UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e Matematiche
Corso di Laurea Triennale in Informatica

NLP application in Customer Service: how to improve Customer Success Rate and Satisfaction using AI

Relatore:
Prof. Riccardo Martoglia

Candidato:
Federico Scaltriti

Anno Accademico 2020/2021

Ringraziamenti

Desidero ringraziare il prof. Riccardo Martoglia per la disponibilità, l'assistenza e l'essenziale puntualità nell'aiutarmi in questo progetto.
Ringrazio mia mamma, per avermi sostenuto in questi anni e avermi aiutato a fare scelte sagge, che mi hanno portato fino a questo punto.
Grazie anche ai miei amici e ai compagni di studi che, anche nelle difficoltà, mi hanno aiutato ad affrontare ogni materia ed esame sempre con lo spirito di imparare cose nuove e saperle applicare nella vita reale.

Parole chiave

NLP
Text Classification
Customer Service
Deep Learning
Text Anonymization

Contents

List of Figures

Introduction

When Alan Turing firstly thought about his famous Turing test [1], his purpose was to emulate a human-specific skill with a machine's ability. The spectrum of the capabilities was wide, but he decided to challenge a linguistic task.

He designed what is now the famous *Imitation Game*: a machine capable of cheating the human it's facing, thinking its responses come from a human too.

However, it was a primitive way to process the natural language and it had its weaknesses. In the following years until nowadays, technologies that concern the natural language treatment made big steps in the progress direction and the huge amount of data increasingly pushed researchers to develop new strategies to process them in a more efficient and less time-consuming way.

This thesis is about applying some of these techniques to provide a solution to a recurring problem in the customer service system. With the great availability of the Internet today, the amount of interconnected people is continuously increasing and so is the time spent on e-commerce and other websites. The more people interact with these kinds of services, the more pressing is to supply appropriate answers to their requests.

Automation through AI represents a more complex solution in the short term but, at the same time, a great saving of energy and time in the long run.

Part I - State of the art

In the first chapter I'm going to discuss more extensively about what is the scope of this project. It's presented a description of how customer service support works today and why specifically AI represents the answer, listing the advantages of such a solution.

Chapter 2 explains what is intended when talking about Natural Language Processing (often shortened with the acronym NLP). Having to deal with natural

language coming from different sources and people it seemed fair to reserve a special chapter about it.

Chapter 3 is about distinguishing the difference between Artificial Intelligence, Machine Learning, and Deep Learning. It presents some of the core concepts that will be talked about in the rest of this document to have a better understanding of what is the work behind the solutions I proposed. It will mention some of the algorithms that leveled up the state-of-the-art in this branch of study too.

The subsequent chapter is about the nature of the data I had to deal with. It shows the main performed transformations that it was subjected to, based on the task to accomplish.

To close up the first part of the thesis, chapter 5 lists all the exploited technologies: from the programming language to the containerization tool. An interesting view is given to the whole ecosystem in which the developed service has been placed, to understand the interaction between each main component.

Part II - Text Classification Techniques

This part is focused on the realization of the delivered issue. It follows a logical path beginning with the first of the two realized algorithms. The chapter illustrates the development phases and a pair of tools that helped me through the experience and were suddenly used with the second algorithm.

The second chapter is the most important since it explains the main algorithm. The steps that led me to the solution were many and the possible solutions themself were diversified over different approaches. Each step and enhancement is described here.

A special focus on the future developments is reserved in the second-last chapter: since the explained work it's just a beginning to an improvable solution there are some possible enhancement routes to undertake.

The last chapter closes the discussion proving thoughts about if the solution reached or not the prefixed objective and summarizes what has been the winning strategy.

Part I

State of the art

Chapter 1

Problem introduction

The delivery in which I was involved has been commissioned by a business user in the fashion industry. Thus, the project developed with this thesis responds to a real need present in various companies.

The organization of a **help desk ticketing service** for support and assistance within a company is essential in order to provide an efficient and quality service to users or to its customers.

As the business begins to grow, it's clear that just one level of support is not enough to guarantee optimal service. This is why lots of companies are moving towards multi-tier support systems. The reason for using such a system, instead of a single generic support group, is to provide the best possible service in the most efficient way.

Thanks to escalation, maximum efficiency of the management process are obtained. Groups of solvers are organized, and, if possible, the involvement of the most pertinent solvers to user requests is automated through routing algorithms. Escalation makes it possible to better comply with any **SLAs** (Service Level Agreement) defined on the service or possibly specific to the customer.

The way escalation is managed depends on the ticketing software. Still, the more the applications allow to automate this functionality, the more flexible and efficient the systems will be, helping companies grow by optimizing resources.

The ticketing software is considered a bug tracking platform. One tangible example is Jira. This allows the business user and a client to communicate and, at the same time, maintain agile project management.

When a business user opens a ticket on the platform, people try to assign

a project or a team to the presented issue. Most of the time, managers of CED areas are not sure who can deal with the presented ticket. Examples of problems could be the user database, the raw material supplier, or a management software's functionality.

Practically, each business user has its own **AM Team** (Assistance and Maintenance of the applications Team). Normally, there is a person in charge of coordinating the distribution of customers' requests to people.

Each labeled class corresponds to a working area. One of the available teams will be in charge of solving the related open issues. Based on the workload, the team could be formed to take care of an application or functionality.

There is not a single correct number of how many levels the ticketing software should manage. It depends on the management objectives and on how the functionalities can meet the needs of different companies in terms of configuration and flexibility.

The AM team of the business user the solution was suit for is divided into two different levels:

- 1st level

  people who solve problems that do not require access to the code base. Usually, it's about the frontend interface or a how-to explanation to the customer. In general, people deal with simple requests such as problems in using the software. If the anomaly is known and a workaround or solution exists, the solver can instruct the user on how to resolve the problem.

- 2nd level

  people are in charge of fixing problems in the code base. The technicians responsible for this level initially determine whether the problem is new or whether the report refers to an already known and not yet resolved problem.

  If the bug takes more than five working days to be resolved, it's called a *change request* and requires a more detailed development. If it's needed, it could be passed to a specialised team.

The objective of the project is to match the right class of problems for an opened ticket. The base from which to start the solution has been a collection of closed tickets, each one labeled with what was supposed to be the correct class.

In this context, AI represents the answer to the addressed problem.

The developed deep learning algorithm has a set of classes from which to choose and, after a period of training, will be able to classify each new ticket to the corrected class. An interesting option is the chance to change the predicted classification **by the user**. The algorithm will provide what it thinks is the most suitable class, but the user has to decide. They can revise the prediction and retrain the model to improve the learning process.

Thanks to the developed models, the AI addresses this time-consuming process, and that person could work on something more productive than just label the incoming issues. This process is present at both levels, and the developed AI models can get rid of both these assignments.

Some business users rely on Google or other external providers to benefit from some services. Because of this, the user is constrained to be stuck to what the provider has to offer. What if the business user is not satisfied with what its provider has to offer?

With an on-premise solution instead, the business user

- is not forced to use the software that its provider offers (often at a higher price) due to a signed contract
- could customize it in every aspect, even the algorithm.

1.1 Improvements

This kind of solution leads to two main improvements:

1. **Speed of execution**

   Once the ticket has arrived to the business user platform, it is handled **instantly** without waiting for a physical person to be available to do that job.

2. **FTE reduction**

   The Full-Time Equivalent is a method to evaluate the workload of employed people in a way that makes them comparable across different situations. It's a numeric value between 0 and 1. An FTE of 1.0 represents a

full-time worker, while a value of just 0.5 signifies half of the full work. It could be used to improve the cost reduction in a production process too.

The time dedicated to this specific activity sometimes could be crucial in order to respect deadlines and project deliveries. Hence, it is fundamental to optimize the time where it's possible.

1.2 Text anonymization

Before proceeding with the text classification procedure, it was possible to **anonymize the input data** using a specific deep learning model to clean the text from any useless sentence, especially greetings and thanks.

The anonymization had been possible just because of the **project's scope**: to determine to which class the text belongs, it's not necessary to refer to the actual rules of the GDPR. Some real examples of personal data are names, surnames, and phone numbers.

A feasibility study a priori highlighted this important aspect. Respecting this UE privacy regulation using this kind of information means that there should be at least one legal basis to process a person's data, unless they are aware of the treatment. Therefore, writing a GDPR document was unnecessary because this type of data is not present and thus not treated.

Chapter 2

Natural Language Processing

Natural Language Processing (NLP) is the field that studies the human natural language in computer science and linguistics. In computer science it concerns the process of teaching a machine how to interpret large amounts of natural language data. There are plenty of remarkable applications and researches in this discipline.

Nowadays is one of the most captivating fields of study since businesses run over massive quantities of unstructured, text-heavy data and need a way to efficiently process them. The data we are generating in the last years is largely consistent with text and there's no way humans can process it on their own in a non time-consuming activity. And that's why NLP is so important. In addition, Machine Learning and especially Deep Learning algorithms can now efficiently interpret even vague types of elements that were previously bad at.

2.1 Preprocessing phases

Often these data have to be processed a priori before feeding any algorithm with them. It's now presented a list of the most common operations applied to let an algorithm perform better.

- *Lexical Analysis*

  This includes the action of converting characters' sequences in a token stream. There are some symbols that could be complex to treat. It's the case of numbers, capital letters, dashed-words (-) and dots (.) Based on

the context these sequences should have a different meaning (e.g. not every time a dot means it's going to start a new sentence, like in dates dicitures).

- *Stopwords detection and deletion*

  The most repetitive words have not any discriminant power in distinguishing a specific text so it's a wise decision to detect and delete them to save useful space since the dataset occupies a lot of disk memory. This leads even to a little speed up in the text processing performed by the algorithm.

- *Stemming or Lemmatization*

  With the term *stemming* is intended the truncating operation applied to each single word. For instance, the word "branches" would be truncated to "branch". With *lemmatization* one word converges to its base-form instead. A significant example could be the word "saw" that with lemmatization is reconductible to the verb "see". In this case, if a stemming operation is applied it does not have any effect because "saw" cannot be truncated in any way. These enable the algorithm to treat similar words in the same way, optimizing even more the disk space.

- *Word Sense Disambiguation*

  This includes deriving the right meaning of a word in the context it's been used in. For instance, let's briefly analyse the use of the word "arm" in the next sentences:

  "I have a dog bite on my arm."

  "It's important to arm yourself with a solid education."

  The word and the pronunciation is the same but only the context can give us the right meaning of the word. Algorithms that use this approach can understand when the word "arm" is referred to a human body limb or to the verb.

2.2 NLP tasks

In the last decade the advent of many deep learning methods led this field to reach new state-of-the-art performances regarding precision and accuracy in a lot of different tasks. They have a great potential that's currently exploited by a lot of companies.

- *Speech Recognition*

  This term includes all the techniques with the aim, given a audio track of a person speaking, to determine the textual representation of that very speech. Many problems could come from the different pronunciation of the same letters' combination in different languages or to the adopted alphabet. Some keyboard apps like Gboard and SwiftKey Keyboard use this approach.

- *Named Entity Recognition (NER)*

  Given a single or multiple sentences, it includes the techniques involved in mapping the right grammar role to each term as proper noun, verb, pronoun and so on. A common problem is caused by the different treatment reserved to the same words.

- *Sentiment Analysis*

  Given a sentence, this branch of NLP is aimed to interpret and find the polarity of the sentence itself: is the speaking person angry or happy? Are they satisfied or disappointed?

  The use of negative words such as "not" should have a different weight based on the context. Consider the next two phrases:

  "I'm not sure to have passed the exam".

  "I'm sure I have not passed the exam".

  They are similar, they use almost the same words but in the first case the student is not secure about the mark is going to receive. In the latter one instead there is the conviction of having to repeat the exam. And this makes a big difference.

- *Text Classification*

  Given a sentence or a document, the algorithm involved is in charge of assigning the text to a specific class from a pool of possible choices. An issue could come from classes: maybe it's not really clear the semantic separation of two classes and the algorithm has a high probability of wronging the class to which it belongs.

- *Machine Translation*

  It's the process of translating some text from a spoken language to another one without human intervention.

  A lot of mainstream tools such as Google Translate and Reverso Context apply techniques to perform this significant task.

- *Natural Language Generation*

  This involves algorithms that analyse unstructured text and then produce content based on the processed data. A concrete example is the GPT-3 [2] language model, whose ability is to generate believable articles from the text it was fed to.

Chapter 3

Machine & Deep Learning

Artificial Intelligence is one of the fascinating fields of study in computer science nowadays. It refers to systems and machines trying to emulate what a cognitive human brain can achieve.

The two most promising branches are the previously-called Machine Learning (ML) and Deep Learning (DL).

Lots of people don't know the difference between these terms and the general term "Artificial Intelligence" and thus use them interchangeably.

With the term "Artificial Intelligence" we refer in general to the theory and development of software capable of performing tasks not normally deliverable to machines. They can vary from speech recognition to decision-making, and even translation between languages.

Machine Learning is just a subset of AI and refers to the techniques that enable computers to figure things out from the data and deliver AI applications. Furthermore, Deep Learning is a subset of Machine Learning and it requires computers to solve more complicated operations. This is why solving issues with DL techniques requires more data than ML. Such algorithms must have the highest possible number of entries available.

Each of these methods has in common the automatic learning as the approach to solve the required task.

Let's now overview some of the core concepts this thesis will recall.

3.1 Core concepts

3.1.1 Feeding data

Usually, the data to input into the algorithm are in a file-based format (.CSV and .XML are some of the most common ones). This file, more commonly known as *dataset*, is split in multiple sections based on the training phase the programmer is considering. One standard division that I used throughout this experience involves the presence of a *training set*, a *validation set*, and a *test set*.

A training set consists of the entries used to train any chosen algorithm to obtain a specific behavior.

The validation set is the one used to compare the performance of each algorithm to understand which is better.

As the name suggests, the test set validates how the best model performs over never-seen data.

It is a rule of thumb that the train set has the majority of the samples, but the precise percentual can vary from dataset to dataset.

3.1.2 Learning approaches

There are different approaches to a ML/DL problem. Based on the issue, it's more convenient to proceed with one rather than another.

- *Supervised learning*

  In this approach, every entry of the training set has the desired solution, hence a **label** associated with it. A typical task in this sense is **classification**. The algorithm learns being explicitly told which class an entry belongs to.

- *Unsupervised learning*

  In this approach, as the name implies, the training data is **unlabeled**. It's as if a student studies without a teacher and has to find the solutions by himself. A typical task is **clustering**, where the algorithm has to gather different entries based on their features. The entries with similar features will be grouped together.

- *Semi-supervised learning*

  Since it's a human job to label each entry and it's really time-consuming, it's easy to deal with a dataset with plenty of labeled and unlabeled instances.

- *Reinforcement learning*

  In this context, the learning system can **observe** an environment, pick an action and **obtain a reward** in response (either negative or positive). As the time passes by, it learns by itself which is the best strategy to pursue, avoiding dangerous decisions. We can say it really learns from its mistakes.

3.1.3 Results evaluation

One core step is the human evaluation of what has been the model results. There are three different kinds of results:

- *Right fit*

  The model has the suitable complexity to generalize the input data well, abstract the inner features and predict a correct, or at least trustable, output.

- *Overfitting*

  It means the model performs well on the training data, but it **does not generalize well**. It's as if, when a student prepares for an exam, they memorize everything without really understanding the concepts. This problem occurs when the model is **too complex** relative to the amount and loudness of the data.

- *Underfitting*

  It's the opposite of overfitting. It happens when the model is **too simple** to learn the underlying structure of the data. Stepping back to the student metaphor, it's as if they had not studied enough.

The ideal situation is when the model right fits the input data. When it's in the overfitting or underfitting situation, there are many possible solutions to tackle the problem, based on the as-is state.

If the model is overfitting, it could be simplified by selecting one or fewer parameters, gathering more training data, or reducing the noise in the data itself.

When in underfitting, the programmer could choose to select a more robust model or feed better features to the learning algorithm or even reduce the constraints on the model.

3.2 Deep Learning

Since I worked on two DL algorithms, I thought it would have been necessary to keep a special section about it.

3.2.1 Neural Network

What's behind most of the DL algorithms is a structure called Neural Network. It is essentially a network that uses neurons as **layers of knowledge** to conduct the network itself to have, from an input `X`, one or more output `Y` [3].

The name comes from the initial insight to emulate the interaction between neurons in the human brain.

Neural Networks have gained lots of attention recently because the amount of labeled data is enlarging, and large NNs are now reaching new state-of-the-art performances every passing year.

They have an input layer, one or more hidden layers, and an output one. Each artificial neuron connects to another and has an associated weight and threshold. If the output of a single neuron is above a specified threshold value, given by a what is called an **activation function**, that node is activated, sending data to the next level of the network. Otherwise, no data is passed.

There's a vastness of different structures, and there's not a single model that performs better than any other one in all the tasks, so it is the programmer's job to build the model that suits better for the desired task.

For each input pattern, the network computes an expected value associating the given pattern to a particular class or numeric value.

