and not only the intermediate encoder state. If the input is a word, several attention vectors are produced for each input representing the interaction between the word and the rest of the sentence. An average of these vectors are then produced, which will be the final attention vector for that word. When using several attention vectors in this way, it is called *multi-head* attention.

Architecture

Previously we had to feed in the sequence one input at a time since the input states were dependent on their predecessor. However, the clever construction of the transformer architecture allows the multi-head attention to access all the hidden states simultaneously, ultimately speeding up the whole process. At first glance, this architecture will look quite intricate (see Figure 2.17 below), but upon closer inspection, it becomes clear that it is built mainly by reusing building blocks to make up the whole. The original transformer architecture was made for language translation. During training, the inputs are the entire sentence in the original language (left lower part of the figure) and the sentence in the translated language (right lower part of the figure). With the ultimate goal of creating a correct output probability of the next word in the translated language. [59]

The inputs both to the encoder and the decoder will be word embeddings. As seen previously, this will represent a mapping of words into an embedding space where similar words are closer to each other in space. However, the similarity of words is only sometimes enough since their meaning can differ depending on where they are put in a sentence. This is what the *positional encoding* is meant to solve by transforming

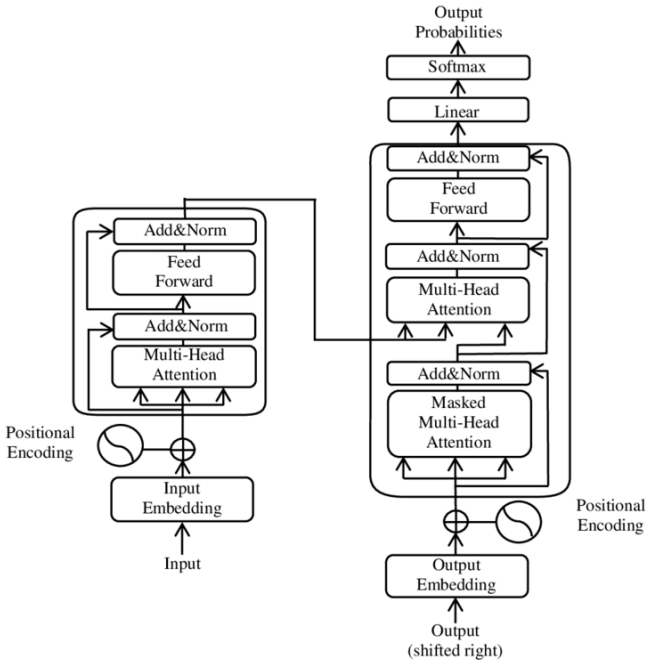

Figure 2.17: Overview of the original transformer architecture displaying the encoder (left) and the decoder (right). The figure is taken from Wikimedia Commons free media repository. [60]

the embedding vectors to not only represent words but also add positional (context) information to them. Next, in the encoder part, the embeddings are passed as input to a multi-head attention block producing attention vectors for each word as discussed above. These attention vectors are then each passed to a feed-forward network to ensure that they are in the right format to be handled by the decoder they will be sent into next.

In the decoder part, another multi-head attention block can be found that takes the output of the encoder and the output embeddings after it has been sent through a *masked* attention block. This block will turn all the right-most parts of the output embeddings to zero, masking this part of the sentence. If the sentences were not masked, the transformer would trivially be able to find the next word. The output embeddings are unmasked one at a time, making an output probability prediction of the next word at every instance and updating the weights accordingly.

These transformer blocks can be combined into a larger structure where several encoders and decoders can be stacked on top of each other. In the original paper, six both encoder and decoder blocks were used.

2.4.4 BERT model

The transformer has been a monumental shift in language models and has inspired other architectures that build upon its principles. One of the most popular imple-

mentations for tasks such as text classification is the BERT (Bidirectional Encoder Representations from Transformers) model. This model structure is based on the encoder part of the transformer, where several such encoders, here called *attention layers*, are stacked on top of each other. The original BERT consisted of 12 such layers.

Training this BERT model to handle a specific task consists of a two-step process. Firstly, the model has to be trained to incorporate an understanding of the language it is trying to model, and this is the most time and resource intense part called the pre-training. Then, in a much less intense step, the pre-trained model can be trained to perform a specific task during so-called *fine-tuning*.

The pre-training consists of getting the BERT model to perform well on two tasks requiring a sophisticated understanding of the language. In part, the training goal is to fill in the blanks of a sentence where one or more words have been masked, so-called *masked language modeling* (MLM). The other part of the training consists of verifying whether a pair of sentences is correctly put next to each other, called *next sentence prediction* (NSP). In Figure 2.18 (taken from the original paper introducing the BERT model), the words are put in as tokens where some of the tokens have been masked and also including some special tokens marking the start of a text ([CLS]) and the separation of sentences ([SEP]). Inside the model, the word tokens have been embedded by summarizing a pre-trained embedding vector incorporating the relation between words, a segment vector with information about the sentence a word belongs in, and finally, a position embedding vector. The model's output consists of one token, C, indicating a boolean value of the NSP prediction, along with the MLM predictions of the masked tokens. These tokens are then put into a softmax layer with as many neurons as there are words in the vocabulary to get a probability vector over the predicted word. This vector is then compared to a one-hot-encoded vector of the correct masked token with the intent to minimize a cross-entropy loss. The more certain BERT is about the correct token, the lower the loss will be. This, of course, needs a lot of training examples, and with the 12-layer structure of BERT consisting of over 110 million parameters, this becomes a resource-heavy task.

Once the model is pre-trained, it can be used for a variety of different, more specific tasks. The nice thing here is that we can use the knowledge already incorporated in the model too. The inner structure of the network consisting of neurons using the *GelU* activation function (a slight modification of the RelU function) is kept as is. The linear outermost fully connected layer, however, is replaced by a new (untrained) network layer that is able to represent the new task. For example, this can be a network used to produce a vector indicating whether the text is positive or negative. When training the model on a new dataset that represents the task we want it to solve, the only thing that must be trained from scratch is this new output layer and some minor tweaks to the interior weights of the model, making the fine-tuning a much faster and less resource-heavy procedure. [61]

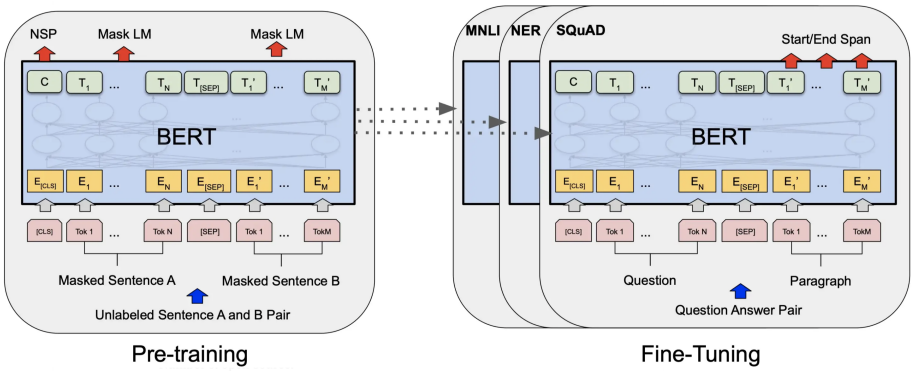

Figure 2.18: Overview of the two-step training process of fine-tuned BERT models, pre-training (left) and fine-tuning (right). The figure is taken from the paper introducing the BERT model by Devlin et al. 2018. [61]

2.4.5 Knowledge distillation

In NLP, usually, the larger a model is, the more complex patterns it can find, and it will ultimately be able to model languages more correctly. However, the trade-off is that it will take up more memory and require more computational resources during training and run-time. To battle this, a way of trying to make smaller models that can retain equivalent or near equivalent results is through *knowledge distillation*.

During knowledge distillation, a sizeable well-performing network called the *teacher network* is used when training a smaller, *student network* to perform the same task as the teacher [62]. Much like a human mentor, the hope during training is to transfer useful information about the task from the teacher so that it is echoed in the weights of the student network.

2.4.6 DistilBERT

The teacher-student technique of knowledge distillation has been successfully implemented on BERT models, producing significantly smaller and faster models that can reproduce comparable results. The simple idea behind the DistilBERT model is to copy the attention-layered structure of the original model but reduce the quantity, effectively reducing the number of parameters. According to experiments made by Xiaoqi Jiao et al. in 2019 [63], it is most effective to keep every other layer of the teacher model when producing the student model.

