BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

Genrifier - Judging a Book by Its Cover

Supervisor

Lect. dr. Zsuzsanna Marian

Author

Iulia Harasim

2018

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ ENGLEZĂ

LUCRARE DE LICENŢĂ

Genrifier - A judeca o carte după copertă

Conducător ştiinţific

Lect. dr. Zsuzsanna Marian

Absolvent

Iulia Harasim

2018

Contents

Chapter 1

Introduction

“Don’t judge a book by its cover”, “what matters is on the inside”, “looks aren’t everything”. These are some of the idioms frequently used to express the same idea: it is shallow to judge something based on its appearance. Actions, however, do not follow popular belief. We constantly use make-up, good looking clothes, accessories and perfume, if not to better our appearance, then to “show our personality”. Some of the things that we say or do can also be deceiving. All these actions represent our efforts to make a *good impression.*

First impressions are out of our control. Our ability to rationally decide whether we like or dislike something is rarely manageable, if not impossible. However, impressions do not fall into a black or white category, they have a range of possibilities. We can perceive something as impressive, fun, attractive, boring, repulsive and the list can very well continue. Due to the impact that first impressions have on our behaviour towards something, marketing has become one of the necessary sections of any firm. A good first impression (achieved through an excellent ad or a nicely designed packaging) is crucial in sales. Therefore, firms strive to present their products in a good light, trying to make them as attractive as possible for their group target.

Books are no exceptions to this trend. The cover of a book is our first point of contact with it: we notice the image and read the title; our judgment or first impression is based on these two channels alone. Publishers have started to use this medium in order to provide additional information about the contents of the book. A brightly colored cover, illustrating an epic fight between Superman and the antagonist gives us a hint of the possible intrigue of the action. We can subsequently obtain the following information from the previous example: the book has superheros in it, therefore is it Fantasy; the characters are cartoonishly drawn, therefore this may be a Comic Book. We are thus able to make an educated guess about the contests of the book, especially about its *genre*.

This paper wishes to extract the additional information included in book titles and covers in order to predict its genre. There is an abundance of book genres and books can simultaneously be classified into multiple categories. In order to prevent confusion, we are going to choose a subset of genres such that they are as mutually exclusive as possible. We will train an intelligent system to differentiate between genres and strive to make it as accurate as we can. We are then going to provide a means through which humans can use the built system to label books: a Command Line Interface API and an Android mobile app.

Chapter 2 will give a broad explanation of the problem at hand and discuss the state of the art of this domain. Chapter 3 will discuss the theoretical aspects of the intelligent model proposed by this application and will cover a broad area of subjects from machine learning: Convolutional Neural Networks, Linear Support Vector Machines, Random Forests and Stacked Generalization. Chapter 4 will discuss the practical aspects of this application: starting from statistics about the data used and evaluation results of our application to how someone can use the system in order to categorize books automatically. Chapter 5 will contain the final thoughts regarding the system and discuss possible areas of improvement.

Chapter 2

Book Genre Classification

Encyclopedia Britannica defines a literary genre to be “a distinctive type or category of literary composition, such as the epic, tragedy, comedy, novel, and short story”[27]. At the starting point of literature, there were few genres in which literary texts could be classified, the earliest being the epic, the tragedy, the comedy, and the creative nonfiction. However, the unquenchable shift in social norms and every-day affairs gave life to a large diversity in literary genres, authors adapting their style and content to suit the fantasies and wishes of the masses. For example, with the exponential advancement in science and technology, a now popular genre came into existence which, in contrast to other genres, glorified the future instead of the past: Science Fiction. In 1818, *Mary Shelley*, which is now considered to be the progenitor[26] of the Sci-Fi genre published the novel *Frankenstein*. Jules Vernes, H.G. Wells, Isaac Asimov and Arthur C. Clarke are some examples of authors which followed in Mary Shelley’s footsteps, producing masterpieces in the Sci-Fi genre.

Nowadays, there are 32 genres[11] in which books are classified on the Amazon website, many having sub-genres. The total amount of genres and sub-genres exceeds 100. The title and cover of a book have a big influence in sales. The cover is the first thing a potential buyer sees when shopping, let that be online or in a book-store. This is easily seen in children, who are easily attracted by colorful-looking covers. They are more prone to ask their parents for a children’ book having on front cheerful looking animals than for a scientific book with a monochrome cover. The title is another crucial part of the first impression. Suppose that you are looking for a book on the history of Romania. A relevant title, which gives a strong hint as to what is the subject of the book is going to be more easily noticed instead of a history book which has a more abstract title. Publishers have a strong incentive for their book to give buyers the right first impression, therefore titles and covers have come to contain strong hints about a book's content and its target group.

Multiple book-stores tend to arrange books by genre. A potential buyer would not need to see the "Sci-Fi" sign about their head in order to realize that they are in that particular section. Noticing the futuristic titles and the abundant spaceships on the covers would be enough to realize that all the books before them have a strong similarity.

2.1 State of the Art

Over the past couple of years, there have been multiple attempts at creating automated systems which identify a book's genre. This is, perhaps, to show the increasing similarity in the visual aspect of books or to further increase the automatization process when adding new data in libraries or bookshops. However, not all classification systems compute their guess based on the cover and title alone. Emily Jordan's paper[34] describes an automated system which tries to classify a book based on its summary. Iwana's paper[33] makes its classification with the cover photo as the only input. The paper *"Judging a Book by its Cover"* gained quite a bit of traction, appearing in several newspaper articles[23], laying ground for the argument of visual similarity.

At the time of writing, there are two other papers which have used the same input material proposed by this thesis, which is the front cover and the title. The first one, written by Holly Chiang[18], classifies books in 5 different genres making use of the color histogram of the cover, some extracted features from a

Convolutional Neural Network trained on the ImageNet dataset and some results from two Natural Language Processing Classifiers. The second paper, written by Sigtryggur Kjartansson [47], aimed for classification into 10 genres and obtained the best results with a Residual Neural Network for the cover and a Convolutional Neural Network for title classification.

2.2 Genrifier

This paper aims to build a system which excels in classifying books into genres, based on their title and front cover. `Genrifier`, the name of this application, is not meant to compete with the approaches of the two papers previously mentioned. This is due to the fact that both have used a relatively small dataset (1,900 instances per genre), which I believe hinders our quest of creating an accurate classification system. Instead, the application in this paper uses an extended dataset which provides us almost 4 times more data and will therefore increase our chances of success in building a more accurate system. Therefore, the results of the system proposed in this paper will not be directly comparable with the results of the other papers.