The learning is obtained by setting at the beginning weights with casual values and applying an activation function (referred as *sigma* σ in equation 3.1) to perform the computation of the expected value.

$$output = \sigma(\sum_{i=1}^{m} w_i * x_i + b) \tag{3.1}$$

Based on the predicted value, the weights will be updated.

Considering the supervised learning case, every input sample is labeled, i.e., it has a correct value or class. If the prediction is equal to the actual value, then the weights will be kept as they are, but the weights will be updated if it's the contrary. This behavior is performed to ensure that neurons wrongly on (or off) could change their state in the following case to take into account.

In general, to improve the weight variation, it's common to define a delta rule that describes what is called "the gradient descent": it is a deterministic method that leads the values of a function to follow the direction defined by the gradient. It slows down in an approximation of the minimum value.

The process of computing the predicted value is called **forward propagation**. In contrast, the process of updating the weights through the hidden layers is called **backward propagation** and represents one of the most complex aspects of a neural network.

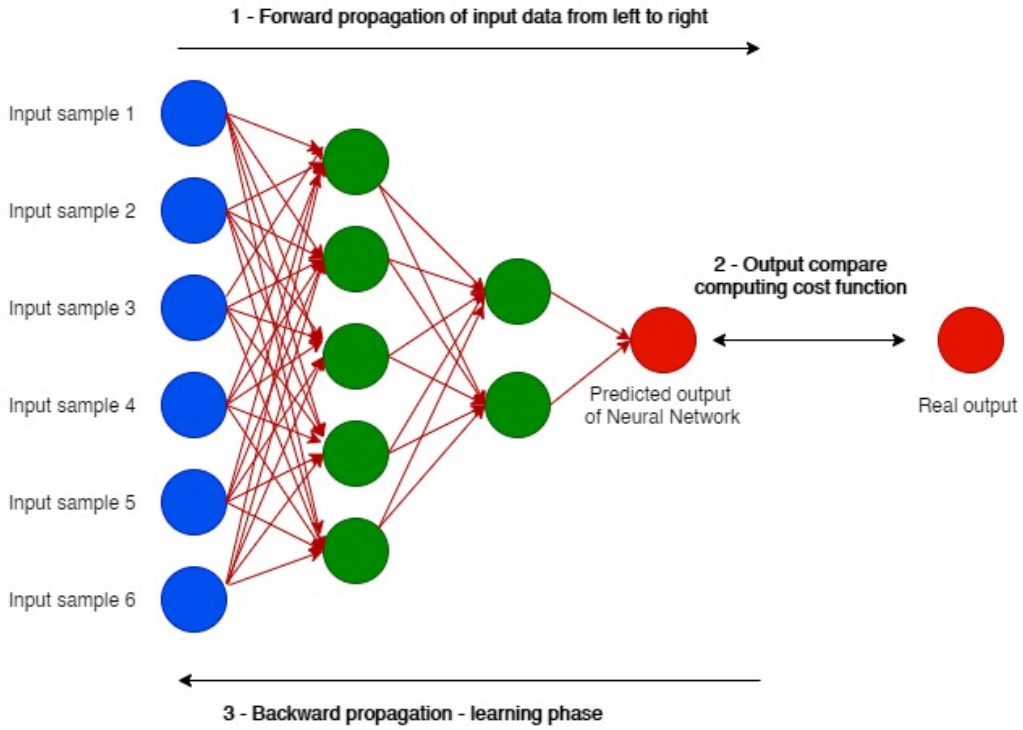


Figure 3.1: Structure and process of learning in a neural network

3.3 NN structures in NLP

Neural networks are grouped in multiple types based on the need.

The following are the most commonly used kind of NN nowadays:

- *Multi-Layer Perceptron (MLP)*

  It's the most commonly used and the most straightforward network on the list. It's constituted by one input layer, multiple hidden layers, and one output layer with one or multiple neurons. It's at the base of the creation of more complex constructs as follows.

- *Convolutional Neural Network (CNN)*[4]

  This type of network is similar to the MLP. Still, it relies on a mathematical operation called *convolution*: given two functions, this operation consists in processing them to produce a third one that expresses how one function

influences the other, looking at the shape. Essentially, after reversing and shifting one of the two initial function, it is calculated as the integral of their product. The convolution function is then produced after the integral has been evaluated for all values of shift. This kind of architecture is mostly used to find patterns in image recognition and computer vision but some layers that create a convolution kernel over a single spatial dimension could be applied to NLP too.

The input object is tipycally a matrix, that well represents images, if considering to computer vision's tasks. In case of NLP tasks instead, each row of the matrix corresponds to one token, generally a word. Thus each row is a vector that represents a word. Typically, these vectors are word embeddings like **word2vec** [21] or **GloVe** [20], but they could also be one-hot vectors that index the word into a vocabulary.

- *Recurrent Neural Network (RNN)*

  This kind of neural network looks very much like a feedforward neural network (also known as MLP), except it also has connections pointing backward. It's a type of neural network capable of handling sequential data and this is why it often works well on text. The difference from a simpler network comes when dealing with very long sequences.

  Every neuron is called recurrent neuron and at each time step receives the inputs, as a normal neural network, as well as its own output from the previous time step. In this way it can have a sort of memory of what was processed before some word. Therefore this neuron has two sets of weights.

  Two of the most common transformations are the Long-Short Term Memory (LSTM) [5] and the more recent Gated Recurrent Unit (GRU) [6].

- *Attention* [7]

  Introduced in 2014, this constituted a groundbreaking idea. It relies on a structure composed by an encoder and a decoder of text. This technique allows the decoder to focus on the appropriate words at each time step. It allowed a significant improvement on the state-of-the-art softnening the effect of the limitations of RNNs in very long sequences.

In the last years, this idea was perfected and it led to the next improvement that follows.

- *Transformer* [8]

  Applied in Neural Machine Translation, this architecture uses just attention mechanisms (coordinated with other kinds of layers) but without any RNN or CNN layer. It's faster to train and easier to parallelize.

  The improvement comes from the encoder's Multi-Head Attention layer that encodes each word's relationship with every other word in the same phrase. The result is a different attention to each word. Even the initial positional encodings are fundamental: dense vectors representing a word's position in the sentence. This provides the model with access to each word's position, which is crucial because all the following layers have no way of knowing such aspect.

- *BERT* [9]

  Developed at Google, this structure shows the effectiveness of self-supervised pre-training on a large corpus. The authors stacked many Transformer modules. Then they fine-tuned them on various language tasks.

  This model results to be bidirectional, as the name suggests (Bidirectional Encoder Representations from Transformers). To understand the importance of this model, it just needs to say that this model can predict whether two sentences are consecutive or not. This is challenging when dealing with tasks such as question answering.

- *MUM* [10]

  This is a multimodal architecture. It means it's been trained over different modalities of input information at the same time. It understands what a person says and can handle complex requests that usually take several searches to be covered. It relies on a Transformer-based structure too, and Google experts say it's 1000 times more powerful than BERT. It's also transversal, taking into account 75 different languages. Its power resides in the capability to consider various aspects regarding the same question.

Therefore, it's been thought of as a new DL model to improve the quality of any Google search.

- *LaMDA* [11]

  The acronym for "Language Model for Dialogue Applications" is a brand-new architecture based on the previously mentioned Transformer and can follow a conversation even if the subject changes rapidly. It's considerable, knowing that the more commons chatbots can merely handle a linear conversation. Its application could rely primarily on the virtual assistants' features to improve how they respond to us.

3.4 Evaluation metrics

To obtain good insights of any machine learning algorithm, metrics are one of the crucial points to pursue. This choice influences how the performances are measured and compared and when the developer could be satisfied with what one has achieved so far. Different metrics are applied considering the task that is tackling. A classification problem is divergent and requires a different approach from a regression problem.

I had to tackle two classification problems on this job, so I'll focus on metrics regarding this branch in this section.

- *Classification Accuracy*

  This first metric represents the number of correct predictions as a ratio of all predictions made.

  It's one of the most commonly used values, but it is significant only when considering a *balanced dataset*. It could result in a misleading metric if the dataset is not well preprocessed before the train phase.

  For instance, imagine you have to solve a binary classification problem. You only have two classes where to choose from. A model scores a 90% accuracy. It's impressive at first sight but analyzing the dataset, you realise it's composed of 90% of entries belonging to class A and a mere 10% to class B.

The developed model will likely predict every new entry as part of class A as it's been trained over many more examples of that class. But in the real world, when it will be asked to recognise an entry of class B, it will probably wrong the assumption.

This case illustrates well how just a single metric is actually not enough to be satisfied.

- *Log Loss*

  Logistic loss is a metric to evaluate the predictions of probabilities of membership to a given class. It essentially measures how the model is sure to assign a class to an entry when it is presented.

  It's a measure of confidence for a prediction by an algorithm and it can have a real value between 0 and 1.

  This is useful to punish or reward the model based on its confidence in the prediction made. Given model parameters, this kind of reward/punishment model involves selecting a likelihood function that defines how likely a set of observations is. Because it is more common to minimize a function, the log-likelihood function is inverted by adding a negative sign to the front.

- *Area Under ROC Curve [ROC AUC]*

  This is a particular metric for binary classification problems.

  The AUC describes the model's ability to discriminate between positive and negative classes. A numeric value of 1.0 is surely utopic and means the model never wrongs a prediction. An area of 0.5 means the model is not more accurate than a random guesser instead.

  In this context, the numeric value is better explained with a plot of the positive and the false positive rate for a given set of probability predictions at different thresholds, also known as the Receiver Operating Characteristic (ROC) Curve.

  The false-positive rate is the ratio of negative instances that are incorrectly classified as positive.

  A good classifier tries to stay as far away from the center of the plot as possible (therefore toward the top-left corner).

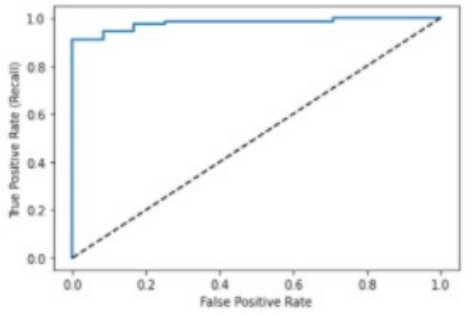


Figure 3.2: ROC AUC graphic

The image 3.2 represents an example of how a good ROC AUC curve looks like.

- *Confusion Matrix*

  This is a technique used to determine the types of errors the model makes. It returns a multidimensional array where the rows are the actual classes, while columns represent predicted classes.

  Using an image representation could help a lot when examining where the model could be improved.

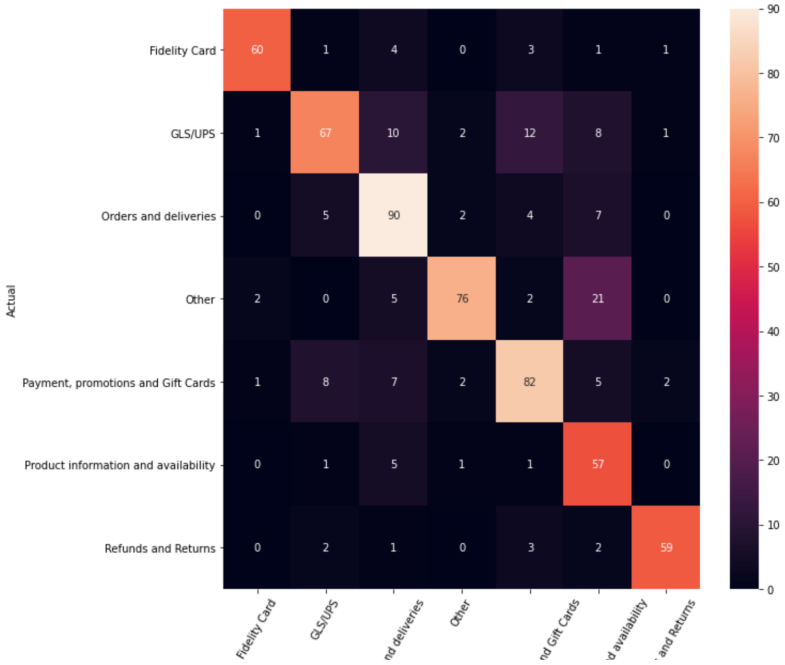


Figure 3.3: Confusion Matrix graphic

Image 3.3 illustrates how a graphical representation of a confusion matrix looks like. The ideal situation is when the main diagonal contains all the samples. That would mean the model has always chosen the right classification.

- *Precision*

  An interesting information to get from the confusion matrix is what's called precision: the accuracy of the positive predictions. It's obtained with the following formula 3.2

$$P = \frac{TP}{TP + FP} \tag{3.2}$$

  where:

  P = Precision
  TP = True Positive instances
  FP = False Positive instances

- *Recall*

  Even called Sensitivity, is the ratio of true positive instances that are correctly detected by the classifier.

  This aspect also includes the False Negative instances or those that have been wrongly assigned to another class, as the formula 3.3 illustrates.

$$R = \frac{TP}{TP + FN} \tag{3.3}$$

  where:

  R = Recall
  FN = False Negative instances

- *Harmonic Mean*

  It is often convenient to combine precision and recall into a single metric: the harmonic mean. Often called "F1-score", it's useful because it gives

much more weight to low values. This means this value will be high just if both precision and recall are high, as shown in equation 3.4

$$F1 = \frac{2}{\frac{1}{P} + \frac{1}{R}} = 2 * \frac{P * R}{P + R} = \frac{TP}{TP + \frac{FN+FP}{2}} \tag{3.4}$$

where:

F1 = Harmonic mean

The last three metrics are worthwhile but, depending on the task, the developer could not be interested in having the three of them similar values. In some cases, it's more important to care about precision instead of recall or vice versa.

In my case, when I used them to classify over multiple classes, I was interested in reaching a good result with every metric. But this is difficult considering that increasing precision reduces recall and vice versa.

This is why everyone talks about precision/recall tradeoff.

Chapter 4

Dataset

One of the most important parts in the development of a DL/ML model is the dataset, or the source of learning information the model has to tap into.

4.1 Initial dataset

The image 4.1 illustrates how the initial information was represented.

Case Number	Subject	Description	Type	Sub Type
414548	Re: Hai ricevuto il tuo ordine ? Numero (	Oggi non ho ricevuto il mio ordine. Vi prego di verifica	Orders and deliveries	Delivery errors
422829	Resi e rimborsi	Devo rendere tutti i prodotti, ho provato 4 volte ma av	Refunds and Returns	Return procedure authorization
414550	Orders and deliveries	Hi May i know the length in cm from the shoulder to th	Orders and deliveries	Order information
422831	Ordine n. 37106113	Pacco reso a magazzino completamente macchiato d	GLS/UPS	Claims procedure
414551	Disponibilità e informazioni sul prodotto	Desidero sapere se la taglia M corrisponde alla 44 op	Product information and availability	Fit and measurements
422832	Resi e rimborsi	Vorrei rendere il cappotto acquistato perché della tagl	Refunds and Returns	Return procedure authorization
414553	Resi e rimborsi	vorrei rimborsare il prodotto ordinato (Numero d'ordin	Orders and deliveries	Change / cancel order
422833	Ordini e spedizioni	Salve, vorrei sapere , quando è prevista la consegna	Refunds and Returns	Return progress
414554	Ordini e spedizioni	Si prega di annullare l'ordine. Mi dispiace disturbarLa.	Orders and deliveries	Change / cancel order
422834	Support technique	Je n'arrive pas à créer un compte.	Technical assistance	My Account subscription
414555	Réclamations vendeur	Bonjour,J'ai reçu un cadeau de mon mari aujourd'hui,	Product information and availability	Products complaints

Figure 4.1: Some samples from the original dataset

I was presented with a report containing `30.811` entries in many languages where the most consistent were Italian, English, German, French, and Spanish. Some entries in Croatian and Dutch were present too but they were a really tiny minority and it was impossible to consider them in the development of a full-functional DL model.

Column description:

- *Case Number*: the unique identifier for each row
- *Subject*: the title of the customer request
- *Description*: the description of the issue the customer is experiencing
- *Type*: category to which the issue was classified by the customer service
- *Sub Type*: sub-category. It constitutes a specification of the Type column

I was asked to initially consider the Sub Type column as the target to label each customer issue.

I took advantage of all the languages in the cleaning-text phase, but when it came to developing the most important text classification algorithm, I just used the Italian tickets. They were `25.683`: the 83,3% of the initial dataset.

Figure 4.2 illustrates how the entries are distributed through the different subtypes. It just considers the Italian dataset, as mentioned before.