During training, the student model is trained just like the larger teacher model was by producing a loss representing how well it was able to produce the correct answer. However, here the training step is also combined with two other functions that use the teacher model with the hope of helping the student find a suitable configuration more effectively. [64]

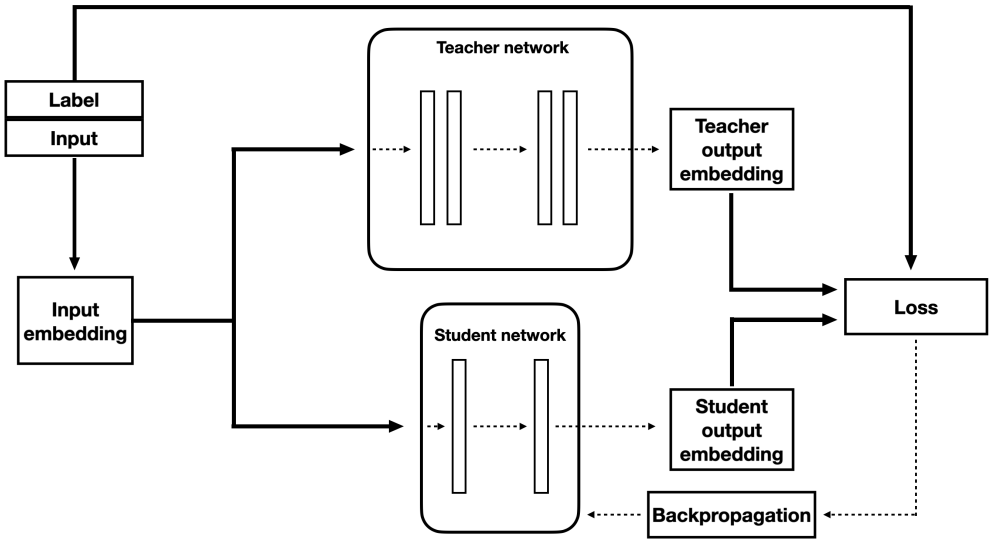

Figure 2.19: An overview of the distillation training process of the larger teacher model and the smaller student model.

The first such loss compares the output probabilities produced by the output softmax layer from the teacher model t_n with the output of the student model s_n in a loss function called the *teacher-student cross-entropy loss*. With the goal of minimizing the loss making the student behave as the teacher.

$$L_{tsCross} = -\sum_{n=1}^{N} t_n \cdot log(s_n)$$

The second function is not trying to make the two outputs as close as possible, and it is instead concerned with trying to make the student output vector $S(x)$ aligned with the output vector of the teacher $T(x)$, disregarding the norms and origins of the vectors. This information is given by a *cosine loss* as follows:

$$L_{cosine} = 1 - cos(T(x), S(x))$$

The final loss is then calculated by averaging the three respective losses with the goal of minimizing them.

2.5 Evaluation metrics

To obtain a model that produces favorable results, the training and testing must be measured to represent how we would like the model to behave. When the task is to classify the input into different pre-determined categories, one of the most popular approaches is to use the so-called *f1-score*. This method combines two other more straightforward metrics (*precision* and *recall*), obtaining the harmonic mean between them as the basis for evaluation.

Precision

Say that we want to predict some input examples that belong to several different classes. Precision is a measure taken for every class that shows how well the model recognizes that inputs should correctly belong to the corresponding class. The prediction for some class X is calculated as:

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

- TP is the true positives of class X, I.e., the outcome where the model correctly predicts the class X.
- FP is the false positive of class X, I.e., the outcome where the model incorrectly predicts the class X.

Recall

Recall is the second component of the f1-score and measures how many instances, out of all instances of some class (X) that were correctly predicted. For every class, the recall can be calculated as follows:

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

- FN is the false negatives of class X, I.e., the outcome where the model incorrectly predicts an input belongs to another class that actually belongs to class X.

F1-score

Now the f1-score can be calculated for every class. This is done by combining the precision and recall into a mean as:

$$\text{f1} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 * \text{TP}}{2 * \text{TP} + \text{FP} + \text{FN}}$$

When executing this calculation for every class, the arithmetic mean can be taken over all classes to achieve a single value of the model performance.

3 Method

3.1 Use case

In order to compare the different models, they are evaluated on a specific use case regarded as realistic for how Sigma Technology Cloud would use the service. The use case implies handling a sporadic input of collected documents of Swedish text two times every week (with equally sized inputs) for a month. In this scenario, having low latency of the model will not be considered a requirement. Instead, the evaluation will be based on the model's ability to accurately find the text's sentiment and its total cost.

For simulating this scenario, the training of the models is done on a relatively small set of inputs (10000 samples) and evaluated on an even smaller validation set (2000 samples) of Swedish text. A real-life example of such a use case would be if Sigma Technology Cloud developed a new product that they make available to the public. Then collecting unlabeled reviews from users of the product, they want to find the sentiment across the reviews. This is done by first training the model on a representative set of documents. They then collect and feed the texts to the trained model twice a week for a month. For this, they are interested in the model's performance and its cost.

3.2 Data set

To resemble the use case described above, the experiments will use a data set consisting of Swedish reviews expressing either positive or negative sentiment. The reviews are scrapped from the publicly accessible data on reco.se, incorporating several different products/services from several companies in various industries.

The reviews consist of a short text and a rating of 1-5 stars given by its author, depending on their feelings about the product/service. Reviews with 1-2 stars are considered negative and 4-5 stars as positive. The reviews that were given three stars are considered neutral and, therefore, not a part of this binary data set.

1-star review:*"Usch, personalen är seg, och alltid när jag kommit hem har jag upptäckt att det fattas mat, ALLTID!.."*

5-star review: *"Otroligt trevlig och proffessionell mäklare, kan varmt rekommendera!.."*

The data set was put together by Tim Isbister, who gave his permission to use the data in this thesis [65]. The original data set consists of over 100000 text examples with corresponding labels. However, only a subset of these examples was used to simulate more scarce access to relevant data. First, for training data, a balanced set of 10000 samples (5000 samples for each label) was taken out from the original set. Then, two nearly balanced sets (see the bar chart in Figure 3.1) of 2000 other samples were picked out, one for testing and one for validation.

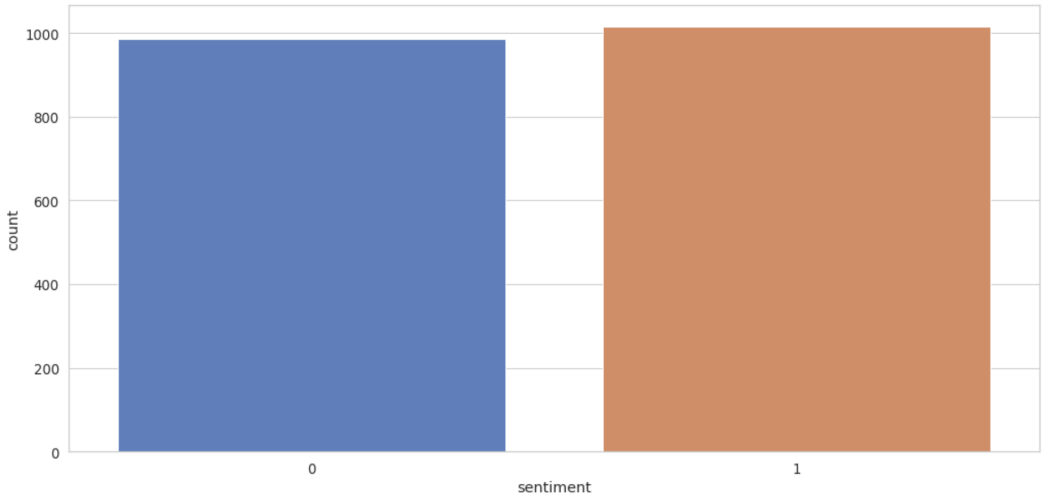

Figure 3.1: Count of the two sentiment labels, negative (0) and positive (1), in the test data set

The original data set was downloaded from GitHub and saved as a CSV file inside an S3 bucket. The data was divided into the subsets used in the experiments with the built-in functionality of the scikit-learn library, allowing for the automatic extraction of a balanced data set, taken from samples in a randomized fashion. The resulting data sets were then stored back into the same S3 bucket.

3.3 Evaluating Amazon Comprehend

In order to access the built-in version of the Amazon Comprehend sentiment analysis tool, the experiment uses the AWS SDK for Python (boto3). This allowed for the use of the service's functionality and for reading and writing data from an S3 bucket to be done programmatically. The sentiment analysis service provided by Amazon Comprehend does not require any pre-processing before being fed into the model, so the text retrieved from the saved data sets inside the S3 bucket was taken as is.

3.3.1 Built-in version

As stated in the background section, the limitations of using the built-in version in real-time are that only a single text can be fed in at a time and that it can not be larger than 5 KB. However, since the built-in version does not use any training, only the validation data set must be considered, and, in this case, it does not violate the constraint.

The built-in API is reached by creating a low-level service client in boto3 that can call AWS services and reach resources that are stored in the same region (if allowed by access policies). For real-time analysis, the texts of the validation set were first read from the S3 bucket before being looped through one at a time, using it as input to the client's *detect_sentiment* function. This function's return value is a dictionary containing the different probabilities for each sentiment class and the sentiment the model finds most probable. For each result produced the most probable sentiments were saved and stored for evaluation.

For using the built-in versions asynchronously, the same client can be used as above, but instead, initiating the analysis job by calling the function *StartSentimentDetectionJob*. Now, instead of reading the input from the S3 bucket, the location of the data set (specified by the S3 bucket URI) can be given as a parameter. Along with the language code, the separation token is used to distinguish the different texts and the URI for the output bucket. When finished, this will produce a JSON file containing the results for each text, given in the same dictionary format as the real-time analysis.

Regardless of how the built-in version is used, it does not allow for custom labeling. Since the data set is meant to represent sentiments that strictly fall into one of the binary categories, a *neutral* or *mixed* label could be considered wrongly classified. Because of this, when creating estimations against the actual labels of the validation set, predicted neutral and mixed labels were replaced with purposefully wrong labels.

3.3.2 Custom version

Just like the built-in version, the API was accessed through the Python SDK and set up on a jupyter notebook instance on Amazon SageMaker. After constructing the training data to fit the two-column format used by Comprehend, it was stored in an S3 bucket. The location of this training data and the language code were then used as parameters when initiating the *CreateDocumentClassifier* operation. All the supported languages were used for creating model versions, and after each model had been created, they were stored on AWS.