One other approach that I decided to take when building this system was to not build Genrifier with a fixed model in mind. Each classification problem is different and one needs to test and tweak several models in order to figure out what best suits the problem at hand. Therefore, the criteria on which the final model was chosen lies in how good the model performed on a test set. In Chapter 4, section 2, we will compare the results of several models and prove that the final model (which will be further discussed in Chapter 3) was the best performing one out of all those tried.

Chapter 3

The Ensemble Model

The model proposed in this paper is an *ensemble* system. An ensemble system aims to improve the results of individual classifiers through combining them in all encompassing classifier. Let us begin by giving a high-level overview of how the ensemble system in this application works. First, the covers and titles are going to be processed separately. Two different classifiers (one using Inception and another using Linear Support Vector Machines) will be trained separately, using the same training set. Each of the two classifiers will output 8 numbers: $p_1, p_2 ... p_8$, where p_i represents the probability that the input belongs to the i-th genre. The 16 combined probabilities will then be passed through a third classifier (based on Random Forests) which will output the most probable genre. This ensemble method of combining the results of different classifiers through another classifier is called *stacking*. The following sections discuss in detail the models used in this application: Inception, Linear Support Vector Machines, Stacking and Random Forests.

3.1 Inception

Inception is the name of a deep convolutional neural network architecture proposed in the paper *"Going deeper with convolutions"*, written by several Google

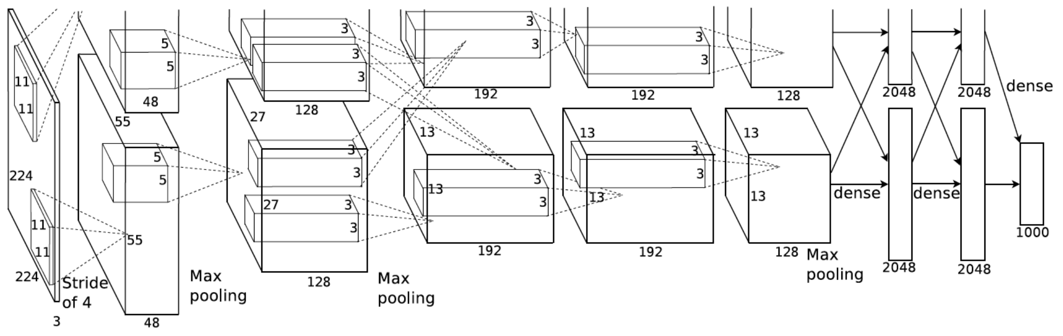


Figure 3.1: AlexNet architecture[35]

employees [52]. It was implemented in GoogLeNet and won the annual competition *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* in 2014[1].

3.1.1 ILSVRC and GoogLeNet predecessors

According to the competition's website[32], *"The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) evaluates algorithms for object detection and image classification at large scale. One high level motivation is to allow researchers to compare progress in detection across a wider variety of objects – taking advantage of the quite expensive labeling effort. Another motivation is to measure the progress of computer vision for large scale image indexing for retrieval and annotation."*. Its first edition took place in 2010 and it has been an annual contest ever since, at present day attracting submissions from more than fifty institutions per year[43]. The format of the contest is relatively straight forward: given a set of images, the participants need to train a system to classify them in 1000 different categories (including 90 different breeds of dogs).

In 2012, *AlexNet*[35] was the first network to revolutionize the field of image recognition. It obtained results 10.8% more accurate than the runner-up, with a total top-5 error of 15.3%[30]. It has a relatively simple architecture compared to the nowadays state of the art convolutional neural networks: a total of 8 layers; the first 5 were convolution layers and the last 3 were dense, fully connected layers, having a total of 60 million parameters and 650,000 neurons[53].

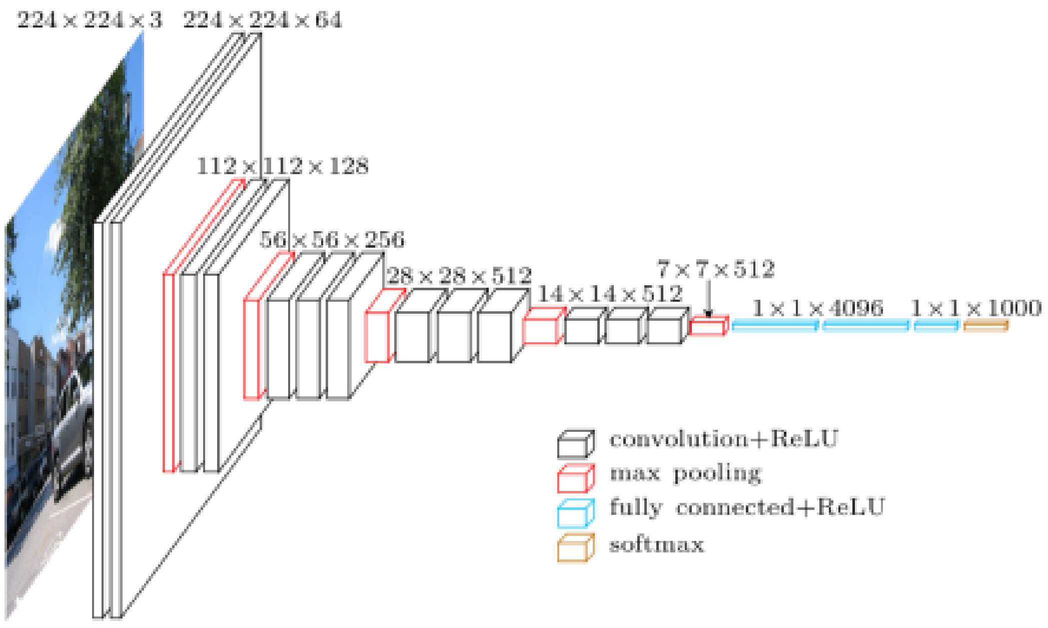


FIGURE 3.2: VGGNet architecture[2]

In 2014, another paper described an even better deep convolutional neural network architecture: VGGNet[48]. As opposed to AlexNet, it replaces large kernel-sized filters (the first 2 layers, as seen in 3.1, have a size of 11 and 5 respectively) with multiple 3x3 kernel-sized filters, one after another. It has a very appealing architecture (see 3.2) due to its uniformity: it has a total of 16 convolution layers, all having filters of size 3x3. The weight configuration of the VGGNet is available to the public, however, it consists of 140 million parameters (more than twice the number of parameters that AlexNet has) which results in long training times and the need of extensive computational resources. VGGNet was the runner-up at the 2014 edition of ILSVRC[44].