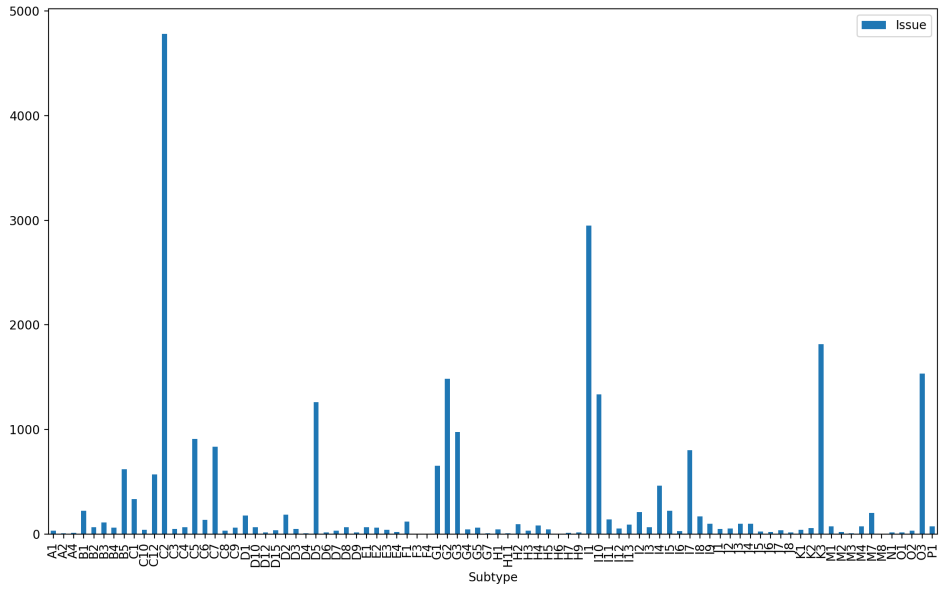


Figure 4.2: The Italian instances distribution

Each subtype has been labeled with an appropriate acronym to detect immediately the Type they belong to.

The initial class names were with a complete sense. Some examples are "Refunds and Returns", "Comments and Suggestions", or "Technical assistance". Even if a human would prefer the full-sense nomenclature, it was more convenient to keep track of each class with the acronym method in the development phase. Classifying by Type was always an option to pursue eventually. In this way, I immediately switched from one target to another (Type or Subtype).

4.2 Derived dataset

To develop the clean-text model I reworked the initial dataset.

I discovered I had to remove entire phrases and not just single words. Thus it became necessary to have my initial data in a different way.

As I will describe in section 6.1.1, the dataset was divided as follows:

Grazie Saluti	0
Si prega di annullare l'ordine.	1
Mi dispiace disturbarLa.	0
vorrei sapere come mai il mio ordine fatto il 5 gennaio ,oggi 9 gennaio,è fermo..non risul	1
Gent.li Sigg.ri da qualche giorno tento di effettuare un ordine, ma il Vs. sito non me lo co	1
Desideravo sapere se siete a conoscenza di problemi al sistema.	1
Grz e cordialità	0
Non riesco ad accedere nel mio account	1
Sarà disponibile la Taglia.38?	1
Ho fatto un ordine il 2 gennaio di 154 euro, già addebitati, ma ad oggi, risulta ancora in s	1
Quando arriverà?	1
Cancel please	1

Figure 4.3: Some samples from the derived dataset

The first column represents the original text but divided by sentences.

The second one has a binary value (0 or 1) indicating the importance of the sentence. The 0 states the phrase should be considered as irrelevant, while the 1 value indicates that sentence is important and should be considered in future evaluations.

For instance "Grazie Saluti" is just a greeting sentence, adding no information to a possible context and thus should be deleted. At the same time, when a customer says "Si prega di annullare l'ordine", it's clear it is indicating the experiencing issue. That is the kind of message we want to keep.

4.3 Data problems

The first problem I had to deal with was the presence of **a lot of languages**. I was able to overcome it when I developed the simpler deep learning algorithm, which focused on cleaning the text from sentences containing sensible data and the useless ones such as greetings and thanks.

The much trickier task to classify each message in one of the multiple target classes was not worth it because of a substantial loss in performance.

A second problem was the **unbalancing nature** of the dataset, clearly highlighting how some classes would have been more recognizable than others like C2 or I1, as the image 4.2 expresses.

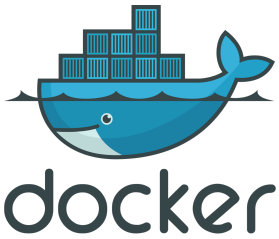
A deep analysis of the entries helped me perform an efficient categorization and eventually exclude some irrelevant classes.

Chapter 5

Technologies

To accomplish what I was supposed to do, I took advantage of different kinds of technologies.

- *Docker* [12]



Docker is a containerization software.
It offers an efficient volume management of any application. It lets you integrate an application regardless of the running Operative System. It uses virtualization to deliver software in packages called containers.
It has been used to encapsulate the software in a pluggable microservice, exposing APIs to exploit the realised functionalities.

- *Tensorflow* [13]

Tensorflow is an open-source end-to-end machine learning platform. It offers several features for modelling the input data in tensor format and developing and training ML models.
It is used within the microservice to build and train the text classification models.

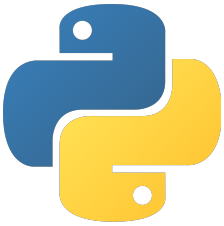
- *Pandas* [14]

Pandas is one of the most used tools to perform data analysis and datasets manipulation.
It is used within the microservice to load and manipulate the input content of the various APIs.

- *Scikit-Learn* [15]

Often abbreviated as Sklearn, is a tool that offers ML implementations of some of the most used ML algorithms (Naive Bayes, Linear SVC, and many more).
It's been used to try different and simpler solutions than the chosen one and to compute statistics and useful graphs to understand how the models were performing.

- *Python* [16]



Python is a high-level programming language.
It's the main language used in the data science industry.
It is used to develop the entire backend codebase.

- *Google Colab* [17]

Colab is an online platform hosted by Google that lets you experiment with your code with the possibility to use Google's calculus power.
It's been used to develop the DL model structures quickly and test them using the GPU and TPU temporarily offered.

- *MLflow* [18]

Mlflow is a recent tool to automate the handling of an ML model lifecycle. It's been recently added to the tools used in the company, and it gives you the chance to handle the versioning of a ML model and serving it over different environments. It also lets you store information about metrics and tags used to train and recognise the model. It's currently applied to keep up-to-date the models integrated in the microservice I developed.

It allows the user to carry out a standardized deployment of an ML model.

- *Flask* [19]

It is a backend micro-framework at the base of the microservice structure.
To merely expose some APIs, it was not necessary to adopt a more complex framework.

5.1 The ecosystem

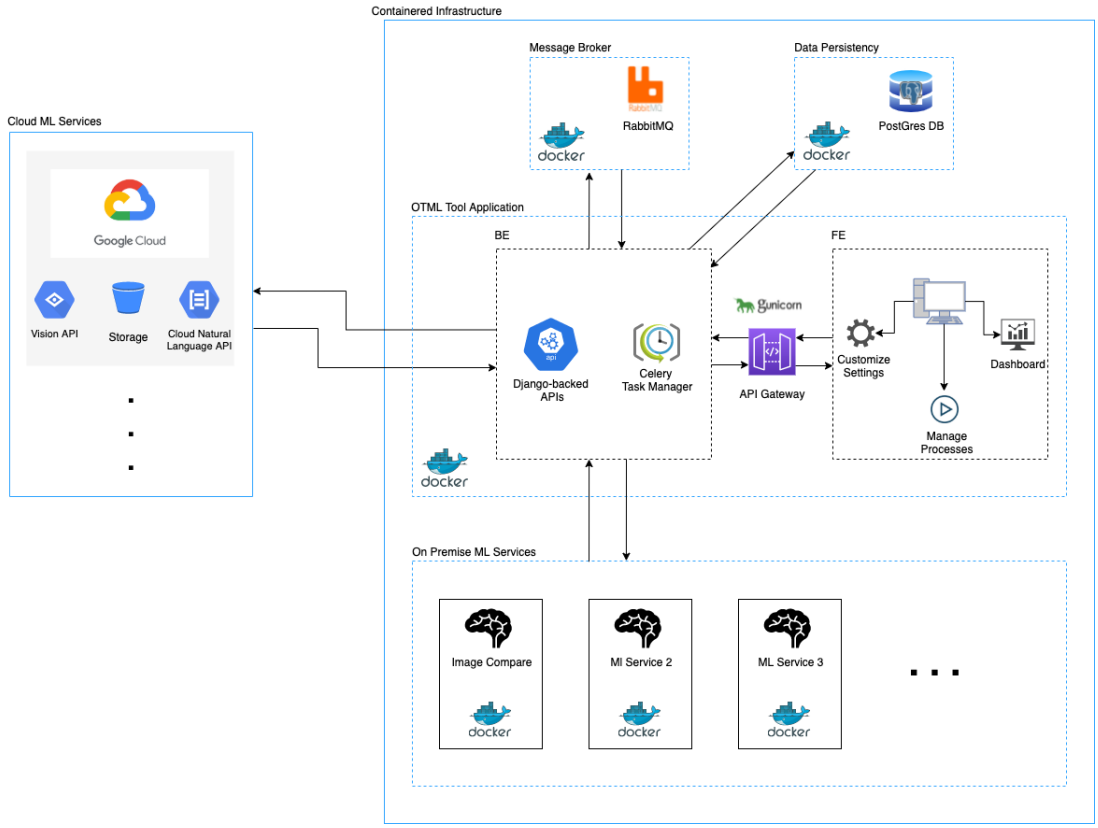


Figure 5.1: The entire ecosystem

The base component of the infrastructure is the Docker Network. Its scope is to hold all the containers. The structure is composed of four types of microservices.

1. *Message Broker*: the microservice responsible to handle the task queues
2. *Data Persistency*: the microservice used as a database to store the data
3. *OTML*: it's the main tool. In the frontend side there's an interface built with the open source framework Angular. At the same time, for the backend, the team has chosen to code it in Python with the help of the framework Django and Celery, a software to handle tasks asynchronously.
4. *ML services*: the set of ML microservices. The text classification microservice is part of these entities.

How does it work

When a user acts as the frontend interface, in the backend side Django communicates to the message broker which tasks need to be processed, and Celery will generate the **task_id**, the unique identifier of each task. As a consequence, the message broker will hold the queue of documents. The number of setted **workers** constrains this asynchronous communication.

Each worker is a process unit; it tells how many tasks could be processed in parallel. The software whose work is to set this kind of parameters is the Python WSGI Gunicorn. Celery even has the role of enqueueing a document when it's ready to be processed.

A practical example:

5 documents are loaded by a frontend API. After that, a proxy server called NginX will deliver the request to the backend. Then Django, with the support of Celery, communicates to the message broker microservice to process these 5 documents. As a consequence, the tasks will be registered in the RabbitMQ instance.

For each document there's a *unique task*. An example would be to get the entities from an input text. This task is computed for each document.

If the system is set to work with two workers, then two documents will be processed simultaneously while the three remaining will wait until the workers are available again. When each task is being processed, its status will be registered and sequentially updated on the Postgres database.

These tasks usually exploit some features exposed in the ML microservices. This means that when a task is processed, it will make some calls to the microservices that expose the features that respond to what the user requested.

The tool' structure's advantage is that it can be sold to a specific customer based on its needs. Being constituted by pluggable microservices is convenient. Once the customer has purchased the use of the Tool, the settings will be such as to guarantee only the access of the purchased services.

To be usable by more users, some features have been developed even exploiting Google services. So the offer includes both on-premise and external provider features.

Part II

Text classification techniques

Chapter 6

Text anonymization

6.1 The model

In this chapter I'm going to focus on the design of the first of the two AI models I developed in this experience.

It has as the main goal to clean every ticket from any sensible data and useless words such as greetings and thanks, looking to the next classification and especially to the future developments (see chapter 8).

What motivated me to build up this model was a previously done study of the initial dataset. Looking to a possible sentiment analysis, I tested many ticket messages over what is the Google Sentiment Analysis API. It was clear that marginal expressions like "Thanks", "Cordiali saluti", "Best wishes", "Have a nice day", and many more were influencing the judgment of the API such that some clearly negative messages were turned as neutral or even a bit positive. The latter is the case when a negative message is concise. Those kinds of words usually have a big impact and indicate the sentiment of the main message. However, my situation was not like this.

Later on, it had been more apparent that even in a text classification that words wouldn't have a minimal discriminative power. Their deletion would have been necessary.

As explained in section 1.2, the additive presence of sensitive data could have been interpreted as a way to discriminate whenever a sentence is positive or negative. A machine cannot have a bias that, for instance, when there's the name "Giulia" is a more positive sentence than when a name like "Mike" appears.

The need to rework the input text was clear, and a stopwords list would not have been enough to comprehend all the possible cases.

6.1.1 Preprocessing

One of the first questions that came to my mind was, "How can I treat a dataset like that?" After all, I had just what was supposed to be the dataset to feed the next AI model.

I spent some days looking over the web to find if someone had passed before a problem like this. Knowing some messages came from mail threads, I read the paper "Email data cleaning" [22] thinking it would have given me some insights.

I was at the first experience in the field with a machine learning model and it seemed not to be the right route to pursue as a beginner.

With the help of Pandas, I examined one more time the dataset, and I discovered that what I had to remove were entire phrases and not just single words. Then I used a little script to process the input dataset and rework it in the format exposed in section 4.2.

That problem turned out to be a **binary classification**. That way, I would have been able to easily classify each input phrase as relevant or not.

The dimension of the dataset was not a problem since I was presented with more than 1000 sentences for each of the main languages: Italian, English, French, Spanish, and German. That number was enough because there were an almost equal number of positive and negative cases for each language.

I composed the train set of 80% of the original dataset and then I split the remaining 20% between the validation and the test set equally.

Each set of samples was then converted to be a **tensor**, a more suitable object for the model I was about to develop.

6.1.2 Model development

The choice of the ML development platform fell back on TensorFlow since it was the one I used in previous courses I enrolled in, and the company too pushed me in that direction.

The objective of this model's development was to use just one model independently of the main language. To have an adaptive model like this makes this task more complex.

Then I moved on remembering the **transfer learning** approach, and I started looking around to find something that should work for me.

I came across a Google implementation of a multilingual text embedding [23] that covers 16 different languages and is based on CNNs (see section 3.3 for a more exhaustive explanation). In the embedding description, one of the intended uses is text classification and is optimized for a multi-word length text, like sentences. This text embedding is inspired by the paper "Learning Cross-lingual Sentence Representations via a Multi-task Dual Encoder" [24].

Then the only job remained was to integrate it and see with a simple architecture how the model would perform.

Hyperparameter tuning

One of the most important steps in developing a DL model is the hyperparameter optimization. A **hyperparameter** is a special parameter that could be modelled and affect the learning process. There are two possible types:

- *model hyperparameter*

  This is the kind of parameter that refers to the model selection task as, for instance, the choice of the nature of a NN layer or the number of layers. These values cannot be inferred in the training process.

- *algorithm hyperparameter*

  The nature of these parameters resides in the training process. Parameters such as the learning rate or the batch size, i.e. the number of training samples to process before updating the model's parameters, directly influence the model performances.

The performed tuning phase was conducted principally using the **grid search** approach: it consists of selecting a set of parameters to tune in the training process and, for each one, select a set of possible values to investigate. Grid search then trains the model with each parameters' permutation in the resulting Cartesian product of the chosen sets and evaluates the model's performances. This evaluation is performed over a held-out validation set. The output of the grid search is the setting that reached the best score when testing the model on the validation set.

One more used approach was **early stopping**: it aims to stop training when a specific metric has not improved in a certain number of epochs. It is used when there are many parameters to update each time, and the training process could be uselessly long.

The metric I chose to monitor was the log loss (see the section 3.4). TensorFlow lets you use a callback to do so. Two of the most important parameters to set are the number of epochs to wait from stopping and the flag to restore model weights from the epoch with the best score values. I used these same two settings.

Model compilation

A special paragraph is needed to explain two crucial metrics in the model's configuration for training:

- *the optimizer*

  I adopted one of the most commonly known optimizers: Adam [25]. It's an optimization algorithm. It is more effective than Stochastic Gradient Descent because it merges two advantages from two variants of that technique. It uses indeed

  1. the **Adaptive Learning Algorithm (AdaGrad)**, which maintains a learning rate for each parameter that improves performance on sparse gradient problems. Working with natural language involves this issue;
  2. the **Root Mean Square Propagation (RMSprop)**, which uses a similar approach to AdaGrad, but these learning rates are adapted to how quickly the gradient descent is changing. This means the algorithm does well on online problems, i.e. when data are available in sequential order, and the best predictor is updated at each step.

  The essential is that Adam computes both the squared gradient and the exponential moving average of the gradient itself.

- *the loss function*

  I selected the **binary cross-entropy** function since it is a binary problem. The purpose of this function is to compute the cross-entropy loss between

true and predicted labels. In practice, this function is set to work with probabilities with values in the range [0, 1] instead of working with logits.

Since both set of labels have a fixed number of possible values, I had to deal with two discrete probability distributions. Thus the function is defined as shown in equation 6.1

$$H(p, q) = -\sum_{x \in \chi} p(x) * log(q(x)) \tag{6.1}$$

where

p, q = two distributions
H(p, q) = cross-entropy of q relative to p over a given set
χ = topological space
x = support of a measure in χ

Model Structure

TensorFlow provides you two different APIs to have the chance to build any model: Sequential and Functional.