Each of the trained custom models was first used for (asynchronous) batch analysis on the texts of the validation set. The result is a JSON file just like the built-in version, the only difference being the use of custom labels from the training data.

For the real-time analysis, an endpoint must first be created. When initiating it, the inference unit rate is decided and what model the endpoint will take its predictions from, specified by providing the model's unique ARN (Amazon Resource Name). When created, the real-time predictions of the validation set were extracted by again

going through the texts one at a time and storing the result. Even if it is not being actively used, the endpoint will be kept up as long as the user does not delete it. An overview of different parts of an analysis performed on Amazon Comprehend can be seen in Figure 3.2 below.

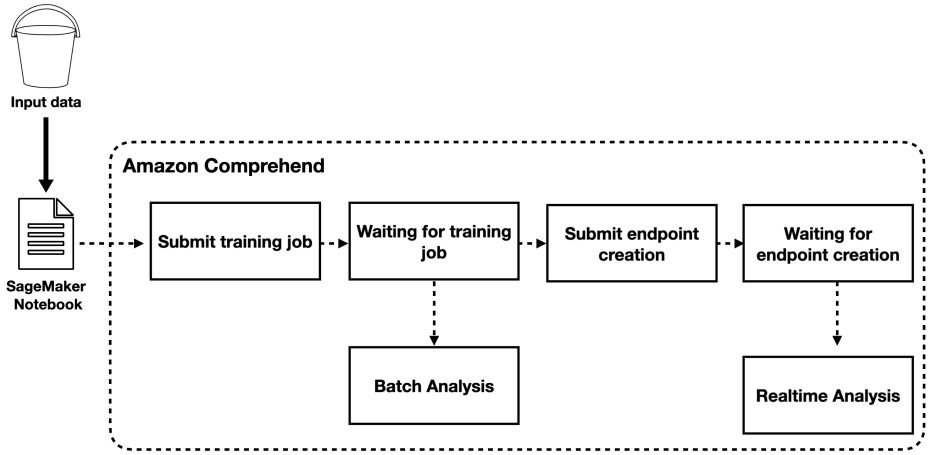

Figure 3.2: Workflow of evaluating the Amazon Comprehend custom version. Creating a batch analysis and a real-time endpoint.

3.4 Evaluating Amazon SageMaker

The models used in this evaluation were constructed and trained on SageMaker jupyter notebook instances. Through the notebooks, the trained models can easily be uploaded to AWS and deployed for inference, either directly from the notebook or from the SageMaker console interface.

Different approaches for the sentiment analysis model were tested, using the classical vectorization techniques combined with classifiers, the built-in blazingtext algorithm, and transformer-based models. The best performing models were then deployed for inference to evaluate the production cost of the solution.

3.4.1 Classical word vectorization

First, models were trained that combine classical word vectorization techniques (bag-of-words and td-idf) with simple machine learning methods for classification. The hope is that despite the models' simplicity, they would be able to solve the task sufficiently and on par with the Comprehend models. Maybe even showing an advantage of cost efficiency since training the models would also be much simpler than training neural networks.

To obtain the vectorization of the text, the in-built functions of the machine learning library sci-kit learn were used. *CountVectorizer* to get the bag-of-words representation

and *TfidfVectorizer* to get the td-idf weighted vectors. These vector representations were then used as input to some of the most common classifiers used for text classification, namely Logistic regression, *Support vector machines* (SVM), and *Naive bayes*. All classifiers were also obtained from the scikit-learn library with the default parameters for each classifier.

Different n-gram features were extracted from the results and used in three different sets of experiments. The first is with only unigram features, where every word is a single feature in the vector. The second set of experiments is when the vector contains the feature representations of both the unigram and the bigram (every pair of consecutive words). And lastly where the vector contains both uni- bi- and trigram (every triplet of consecutive words) features.

The classifiers were first trained on the training data set of 10000 text samples before being tested on the 2000 samples in the validation set. From which the results of the experiments are obtained.

3.4.2 BlazingText

The blazingtext algorithm is the only available option for text classification using one of the built-in models of SageMaker. Just like the algorithms of Amazon Comprehend, it can be used from the console interface. However, in this experiment, the algorithm was retrieved from inside a jupyter notebook instance using boto3.

All the text, both for training and testing, was pre-processed by first removing all special characters using NLTK. Then the text must be converted into a space-separated format (including only words and separation symbols) since this is the format blazingtext expects to handle. For this, the *punkt* tokenizer, imported from the nltk library, was used and applied to every text sample. Lastly, the necessary pre-fix *__label__* <label> was also inserted at the start of every text.

For retrieving the algorithm, it must be identified by its unique URI, called *image uri*, specifying which framework (algorithm) is to be called and from what region. This image URI can then be used to create the *estimator* instance, allowing the algorithm to run on the AWS resources. For this experiment, the smallest possible instance was chosen (ml.m5.large) since it is associated with the lowest costs.

The last step before starting the training will be to set the hyperparameters of the estimator. The *mode* parameter is set to *supervised* in order to obtain not only the vectorization but the whole of text classification from the algorithm. Other parameters (such as learning rate) are set to their default value [66], except for the epochs that were gradually increased in order to obtain the most favorable result.

3.4.3 Pre-trained models

The pre-trained transformer-based models were taken from Hugging Face [67], where they are hosted. There exists a plethora of different models, combining various model configurations with training data of different contents and sizes. For the purpose of this experiment, five different BERT-based models were evaluated against the use case. Two so-called *base* models, one of the original BERT model size (12 layers) called BERT-base, and a smaller distilled version (6 layers), called *DistilBERT-base*. Both the base models have been pre-trained on large corpora of English data and are often used as a base case for model evaluation.

Two other models, *BERT-multilingual* and *DistilBERT-multilingual*, are of the same inner structure as the base models. Although here they have been trained on corpora from Wikipedia, consisting of texts from the top 104 languages with the largest collections of Wikipedia articles (Swedish included).

In order to further explore the importance of the pre-trained data, another model called *KB/BERT-swedish* was also evaluated. This model has been pre-trained by the national library of Sweden and is trained on text from various sources of Swedish text, including books, Swedish Wikipedia articles, government publications, and internet forums.

Fine-tuning

The pre-trained models mentioned above are then fine-tuned using the training data set so to fit the specific purpose of the use case. Before this can be done, the data must go through a couple of preparation steps so the model can process it properly. In this case, the data can be directly downloaded from Hugging Face, where it is hosted as a data set called *swedish reviews*. The data is extracted by using the *datasets* library, which will automatically return the data in the format needed during pre-processing. If the data set were to exist outside of Hugging Face, it could easily be converted from a pandas dataframe into the correct format by using the *Dataset.from_pandas* function from the dataset library.

When the data set has been extracted, it must first be tokenized into the different text fragments used when training the BERT models. This is easily done with Hugging Face models by using the *Auto-tokenization* class from the transformer library. The tokenizer takes the name of a pre-trained model as input, returning the respective tokenizer that can be used to get the right tokens used by the model. When used on the data sets, it returns a new set consisting of the input tokens, the masked segments, and the corresponding labels. The resulting tokenized data is then uploaded to a session bucket in S3 that will be used to access the text during fine-tuning. The training and testing data must therefore be stored on different paths in the bucket.

When this is done, a Hugging Face estimator can be created. When creating the estimator, it will take several hyperparameters as input. Including what model should be used, the number of epochs, what metrics should be evaluated, as well as if the training will use spot training or the training compiler. The choice of instance type was assessed by using the smallest possible instance type that can run the model's training.

The model will then be fine-tuned by calling the fit function of the estimator and giving it the input paths to the training and testing data. After the fine-tuning has been finished, the model is stored on AWS and can be used for deployment.

For insights into the model performance and the training procedure, the metrics that were defined in the estimator can be investigated in the logs created by Cloudwatch. In addition, there will also be detailed information on other metrics such as memory usage, CPU utilization, and the like.

Hyperparameters

Selecting the optimal hyperparameters can significantly affect the final performance of the trained models. Because of this, several configurations of hyperparameters were experimentally tested, and the best performing setup was kept as the model's final result.

In the fine-tuning of the models, learning rates of 5e-5, 4e-5, 3e-5, and 2e-5 were tested using the Adam optimizer. These choices are the same as in the original paper introducing BERT [61]. In the same paper, a batch size of 32 was used. Here the models are using the same batch size as well as added experiments with batches of size 16.

The models were trained on ten epochs in order to capture the trend of the results and the loss function. The epochs were then modified to capture the optimal result when the best performing model had been established.

3.4.4 Deployment

The best performing model (based on the f1-score) was then set up for deployment using the four different options for inference. The model can be extracted from the S3 bucket output path specified when training the model, after which it can be deployed using the *deploy* function of the Hugging Face estimator.

The only thing that must be specified for deploying a real-time endpoint is the endpoint name and the instance it will run on. For this, the model was run on several instances, determining the minimum instance type that would allow the endpoint to work properly.

The serverless option is similar in its simplicity to deploy. Although here, it is not the instance type that must be specified but rather the memory size (ranging from 1024 MB up to 6144 MB). Like in the real-time case, several memory size configurations were tried to reach the smallest possible value.

For the asynchronous inference, the endpoint was set up with an auto-scaling policy so that it would scale down to zero when not actively used. This configuration was set up using the *CreateEndpointConfig* where it is possible to declare the minimum number of instances used by the endpoint (here, 0) and how many instances will be used when invoked (1 instance in this case). The configuration and the URI to the trained model are then incorporated when creating the endpoint with the API *CreateEndpoint*.