The architecture which was able to beat VGGNet is Inception. It won the 2014 ILSVRC (admittedly surpassing the runner-up by only 3.3% accuracy). The official implementation of it is called *GoogLeNet*. However, accuracy was not the only revolutionary accomplishment that Inception achieved. The truly impressive feat was the number of parameters the network contained: 12 times less than AlexNet (i.e. less than 5 million [52])! This was an incredible accomplishment due to the fact that Inception was able to run on a larger area of devices (as less computational power was needed), with no cost in results. The idea behind this

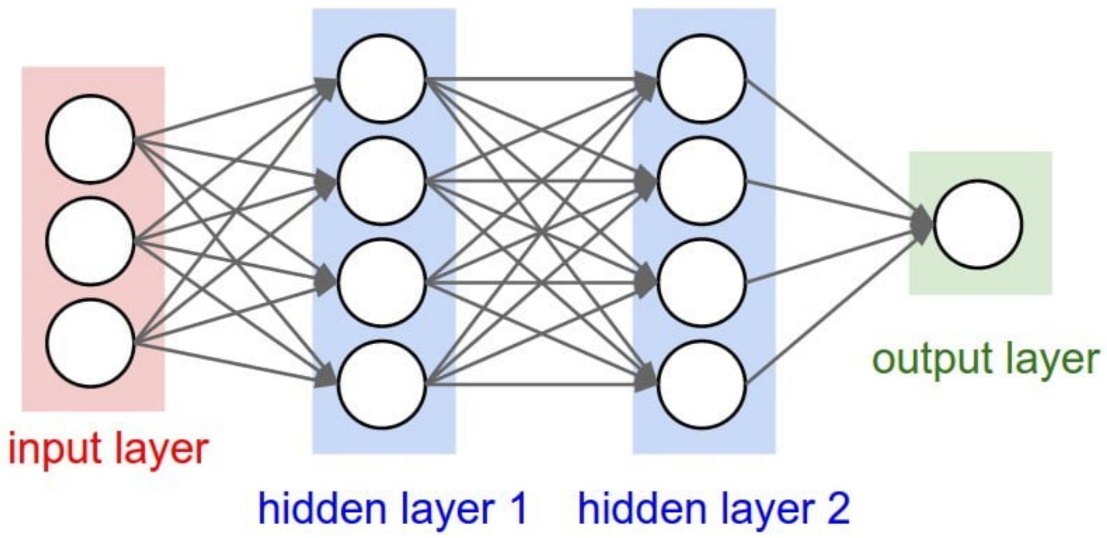


FIGURE 3.3: A simple Artificial Neural Network with 4 layers and 12 neurons[4]

architecture relies in dimensionality reduction using 1x1 convolutions (or Network in Network). Before explaining how this works, a bit of background in what Neural Networks are and how convolution works is needed.

3.1.2 Neural Networks

Artificial neural networks (ANNs) are computational models that are loosely inspired by their biological counterparts[3]. In 1943, Warren McCulloch and Walter Pitts proposed a computational model for neural networks, based on mathematics and algorithms, called threshold logic[37]. This model sparked the interest of researching how neural networks could be applied to artificial intelligence.

There are several key words related to Artificial Neural Networks that need to be defined in order to explain their inner workings:

- *neurons* are entities which try to mimic the biological neuron: they are *"connected"* with other neurons and can communicate with each other in order to learn. They are the elementary units of an artificial neural network and, at their core, are represented as mathematical functions. There are 3 types of neurons in an ANN: input neurons (they have no predecessors and they form connections only with successive neurons), neurons in hidden layers (which have connections with predecessors and successors alike) and output neurons (which, quite predictably, have connections only with predecessors). The quality of the connections is measured through *weights*. Weights, in their mathematical representation, are just real numbers. A high positive weight will give a hint to the receiving neuron that the information gained from the sending neuron is of a high, positive importance to the result. A low weight will signal the opposite meaning: the information received is of negative importance to the result from the point of view of the current neuron.

- *layers* are groups of neurons of the same kind. They can have any positive number of neurons and together form the structure of the ANN. The neurons inside a layer do not form connections with each other; they can only be connected to neurons that are in layers immediately before or immediately after the current layer. The total number of layers is equal to the *depth* of the network. For example, in Figure 3.3, there are a total of 4 layers: an input layer, 2 hidden layers and an output layer. Any ANN has in its consistency an input layer and an output layer. The number of hidden layers can vary, depending on the desired architecture. A *dense* (or *fully-connected*) layer has the property that all neurons on that layer are connected to all neurons from the previous layer. For example, in Figure 3.3, both the hidden layers and the output layer satisfy this property and are therefore considered fully-connected.

- *activation functions* define how a non-input neuron will process and forward the information it receives; they are also the mathematical representations of neurons (as previously hinted at). In order to understand what they are, we first need to define how the input is processed by neurons. Let's denote by $x_1, x_2...x_l$ the input information received by the neuron (as seen in Figure 3.4). The weights $w_1, w_2...w_l$ are associated to $x_1, x_2...x_l$ respectively. First, the neuron will compute the weighted sum of the inputs $\sum_{i=1}^{n}(x_i w_i)$. The result is then passed through the *activation function*. On of the most used activation functions is called *ReLU (Rectifier Linear Unit)*. It has been

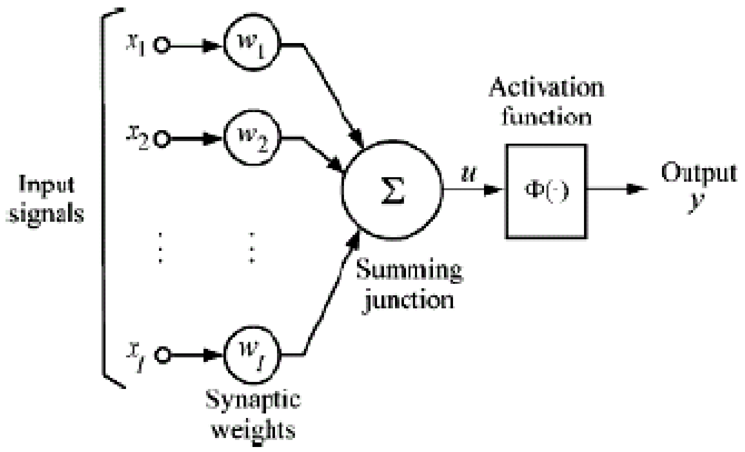


FIGURE 3.4: An artificial neuron with l inputs[5]

shown that, as opposed to other activation functions, ReLU accelerated the convergence of the training algorithm[35]. It has the following definition:

$$ReLU(x) = max(0, x)$$

There are a lot of existing activation functions, however, for the purposes of this explanation, ReLU will suffice as the sole example. The result of the activation function is passed through the output connections of the neuron. It is through this way of communication that information passes from the input layer towards the output layer.