1. **Sequential**

   It's the easiest way to create a neural network. Basically, it allows you to create the model layer-by-layer but not sharing or branching layers. It suits better for models with just one input and output.

2. **Functional**:

   the advantage of using this API is that it is more flexible and covers all the lacks of the previously mentioned API regarding branching, sharing them, and having multiple inputs and outputs.

The choice to use one instead of the other is up to the nature of the task to solve. In my case, I had a single output, the text sentence, and I had to predict just a single data, whether or not the sentence was valid. It was clear my choice would have taken the side of the **Sequential API**.

The next step was to choose the number and the dimension of the layers to stack after the text embedding. Observing the model's performance initially

adding just an output layer of 1 neuron, I increasingly added layers and neurons. I ended up with two hidden layers of 96 neurons each and an output layer of just 1 neuron since the possible choices were just two and were mutually exclusive. The nature of these layers is **Dense**, meaning that every neuron of a previous layer is directly connected to each neuron of the next one.

Figure 6.1 shows the final structure of the developed model and shows another interesting piece of information: the number of parameters.

The embeddings constitute the majority of parameters. Some predefined text embeddings are set to leave the weights as they are, but this is not the case.

I thought the chance to change all the weights slightly would have been an opportunity to suit that layer to my task better and thus achieve better performances.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer (KerasLayer)     (None, 512)               68927232
_________________________________________________________________
dense (Dense)                (None, 96)                49248
_________________________________________________________________
dense_1 (Dense)              (None, 96)                9312
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 97
=================================================================
Total params: 68,985,889
Trainable params: 68,985,889
Non-trainable params: 0
_________________________________________________________________
--
```

Figure 6.1: The structure of the text anonymization model

6.1.3 Model Saving

Looking to the TensorFlow documentation, there are two format to save the model: SavedModel and HDF5.

1. *SavedModel*:

   This format contains a complete TensorFlow program with computation and trained parameters, resulting in an execution graph. It has a significant advantage: it does not require the code used to build the model to run it; thus it's more convenient when it's time to deploy it.

It's even possible to just save and load weights using checkpoints while training.

2. *HDF5*:

   The acronym for Hierarchical Data Format (version 5) uses object configurations to save the model's architecture. It's older than the SavedModel format and was defined back when Keras, the main library now integrated in TensorFlow, was independent and aimed to support multiple backends without being constrained to anyone.

   With the advent of the version 2.x of TensorFlow, it should be used just if a file compatible with older versions is needed.

Working with TensorFlow in its 2.5 version, it seemed natural to adopt the SavedModel format to save any model I had to develop since these would have been integrated into a brand-new tool with no previous dependency.

6.2 Time

One core aspect in training, especially in hyperparameter tuning, is the needed time since it's not a deterministic computation.

In this context Google Colab turned out to be a time-saver. As explained in chapter 5, it makes available the power computation Google has. In this way, instead of waiting half-hours like in a normal training that use the PC's CPU, I used the TPU [26], and it occured me just a few seconds.

This resource is limited, and just the users with a contract for Google's G-Suite can use it, and just for a limited time, but it is enough for a full-time day spent almost entirely on training.

6.3 TensorBoard

An exciting option that comes with the use of TensorFlow is the tool TensorBoard. It gives the chance of examining graphically and clearly the training trend of a specified model.

Figure 6.2 shows one tab of the interface.

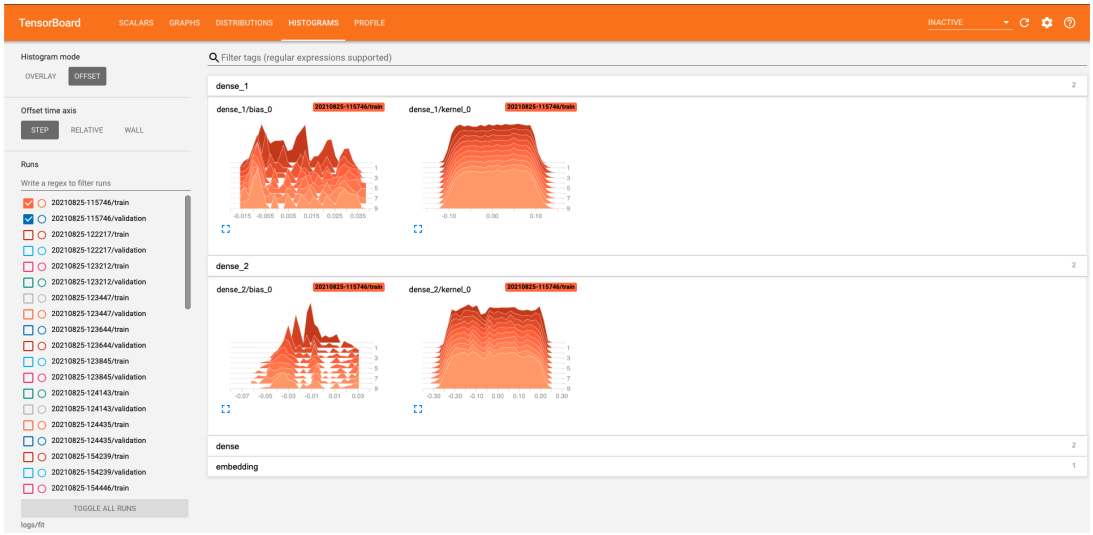


Figure 6.2: The typical TensorBoard interface

On the left side, under the section "runs" it shows all the runs memorized when training and which have been selected to be displayed on the right.

The *Histogram model* and the *Offset time axis* options let you change the way the graphics are depicted.

Talking about the main tools offered by the navigation bar, the **Scalars** tab shows the progression of loss and other metrics through the epochs. Some trackable values are the learning rate, the accuracy, and the training speed.

The **Graphs** tab shows the structure of the model and how each layer is connected. This is useful to check if the model looks as intended.

The **Distributions** instead is referred to the tensors' weight distribution. It lets you have a look at how weights and biases change over the epochs.

The photographed tab, **Histograms**, shows in different formats the tensors' distribution in training for each layer (embedding, dense, dense_1, and dense_2).

6.4 Results

The evaluation of this model was satisfying. The model had been tested over the 5 main languages of the initial dataset: Italian, English, German, Spanish, and French.

Even though the dataset was composed originally by a larger number of Italian tickets, thanks to the rework exposed in section 4.2, I was able to obtain

a discrete number of single sentences for each language.

I consequently divided the dataset by language in 5 subsets, each one composed by 1000 sentences coming from both thread mail and user requests made from the business user website.

The division of each dataset was constituted as follows:

- training set: `80% - 800` samples
- development set: `10% - 100` samples
- test set: `10% - 100` samples

I trained the model for `10 epochs` and with a `batch_size of 16`. The obtained results were remarkable with various metrics.

I initially based my judgement looking at the accuracy and the log loss. In both training and validation sets the numerical values are quite good.

The graphic exposed in figure 6.3 shows that. There's a moment in which the model does not improve its predictions anymore, showing a bit of overfitting, especially from the 6th epoch.

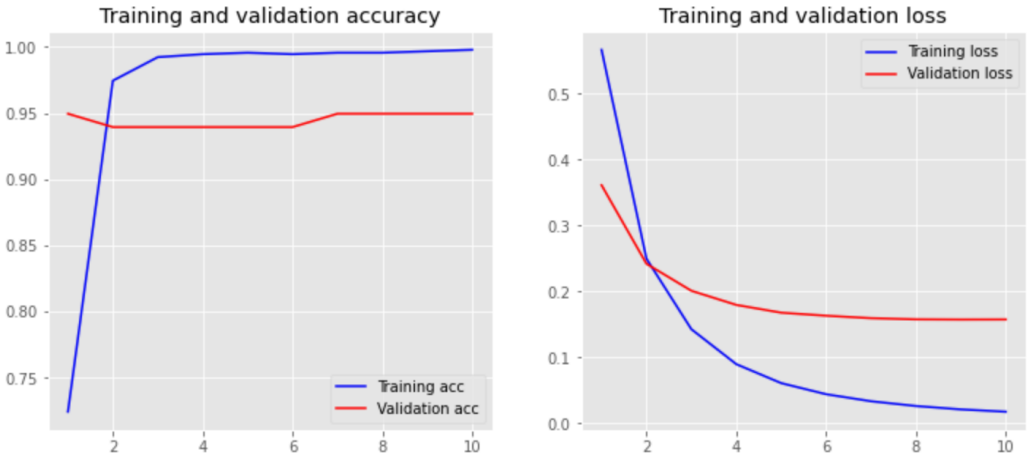


Figure 6.3: Graphics' performance of the anonymization model

The results in the test set showed an `accuracy of 91,2%` and a `loss value of 0.192`.

Even if these metrics gave me good insights of how the model was performing, I needed to be really sure that evaluations were not a coincidence.

Consequently, as anticipated in the 3.4 section, I then adopted the commonly used ROC AUC curve.

Figure 6.4 is the graphical representation of the ratio between the `true positives` and `false positive` rates.

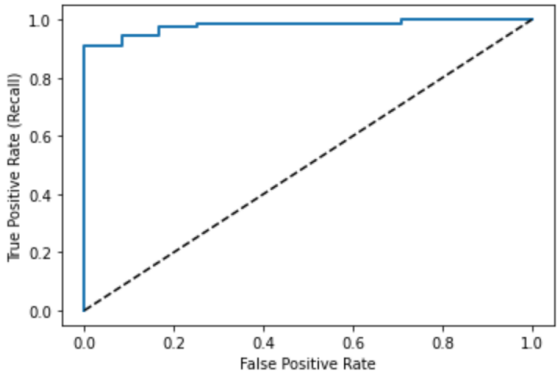


Figure 6.4: ROC AUC curve graphic

It is also possible to extract a numeric value that indicates how good the graphic is. It's a number in range [0, 1]: the nearer to 1 the value is, the better it means.

The rounded value associated with this graphic is 0.979. This metric convinced me that the model was ready and I could have moved to the next, and more complex, task.

Chapter 7

Multiclass Text classification

The solution I will explain is to resolve the original problem: automate the labeling process performed by humans.

A person has to choose between many different choices, each one mutually exclusive. Thus the problem I was facing was a *multiclass* one and not a multi-label problem.

The difference is that the latter has more labels associated with the same input as a possible output, whereas in multiclass problems, I can have just one correct label.

When it came to developing the main algorithm, one of the initial thoughts was to replicate the structure of the previous model. The performances, however, were poor, even changing the last layers' structure. I understood immediately that a significant change was needed.

7.1 A classical approach

After talking with professor Martoglia, I decided to try something simpler but maybe more effective: this led me to try classical ML algorithms.

In these terms, Sklearn implements many of them: I took advantage of these very implementations, letting myself be guided by what Sklearn developers suggest [27].

Following the schema reported in figure 7.1 from the **START**, I discovered what were the algorithms that would suit better: I had more than 50 samples,

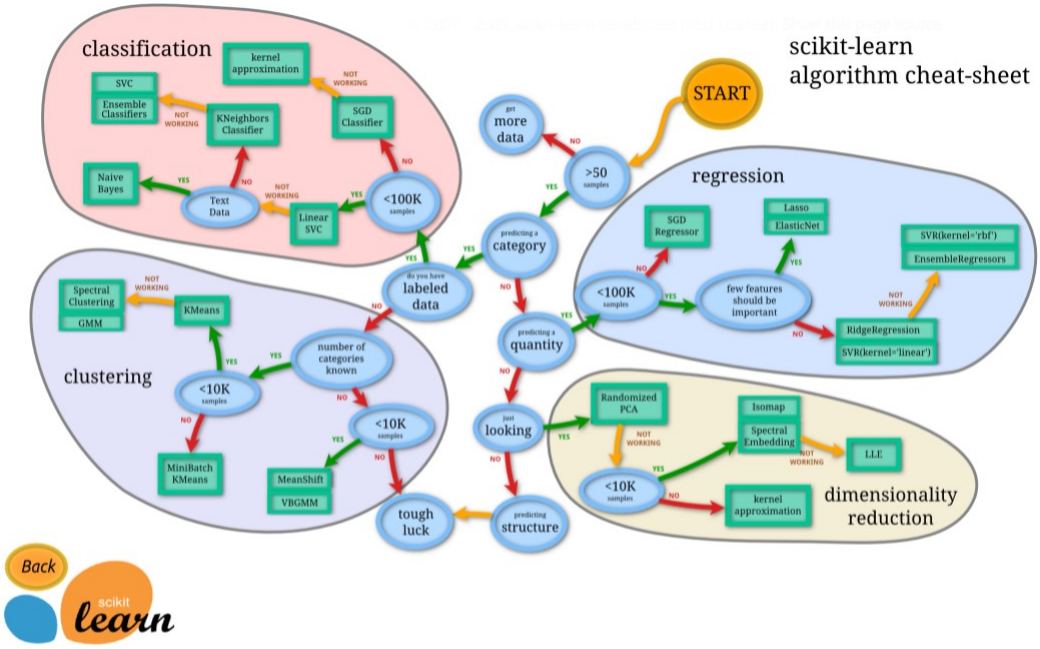


Figure 7.1: The cheat-sheet available at [27]

I wanted to predict a category with labeled data, but I had available less than 100k samples (25683 to be correct).

The first choice, therefore, was the Linear SVC

7.1.1 Linear Support Vector Classification

The **Linear SVC** implements the supervised algorithm SVM (Support Vector Machine) applied to classification tasks.

A support vector machine builds up a hyperplane or a set of hyperplanes (depending on the specific implementation) in a multi-dimensional space, which can be used for classification.

In general, a good separation can be obtained from the hyperplane, which has the greatest distance from the closest point of each class. The more evident the separation between the classes is, the easier to classify a sentence will be.

In the case of a linear SVM, a sentence is represented as an n-dimensional vector, where `n` is the length of the vector itself. In a hyperplane of `n-1` dimensions, that vector is presented as a data point. The objective of the algorithm is to find an optimal boundary between the outputs. This means maximizing the distance from the hyperplane itself to the nearest data point of each class.

Figure 7.2 illustrates how a good hyperplane would be represented.

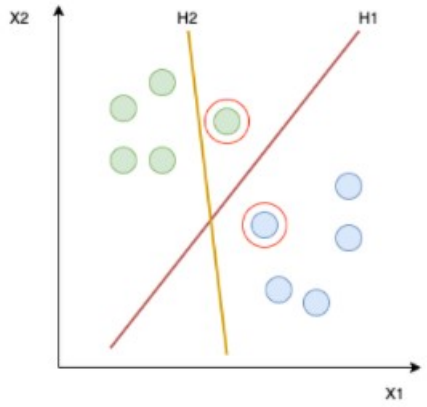


Figure 7.2: Examples of good and bad SVM generalizations

The hyperplane H1 perfectly divides the two classes with a good margin from the nearer data points (the two circled in red).

H2 does not represent a good generalization since it does not separate the classes.

Preprocessing

As in previous experiments, I merged the *Subject* and *Description* columns to have a more significant input, deleted any stopword, lowered and tokenized each word.

After this process, I converted both text and labels in the **TF-IDF representation**. The Term-Frequency times Inverse Document-Frequency is a measure to count each word's occurrences and give them an appropriate weight [of importance].

Sklearn offers different classes to perform this operation [28]. The chosen one is `TfidfTransformer`, that evaluates the idf as follows:

$$idf(t) = \log \frac{n}{df(t)} + 1 \tag{7.1}$$

where:
n = number of instances in the dataset
t = analysed term
$df(t)$ = number of instances in the dataset that contain the term t

The tf-idf value is then calculated by:

$$tf - idf(t, d) = tf(t, d) * idf(t) \tag{7.2}$$

Each instance is then normalized as the equation 7.3

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + ... + v_n^2}} \tag{7.3}$$

where:
$||v||_2$ = Euclidean norm of the vector v

To process the classes, I used a `LabelEncoder` object that allowed me to convert text labels to a numerical form.

Results

With a naïve approach to the linear SVC algorithm, I obtained an average accuracy on a validation set of just 54%. That was not an encouraging result, but I did not give up.

Since I was working with sparse data, i.e. data comprised of mostly zero values, I chose to apply a **Single Value Decomposition** [29] to reduce the input text dimensionality. I then applied a grid search to fine-tune different parameters such as "penalty" and "C" (which dictates the strength of the regularization). These upgrades were helpful and enhanced the accuracy up to 58,3%. Still, this was not satisfactory as the company expected.

7.1.2 Naïve Bayes

Following the Sklearn map (figure 7.1), the next question was if I had to deal with text data, and the answer was obviously yes. Then I moved on trying with the **Naïve Bayes** algorithm. It is based on Bayes' theorem with naïve independence assumptions between features. It's then a probabilistic classifier.

The Bayes' theorem says what is the outcome of a posterior probability as follows:

$$p(y|x) = \frac{p(x|y) * p(y)}{p(x)} \tag{7.4}$$

where:

x, y = events to which a probability is associated

$p(y)$ = prior term

$p(x)$ = normalization term

$p(x|y)$ = likelihood

In algorithm terms, x is a feature vector of m components $[X_0, ..., X_m]$ and y is the target to predict in a set of Y possible classes.

This algorithm assumes that words' position does not matter and considers just the occurrences of the most relevant words. This is called **Bag of words assumption**, and it leads to having conditional independence between each feature of the text sample given a specific class y.

In terms of equations, the previous paragraph is interpreted with the formula 7.5:

$$P(x_1, ..., x_n|y) = P(x_1|y) * P(x_2|y) * ... * P(x_n|y) \tag{7.5}$$

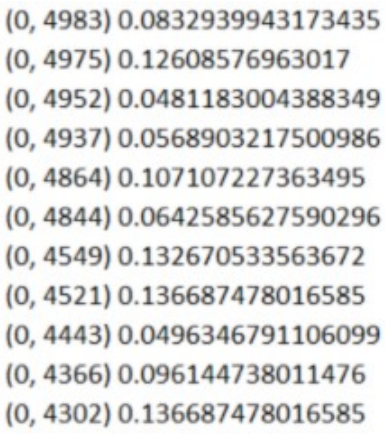
One of the most popular implementations is the one Sklearn has to offer: the *Multinomial Naïve Bayes classifier* that, as the name states, has been developed for multinomial models.

In this context, to predict the most likely class, the equation 7.6 is used.

$$y_{map} = \underset{y \text{ in } Y}{\arg\max}\, P(x_1, x_2, ..., x_n|y) * P(y) = \underset{y \text{ in } Y}{\arg\max}\, P(y) \prod_{x \text{ in } X} P(x|c) \tag{7.6}$$

Preprocessing

Even if the Multinomial NB classifier is suitable with discrete features, in practice counts like tf-idf work too.



I reused the preprocessing undertaken with SVC and used a TfidfVectorizer object to set a limit of features to take into account.
This way each sample of the train, validation, and test set would be represented as a record of unique numbers along with their weight calculated by TF-IDF, as shown in the figure aside.
The first tuple value represents the sample id, the second one is the unique integer of each word and the real value is the TF-IDF score.

Results

Before proceeding with evaluating multiple metrics, I considered as fundamental the accuracy value. However, using the `MultinomialNB` the accuracy score was disappointing, presenting just a 50,82% of accuracy.

This outcome led me to look around to other classical solutions.

7.1.3 Logistic Regression

The last trial I had with classical ML algorithms was directed to the Logistic Regression, specifically, the extension named `MultinomialLR`, as suggested in a consulted article [30].

Even if the premises were different, as in the article the algorithm is applied on an almost perfectly balanced dataset with more samples, I wanted to give it a shot. I had more than two possible categories, and I had to deal with a categorical variable, as there was the chance to acquire one of the possible values that cannot be ordered in an objective way.

Basically, the idea is to have weights w for any class k. This kind of algorithm is based on probability, too (see equation 7.7).

$$P(y = k|x) = \frac{\exp^{w_k^T x}}{\sum \exp^{w_i^T x}} \tag{7.7}$$

The chosen class will be the one with a higher probability.

In the context of a multiclass classification, one choice is to approach the problem with a strategy between the One-vs-Rest (OvR) and the One-vs-One (OvO). These are two heuristic methods to apply binary classification for multiclass classification:

1. *One-vs-One*:

   the original dataset is split in a binary one for each class versus every other class. This means more datasets and models, precisely:

   $$N = \frac{|Y| * (|Y| - 1)}{2} \tag{7.8}$$

   where:

   Y = set of classes

   N = number of datasets

   This approach is suggested for SVM because the performance does not scale, looking to the dataset size.

2. *One-vs-Rest*:

   the difference between the previous strategy is that applying the divide-et-impera technique, the dataset will be split into multiple binary classification problems, one for each class.

   This approach is broadly used with algorithms that predict probabilities, just like Logistic Regression. This is why I pursued this choice.

However, neither this solution was suitable for the problem, achieving the worst accuracy score out of the applied methods until that moment: 39.5% accuracy, showing at the same time awful outcomes in the test set even with precision (28%), recall (37%), and harmonic mean (29%).

Performance summary

Here's a summary of what are the achieved results of the three classical algorithms.

Clearly, what the Sklearn developers suggested 7.1 was right as the first selected choice is the one that achieved the best results, even if not satisfying.

	Linear SVC	**Naïve Bayes**	**Logistic Regression**
Accuracy	58,3%	50,82%	39,5%
Other metrics			precision: 28% recall: 37% F1: 29%

7.1.4 Ensemble Learning

When we talk about Ensemble Learning we mean a series of methods that are targeted to use multiple models to gain a better predictive performance as compared to each single model.

In more human terms, it's like comparing and combining the thoughts from different people that caption an event from different perspectives to obtain a more realistic view of what happened. This is due to the risk that a single person cannot consider the whole environment in which something happened and omits some details.

Three are the main used techniques:

1. *bagging*:

   It aims to create a set of classifiers having the same importance. At classification time, each model will vote on the outcome of the prediction, and the overall output will be the class that has received the most votes.

2. *boosting*:

   Each classifier affects the final grade with a certain weight. It will be calculated based on the accuracy error that each model will commit in the learning phase.

3. *stacking*:

   a further classifier is introduced (called meta-classifier), which uses the predictions of other sub-models to carry out further learning.

One of the most powerful libraries in the ML field is `XGBoost` that, as the name suggests, pursues the boosting technique.

It has different advantages:

- *regularization*:

  this algorithm is known to be a regularized boosting technique.

- *parallel processing*:

  it implements this kind of processing that makes it really fast.

- *high flexibility*:

  it allows to customize optimization objectives and criteria.

- *handling missing values*:

  the algorithm tries different approaches when encountering a missing value and learn how to deal with them in the next predictions.

- *tree pruning*:

  when defining a maximum depth of the decision trees, the algorithm starts to reduce the tree backward just when reaching that level and removes the division when it is not worth it anymore.

- *built-in cross-validation*:

  it has that technique as predefined at each iteration.

To demonstrate this claim, it has been used in countless winning solutions in competitions on Kaggle [31], a famous platform where a lot of challenges are planned to improve the skills of any data scientist.

Because of these premises, it is right to explore this approach as well before trying something mathematically more challenging and complex.

Model development

The choice to use a `XGBClassifier` let me choose a great variety of parameters to obtain the best possible results.

While developing it, I used one more time the grid search approach to fine-tune each of the possible parameters of the resulting decision tree:

- *learning_rate*
- *max_depth*: it states the maximum depth of the resulting tree
- *min_child_weight*: the minimum sum of weights of all observations required in a child
- *gamma*: it represents the minimum loss reduction required to do a split
- *colsample_bytree*: the fraction of columns to be randomly sampled for each tree
- *objective*: the required classification. In my case, it always had the values "multi: softmax", indicating the kind of classification and its evaluating function
- *nthread*: used for parallel processing, defying the number of system's cores to exploit
- *scale_pos_weight*: : to help fasten convergence

Results

This algorithm achieved the best performances speaking of classical ML algorithms and this constituted a real improvement, even if it had not been the definitive, since metrics did not pass the imposed thresholds to consider an excellent classifier to my task.

Accuracy	**Precision**	**Recall**	**F1-Score**
71,5%	72,0%	75,1%	73,0%

The confusion matrix in figure 7.3 gives an additive hint of the metrics' values:

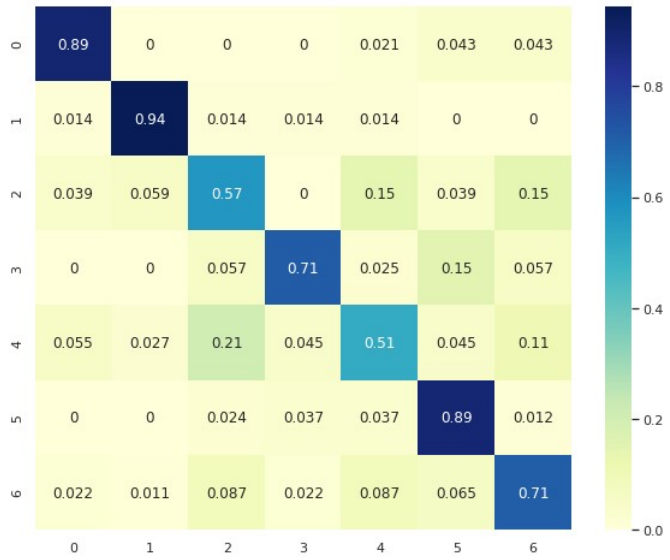


Figure 7.3: The confusion matrix of the XGBoost classifier

The 21% of error when misunderstanding the class 4 with the 2 and the multiple 15% errors when classifying the class 2 made me think there still were high and not acceptable misclassifications. It was prohibitive to consider it as the definitive solution.

7.1.5 CNN

Knowing some researchers demonstrated CNNs in some NLP tasks could perform well, I tried to pursue that route. I firstly merged together the columns "Subject" and "Description" (see section 4.1 to deepen into the characteristics of the dataset) in one more exhaustive column.

Since most DL algorithms are number-based, I then used a tokenizer that converts words to numbers to create a words' index. To do so, I sequentially cleaned up the text, removing punctuation and lowercasing any word. The input entries were then padded because when dealing with NN in NLP, the sequences need to be of the same size. The sequence dimension is based on a measure of the average length of the message tickets and the distribution of the number of words.

The same was performed on the labels, with the extra to turn list of labels in numpy arrays since it's the expected format from the algorithm. The number of labels was vast: `88` different subtypes were presented in the dataset.

Model structure

The complexity of the task came out when developing the requested model. I relied on the Sequential API, since the concepts of input and output objects were the same as the model whose development is expressed in the previous chapter.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 150, 75)           375000
_________________________________________________________________
spatial_dropout1d (SpatialDr (None, 150, 75)           0
_________________________________________________________________
lstm (LSTM)                  (None, 75)                45300
_________________________________________________________________
dense (Dense)                (None, 88)                6688
=================================================================
Total params: 426,988
Trainable params: 426,988
Non-trainable params: 0
_________________________________________________________________
```