Lastly, the batch transform analysis was tested, which can easily be initiated from programmatically using boto3 by using the API *CreateTransformJob*. Where the URI to the trained model is provided, along with the URI for the S3 bucket containing the input data, as well as the bucket where the resulting predictions should be stored.

4 Results

4.1 Amazon Comprehend

When using Amazon Comprehend, comparing different language codes showed that regardless of which of the two versions was used and how the analysis was set up, English always proved to be the best language for maximizing performance. An important note on the final results is that the way the models were set up (synchronously or asynchronously) did not affect the model's performance in any of the configurations. In the results presented below, the performance (f1-scores) are taken from the evaluation of the test data set.

4.1.1 Built-in version

Performance evaluation

When making predictions on the validation set texts, the number of labels that were either *mixed* or *neutral* were 205 out of 2000 inputs total. Therefore some modifications to the results had to be made so that it would be compatible with the intended (binary) use case. This was also necessary in order to make a fair comparison between the different services and configurations.

One way of modifying the result of the model is to simply remove all the predicted labels that do not fall into one of the categories *positive* and *negative*, then make an evaluation of the remaining predicted classes and their corresponding labels. The complete evaluation of all the languages available can be found in appendix A.1. The best performing language code in this case turned out to be English with an f1-score as shown in Table 4.1 below:

Built-in model, English (binary)	
F1-score	0.93

Table 4.1: Performance of Amazon Comprehend using English as language code. Here inputs labeled as either mixed or neutral have been removed from the evaluation set.

However, after these labels were removed, the model can be thought of as only evaluating the inputs of which it has a high certainty of falling into one of the extreme categories, effectively removing the examples that would be harder to classify. One method for adjusting for this fact would then be to keep the 205 inputs that were

labeled as neutral and mixed, but assign them a wrongfully classified prediction. As can be seen in Table 4.2, this result in a significant decrease of the models performance.

Built-in model, English (adjusted)	
F1-score	0.84

Table 4.2: Performance of Amazon Comprehend using English as language code. Here inputs labeled as either mixed or neutral were deliberately assigned the wrong corresponding label.

Production cost

When using the built-in version, both for real-time and asynchronous analysis, the number of units for the test data is rounded up to 5563 at the cost of 0.0001 USD each. With no added cost for training or storing the model, like-sized inputs processed eight times over a month result in a total cost of approximately 4.45 USD.

Built-in model	
Deployment	**Cost (USD)**
Asynchronous	4.45
Real-time	4.45

Table 4.3: Comparison of the total cost for one month in production of the two different deployment methods. Showing that no matter what option is used, there is no difference to the overall cost.

4.1.2 Custom version

Performance evaluation

When evaluating the results for the custom version of Amazon Comprehend the models were trained using all the different supported language codes. The results show that the model solves the problem well with some training, and the difference between languages is not severe. The very best performance, however, comes from using English as can be seen in Table 4.4.

Custom model	
Language	**F1-score**
French	0.92
German	0.93
English	**0.94**
Italian	0.90
Portuguese	0.90
Spanish	0.89

Table 4.4: Comparison of different language codes when training the custom version of Amazon Comprehend. Where the English language code yield the best result by a small margin.

Production cost

The total training time of model was 58 minutes, with an hourly rate of 3 USD. For the asynchronous analysis, the cost is calculated based on the number of units being processed. The number of units for the test data is the same as before (5563), but now at 0.0005 USD each. With the added cost of the training and 0.5 USD for storing the model per month, the total cost of 8 like-sized inputs (two per week for a month) becomes:

$$(5563 \cdot 0.0005) \cdot 8 + (58/60) \cdot 3 + 0.5 = 25.66$$

Deploying the model as a real-time endpoint with the capacity of 1 inference unit gives the hourly rate of 1.8 USD, and keeping it up for the entirety of a month results in 1296 USD. The total production cost of the custom model deployed as a real-time endpoint is calculated in the formula below and a comparison between the two costs can be seen in Table 4.5.

$$1296 + (58/60) \cdot 3 + 0.5 = 1299.4$$

Custom model, English	
Deployment	**Cost (USD)**
Asynchronous	**25.66**
Real-time	1299.4

Table 4.5: Comparison of the two deployment opinion for the custom version of Amazon Comprehend. Showing a clear difference between the final costs.

4.2 SageMaker model

4.2.1 Classical word vectorization

Evaluation (f1-scores) of classical word vectorization and TF-IDF were derived by using uni- bi- and trigrams, combined with classifiers extracted from the scikit-learn library.

The results show that across all n-gram experiments can be seen in Table 4.6, where the support vector machine (SVM) method obtains the best performance. Interestingly, although the use of tf-idf scores decreased the performance for the two other classifiers, it did increase the performance of SVM. Furthermore, there seems to be a measurable increase in performance overall when increasing the inputs from only unigrams to including bigrams as well. However, the effect appears to level out, and in some cases even decrease, when adding the trigrams to the features. Therefore a support vector machine classifier combined with tf-idf vectorization with a unigrams-bigrams combination yields the overall best result.

Unigrams			
Classifiers	**Logistic Reg.**	**SVM**	**Naive Bayes**
Bag-of-words	0.928	0.914	0.907
TF-IDF	0.926	**0.933**	0.898

Unigrams + Bigrams			
Classifiers	**Logistic Reg.**	**SVM**	**Naive Bayes**
Bag-of-words	0.929	0.929	0.911
TF-IDF	0.921	**0.937**	0.896

Unigrams + Bigrams + Trigrams			
Classifiers	**Logistic Reg.**	**SVM**	**Naive Bayes**
Bag-of-words	0.930	0.928	0.913
TF-IDF	0.912	**0.934**	0.898

Table 4.6: The three tables above show the different options for how to derive n-grams from the text. Cross-comparing them with both word vectorizations and the use of different classification methods. The use of SVM combined with tf-idf scores and a combination of uni- and bigrams turns out to show the most prominent result.

4.2.2 Built-in algorithm

Running the built-in blazingtext algorithm for a different number of epochs showed that the most promising performance occurred after about 30 epochs. After this point, signs of over-fitting starts to take place, worsening the performance on the test data set. Using this optimal number of epochs, the algorithm shows promising results (seen in Table 4.7 and Figure 4.1 below) but is still beaten by the best performing approach in the previous experiment.

Figure 4.1: The f1-score of the blazingtext algorithm when trained for various numbers of epochs. The optimal result is obtained by stopping the training at around 30 epochs.

Blazingtext, 30 epochs	
F1-score	0.922

Table 4.7: The performance of the blazingtext algorithm when being trained for the optimal number of epochs.

4.2.3 Pre-trained models

When training the distilled versions of the BERT model, it was able to run on an ml.p3.2xlarge instance type. This was the smallest possible instance that was able to perform the training. The cost for running this instance is 3.825 USD per hour of usage.

The larger BERT models, however, were not able to be trained on this instance type, and instances with more resources were not evaluated since the experiment did not have access to these larger instances. However, because of the training compiler's efficiency benefits, the larger models could be trained on the ml.p3.2xlarge instance when implemented.

Training the models for ten epochs was sufficient to show the stabilization of the results, as more epochs beyond this number gave a very marginal increase, or decreased, performance.

Fine-tuning the DistillBERT-base model (without using the training compiler) it showed outstanding performance. The performance stabilized after about nine epochs and was then able to almost perfectly classify the different sentiments of the text with an f1-score of 0.998. Moreover, as seen in Figure 4.2 the loss of both the training and the evaluation data decreases during all epochs, indicating that the model does not over-fit to the training data. The training took about an hour to finish, with a total of 3130 billable seconds and 2800 seconds to reach the ninth epoch.

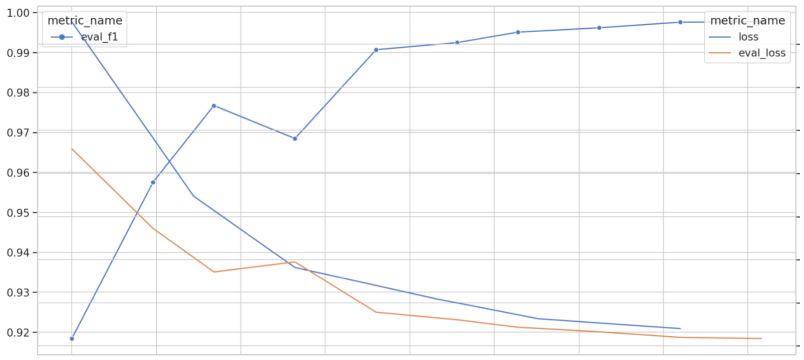

Figure 4.2: F1-score of the evaluation set during 10 epochs of fine-tuning the DistillBERT-base model. Loss of the training set (blue, downward graph) and the loss of the evaluation set (orange, downward graph).

The DistillBERT-multilingual model is not able to achieve the same results as the base model. As shown in Figure 4.3, this seems to be contingent on a case of over-fitting to the training data from early on in the training procedure. After about three epochs, the evaluation loss increases as the model's performance stagnates around 0.94 f1-score. Interestingly, the model does not seem to achieve the same level of performance as the base model compared to their respective first three epochs. The training time is similar to the base model, with a total of 3001 billable seconds.