- *backpropagation* is the algorithm used for training an ANN. A famous paper, *"Learning representations by back-propagating errors"*[42], written in 1986, described some approaches where neural networks worked far faster than earlier approaches, making it possible to use neural nets to solve problems which had previously been insolvable[39]. The goal of this algorithm is to calibrate the weights such that the network will output results as close as possible to a desired value. Suppose that we're training a network which, given as input the number of rooms of an apartment alongside the latitude and the longitude of its location, wants to determine the rent price of the apartment. We would start the training by giving as input the information of the first apartment through the input layer (see Figure 3.3). We have exactly three neurons in the input layer so the number of rooms will be attributed to the first neuron, the latitude to the second and lastly, the longitude to the third neuron. Once the information of the apartment has been submitted, the neurons on the first hidden layer will use those values to compute the weighted sum and pass the result through the activation function to the next layer. This operation is repeated layer by layer. When the single neuron in the output layer will finally finish processing the data received from the second hidden layer, we will have reached a result (i.e. a number representing the rent for the apartment). However, quite predictably, the weights are not well calibrated in the first run of the algorithm, so the system needs to perform some changes, depending on how *accurate* the prediction was compared to the actual price of the apartment. First, the *cost function* will be used in order to compute how "far off" the prediction was from the actual result. A frequent cost function used in regression is *Root Mean Square Error (RMSE).* The formula is:

$$RMSE(x) = \frac{1}{n}\sqrt{\sum_{i=1}^{n}(y_i - prediction(x_i))^2}$$

where n is the total number of instances in the training set, y_i is the actual price of the apartment of the i-th instance and *prediction(x_i)* is the value resulted from the Neural Network for the i-th instance. In order to obtain good results, we need to have a network which has a small error: we want to predict rents as accurately as possible. The question is: what algorithm would provide us a way to modify the parameters so that they converge to the optimal ones needed for a small RMSE?

- *gradient descent* is an iterative optimization algorithm used to find the minimum of a function. An intuitive way of understanding what gradient descent does is the following: suppose we were on top of a mountain and we wanted to reach a cabin which is right at the base of the mountain; one algorithm that we could follow in order to reach the ground is: looking around before taking any step and deciding to move forward the path that is the steepest

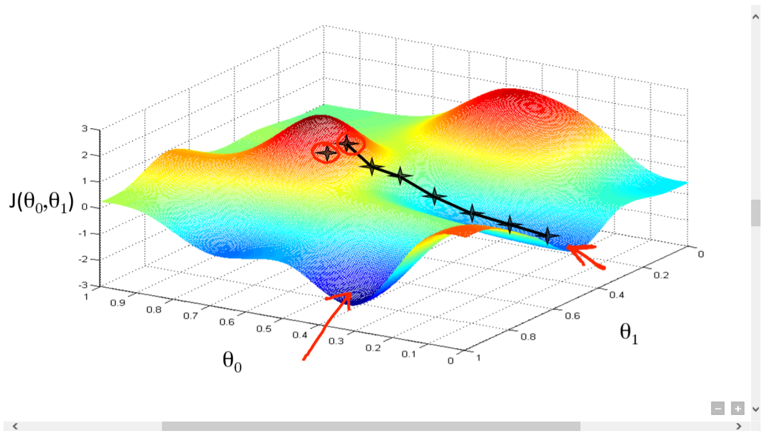


FIGURE 3.5: Gradient descent trying to find the minimum of a function[6]

in a downwards direction. For example, in Figure 3.5, we would start from the red circle, atop of the mountain, and the decisions that we would have made follow the black arrows exactly. We will happily reach the base of the mountain. Unfortunately, the location that we will find ourselves in is not the location of the cabin. In fact, looking at the Figure, one can notice that the minimum in which we would find ourselves in is not the global minimum. This is due to the fact that the algorithm is not guaranteed to find the global minimum but rather is guaranteed to reach a local minimum. Gradient descent has the exact same principle as the algorithm used to climb down the mountain: at each iteration, it will look for the steepest downwards slope and will decide to modify the parameters such that the next value of the function will converge to a local minimum. Fortunately, in math, there is a very specific way of finding what "direction" the algorithm should take: the slope of the tangent which is conveniently enough equivalent to the gradient.

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

The formula above describes the iterative process of the gradient descent. θ_i represents the i-th feature of the model. A feature, in this context, represents the i-th weight of the node for which we apply the formula. For example, the output node from Figure 3.3 has 4 features, therefore we would need to repeat the formula 4 times, computing in total 4 partial derivatives with

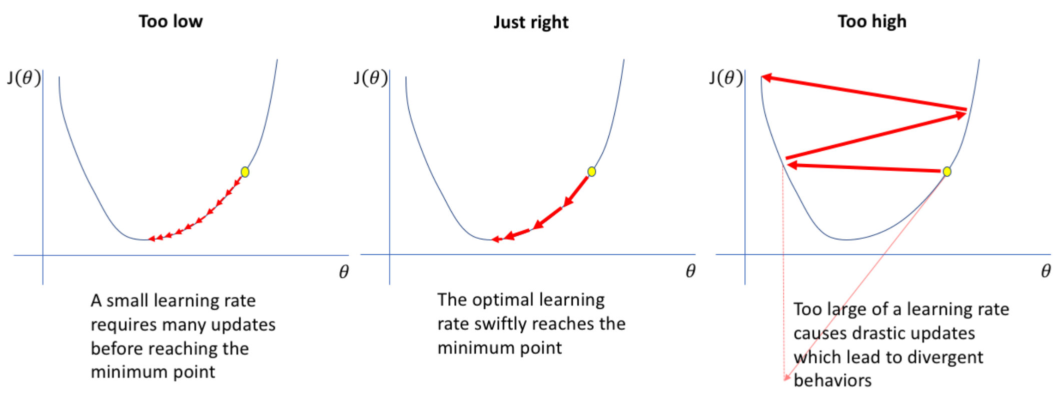

Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

FIGURE 3.6: Possible scenarios with different learning rates[7]

respect to the weights. J is the mathematical notation of the cost function. In our example, it is equivalent with RMSE. α is the *learning rate* i.e. it defines how drastically we will modify the weights at each iteration. You can intuitively think of α as the size of the step that we would make when descending the mountain. The appropriate choosing of α is important for the convergence of the algorithm. Suppose that we would have a big α, therefore, we would make huge steps when deciding to move along a path. The step might be so big that we might find ourselves on a higher surface than before. Conversely, if we were to choose a tiny learning rate, we would have to make a lot of decisions and decline in a slow manner. Figure 3.6 illustrates the possible scenarios that result from choosing a learning rate.