Figure 7.4: The structure of the realized model

As shown in figure 7.4, I used an embedding layer that takes in input an integer matrix of size `(batch_size, input_length)`. The output is a 3D tensor with the shape `(batch_size, input_length, output_dimention)`.

The role of this layer is to store one vector per word. Then, when called, the sequences of word indices are converted into sequences of vectors.

A LSTM layer was then applied. This operation would have been necessary to learn better from long-term dependencies.

The next step was to stack over a Dense layer of 88 neurons, one for each of the possible outputs, selecting **softmax** as the activation function; this function's choice is soon explained. The output class of the model is modeled as a probability distribution. When dealing with a multiclass problem, the output conditioned distribution appears to be a multinomial distribution.

When in the last layer of the NN, the model has computed the score $s_k(x)$ of every class for an instance x. This function then computes the probability $\hat{p}_k$ that the instance belongs to a class k as expressed in formula 7.9.

$$\hat{p}_k = \sigma(s(x))_k = \frac{\exp(s_k(x))}{\sum_{j=1}^{K} \exp(s_j(x))} \tag{7.9}$$

where:

K = number of classes

$s(x)$ = vector with the scores of each class regarding the instance x

$\sigma(s(x))_k$ = estimated probability of x belonging to a class k when given each class' scores for that instance x

The exponential of every score is normalized dividing by the sum of all the exponentials. These outputs are often called *logits*.

As the last operation, every logit that is not the one with the highest value is rounded to 0, while the maximum is rounded to 1 instead, resulting in the value of the performed prediction.

Between the LSTM and the Dense layer, I added a spatial dropout layer. Dropout [32] is a popular technique to help prevent overfitting in a neural network: it randomly "turns off" with a specific probability a set of neurons each time a weight update is going to be processed. The objective is thus to realize a network less sensitive to the specific weights of neurons. When adding this type of technique, I considered two tips that came out from the previously mentioned paper [32]:

1. to set a probability value around 20% to have a tangible effect
2. enlarge the network to have more chances to learn independent representations.

To help in finding better results, I applied a grid search over numerous parameters: the batch size, the number of epochs, the optimizer, the learning rate, the momentum (a variant of the stochastic gradient descent, specifically a coefficient that is applied in the weights update), and the activation function.

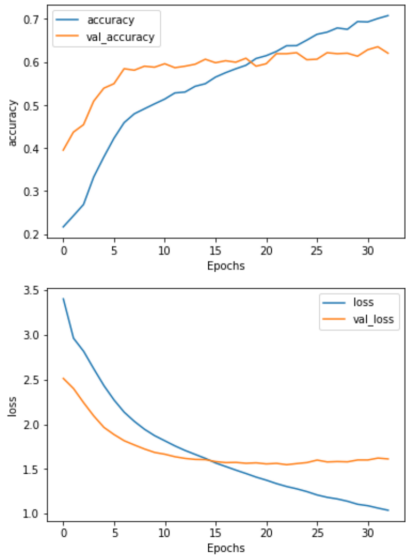
Callbacks

- *Early stopping*: it was applied to monitor the log loss on the validation set when there's not an improvement to prevent excessive overfitting
- *Learning rate scheduler*: it was used to slow down the learning rate since I initially found out the model was overfitting too early
- *Reduce Learning Rate On Plateau*: sometimes this callback turns out to be useful when a plateau is reached, considering a certain metric; in my case, it was the validation loss

Anyway, these techniques were not enough.

Results

The considered baseline was constituted by a random guesser: it would have had the chance of guessing the right label around of 1%.



The improvements as compared to that value are massive but not satisfying.
I used as the input data both cleaned and dirty tickets, where "dirty" means not processed with the text anonymization neural network.
The maximum result I achieved was 60% accuracy and a loss of 1.5 in the validation set.
The graphics in figure aside (7.1.5) show that.

7.2 An approach change

7.2.1 Introduction

After facing the previously mentioned troubles, a collation was seemed necessary. In the crucial meeting at the beginning of May, I expressed my perplexity in achieving the wanted results by looking at the data I had to deal with.

We have concluded that, in the face of only 30.811 samples divided into five languages, it is not possible to create a classifier capable of achieving satisfactory levels of precision.

Collecting further data reports has been identified to obtain greater accuracy in the model without running into the problem of overfitting caused by the complexity of the model itself and an eventual addition of artificial samples for the minor classes.

Since getting further data is unfortunately not immediate, it was decided to restart by analyzing just a subset of the already given labels.

Having obtained in the first task the most satisfactory results in terms of accuracy and execution time using a DL model based on neural networks, I decided to start from this approach in such a way as to improve and refine this model over time when new subtypes are inserted to be classified. I preferred this solution instead of using a more traditional ML algorithm with the risk of not being able to use it anymore when the amount of labels increases. This should therefore be beneficial in terms of

- time
- effectiveness of the solution
- modularity

Data analysis

From an analysis of the balancing of the dataset; it was decided to proceed with the following classification alternatives:

- since the Types column has fewer possible classes than the Subtypes one, a workable solution could have been to consider it as the target

- to reduce the number of deemed subtype classes, imposing a specific threshold (i.e. 1000, then 750 and finally 500) of samples per subtype.

The first undertaken route was the first listed.

7.2.2 Classification by Type

Data loading

Having the classes labeled in the format "Letter-Number" was just necessary to extract the letter.

Using the Italian dataset, I decided to divide it into the following percentages:

- 90% training set - 23.115 entries
- 5% validation set - 1284 entries
- 5% test set - 1284 entries

The division by type resulted in being as described in figure 7.5:

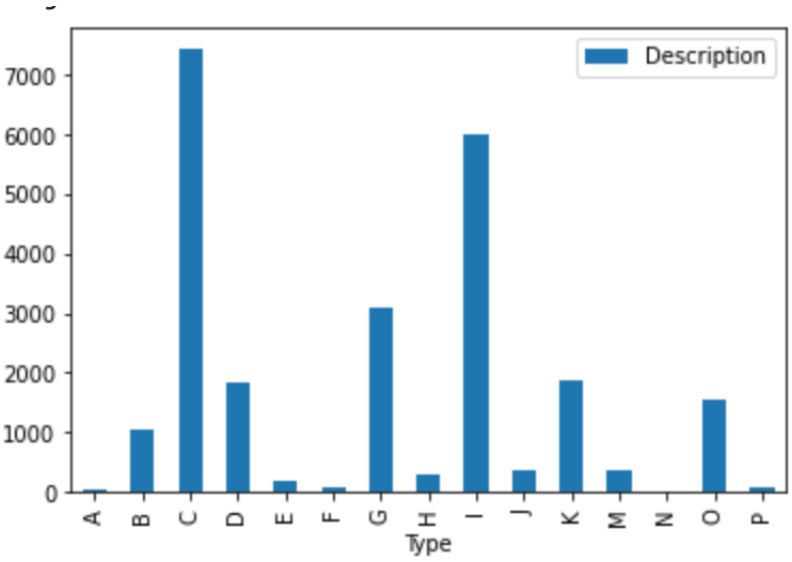


Figure 7.5: The balance of the dataset grouped by Type

The number of classes decreased from `88` to `15`. However, the unbalancing problem persisted, so I decided to cut off some categories. Looking at the data, it's evident there's a great gap between classes like B, with 1082 samples, and J, with just 387 samples. The configured threshold was 750 samples, resulting in a dataset as follows in figure 7.6:

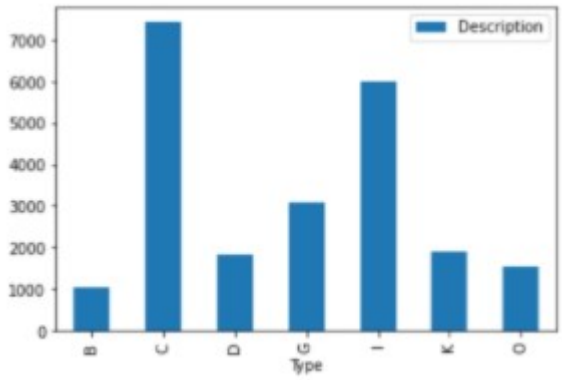


Figure 7.6: The remaining classes with a number of sample greater than 750

This decision sensibly reduces any possible reason of bad performance due to the unbalancing of the dataset, having enough samples for each remaining class.