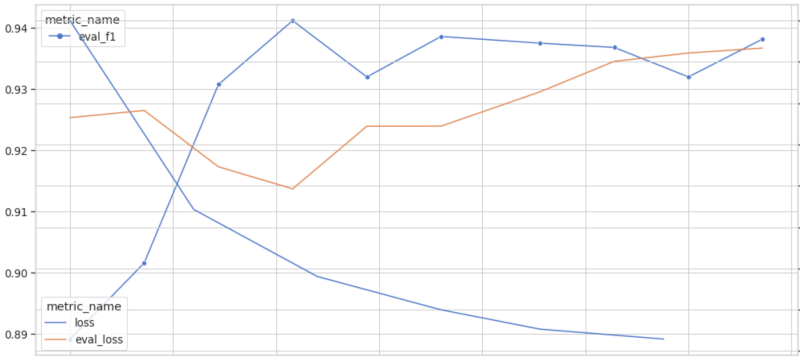

Figure 4.3: F1-score of the evaluation set during ten epochs of fine-tuning the DistillBERT-multilingual model. Loss of the training set (blue) and the loss of the evaluation set (orange).

The larger BERT models (trained using the training compiler) all showed a similar pattern to that of the distilled-multilingual model. Where over-fitting became apparent after just a small number of epochs (see Figure 4.4 below), and the overall performance was not able to reach the same results as the distilled-base model. The training time was, on average, 46 minutes for all the BERT models. This means that even when using larger models, the training time was decreased by using the training compiler.

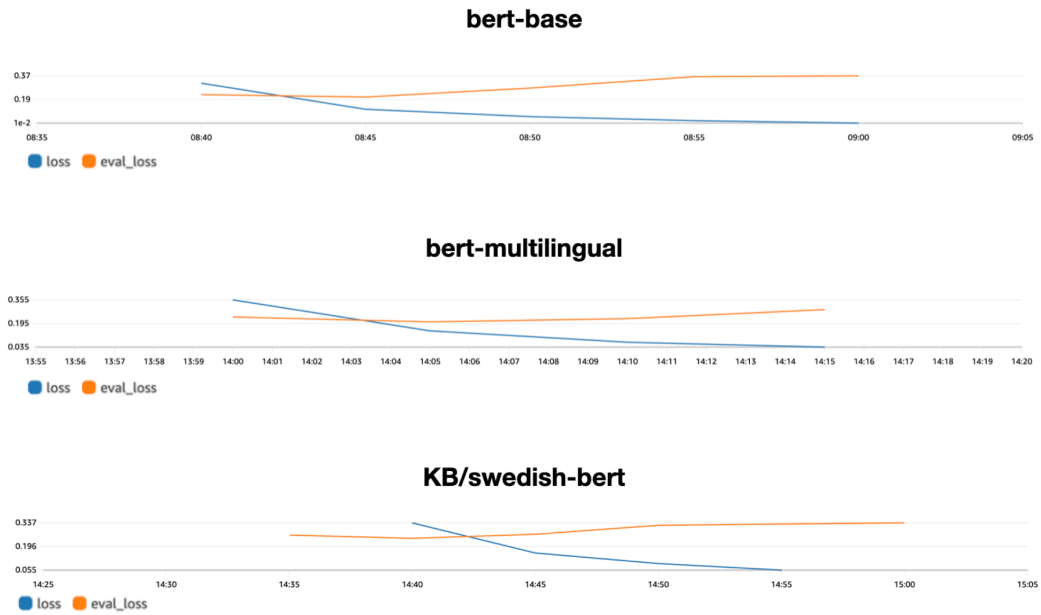

Figure 4.4: The loss function on training data (blue) and the loss for the evaluation data (orange) for the three BERT models.

The distilled models were also trained using the training compiler in order to verify if this configuration is the larger model's issue. However, this does not seem to be the case, as the same result of over-fitting was still absent in the case of the distilled base model and, just like before, it does appear when using the multilingual version.

The training time was improved and decreased about 1.4 times from before. However, the model's overall performance deteriorates by a couple of points when implementing the training compiler. Leading to the conclusion that the best performing model turns out to be the DistillBERT-base model without the training compiler implemented, as shown in Table 4.8 on the following page.

Pre-trained models	
Models	**F1-score**
BERT-base + training compiler	0.946
BERT-multilingual + training compiler	0.949
KB/Swedish-BERT + training compiler	0.929
DistilBERT-base + training compiler	0.935
DistilBERT-multilingual + training compiler	0.931
DistilBERT-base	**0.998**
DistilBERT-multilingual	0.942

Table 4.8: A summary of the perfromance obtained by using different language models. The best result is obtained by using the DistilBERT-base model.

4.2.4 Deployment

The best performing model from the experiments above is the DistilBERT-base model. This model was therefore deployed according to SageMakers' four different inference options to evaluate cost and utility during production.

Real-Time endpoint

Several instance types were tested to avoid time-out errors from exceeding the latency limit and avoid using insufficient memory. Going from the most minor instance available, the smallest working instance available for experimentation was the ml.t2.xlarge instance, with a cost of 0.223 USD per hour. The deployed endpoint has to get the test samples consecutively one after the other, returning each of the predictions immediately. Figure 4.5 shows the CPU utilization when predicting the entire test set in this manner for a total of about 22 minutes. However, the cost for using the real-time endpoint is independent of its usage, and running it (excluding model training and storing) with this instance for a month thus becomes:

$$0.223 \cdot 24 \cdot 30 = 160.56 \text{ USD}$$

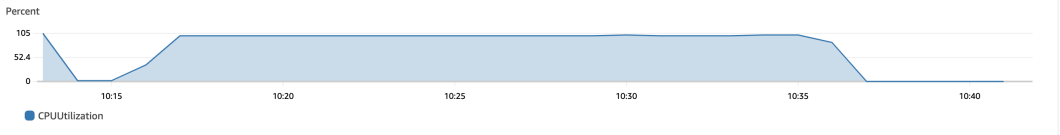

Figure 4.5: CPU-utilization of the real-time endpoint when predicting the test set consecutively. Here the endpoint still costs even when no CPU resources are utilized.

Serverless endpoint

Like the real-time endpoint, the most minor instance was tested first, gradually increasing the resources until the inference could be made. In the serverless case, it is not the instance type but instead the memory allocation that was tested. For setting up the model and running the test set, a memory of 4096 MB was sufficient. Running it with this memory configuration costs 0.00008 USD per second that the endpoint is used. Figure 4.6 shows how the endpoint is initialized (cold start), gets to full capacity of the CPU, and is then shut down again. The cold start before was on par with that of the real-time endpoint. While the entire run time was faster, around 15 minutes. In the serverless case, the cost is only increased by the time the endpoint is active, making the cost for running the endpoint, exclusive of model training and storage, as follows:

$$0.00008 \cdot 60 \cdot 15 \cdot 8 = 0.576 \text{ USD}$$

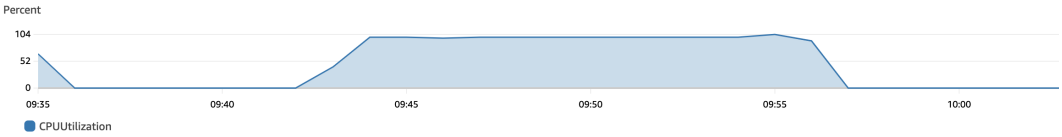

Figure 4.6: CPU-utilization of the serverless endpoint when predicting the test set consecutively. Different from the real-time endpoint there is no cost when the utilization goes down to zero.

Asynchronous endpoint

As with the real-time inference, the smallest instance able to deploy the model was the ml.t2.xlarge instance. The scaling policy was set to zero when the endpoint was inactive. When being called, the endpoint is scaled up to use one instance for inference. In Figure 4.7, the orange line indicates an average of the text samples being processed, and the blue line shows CPU utilization. At time 16.47 in the graph, the entire test set has been processed, and the down-scaling initiates. The total active time for running the endpoint came down to around 9 minutes, and the total cost for a month of deployment would be:

$$(0.223 \div 60) \cdot 9 \cdot 8 = 0.268 \text{ USD}$$

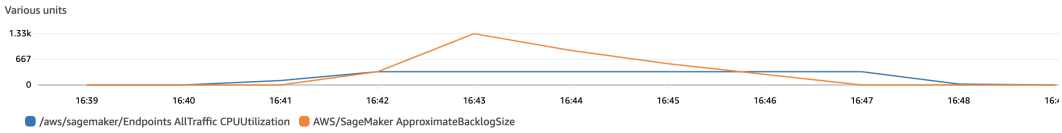

Figure 4.7: CPU-utilization of the asynchronous endpoint (blue) and the size of the input to the endpoint (orange). As seen in the figure the CPU allocation starts to decline as the input goes down to zero.

Batch transform

The batch transform could also use the ml.t2.xlarge instance to host the model. The total time for performing the job on the entire test set was around 14 minutes from start to finish. The total cost for the entire month would be as follows:

$$(0.223 \div 60) \cdot 14 \cdot 8 = 0.416 \text{ USD}$$

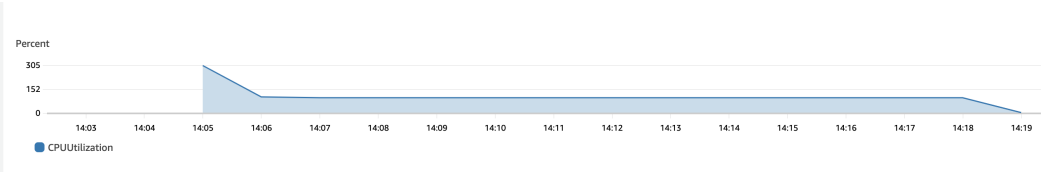

Figure 4.8: CPU-utilization of the batch transform when predicting the test set consecutively.