The backpropagation algorithm will successively modify weights starting from the neurons on the rightmost layer to the ones on the leftmost layer (in a "backwards" fashion, hence the name). The cost function will start to decrease with each iteration. The stopping criterion of the algorithm depends on the user. For example, one could choose to finalize the training process (i.e. the backpropagation algorithm) when the we have completed a certain number of iterations.

Having explained what a neural network is and how we could use it for training, we could start thinking how we can use such a system in order to classify images. We could, perhaps, have one neuron in the input layer responsible for one pixel of the image. Therefore, for an image having a resolution of 100x100, we would need exactly 10,000 neurons in the first input layer. The output layer could have exactly the same number of neurons as the number of categories in which we need to classify the images, the value in each such neuron representing the *probability* of belonging to a particular class. This is a perfectly valid way of training ANNs to classify images, however, the shape and the structure of the image will be lost in the process. Intuitively speaking, we as humans do not need to break down the image pixel by pixel and transform it into a one-dimensional structure. The multi-dimensionality of the image is crucial towards accurately classifying it.

3.1.3 Convolutional Neural Networks

The paper *"Gradient-Based Learning Applied to Document Recognition"*[36], released in 1998 introduced the term of a *Convolutional Neural Network (CNN).* LeCunn created a CNN that classified digits in hand-written form, given as 32×32 images. The CNN was used by several banks to recognize hand-written numbers on cheques. LeNet-5 (the name of the CNN) revolutionized image recognition in the artificial intelligence domain. Ever since, CNNs have become widely used in a variety of domains, including video recognition, recommender systems and natural language processing.

CNNs introduced two different layers to a classic ANN architecture:

- *convolutional layers*
- *pooling layers*

Before explaining what each of these layers do, we need to focus on how data is fed in CNNs. As opposed to ANNs, where the input layer has one dimension, CNNs receive the input in a 3D structure representing the image (the input is commonly referred to as *volume*). The first dimension represents the width, the second the height and the third the number of *channels* (or *depth*). In the case of a colorized image, the number of channels will be three: one for the red value, one for the green value and another one for the blue value. If we were to process grey scale images, the number of channels will be one.

The first layer in any CNN will be a *Convolutional Layer.* A good analogy for how convolutional layers work is the following: we are in a dark room, with no light source whatsoever other than a flashlight which can illuminate at one point in time exactly $W \times H$ cm. If we were to have in front of us a painting, we would need to slide the flashlight along the painting gradually in order to see it. To notice every single detail of the picture, we could start from the top-left corner, slide to the top-right corner, then move down and continue in the same right-down fashion until every single piece of the painting was seen.

A convolutional layer works in a similar manner: a *filter* (i.e. flashlight) is slid along the image. A filter has 2 properties associated to it: *the receptive field* (i.e. the area it covers; in our previous case it was $W \times H$; however, in CNNs, the width and depth are equal thus resulting in a square with an $N \times N$ resolution) and *the depth.* The depth of the filter needs to be the same as the depth of the input volume. Therefore, for an input volume of size $32 \times 32 \times 3$, the filter has a volatile receptive field (e.g. 3×3, 5×5, etc), however, it has a fixed depth of 3. Each cell of the 3D structure of the neuron holds a value (a *weight*) which will later be calibrated in the training process.

By definition[21], a convolutional layer has 4 parameters which define it:

- *the number of filters* (K).
- *the spatial extend* of its filters (F); the *receptive filed* is $F \times F$.
- *the stride* (S), which defines the "speed" at which the filter will be slid across the volume. A stride of 1 means that the filter will move one pixel at a time. The stride can be expressed in two dimensions, thus defining how many pixels we will move depending on the dimension we will move the filter on. For example, if we were to have a stride of (2, 1), when shifting the filter to the right we would move 2 pixels at a time; however, if we were to move

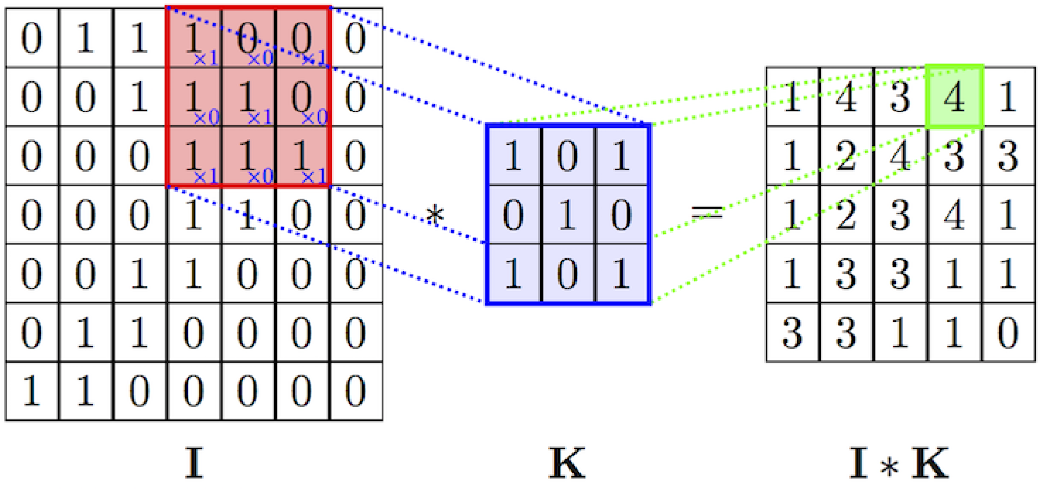


FIGURE 3.7: Convolution applied to a 7 × 7 × 1 input volume. The kernel has a spatial extend of 3. Judging by the size of the output, we can deduce that the stride is 1 and the padding is 0.[8]

the filter down, we would move only one pixel at a time. The most common values for the stride are 1 or 2.