Data preprocessing

As a preprocessing task, I merged the "Subject" and "Description" columns from the dataset to have just a single input text.

I then used a pair of regular expressions to substitute any bad symbol like numbers, and letters followed by one or more *underscores* and replace any symbol like parenthesis, brackets, ats, and punctuation marks with a blank space.

To improve this process, I then removed any stopword and any possible Italian greeting. I then tokenized any input text, padded it, and transformed it to form a new tensor of sequences of shape `(number_of_samples,100)`, where 100 is the maximum number of words permitted in each sample.

The pad action aims to insert, if needed, to the left any 0s to achieve a length of 100 numbers, each one representing a word in the word index. This is just to have a nicer input to the model I was going to develop.

Model development

As a substantial and meaningful baseline, I decided to use the **Zero Rule** better than a random baseline.

In classification problems, it aims to use the most present class in the dataset to predict each entry. In my case, the class with the most number of entries was `C`, and subsequently, the value of the Zero Rule is 33,44% precision. That

undoubtedly raises the minimum level of acceptance since a random guesser would have had the 14,29% precision.

The TF Callbacks could be extremely useful. Hence, I set an early stopper, a learning rate reducer on a plateau, and a learning rate scheduler, just like I did in the initial developments (see section 7.1.5). The idea was to try each of them and see what combination would be better, but after achieving decent results with metrics such as accuracy and log loss.

Next, I began to develop the model with two different approaches:

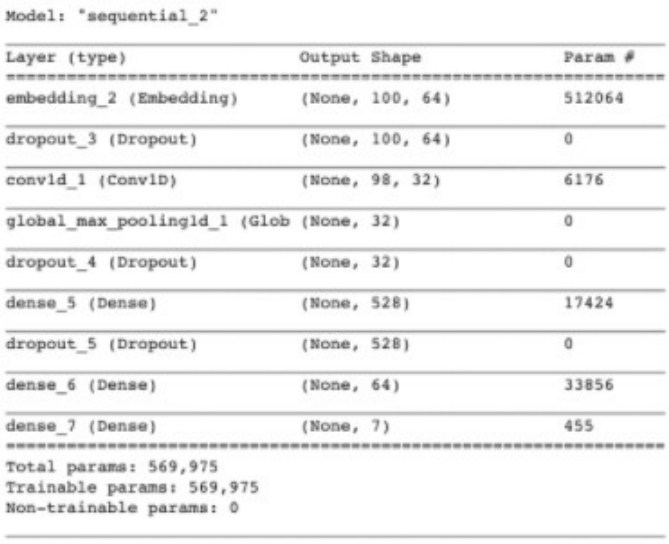
1. *CNN*

   Figure 7.7 represents the resulting structure.
   The initial embedding [33] has a shape of `(None, 100, 64)`, representing respectively the batch dimension (not restricted to any size), the input size (a sequence of 100 numbers), and the embedding dimension (maximum 64). This means the model will have as input a matrix of integers whose size is `(batch, input_length)`. Moreover, the largest input's integer should be no greater than the vocabulary size, which in my case is 8000 words, the most frequent ones. The latter value was chosen based on the number of unique tokens found: 17902. Maintaining about half of the values, it's a good approximation.
   Any couple of hidden layers was interspersed by a dense layer, with the scope of preventing overfitting.

   The next is a 1D convolution layer [34] that has the role of creating a convolution over a single dimension: the input layer is convolved with a convolution kernel to produce the output to pass to the pooling layer.
   The padding argument is one of the most important when defining such a layer. It represents the way output data should be processed in the borders. Since I didn't want to introduce bias, I defined it with the value "valid," which means no padding.

   The following is a Global Maximum Pooling over 1 dimension [35]: it aims to reduce the dimension of the input object by selecting the maximum value over the time dimension. Often it is used to level up performances.

   The remaining layers are some Dense [36] of different shapes: the most relevant is the final, which has 7 output neurons as the classes' number.



```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 100, 64)           512064
_________________________________________________________________
dropout_3 (Dropout)          (None, 100, 64)           0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 98, 32)            6176
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 32)                0
_________________________________________________________________
dropout_4 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_5 (Dense)              (None, 528)               17424
_________________________________________________________________
dropout_5 (Dropout)          (None, 528)               0
_________________________________________________________________
dense_6 (Dense)              (None, 64)                33856
_________________________________________________________________
dense_7 (Dense)              (None, 7)                 455
=================================================================
Total params: 569,975
Trainable params: 569,975
Non-trainable params: 0
_________________________________________________________________
```

Figure 7.7: The structure of the model based on the use of a convolutional layer

2. *RNN*

   Figure 7.8 represents the alternative: a model made up to exploit the RNN potential. It has the same embedding layer as input, followed by a Spatial Dropout layer over 1 dimension [37]: it's a particular type of dropout. It takes as input a 3D tensor, but instead of dropping just a single value, it considers a whole 1D feature map. This way, it can push independence between this kind of data. This has, as a consequence, the effective regularization when dealing with neighbor data strictly correlated.

   However, the most important layer is the LSTM [38]: this is the mainstream layer to have an RNN (see section 3.3 to go deeper with what's an RNN). The only positional argument I had to set is the units, which tells us how big the output space will be. I set it to 100 neurons and applied a dropout of 0.2 to prevent some neurons from being influenced by the linear transformation inside the layer.