4.3 Comparison and recommendation

The total cost for training and deploying the Comprehend models and the best performing SageMaker model (DistilBERT-base) is shown in Figure 4.9 below. The initial cost at unit zero is the cost for training and storage, and the rest of the graph shows the cost development for processing an increasing number of units. It is clear that although the training cost for batch analysis with Comprehend and the SageMaker model is quite similar, a larger gap arises as more units are processed.

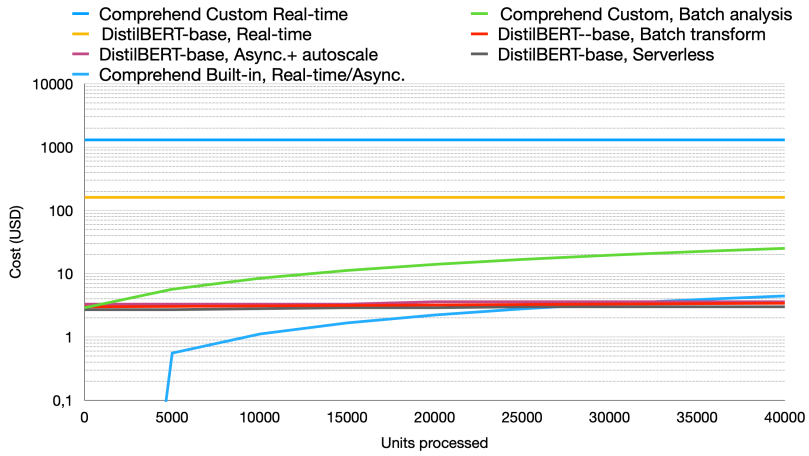

Figure 4.9: Comparing cost development as an effect of the number of units processed. The real-time endpoints are only affected by deployment time and are therefore portrayed here as straight lines. While the other alternatives increase gradually, eventually reaching a break-even point after about 30.000 units where SageMaker alternatives (except the real-time option) become the most cost-effective. The cost for the DistilBERT-base alternatives are so similar across the number of units processed they appear to overlap in the graph.

The bubble graph below shows a comparison of Comprehend and the best-performing model. The green area indicates that the model is more cost-efficient and better performing than the best performing Comprehend model. As can be seen, this is the case for deploying the DistilBERT-base model in all cases except for the real-time deployment.

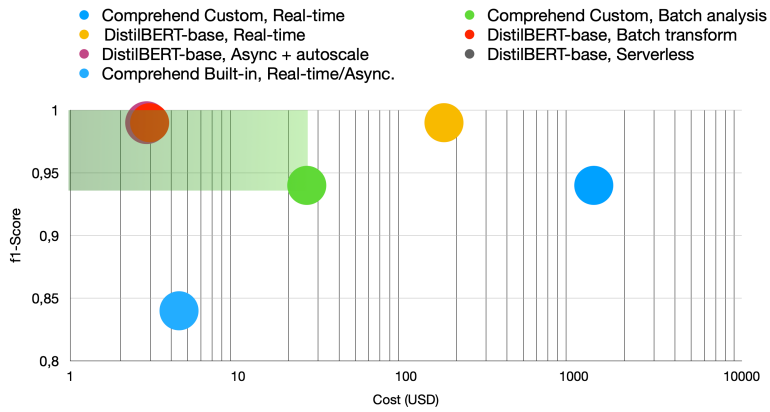

Figure 4.10: Comparison of cost and performance between Comprehend and the best performing SageMaker model. As seen in the graph, the model deployed on SageMaker obtains better performance then Comprehend for all its options and a lower cost for three out of four deployment alternatives. Again the cost and f1-score of the models are so similar they appear to be overlapping.

As shown by the graph, when using the Comprehend model, the batch analysis is the most advantageous since it achieves good performance while maintaining a relatively low cost.

When evaluating the SageMaker model, the three deployment options, except real-time, were more or less equal in cost. These deployments work equally well for this use case, but for scale, it might be more beneficial to use a deployment that allows for more extensive data loads. Because of this, the serverless option might not be the best, considering the limited input size and the fact that predictions must be input one at a time. Leaving the asynchronous or batch transform inference. Although batch transform can handle even larger amounts of data, for this application, where the ability to integrate with other systems might be crucial, being able to set up the model as an endpoint can be beneficial. Ultimately, this comes down to recommending the asynchronous inference option combined with a scaling policy that goes down to zero when the endpoint is inactive.

Summarizing the results, the tables below shows the best configurations of both the Comprehend model and the SageMaker model for the best performance at the lowest cost.

Best performing Comprehend configuration	
Version	**Custom**
Language code	**English**
Inference option	**Batch analysis**
Performance (f1)	**0.94**
Estimated monthly cost (USD)	**25.44**

Table 4.9: Summation of the best configurations for Amazon Comprehend

Recommended SageMaker configuration	
Models	**DisitllBERT-base**
Instance type for training	**ml.p3.2xlarge**
Training compiler	**No**
Inference option	**Asynchronous + auto-scaling**
Instance type for inference	**ml.t2.xlarge**
Performance (f1)	**0.998**
Estimated monthly cost (USD)	**3.41**

Table 4.10: Summation of the best configurations for Amazon SageMaker

By utilizing Amazon SageMaker with the recommended configurations a user gains numerous benefits. Firstly, they can experience a substantial improvement in classification performance, achieving results that approach near-perfect accuracy. This enhanced performance enables more reliable decision-making based on highly accurate insights. Additionally, the need for scheduling analysis jobs is eliminated as the endpoint remains available at all times. This continuous availability ensures seamless access to classification services. Moreover, utilizing Amazon SageMaker with the recommended configurations offers significant cost savings, being approximately ten times cheaper than the best Comprehend configuration.

5 Discussion

Deploying a machine learning model to production is always an iterative process. Therefore, the findings proposed in this project should be considered as a study of a possible starting point rather than a definite proposal for deployment. Since in a real-life circumstance, there would be many factors that were considered outside the scope of this thesis. For example, when evaluating the different service options, the cost of having a developer make the solution would have to be considered, along with monitoring, evaluating, and possibly re-configuring the solution when in production.

Also, the findings depend very much on the specific use case on which the alternatives were evaluated. One noticeable change that would have much impact is if the desired requirements specification were to change (sometimes called a *concept drift*). For example, if we want the model to handle a different language besides Swedish, the model would have to be retrained. Alternatively, if requirements on latency, data handling, or compute resources changed, meaning the deployment method would likely have to be altered. There can also be so-called *data drifts* where the underlying distribution of data changes while the model is in production. This evaluation used a fairly balanced data set to train the model. However, if that dispensation shifted, we would likely have to retrain the model to obtain peak performance.

To handle these issues, continuous maintenance would have to be applied to the process, where logging incoming data and the model output could be one efficient way of alerting to possible data drifts and the need for conceptual changes. In conjunction with this, understanding the implications and trade-offs between different model and deployment options would be a key factor, where this thesis could serve as a perfect commencement.

5.1 Evaluating findings

In order to deepen the understanding of the findings, different aspects could be changed one at a time to see how they may affect the outcome. Moreover, when doing so, it will also be possible to try out a different hypothesis about what may cause a certain output. For example, one hypothesis regarding the underperformance of the large (BERT) language models was that it might be caused by over-fitting to training data. An effective way to reduce such under-fitting is usually to increase the amount of data the model is trained on, which will be tested and discussed in the following section.

5.1.1 Increasing training data

The amount of data used in the original experiment simulated a scenario where the supply of training data was limited, thus only including 10.000 examples for fine-tuning the language models. In this experiment, the same test set was kept as before to make a fair comparison between the evaluations. However, the size of the training was increased multi-fold to contain 80.000 training examples now. The examples are taken from the original data set used to obtain the training data previously, which contains a total of 103.000 reviews written in Swedish.

The setup for training the models is the same as in the original experiment with the same configurations of instances. This is despite the increasing number amount of training data increasing the run-time of the fine-tuning in all three models. After running the newly trained models on the test data, it showed no indications of an increase in performance. By looking at the loss functions, it is apparent that the problem of overfitting is still occurring. Thus, the problem likely lies in the problem's simplicity contra the model's complexity. An even higher amount of training data might show some indication of lessening the overfitting, but more likely, the model would have to reduce its complexity. For example, by performing pruning, regularization, or, as was proven effective in the original experiment, distillation. The final result of the impact of increasing the amount of training data can be seen in Figure 5.1 below:

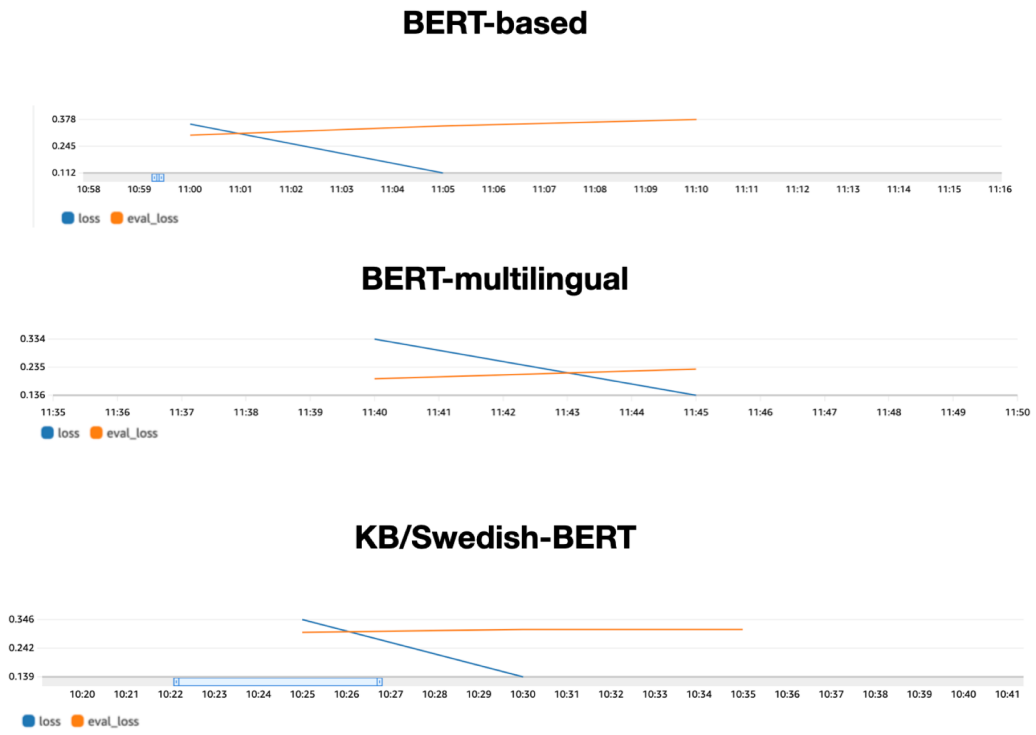

Figure 5.1: The loss functions from the large language models when using an increased amount of training data. As shown in the figure the problem of overfitting is still apparent and the increase does not correlate with an increase in performance.

5.1.2 Evaluating performance on text with mixed languages

The training and test set used in this study was made exclusively from reviews written in Swedish. However, a more realistic scenario is that the texts collected include a mixture of both Swedish and English. Since the Comprehend service did not have an option for the Swedish language when performing sentiment analysis, it would be interesting to see how such a change might affect the outcome. Therefore a new composite of the training and test data can be used to investigate the results of the Comprehend alternative.

In this experiment, 20 percent of the reviews from the original sets were translated into English by using Google's translating tool. Keeping the other 80 percent as is and shuffling the translated text into it creates a single data set. This will simulate the scenario of collecting reviews from a Swedish site where it is likely to find the majority of the texts written in Swedish and some smaller sub-set in English.

When using the different mixtures in the data set, it is unlikely that the price will be affected to any significant degree. Since it is based on the length of the texts and this will only differ slightly from the main evaluation in a minority of the samples. What might make a difference is the performance, which will be the metric of interest for this experiment.

In the case of the built-in version of Comprehend, English was again chosen as the language code. With the same preceding for adjusting for the fact that the result is not binary as in the main experiment. The resulting performance on the new test data set became as follows:

Comprehend's Built-in version, mixed languages	
F1-score	0.87
Language code	English

Table 5.1: Result of using Comprehend's built-in version using the English language code for sentiment analysis. The sample text included 400 samples of English text and 1600 samples written in Swedish. Resulting in an increase of 0.03 f1-score points from samples consisting of only Swedish text.

The finding in Table 5.1 shows that the results were improved slightly from the previous experiments of exclusively Swedish reviews. One possible explanation is that it was now possible for the model to rightly classify some of the reviews correctly that were previously seen as neutral. Furthermore, the increase in performance seems to be proportional to the increase in the number of English reviews where the model (using this language code) can be expected to do a better job.

For the custom version of Comprehend, the same ratio between the two languages was used in both the training and the test data set. Here the English language code is used as well, and the results are shown in Table 5.2.

Comprehend's Custom version, mixed languages	
F1-score	0.93
Language code	English

Table 5.2: Result of using Comprehend's custom version using the English language code for sentiment analysis on texts written in both Swedish and English. The data set used for both training and testing consisted of a majority (80 percent) of Swedish. This change did not improve the result from previous experiments, which decreased by 0.01 f1-score points.

Interestingly, the same performance increase as in the built-in version can not be found here. Actually, there is instead a slight decrease in performance (0.93 compared to the previous 0.94). The change that the mixture of language introduces is so small that it should not be considered an indication of any changes in the actual performance of the model. As it is natural for the model to show some fluctuation in results between different training runs. This is combined with the fact that the translation might not be able to fully and accurately convey the intended meaning of the original text. That can be caused by deficiencies in the translation tool making wrongful judgments of translation, or formulating the text in ways that are not commonly used in English texts.

5.2 Subject to change

The results of this study, and how they hold up in a real-life setting are volatile on many fronts. First is the change and upgrading of technology discussed in the previous section. New language models are available daily and updates to the very models used in this thesis. Combined with new methods of performing word embedding and innovative algorithms that are able to change the playing field.

But beyond this, the findings are also subject to changes in one specific company's (Amazon) policies. If the pricing of AWS services were to change, the resulting evaluation might fall out differently. It is not unthinkable that AWS will improve the methods for their ready-made solutions (including Amazon Comprehend) in a way that will make this alternative much cheaper for the end consumer. It is also possible that AWS will keep adding languages to their sentiment analysis tool by training their language models on for example Swedish text. This is a likely outcome since the tool has been updated to include more languages each year that it has been available. If this is implemented, such changes to price and performance might be persuasive enough to tip the scales on what recommendation proves the most beneficial for the end user.

6 Future work

The work presented in this thesis explores a specific area of a vast field. In order to improve the findings, future work could include exploring more options, such as the selection of data, models, and deployment options. Also, future work could include making the evaluation more rigorous by looking at more dimensions of comparison than the cost and the performance. For example, this might include ease of using the service, explainability, or compatibility with other systems or micro-services.

Another important aspect is that language models, and cloud deployment are both rapidly evolving technologies. Using a model deployment like the one presented in this thesis will therefore have to be under constant revision to keep up with the latest progress. Therefore, an essential part of further building on this study's findings would be to examine the dynamics of the systems. For example, by implementing solutions or processes so it can effectively keep up with changes. This might include evaluating other model options or developing a robust testing system.

Bibliography

[1] Alan M Turing. "Computing machinery and intelligence". In: *Parsing the turing test*. Springer, 2009, pp. 23–65.

[2] John McCarthy. *What is artificial intelligence*. 2004. URL: http://jmc.stanford.edu/artificial-intelligence/what-is-ai/index.html (visited on 08/02/2023).

[3] Arthur L Samuel. "Some studies in machine learning using the game of checkers". In: *IBM Journal of research and development* 3.3 (1959), pp. 210–229.

[4] Tom Mitchell. *Machine learning*. Vol. 1. 9. McGraw-hill New York, 1997.

[5] Taiwo Oladipupo Ayodele. "Types of machine learning algorithms". In: *New advances in machine learning* 3 (2010), pp. 19–48.

[6] Ajith Abraham. "Artificial neural networks". In: *Handbook of measuring system design* (2005).

[7] Gabriel Furnieles. *Sigmoid and SoftMax Functions in 5 minutes*. 2022. URL: https://towardsdatascience.com/sigmoid-and-softmax-functions-in-5-minutes-f516c80ea1f9 (visited on 08/02/2023).

[8] Sagar Sharma. *Activation Functions in Neural Networks*. 2017. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6 (visited on 08/02/2023).

[9] Kunihiko Fukushima. "Cognitron: A self-organizing multilayered neural network". In: *Biological cybernetics* 20.3 (1975), pp. 121–136.

[10] Sagar Ram. *When Do Language Models Need Billion Words In Their Datasets*. 2020. URL: https://analyticsindiamag.com/language-models-billion-billion-words-datasets/ (visited on 08/02/2023).

[11] MathWorks Editor. *3 Things You Need to Know About Deep Learning*. 2022. URL: https://medium.com/mathworks/3-things-you-need-to-know-about-deep-learning-342b71ba118 (visited on 08/02/2023).

[12] Amazon Web Services. *Amazon Machine Learning Developer Guide*. 2022. URL: https://docs.aws.amazon.com/pdfs/machine-learning/latest/dg/machinelearning-dg.pdf (visited on 08/02/2023).

[13] IBM Cloud Education. *Natural Language Processing (NLP)*. 2020. URL: https://www.ibm.com/cloud/learn/natural-language-processing (visited on 08/02/2023).

[14] Weights and biases. *Sentiment Analysis*. 2020. URL: https://wandb.ai/site/tutorial/sentiment-analysis (visited on 08/02/2023).

[15] Oracle cloud. *What is cloud computing?* 2022. URL: https://www.oracle.com/se/cloud/what-is-cloud-computing/ (visited on 08/02/2023).

[16] Kumar Rahul. *AWS Market Share 2022: How Far It Rules the Cloud Industry?* 2022. URL: https://www.wpoven.com/blog/aws-market-share/ (visited on 08/02/2023).

[17] Amazon web services. *Cloud computing with AWS?* 2022. URL: https://aws.amazon.com/what-is-aws/ (visited on 08/02/2023).

[18] Amazon Web Services. *Machine Learning on AWS*. 2023. URL: https://aws.amazon.com/machine-learning/ (visited on 08/02/2023).

[19] Amazon Web Services. *Machine Learning on AWS: Amazon SageMaker*. AWS. 2023. URL: https://aws.amazon.com/sagemaker/ (visited on 08/02/2023).

[20] Amazon Web Services. *Innovate with machine learning*. 2022. URL: https://aws.amazon.com/ai/ (visited on 08/02/2023).