- *the zero padding* (P), which defines the amount of padding the input volume will receive around its border. A padding size of one means that the input volume will be "surrounded" by a frame of zeros, of size one.

We can simply denote the convolutional layer by a tuple of 4 elements: (K, F, S, P). As input, it receives a volume of size (W_1, H_1, D_1). After passing through the convolutional layer, the output volume will have the following dimensions[21]: (W_2, H_2, D_2), where:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$
- $D_2 = K$

Therefore, the stride, the padding and the spatial extent are used to regularize the width and the height of the output volume, while the number of filters determines the depth of the output volume.

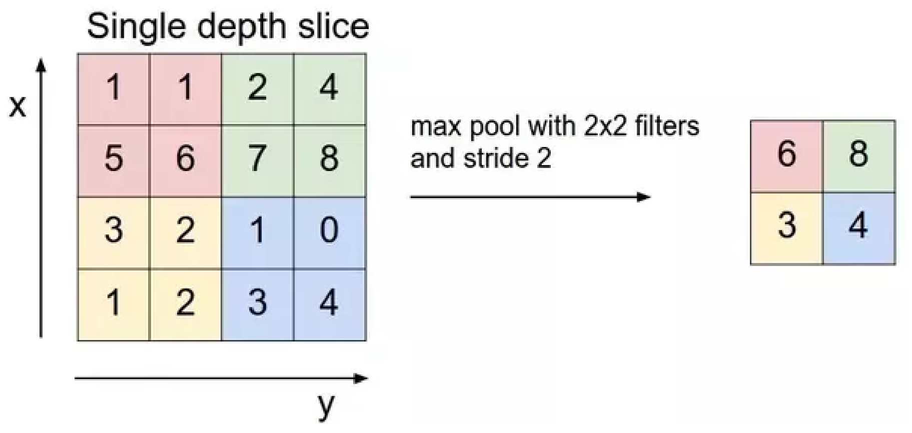


FIGURE 3.8: Passing an input volume through a max pooling layer of spatial extent 2, with a stride of 2[21]

The operation that is applied during the slide of the filter is eponymously called "convolution". It performs a dot product between the weights in the filter and the values found on a particular slice of the image. Figure 3.7 visually exemplifies the the convolution process. In order to introduce non-linearity, the output volume from a convolutional layer is usually passed through an activation function (such as ReLU) before being fed to successive layers.

Pooling Layers are common in CNNs and are usually found immediately after convolutional layers. The basic action that they perform is to reduce a portion of the input volume to a singular value. Just like with convolutional layers, we can image that we are sliding across the input a flashlight. However, this time around, we won't be using the flashlight to notice every detail of a painting: we will be using it to get a certain type of information out of each slice. When looking at a picture, humans are most commonly drawn to a particular detail. When describing an image, we can't illustrate every particularity that it has, however, we resume our description to some key words that we feel that do justice to the picture.

One of the most common types of pooling layers is a *max pooling layer*. It behaves similarly to what we do in our brains: it retains the most poignant parts of a picture (i.e. the *max* value). A visual explanation of how a max pooling layer works is given in Figure 3.8.

Pooling layers have 2 characteristics in common with convolutional layers:

- *the spatial extent* (F), i.e. the *size* of the pooling area (for example, a max pooling layer with a spatial extent of 2, is going to extract the maximum value from a square of 2×2 at a time)
- *the stride* (S). Just like convolutional layers, we can choose to slide the metaphorical flashlight a certain number of pixels at a time.

We can simply define a pooling layer by a tuple of 3 elements: (F, S, A), where A is the *reduce function* applied in the pooling layer. We previously gave as an example the *max* function. Another type of reduce function used in CNNs is the *average* function.

As input, the pooling layer receives a volume of size (W_1, H_1, D_1). After passing through it, the output volume will have the following dimensions[21]: (W_2, H_2, D_2), where:

- $W_2 = (W_1 - F)/S + 1$
- $H_2 = (H_1 - F)/S + 1$
- $D_2 = D_1$

As opposed to convolutional layers, the pooling operation is applied for each depth slice independently, therefore the depth of the output volume will remain unchanged. The general goal of pooling layers are to reduce the spatial size of the volumes and the total number of parameters. Pooling is also a means through which we can prevent over-fitting.

Pooling layers are not necessary. Some papers, such as *"Striving for Simplicity: The All Convolutional Net"*[50], propose getting rid of pooling layers altogether. In order to reduce the size of the representation, they increase the size of the stride for some convolutional layers.

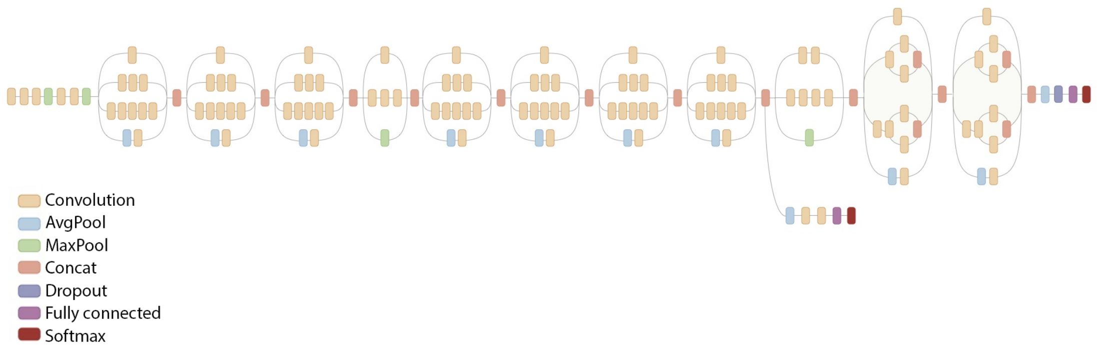


FIGURE 3.9: The architecture of Inception v3[9]

3.1.4 The Inception Architecture

Now that the basic elements of how Convolutional Neural Networks work have been explained, we can move on to exploring the Inception architecture. Firstly, we are going to discuss each layer category in the architecture, then we are going to talk about the core module of Inception.