   The rest are a pair of Dense layers of different shapes, with the last one having 7 neurons just like the number of classes.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 100, 64)           512064
_________________________________________________________________
spatial_dropout1d (SpatialDr (None, 100, 64)           0
_________________________________________________________________
lstm (LSTM)                  (None, 100)               66000
_________________________________________________________________
dense_3 (Dense)              (None, 128)               12928
_________________________________________________________________
dense_4 (Dense)              (None, 7)                 903
=================================================================
Total params: 591,895
Trainable params: 591,895
Non-trainable params: 0
_________________________________________________________________
```

Figure 7.8: The structure of the model based on the use of a recurrent layer

Results

Both models led to encouraging and similar outcomes, achieving respectively 78% and 77,6% accuracy and a log loss of 0.790 and 0.780.
Then a doubt came to my mind: "how would both models perform with original data?" With "original data", I mean data with entailment, greetings, and repetitive sentences. The experiment resulted in a particular outcome: the accuracy is a little lower, but the same happened with the log loss: the next table shows the differences in the model with RNN logic.

	Cleaned	**Original**
Accuracy	77,6%	76,3%
Loss	0,780	0,663

These results hinted at the origin of the problem: the performances are influenced by the data dirtiness but less than I would have expected.
Being an initial test, I couldn't be sure about the previous statement, so I proceeded with the classification by subtype to discover if the way I was grouping in classes had any responsibility.

7.2.3 Classification by Subtype

Having the same amount of samples, I maintained the division in train, validation, and test set as exposed in subsection 7.2.2.

The figure 7.9 represents the classes obtained by filtering out the ones with less than 750 samples, resulting in 11 different classes with `18.680` samples.

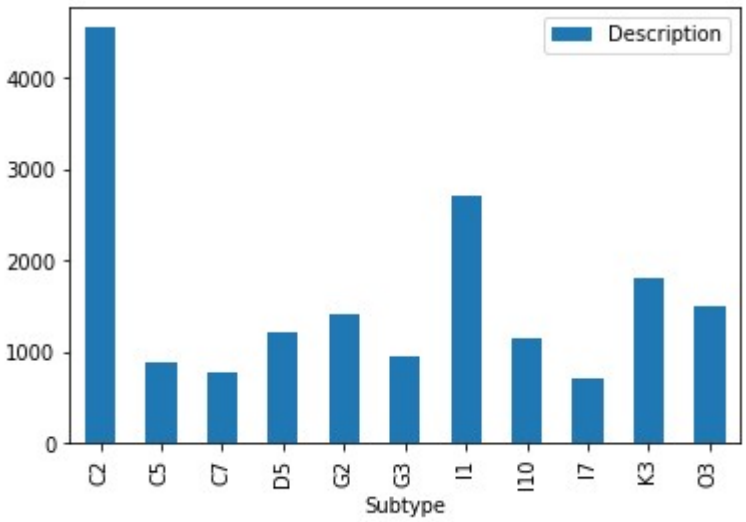


Figure 7.9: The dataset composition after filtering out classes with less than 750 samples

I immediately tried the subsample technique to have a more balanced distribution, resulting in the next dataset of `13.601` samples, as figure 7.10 shows:

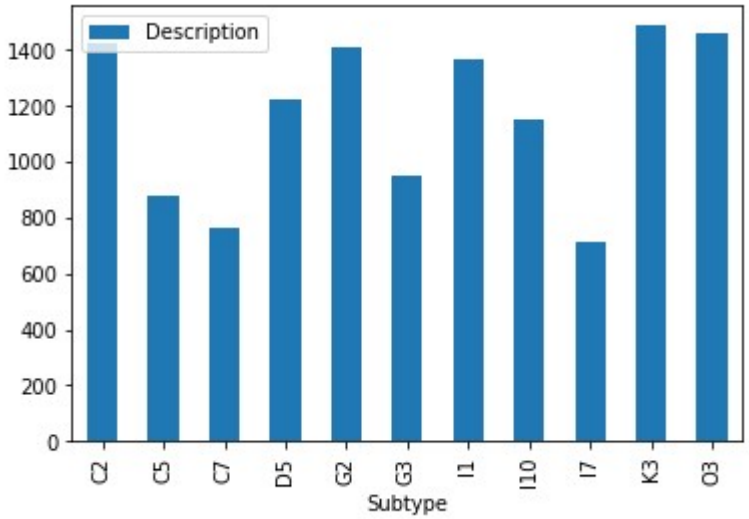


Figure 7.10: The dataset composition after subsampling

As the last operation over the data, I shuffled the dataset to have a better chance to have each class in every subset of the dataset itself.

Data preprocessing

This phase had been realized exactly as explained in the homonymous subsection in 7.2.2 when I was dealing with Types classes.

Model development

I maintained the Zero Rule as the baseline for my model, this time with a value of 26,6% of accuracy.

Keeping things simple, I firstly used a quite simple model to see if it could generalize well. After an initial Embedding layer, I stacked a GlobalAverage-Pooling layer over 1 dimension [39]: it acts like the Global Maximum Pooling layer but instead of taking the maximum value, it considers the average of all the given numbers. After that, it was time to introduce again a pair of Dense layers, this time using a stricter regularizer: the L1. It takes its name from the norm that perform:

$$||w||_1 = |w_1| + |w_2| + ... + |w_n| \tag{7.10}$$

Any regularizer influences the loss function that will be changed with an additive member as follows:

$$L_1 = (wx + b - y)^2 + \lambda|w| \tag{7.11}$$

where the regularisation parameter $\lambda > 0$ is manually tuned.

`L1` regularizer, really like `L2`, is useful: the term λ, when updating the weights of the model, has the role of changing the weights in a manner to reduce the chances of overfitting because it shifts away from the original weight values.

Between these two layers, there was a Dropout one to help prevent overfitting too. All of this just before the last Dense layer with 11 neurons, one for each class.

However, I did not have a particular fortune, as summarized in table 7.1. This made me rethink the CNN model previously used when classifying by type. Even this time, I could not pass the threshold of 80% accuracy; it was the target value for a good initial model.

At that point, I thought to increase the complexity of the model by trying to use a layer I had never used before: the Bidirectional [40]. It is a wrapper for RNNs that combines the output of an RNN layer such as the previously used

LSTM or the GRU [41]. It propagates the input forward and backward through the LSTM layer and finally concatenates the output. When dealing with long sequences, it could help since Italian is a language where it's easy to formulate long sentences and thus harder to interpret.

This model resulted in a better performance than the two previously mentioned experiments but not as much to feel satisfied with what has been done.

The next table shows the obtained results.

	Simple model	**CNN**	**Bidirectional RNN**
Accuracy	63,7%	73,2%	78,3%
Loss	1,083	0,53	0,747

Table 7.1: Results classifying by Type

7.3 The turning point

At some point I was perplexed. The input did not seem to be that difficult to generalize, even though there was a discrete number of different classes.

With the previously obtained results, I understood that the main problem was not either in the classification by Type or Subtype, nor in the dirtiness of the raw data. Surely one approach led me to better results but not as intended.

I then tried to examine even deeper the initial dataset. At that moment, I realized what the problem was.

7.3.1 The crucial metric

While realizing this model, I understood the importance of considering the right metric to examine the produced outcomes.

I did get what was the underlying problem just when I analyzed the **confusion matrix**. I first translated the acronym nomenclature to a more human-understandable one. Then I used that matrix when classifying by type, and what's reported in figure 7.11 is what I discovered:

It is clear that most of the errors are conveyed to the cells "Other" X "Product information and availability". This means the model misunderstands when it's time to classify with these two classes. I'm considering this example because it's the most relevant.

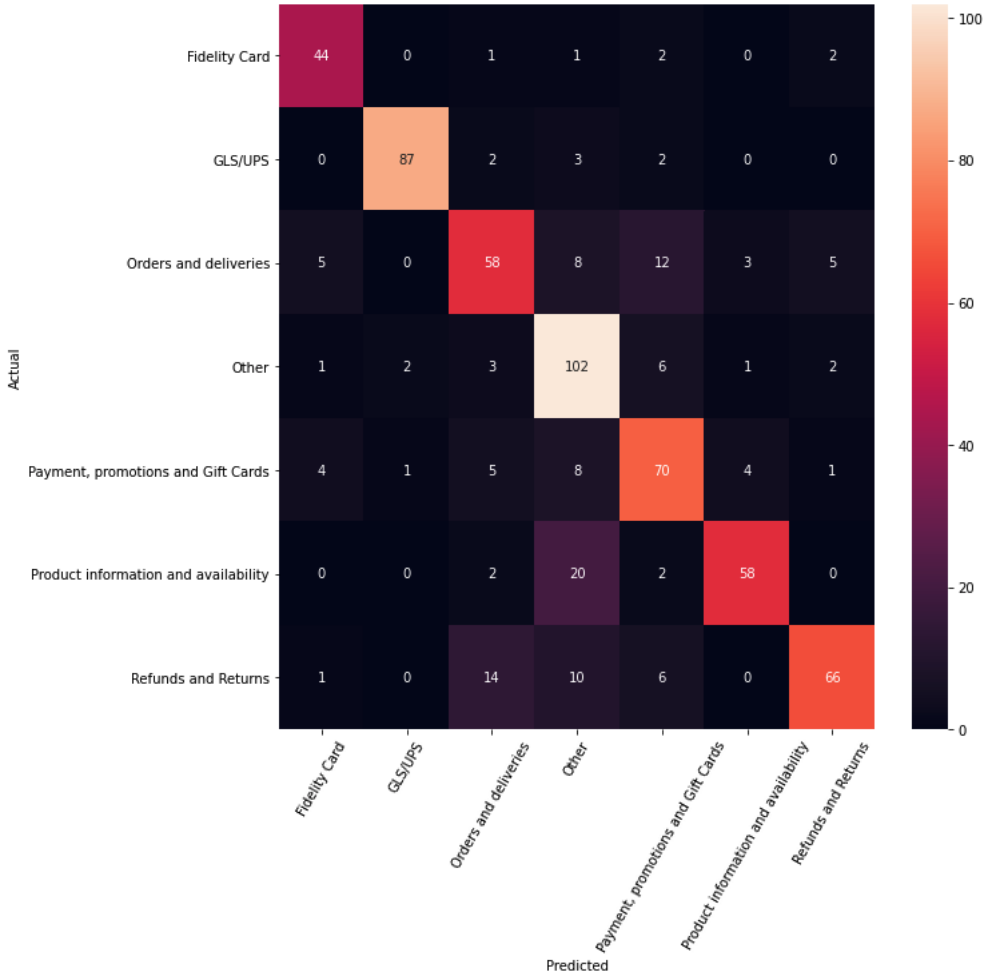


Figure 7.11: The initial confusion matrix

The problem is inherent to the nature of the data. The Description column with the value "Live Chat" appears many times in both classifications (78% in Other and the remaining 22% in Product information and availability).

A more in-depth examination reveals that there are 451 cases classified with "Other" (27% of the cases detected before) and 372 (80% of the data considered before) classified as "Product information and availability" that have the same data in "Subject" and "Description", which are the only columns I can use as input data. The initial human classification metrics differed according to the person who made these associations (see figure 7.3.1).

	Subject	Description
5893	Product information and availability	Live Chat
5570	Product information and availability	Live Chat
2483	Product information and availability	Live Chat
7406	Product information and availability	Live Chat
10818	Product information and availability	Live Chat
...	...	...
3224	Product information and availability	Live Chat
19335	Product information and availability	Live Chat
2110	Product information and availability	Live Chat
25438	Product information and availability	Live Chat
8505	Product information and availability	Live Chat

451 rows × 2 columns

	Subject	Description
6210	Product information and availability	Live Chat
10780	Product information and availability	Live Chat
7162	Product information and availability	Live Chat
5774	Product information and availability	Live Chat
13252	Product information and availability	Live Chat
...	...	...
13971	Product information and availability	Live Chat
5249	Product information and availability	Live Chat
10023	Product information and availability	Live Chat
1497	Product information and availability	Live Chat
9890	Product information and availability	Live Chat

372 rows × 2 columns

Figure 7.3.1: the samples involved in the misclassification of "Other" and "Product information and availability".

Noticing this kind of situation in other couples of classes, I used a little script to disambiguate the wrong classification. This way, texts like the previously mentioned "Live Chat" had been redirected to a single class.

As a positive consequence, this let me have a bigger set of classes to consider while classifying and have a more balanced situation without operating any downsampling, as shown in figure 7.12:

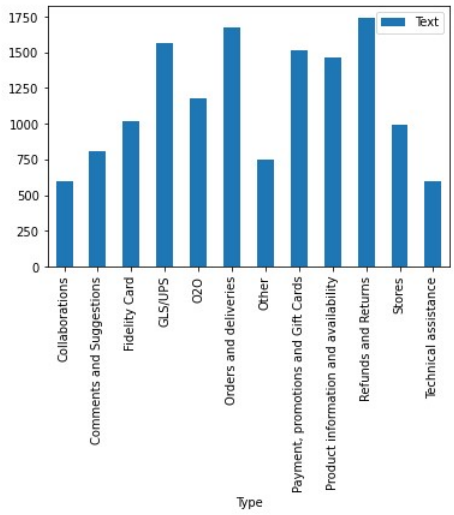


Figure 7.12: The italian dataset after removing any human mismatch

Passing from 7 to 12 passable classes represented a big step forward to a

complete classification.

Initial results

The effects of the data disambiguation led to a performance increase of the same model without changing the structure. I used the model that uses a Convolution layer.

Table 7.2 reports the initial metrics that were much better than what I had achieved so far.

Accuracy	Loss	Precision	Recall	F1
80,9%	0.662	79,5%	83,9%	81,6%

Table 7.2: initial results after removing any mismatch

7.3.2 Upsampling

After encouraging results, I thought I would be able to reach new better performances carrying out an upsampling technique.

This section explores the different techniques I tried:

1. *Translation*:

   Translating and re-translating sentences in the original language of the ticket is not a bad idea to obtain slightly different samples. At the same time it is very limiting because it consists of making requests to the Google Translate API, which are limited. Therefore it cannot be considered a definitive solution, being much less satisfactory.

2. *Markov Chains*:

   The Markov property implies a generation of tokens based just on the last one provided. The agent acting in this process, therefore, has no memory. In text modeling, the token is the word to produce. Thus the model does not know the context of the sentence and anything that concerns it, as when it will end or what the subject is. It is limited to predict the next word that has a high probability of appearing. This property is described with the following formula:

$$P(X_k|X_{k-1}, X_{k-2}, ..., X_{k-n}) = P(X_k|X_{k-1}) \tag{7.12}$$

To use this technique, it is necessary to have a clear input text from any special character such as punctuation, with a uniformed case. It is also useful to define the end of a sentence character.

Setting the minimum number of sentences for each class lets me have a more balanced dataset. This value was set to `1000`. It was not calculated but was so as not to have a high number of synthetic data since just 4 classes did not reach that limit initially (see figure 7.12).

Results

The model performed way better with an improvement on each metric:

Accuracy	Loss	Precision	Recall	F1
82,3%	0.586	81,6%	85,1%	83,3%

Table 7.3: Results applying the Markov chains

3. *Class Weights*:

   Even if the Markov Chains gave me pretty good performances, I tried with this alternative way to think about the problem. The Sklearn library lets me set a weight of importance for each class given the number of samples [42].

   The *'balanced'* option was what I was looking for since it is thought for unbalanced datasets, even if it's an estimation.

Results

The outcomes constituted an additive improvement looking at what I had done before with the Markov Chains, especially regarding accuracy, precision, and the F1-score.

The only worse metric was the loss, stating that there's more uncertainty whenever an assumption is made. However being a difference of just 0,041 it was not a big deal.

Accuracy	Loss	Precision	Recall	F1
84,0%	0.627	83,5%	85,1%	84,3%

Table 7.4: Results applying a different weight for each class

Even the confusion matrix reported encouraging upshots, as reported next.

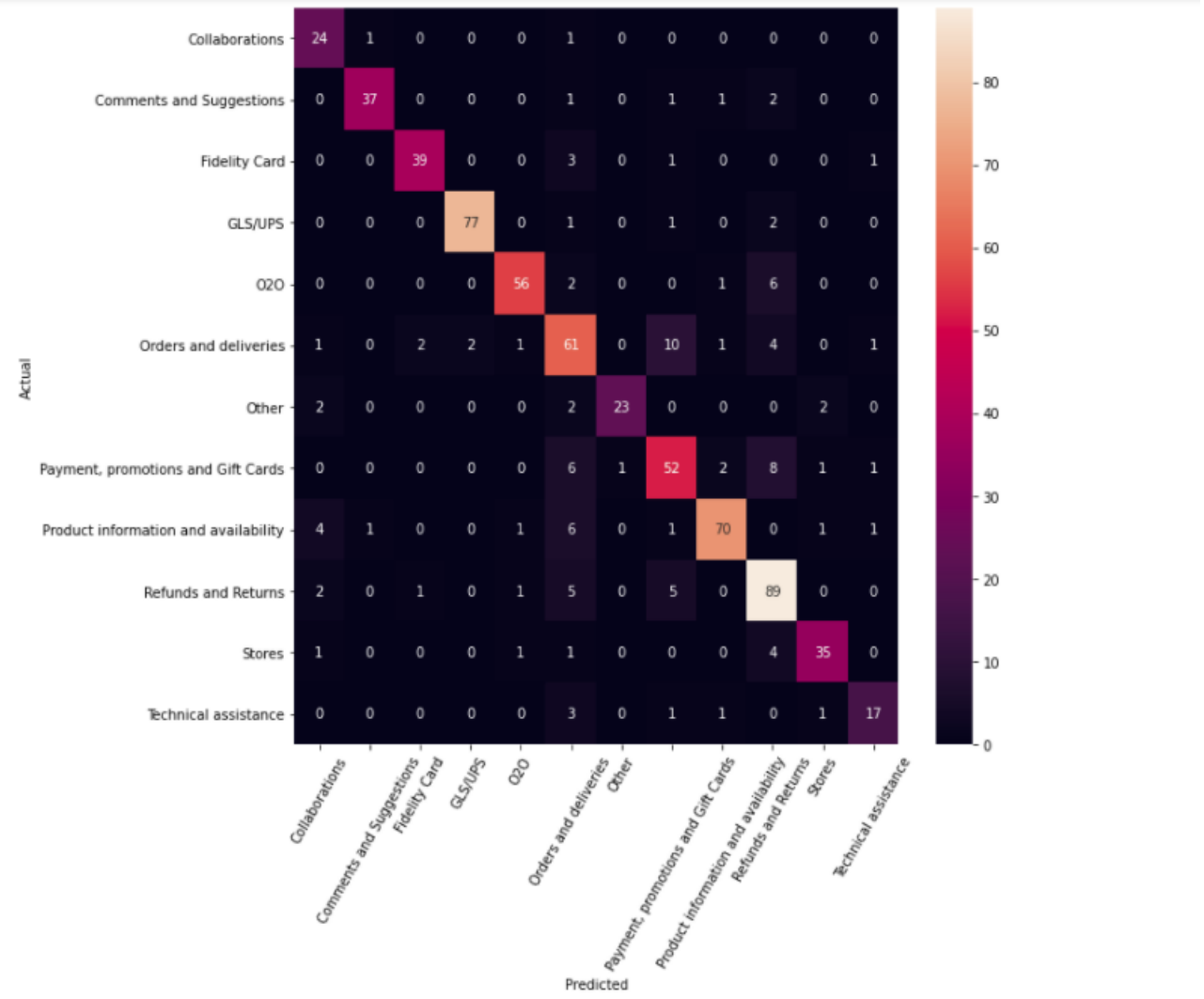


Figure 7.13: The confusion matrix after setting a specific class weight for each class

This option was the selected one to apply even to the Subtype classification.

7.4 The final solution

As the initial column target was the subtype and not the type, I chose to apply what I've previously done when dealing with Type classes. I took a cue from the model structure and experiments to have a confirmation of the obtained results.

7.4.1 Data loading

I still considered the Italian dataset with 25.683 initial entries and filtered out the classes setting a threshold of samples down to 500.

The resulted dataset had 20.525 entries with the following division:

- 18.472 entries to the training set (90%)
- 1026 entries to the validation set (5%)
- 1027 entries to the test set (5%)

To have a more balanced initial situation I performed a little downsampling over the classes C2 (Order Information) and I1 (Return procedure authorization) that had much more samples than the others. I translated the classes' names to a more human-readable nomenclature, merged with the columns "Subject" and "Description" in a more generic "Text", and applied the disambiguation script to throw away any possible controversial human classification.

The resulting dataset is the one represented in figure 7.14.

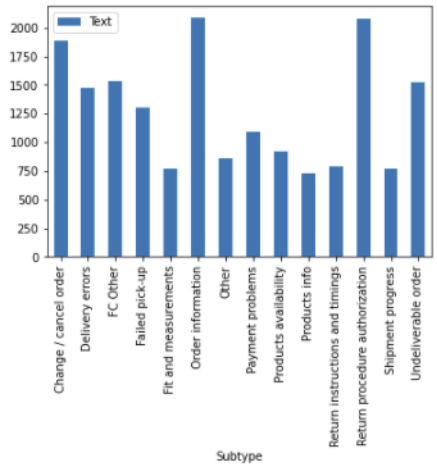


Figure 7.14: The final classes' distribution

7.4.2 Data preprocessing

The procedure was analogous to what has been performed in the classification by Type (subsection 7.2.2).

7.4.3 Model development

The choice to have a different weight for each class was applied to this solution too.

In addition, the three mentioned before callbacks were added:

- the learning rate scheduler
- the learning rate on plateau reducer
- the early stopper

The number of considered classes increased from 12 up to 14. Therefore, when dealing with the structure, I made a few changes to the Type classification model to help it to generalize more.

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 100, 64)	512064
dropout_3 (Dropout)	(None, 100, 64)	0
conv1d (Conv1D)	(None, 98, 32)	6176
global_max_pooling1d_1 (Glob	(None, 32)	0
dropout_4 (Dropout)	(None, 32)	0
dense_3 (Dense)	(None, 528)	17424
dropout_5 (Dropout)	(None, 528)	0
dense_4 (Dense)	(None, 64)	33856
dense_5 (Dense)	(None, 14)	910

Total params: 570,430
Trainable params: 570,430
Non-trainable params: 0

Figure 7.15: The final model structure

As figure 7.15 represents, right after an initial embedding layer, I stacked many dropout layers throughout the structure with a 30% probability of turning off some neurons when updating the weights.

Then a convolutional layer over 1 dimension helped me find relationships between neighboring words with "valid" padding and a stride of 1. This last value

indicates how many positions at each step the filter is shifted when computing the convolution.

The Global Maximum Pooling layer over 1 dimension was applied to reduce the input and level up performances.

Then three Dense layers were added to maintain the discovered features and find some remaining insights. Except for the last Dense layer that has a softmax as the activation function, all of the other layers use the ReLU function.

The **Rectified Linear Unit** function works pretty well with lots of tasks, and it's fast to compute. It results in the function 7.13

$$ReLU = max(0, x) \tag{7.13}$$

Unfortunately it is not differentiable when the input has value 0. Another characteristic is that its derivative is 0 for $x < 0$.

It is important to have this kind of function because it constitutes the way the net learns. Not having a non-linearity between layers means that stacking even a big number of layers is always equivalent to a single layer. Therefore, complex problems couldn't be solved.

Figure 7.16 represents the ReLU and its derivative.

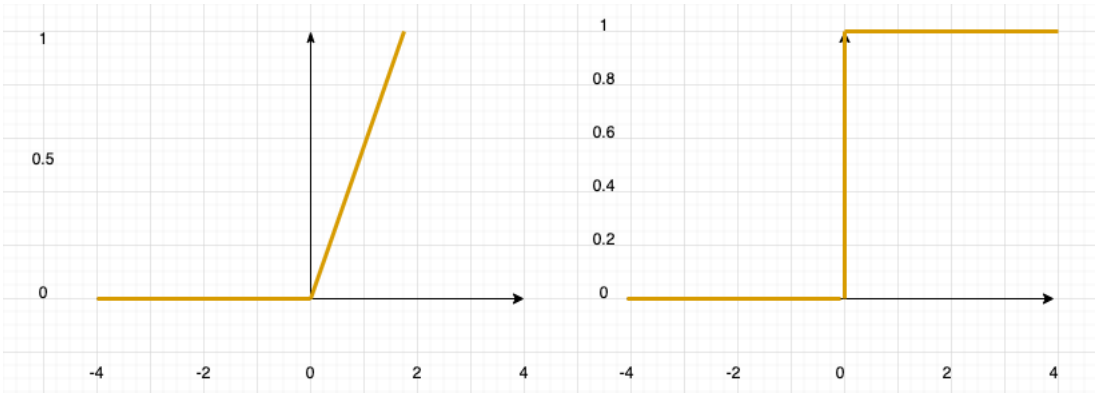


Figure 7.16: The ReLU function and its derivative

The loss function is an important aspect that deviates from the binary classification solution explained in the previous chapter. Instead of a binary cross-entropy, I selected a more suitable **categorical cross-entropy** function. I used it because I had sparse labels since there was just a target class index for each instance.

This function computes the loss through this formula:

$$Loss = -\sum_{i=1}^{O} y_i * \log \hat{y}_i \tag{7.14}$$

where

$\hat{y}_i$ = the i-th value in the model output

y_i = the target value

O = the output size; the amount of scalar values in the model output

It represents a measure of how distinguishable two discrete *probability distributions* are from each other. Specifically, y_i is the probability that the event i occurs. The sum of all the events is equal to 1, meaning just one event occurs at a time.

The minus sign instead has the role in reducing the loss when distributions are more similar and thus closer to each other.

This mathematical explanation made it the best loss function I could use in my case.

7.4.4 Results

When training it with a batch size of 32 samples and 50 epochs, I got better, and mostly acceptable, results that are exposed in the next table:

Accuracy	**Loss**	**Precision**	**Recall**	**F1**
80,8%	0.655	77,2%	81,0%	79,1%

Table 7.5: Final performances classifying by `Subtype`

Even the confusion matrix shown good insights of how the model was performing (figure 7.17).

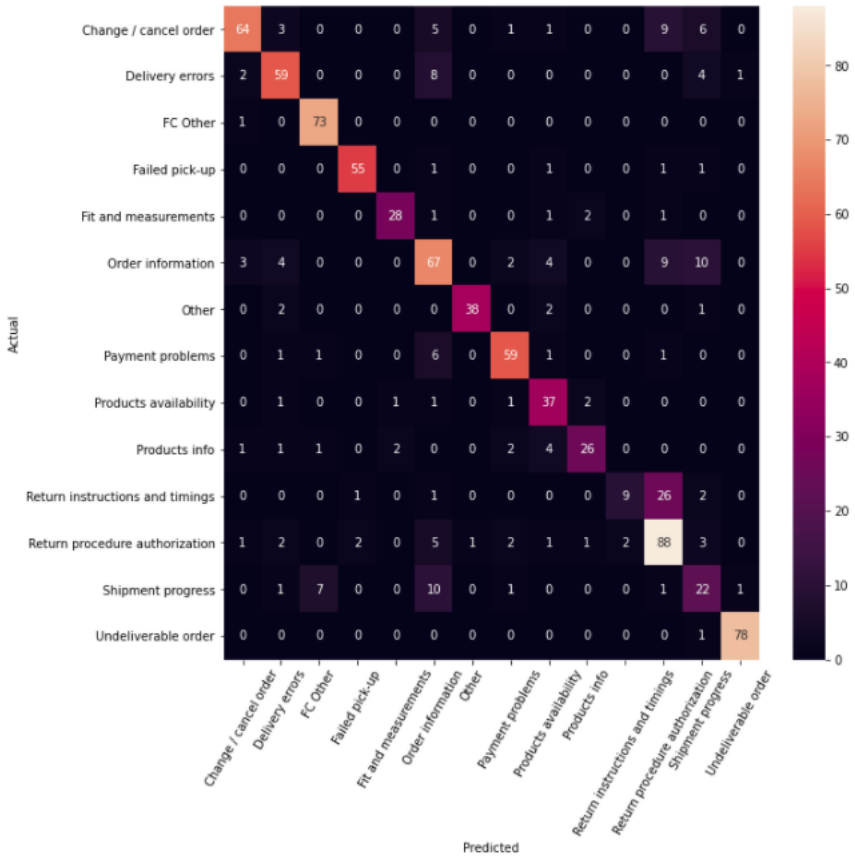


Figure 7.17: Final confusion matrix representation

A relevant data is the 26 that represents misclassifications between the classes "Return instructions and timings" and "Return procedure authorization". It's something I was prepared to face since it's strictly correlated between two Subtype classes belonging to the same Type.

The following image 7.18 show that some words' sequences are present a lot in both classifications. I took an attentive look at the sentences containing the sequence "Resi e rimborsi" but there were many more in common.

Since this kind of sequence is assiduous (looking at this very example respectively at 35% and 21% of the samples), it would have been possible to evaluate their deletion.

After a consultation with the company's data scientist, it seemed more appropriate to deal with this kind of problem in a manner not to distort this data's nature, as explained in the chapter Future Developments.

	Case Number	Subtype	Text
15559	436078	Return procedure authorization	Resi e rimborsi Buonasera, non ho ricevuto il ...
1092	415332	Return procedure authorization	Resi e rimborsi Rendo 1 x Abito con borchie Tg...
20721	443895	Return procedure authorization	Resi e rimborsi Chiedo il reso del dolcevita p...
13340	432996	Return procedure authorization	Resi e rimborsi Ho bisogno di aiuto,non riesco...
11684	422551	Return procedure authorization	Resi e rimborsi Avrei bisogno di una nuova eti...
...	...	...	...
14601	434771	Return procedure authorization	Resi e rimborsi buongiorno vorrei fare il reso...
1137	423679	Return procedure authorization	Resi e rimborsi Buongiorno, relativamente a qu...
12246	431511	Return procedure authorization	Resi e rimborsi Chiedo cortesemente di poter s...
14876	435166	Return procedure authorization	Resi e rimborsi Motivo: non mi piace il colore...
21622	445267	Return procedure authorization	Resi e rimborsi Prodotto 33640990040855 Spediz...

733 rows × 3 columns

	Case Number	Subtype	Text
3605	425356	Return instructions and timings	Resi e rimborsi Cosa devo fare per rendere il ...
18888	441331	Return instructions and timings	Resi e rimborsi Salve, posso rendere prodotti ...
19536	442235	Return instructions and timings	Resi e rimborsi
1502	423941	Return instructions and timings	Resi e rimborsi Gentili signori ho già ricevut...
16437	437847	Return instructions and timings	Resi e rimborsi Buongiorno, dovrei fare un res...
...	...	...	...
19957	442826	Return instructions and timings	Resi e rimborsi Salve, con riferimento al reso...
7090	427856	Return instructions and timings	Resi e rimborsi Buongiorno vorrei sapere se il...
8818	420571	Return instructions and timings	Resi e rimborsi salve ho aperto la pratica di ...
15126	435536	Return instructions and timings	Resi e rimborsi Buongiorno, Non ho ancora potu...
20223	440430	Return instructions and timings	Resi e rimborsi Salve devo effettuare dei resi...

165 rows × 3 columns

Figure 7.18: Controversial classifications in "Return instructions and timings" and "Return procedure authorization"

7.4.5 Deployment

Once the model had been trained, tested, and saved locally, it was time to perform the deployment. It was first added to the new Docker microservice to maintain a pluggable nature with the OTML structure.

To deploy a ML model over MLflow it's essential to define a backend store and an artifact store. These are the components that are part of the Tracking API MLflow makes available. They are responsible for respectively persisting MLflow entities such as runs, parameters, and metrics and, as the name suggests, persisting artifacts like files, models, and images. For internal use, it's sufficient to use a local directory as the backend store and another one as the artifact.

But when deploying and serving the model, it's necessary to use it as the

backend one of the allowed databases. This is so to save the model in the appropriate MLflow registry. Since I was not constrained to use any special database, I took advantage of an instance of SQLite, one of the simplest libraries to implement a SQL DBMS.

This was possible running MLflow with the following CLI commands:

1. `export MLFLOW_TRACKING_URI=http://localhost:500`
2. `mlflow server --backend-store-uri sqlite:///mlflow.db --default-artifact-root ./artifacts/`

After tracking down the training phase of the model, it was possible to visualize it in the MLflow UI in the Artifact section as in figure 7.19

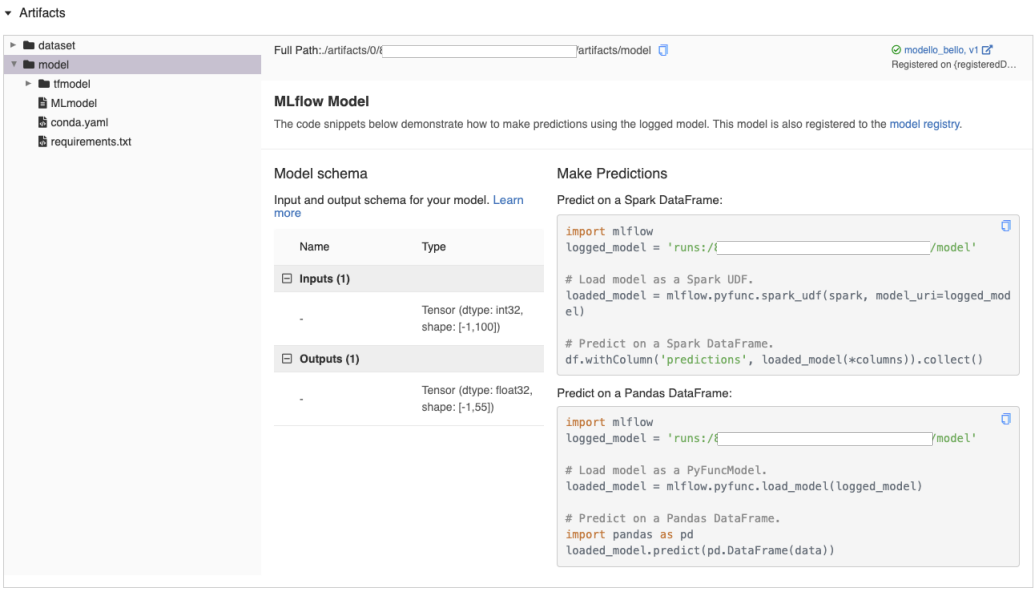


Figure 7.19: Artifact section in MLflow UI

This visualization is useful since it gives information about the stored resources on the left, the schema the model has to be constrained to, and how to make predictions using different dataset formats. Next, it is possible to register it on a specific deployment phase, as shown in figure 7.20:

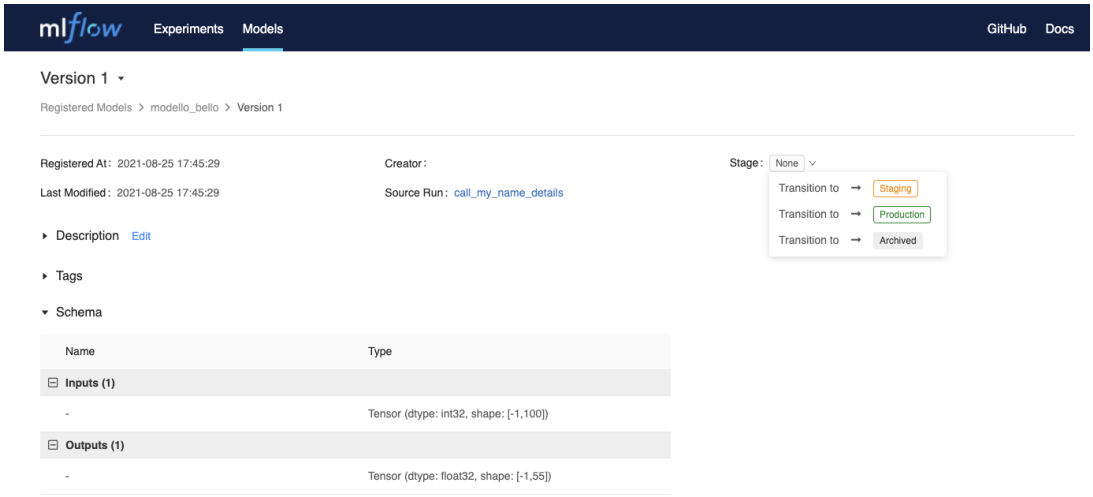


Figure 7.20: Models section in MLflow UI

After transiting the model to a specific area, it's possible to carry out the serving phase. I did it specifying the following command:

```
mlflow models serve -m "models:/model_name/Staging" -p 1234
```

It makes available on the tracking URI at the port `1234` the model called `model_name` to be trained and to use it to make predictions.

Chapter 8

Future developments

Every year, new technologies become available, and people want ever more efficient softwares to deal with.

The microservice has achieved good results but surely there are margins of improvement. The text classification has its limitations: it cannot say when a ticket is urgent and when it's not, and cannot properly classify messages coming from different languages.

That is why we thought about two important improvements.

1. **Multilingualism**

   As shown in section 3.3, there are already various multilingual models that have reached performances that just some years ago would have been literally unbelievable.

   It's then possible to implement this feature that will make it more worldwide than now. As a consequence, this service would be more desirable even to foreign business users.

2. **Class clustering**

   A technique with considerable potential is undoubtedly the GSOM [43], used even by Google to always obtain new labels in its Gmail service. Being able to group any classes that are too specific and very similar in terms of embedding can lead to an improvement in performance.

   Instead, using a clustering algorithm to discover new classes would allow to gain new knowledge from the data that is provided by the customer.

3. **Sentiment Analysis**

   Although most of the time the classification is correct, it does not assign any priority. The sentiment analysis right does so. It computes a real value that states whenever a user is angry, satisfied, stressed, doubtful and so on. The lower is the numeric value, the higher priority an issue gets.

   When dealing with SLA, it's crucial to fix and patch the biggest problems. Every company does not have unlimited time to dedicate to a specific project. The **Service-Level Agreements** indicate this limitation.

   When signing a contract with some business user, the SLAs even say the maximum resolution time expected. Every contract is signed considering the area in which it is related (for instance the e-commerce orders) and the priority.

   They have become a common tool for effectively measuring services. This involves the payment of penalties in case of failure to reach these levels. Thus it is better to first solve the issues with the highest priority.

   To define the latter one there are two philosophies:

   (a) if the user does not set any priority when submitting the request, then the sentiment analysis is useful to cover this lack of information

   (b) if the user sets the priority, often it is a blocker or urgent level. It's understandable from the user's point of view: they want their issue to be resolved as soon as possible. But the reality is that just a minority of cases are blockers. In this eventuality, the sentiment analysis system can moderate that value and scale it back to a more realistic one.

Conclusions

In this thesis two Artificial Intelligence algorithms had been implemented to solve a specific linguistic task: text classification.

There was a single data source; a dataset containing thousands of closed tickets, each one presenting a specific issue encountered by a user. Each ticket was classified to its class.

The first algorithm specifically had the role to anonymize and clean the input text, performing a binary classification to identify whenever a sentence was useful to the scope of the ticket or not. It exploited the Transfer Learning technique, applied to a relatively shallow neural network. The results were high and thus this first easier task was accomplished in a relatively small time.

The second algorithm was instead built up from the grounds layer-by-layer. After an excursus on the classical ML algorithms, it was necessary to switch to a more complex solution, involving again a neural network. Due to the low number of instances of the secondary languages, it was not possible to make it multilingual. For this reason, it constitutes a future development, together with a class treatment with a clustering approach. The class handling presented some issues, even if mostly turned out to be caused by human misclassifications.

The Deep Learning approach is demonstrated to have more calculus power than what the classical solutions have to offer. Nevertheless, this power has to come along with much more data. For this reason, it constitutes a limit that depends on external agents, and consequently, it makes Deep Learning not an always pursuable route.

The initial objective of this project has been achieved, but it leaves room for the addition of new features, such as Sentiment Analysis, and algorithm enhancements. Some various technologies and discoveries can be combined with what has already been developed to be able to make even more accurate predictions.

Bibliography

[1] Alan Turing "Computing Machinery and Intelligence", Mind 49 (1950): 433-460

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, Dario Amodei "Language Models are Few-Shot Learners"

OpenAI, 2020

[3] J. J. Hopfield "Neural networks and physical systems with emergent collective computational abilities"

California Institute of Technology; and Bell Laboratories, 1982

[4] Keiron O'Shea, Ryan Nash "An Introduction to Convolutional Neural Networks"

Aberystwyth and Lancaster University

[5] Hochreiter, Sepp and Schmidhuber, JÃ¼rgen "Long Short-Term Memory"

Neural Computation

[6] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation"

Universite de Montreal, Jacobs University, Universite du Maine

[7] Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio "Neural Machine Translation by Jointly Learning to Align and Translate"

Universite de Montreal, Jacobs University

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin "Attention Is All You Need"

Google Brain, Google Research, University of Toronto

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding"

Google AI Language

[10] The article from the Google blog. https://blog.google/products/search/introducing-mum/

[11] The article from the Google blog. https://blog.google/technology/ai/lamda/

[12] The Docker website. https://www.docker.com/

[13] The Tensorflow website. https://www.tensorflow.org/

[14] The Pandas website. https://pandas.pydata.org/

[15] The Scikit-Learn website. https://scikit-learn.org/stable/

[16] The Python website. https://www.python.org/

[17] The Google Colab website. https://colab.research.google.com/

[18] The MLflow website. https://www.mlflow.org/

[19] The Flask website. https://flask.palletsprojects.com/

[20] The GloVe algorithm website. https://nlp.stanford.edu/projects/glove/

[21] The word2vec website. https://code.google.com/archive/p/word2vec/

[22] Jie Tang, Hang Li, Yunbo Cao, Zhaohui Tang, Bing Liu, and Juanzi Li "Email Data Cleaning"

Tsinghua University, 5F Sigma Center, Microsoft Corporation, Microsoft Corporation

[23] The link to the Google's text embedding layer. https://tfhub.dev/google/universal-sentence-encoder-multilingual/3

[24] Muthuraman Chidambaram, Yinfei Yang, Daniel Cer, Steve Yuan, Yun-Hsuan Sung, Brian Strope, Ray Kurzweil "Learning Cross-Lingual Sentence Representations via a Multi-task Dual-Encoder Model"

Google AI

[25] Diederik P. Kingma, Jimmy Ba "Adam: A Method for Stochastic Optimization"

University of Amsterdam, OpenAI, University of Toronto

[26] The Google's article about TPU. https://cloud.google.com/blog/products/ai-machine-learning/google-supercharges-machine-learning-tasks-with-custom-chip

[27] The Sklearn link to the mentioned cheat-sheet. https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

[28] The Sklearn guide in implementing the TF-IDF representation. https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction

[29] Klema, V. and Laub, A. "The singular value decomposition: Its computation and some applications"

IEEE Transactions on Automatic Control

[30] The link to the explanation of the Logistic Regression to classify text. https://blog.cambridgespark.com/tutorial-an-introduction-to-text-classification-d36769c593ba

[31] The Kaggle website. https://www.kaggle.com/

[32] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"

[33] The link to the Keras' Embedding layer. https://keras.io/api/layers/core_layers/embedding/

[34] The link to the Keras' Convolution 1D layer. https://keras.io/api/layers/convolution_layers/convolution1d/

[35] The link to the Keras' Global Maximum Pooling layer. https://keras.io/api/layers/pooling_layers/global_max_pooling1d/

[36] The link to the Keras' Dense layer. https://keras.io/api/layers/core_layers/dense/

[37] The link to the Keras' Spacial Dropout layer. https://keras.io/api/layers/regularization_layers/spatial_dropout1d/

[38] The link to the Keras' LSTM layer. https://keras.io/api/layers/recurrent_layers/lstm/

[39] The link to the Keras' Global Average Pooling layer. https://keras.io/api/layers/pooling_layers/global_average_pooling1d/

[40] The link to the Keras' Bidirectional layer. https://keras.io/api/layers/recurrent_layers/bidirectional/

[41] The link to the Keras' GRU layer. https://keras.io/api/layers/recurrent_layers/gru/

[42] The link to the Sklearn's class weight explanation. https://scikit-learn.org/stable/modules/generated/ sklearn.utils.class_weight.compute_class_weight.html

[43] Fonseka, Asanka and Alahakoon, Damminda and Bedingfield, Susan "GSOM sequence: An unsupervised dynamic approach for knowledge discovery in temporal data" 2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)