[21] Amazon Web Services. *Storage: Amazon S3*. AWS. 2023. URL: https://aws.amazon.com/s3/?nc2=type_a (visited on 08/02/2023).

[22] Seagate Technology LLC. *How Amazon S3 Buckets Work*. 2022. URL: URL: https://www.seagate.com/gb/en/blog/how-amazon-s3-buckets-work/ (visited on 08/02/2023).

[23] Svensk e identitet. *Vad är API?* 2022. URL: https://e-identitet.se/news/vad-ar-api/ (visited on 08/02/2023).

[24] Amazon Web Services. *Machine Learining on AWS: Amazon Comprehend*. AWS. 2023. URL: https://aws.amazon.com/comprehend/ (visited on 08/02/2023).

[25] Amazon Web Services. *What is Amazon Comprehend?* 2022. URL: https://docs.aws.amazon.com/comprehend/latest/dg/what-is.html (visited on 08/02/2023).

[26] Amazon Web Services re:Post. *Model for Amazon Comprehend sentiment analysis*. 2022. URL: https://repost.aws/questions/QU6hqOWAY1SsmPhsbCXRvfFA/model-for-amazon-comprehend-sentiment-analysis (visited on 08/02/2023).

[27] Todd Escalona. *Detect sentiment from customer reviews using Amazon Comprehend*. 2018. URL: https://aws.amazon.com/blogs/machine-learning/detect-sentiment-from-customer-reviews-using-amazon-comprehend/ (visited on 08/02/2023).

[28] Amazon Web Services. *Tools to build on AWS*. 2022. URL: https://aws.amazon.com/developer/tools/#sdk (visited on 08/02/2023).

[29] Amazon Web Services. *Boto3 documentation*. AWS. 2023. URL: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html (visited on 08/02/2023).

[30] Amazon Web Services. *Amazon Comprehend Pricing*. 2022. URL: https://aws.amazon.com/comprehend/pricing/ (visited on 08/02/2023).

[31] Amazon Web Services. *Analyze insights in text with Amazon Comprehend: How-To Guide*. 2022. URL: https://aws.amazon.com/getting-started/hands-on/analyze-sentiment-comprehend/ (visited on 08/02/2023).

[32] Amazon Web Services. *Developer Guide: Sentiment*. 2022. URL: https://docs.aws.amazon.com/comprehend/latest/dg/how-sentiment.html (visited on 08/02/2023).

[33] Herve Nivon. *Building a custom classifier using Amazon Comprehend*. 2019. URL: https://aws.amazon.com/blogs/machine-learning/building-a-custom-classifier-using-amazon-comprehend/ (visited on 08/02/2023).

[34] Amazon Web Services. *Developer Guide: Guidelines and quotas*. 2022. URL: https://docs.aws.amazon.com/comprehend/latest/dg/guidelines-and-limits.html (visited on 08/02/2023).

[35] George Pipis. *How To Build A Custom Text Classification Model With AWS Comprehend*. 2022. URL: https://predictivehacks.com/how-to-build-a-custom-text-classification-model-with-aws-comprehend/ (visited on 08/02/2023).

[36] Amazon Web Services. *Amazon Ec2*. AWS. 2023. URL: https://aws.amazon.com/ec2/ (visited on 08/02/2023).

[37] Amazon Web Services. *Amazon EC2 Instance Types*. 2022. URL: https://aws.amazon.com/ec2/instance-types/ (visited on 08/02/2023).

[38] Amazon Web Services. *What Is Amazon SageMaker?* 2022. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html (visited on 08/02/2023).

[39] Amazon Web Services. *AWS Lambda*. AWS. 2023. URL: https://aws.amazon.com/lambda/ (visited on 08/02/2023).

[40] Amazon Web Services. *Amazon API Gateway*. AWS. 2023. URL: https://aws.amazon.com/api-gateway/ (visited on 08/02/2023).

[41] Amazon Web Services. *Amazon CloudWatch pricing*. 2022. URL: https://aws.amazon.com/cloudwatch/pricing/ (visited on 08/02/2023).

[42] Amazon Web Services. *AWS Summit DC 2022 - Amazon SageMaker Inference explained: Which style is right for you?* AWS. 2022. URL: https://www.youtube.com/watch?v=bRUNpuRGeZc\&ab_channel=AWSEvents.

[43] Amazon Web Services. *Amazon SageMaker pricing*. 2022. URL: https://aws.amazon.com/sagemaker/pricing/ (visited on 08/02/2023).

[44] Amazon Web Services. *Asynchronous inference*. 2022. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/async-inference.html (visited on 08/02/2023).

[45] Amazon Web Services. *Serverless Inference*. 2022. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html (visited on 08/02/2023).

[46] Tracey Sean. *Introducing SageMaker Training Compiler*. 2021. URL: https://aws.amazon.com/blogs/aws/new-introducing-sagemaker-training-compiler/ (visited on 08/02/2023).

[47] Jason Brownlee. "A Gentle Introduction to the Bag-of-Words Model". In: (2017). URL: https://machinelearningmastery.com/gentle-introduction-bag-words-model/.

[48] Karen Sparck Jones. "A statistical interpretation of term specificity and its application in retrieval". In: *Journal of documentation* (1972).

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

[50] Charles McCormick. *Word2Vec Tutorial - The Skip-Gram Model*. 2016. URL: http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/ (visited on 08/02/2023).

[51] Xin Rong. "word2vec parameter learning explained". In: *arXiv preprint arXiv:1411.2738* (2014).

[52] Amazon Web Services. *Amazon SageMaker's Built-in Algorithm Webinar Series: Blazing Text*. AWS. 2018. URL: https://www.youtube.com/watch?v=G2tX0YpNHfc&ab_channel=AmazonWebServices (visited on 08/02/2023).

[53] et. al Mikolov Tomas. "Bag of tricks for efficient text classification". In: *arXiv preprint arXiv:1607.01759* (2016).

[54] Kavita Ganesan. *Word2Vec: A Comparison Between CBOW, SkipGram SkipGramSI*. 2022. URL: https://kavita-ganesan.com/comparison-between-cbow-skipgram-subword/#.Y3zaY-zMK3I (visited on 08/02/2023).

[55] Amazon Web Services. "BlazingText algorithm". In: (2022). URL: https://docs.aws.amazon.com/sagemaker/latest/dg/blazingtext.html (visited on 08/02/2023).

[56] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. 3rd ed. Pearson, 2010.

[57] IBM. *What is recurrent neural networks?* IBM. 2023. URL: https://www.ibm.com/topics/recurrent-neural-networks (visited on 08/02/2023).

[58] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[59] Ashish et. al Vaswani. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[60] Yuening Jia. "The Transformer model architecture". In: (2019). URL: https://commons.wikimedia.org/wiki/File:The-Transformer-model-architecture.png.

[61] Jacob Devlin et. al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[62] Gou Jianping et. al. "Knowledge distillation: A survey". In: *International Journal of Computer Vision* 129.6 (2021), pp. 1789–1819.

[63] Xiaoqi Jiao et. al. "Tinybert: Distilling bert for natural language understanding". In: *arXiv preprint arXiv:1909.10351* (2019).

[64] Victor Sahn et al. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *arXiv preprint arXiv:1910.01108* (2019).

[65] Tim Isbister. *Swedish sentiment*. 2020. URL: https://github.com/timpal0l/swedish-sentiment (visited on 08/02/2023).

[66] Amazon Web Services. *BlazingText Hyperparameters*. 2022. URL: URL:https://docs.aws.amazon.com/sagemaker/latest/dg/blazingtext_hyperparameters.html (visited on 08/02/2023).

[67] *Hugging Face, howpublished = https://huggingface.co/, note = Accessed: 2023-02-08.*

Appendix A

Appendix

Built-in model	
Language	**F1-score**
English	**0.93**
German	0.93
French	0.90
Spanish	0.91
Italian	0.90
Portuguese	0.90
Arabic	0.87
Hindi	0.86
Japanese	0.87
Korean	0.87
Chinese	0.89

Table A.1: Full table of languages available for sentiment analysis in the built-in tool of Amazon Comprehend.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
June 2023

Document Number
TFRT-6213

Author(s)
Jonas Lilja

Supervisor
William Cloarec, Sigma Technology Cloud, Sweden
Johan Eker, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle

Evaluating machine learning models for text classification

A comparative study of Amazon Comprehend & Amazon SageMaker

Abstract

This thesis will explore the use of AWS machine learning services that enable natural language processing (NLP). More specifically, this work will focus on sentiment analysis of product and service reviews written in Swedish.
To find the most efficient solution to this task, the ready-made sentiment analysis tool available on AWS (Amazon Comprehend) was compared to custom-made solutions built on the AWS platform Amazon SageMaker. Several model options were considered for the custom-made alternative, ranging from simple regression to state-of-the-art language models, along with different ways of deploying the best performing model to the cloud so that its insights could be accessed in the most efficient way.
For the evaluation, a use-case was put forward that is deemed realistic for how Sigma Technology Cloud would use such a service. Based on this, the thesis findings suggest that a scaled-down version of the BERT model called DistilBERT is the best alternative of the models evaluated. Furthermore, this model should be set up as an asynchronous endpoint with a policy allowing it to auto-scale down to zero when it is not invoked. Doing this on the Amazon SageMaker platform results in a solution that is both cheaper and that shows better performance than other alternatives.
Finally, this comparison between the services based on their performance, ease of use, and cost efficiency was put forward as a recommendation as to what model and configuration should be used by Sigma Technology Cloud.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-67

Recipient's notes

Security classification

http://www.control.lth.se/publications/