As seen in Figure 3.9, the architecture of Inception is considerably more complex than those of VGGNet or AlexNet (3.2 and 3.1 respectively). The categories into which layers can be classified are:

- *Convolutional Layers*, which perform basic convolution operations,
- *Pooling Layer*, which perform both max and average operations,
- *Concatenation Layers*, which simply add all input volumes into a large output volume (the volumes are concatenated in depth i.e. stacked onto another)
- *Dropout Layers*, which are used in order to prevent over-fitting. In these layers, randomly selected neurons are ignored during training. Dropout layers have a probability associated to them, which defines the probability of any neuron to be "drooped out" during training.
- *Fully connected layers*, which are basic ANN layers and have been previously explained,

- *Softmax layers*, which are the final, output layers. They combine the results from the fully connected layers and compute the probability of association of the input to each class.

Two stark differences are visible when comparing the images of the three CNN architectures. First of all, Inception has 2 output layers as opposed to one. Not only does the third version of Inception have 2 outputs, but its predecessor (i.e. the first version), had three outputs! We will discuss in the following paragraphs the need of multiple output layers in large neural networks. Second of all, the network is not only *deep* (i.e. having a lot a hidden layers); it is also *wide*: the network is no longer a continuous series of operations which appear to have one dimension, but layers have been stacked "onto" each other, therefore arising the need of a Concatenation Layer to combine the results. Another question that might arise is: how does Inception manage to have such a staggeringly small number of parameters compared to AlexNet and VGGNet when the architecture is irrefutably more complex? The short answer is that dimensionality reduction is applied through 1×1 convolutions and therefore the number of parameters is diminished. We will discuss in the following subsection the importance of such convolutions and how they are able to reduce the number of parameters.

The Vanishing Gradient Problem is a frequent issue in deep neural networks. While adding more layers increases the complexity and the capability of a network to learn, it also raises some problems with respect to the *speed* at which each layer of the network learns. *Speed* is a relatively volatile word, so let's give it a proper, mathematical definition: we define the *speed* of a layer to be the square of the second norm of the vector of gradients for each neuron on that layer (i.e. $||\delta||^2$, where the second norm can be defined as the root of the sum of squares of the elements of a vector). *The vanishing gradient problem* describes the phenomenon in which the speed of early hidden layers is inversely proportional to the number of layers that an ANN has. Michael A. Nielsen gave in his book, *"Neural Networks and Deep Learning"*, an example of a deep neural network which was trained to recognize hand written digits. As he gradually added layers, he plotted the

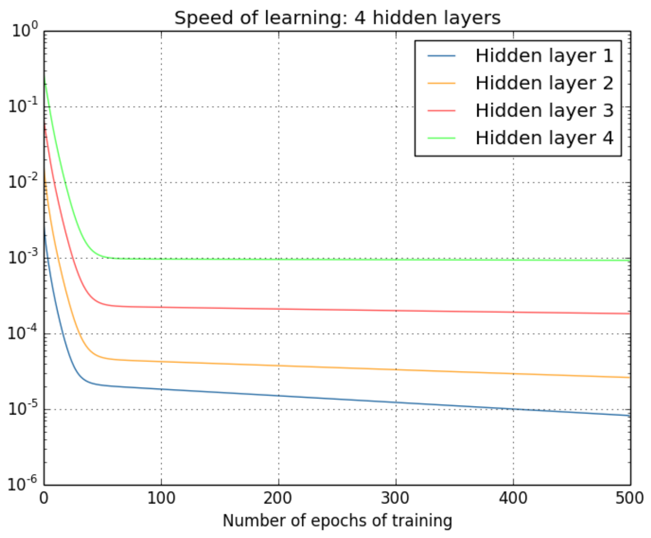


FIGURE 3.10: The speed of hidden layers plotted against the number of iterations, taken from "Neural Networks and Deep Learning"[39].

speed of the layers against the number of iterations (as shown in Figure 3.10) and showed how the vanishing gradient problem became more and more apparent as the number of layers increased (the first hidden layer can be seen to learn 100 times slower than the last hidden layer).

The engineers who worked on Inception realized the gravity of this problem and decided to add auxiliary classifiers. The motivation behind this is that intermediate features are also capable of discriminating between different categories, not only the final layers. Therefore, the loss of the network is actually a combination between two loss functions from two different output layers: *"During training, their loss gets added to the total loss of the network with a discount weight (the losses of the auxiliary classifiers were weighted by 0.3). At inference time, these auxiliary networks are discarded"*[52].

3.1.5 The Inception Module

The Inception module is one of the most important elements of the architecture. It is responsible for dimensionality reduction and it revolutionized the number of parameters that an accurate network “needs” to have. The module also makes the network *wide*, due to the fact that it *“stacks”* layers onto each other.

3.1.5.1 1×1 Convolutions (or Network in Network)

At first, 1×1 convolutional layers do not make much sense. The spatial extent is as small as it can be, so why not use a more comprehensive filter, with a bigger size?

Let’s start with a small reminder from convolutional layers: the depth of the output volume is determined by the number of filters in the convolutional layer. Therefore, if we wish to alter the depth of volumes, without altering the width or the height of the volume, one straight forward solution would be to use a 1×1 convolution layer, with zero padding and a stride of 1. Let’s use the previously given formulas on an example in order to demonstrate this. Suppose that the input volume has a size of (W_1, H_1, D_1) and the convolutional layer it passes through is defined by the tuple $(K, 1, 1, 0)$ (K is the number of filters, the first 1 is the spatial extent, the second 1 represents the stride and 0 stands for no padding). The output will have a size of (W_2, H_2, D_2), where:

- $W_2 = (W_1 - 1 + 0)/1 + 1 = W_1$
- $H_2 = (H_1 - 1 + 0)/1 + 1 = H_1$
- $D_2 = K$

Therefore, the output will not have changed its width or height, however, the depth will be changed according to the number of filters of the convolutional layer. It is through this way that dimensionality reduction (i.e. the reduction of the number of parameters) is achieved.

The architecture of the Inception module is shown in Figure 3.11. It includes a total of 6 convolutions, along with a max pooling layer. As shown in the Figure, 1x1 convolutions are used before each of the 3×3 and 5×5 convolutions. One

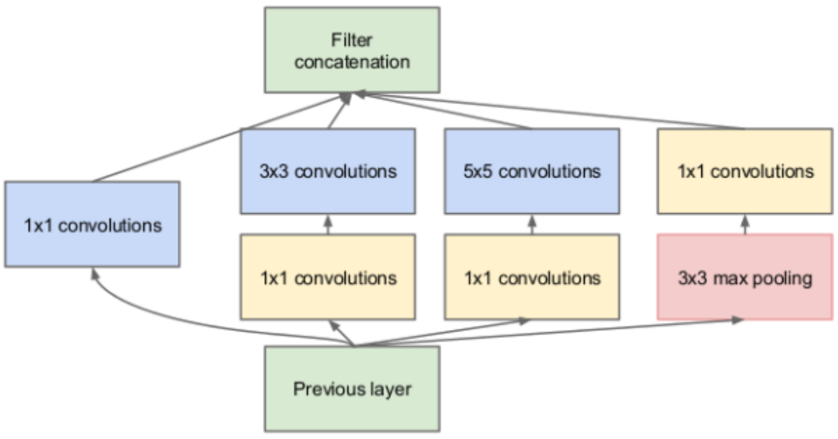


FIGURE 3.11: The architecture of the Inception module[52]

question arises from such a peculiar arrangement: since the 3×3 and 5×5 convolutional layers are going to regulate the depth of the output volume as well, why should one use 1x1 convolutions beforehand? The answer lies in the number of operations.

Suppose that we want to pass a $28 \times 28 \times 192$ input volume through a convolutional layer defined by the tuple (32, 5, 1, 2). The output, according the equations beforehand will be of size $28 \times 28 \times 32$. The depth has changed, however, the width and the height have not! The number of operations needed to compute the output volume is:

$(28 \cdot 28 \cdot 32) \cdot (5 \cdot 5 \cdot 192) = 120,422,400$ operations.

Now, suppose, that we have a *"bottleneck layer"* consisting of a 1×1 convolution, defined by the tuple (32, 1, 1, 0) just before the aforementioned layer of 5×5. The number of operations needed to pass an input volume of size $28 \times 28 \times 192$ is:

$(28 \cdot 28 \cdot 32) \cdot (1 \cdot 1 \cdot 192) + (28 \cdot 28 \cdot 32) \cdot (5 \cdot 5 \cdot 32) = 24,887,296$ operations!

By using a simple 1×1 convolution layer we have reduced the number of operations needed to compute a similar output in size by almost 5 times! Therefore, 1x1 convolutions play a crucial role in diminishing the number of parameters while also speeding up the computational process.

3.1.6 Retraining Inception

Training this model from scratch requires a big training set and can take hundreds of hours to complete. However, TensorFlow developers have provided a script for "retraining" the network. It makes use of *transfer learning* which *"is a technique that shortcuts much of this [the training process] by taking a piece of a model that has already been trained on a related task and reusing it in a new model"*[29]. It makes use of the excellent feature extraction capabilities of the pre-trained network in order to process input and simply trains a new classification layer on top in order to recognize new categories.

The script performs the following operations: first, it computes the *bottleneck values* for each image from the training set; during the second phase, the back-propagation algorithm is run for a certain number of steps on the bottleneck values; at each 10 steps, the validation accuracy is outputted along with the value of the loss function (Inception uses cross entropy as the loss function). According to the official documentation, *"'Bottleneck' is an informal term we often use for the layer just before the final output layer that actually does the classification. (TensorFlow Hub calls this an "image feature vector".) This penultimate layer has been trained to output a set of values that's good enough for the classifier to use to distinguish between all the classes it's been asked to recognize. That means it has to be a meaningful and compact summary of the image"*[29].

The output is a TensorFlow graph, serialized to a file, which can later be used in order to make predictions. Therefore the model used in order to classify book cover images into genres is: the third version of the Inception architecture retrained to recognize new categories.

3.2 Linear Support Vector Classifier

The final model used in this application in order to classify book titles into genres makes use of *Linear Support Vector Machine Classifiers*.

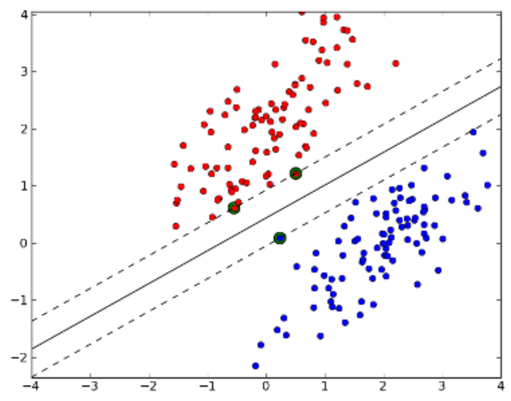


Figure 3.12: The visual separation of objects in 2 different classes[10]

The original algorithm was proposed by Vladimir N. Vapnik and Alexey Ya. Chervonenkis in 1963[54]. In 1992, Vapnik along Bernhard E. Boser and Isabelle M. Guyon[15] suggested a way to create nonlinear classifiers by applying the kernel trick for hard-margin classification. The current soft margin method was proposed by Corinna Cortes and Vapnik in 1993 and published in 1995[20].

The fundamental idea behind Support Vector Machines is best understood visualized. Suppose we would like to classify several objects in two classes. The objects each have two features, therefore, if we were to plot them, we would need an xOy system: the Ox line would represent one of the features and Oy would stand for the remaining one. Let's visually differentiate the two classes: the objects belonging to the first class will be described using red circles and the objects contained in the other class will be represented using blue circles. One such plot might look like the one in Figure 3.12.

A support vector machine classifier (SVC) will try to find a division (i.e. a curve) which geometrically separates the instances of the two classes. Depending on the *type* of the curve, SVCs can be classified into *linear* and *nonlinear* SVCs. Linear classifiers want to separate the instances using straight lines. Nonlinear SVCs use any type of curve in order to separate the instances in the two classes. In the case of this application, the type of SVC used was a linear one so for the remaining of this section we will focus our efforts into explaining how *linear support vector machine classifiers* work.

A good analogy for how SVCs try to find the best division between classes is that they try to fit *the largest possible street between the classes.* The solid line in Figure 3.12 is the “middle” of this imaginary street, while the dotted lines are the “edges” of the street. The edges are determined by the closest instances to the dividing line. These instances, circled in green, are called *support vectors.* The problem of finding the largest possible street is called *large margin classification.* Therefore, it is not enough for an SVC to find any line which separates the two classes, but its goal is to find a line such that the distance between its support vectors is the highest possible.

Large margin classification is subdivided into two different problems:

- *hard margin classification* which strictly imposes that all instances are to be off the street and on the right side (Figure 3.12 is a good example for a solution to such a problem)
- *soft margin classification*, which has the objective of finding a good balance between keeping the street as large as possible while limiting the margin violations, i.e. instances that are “on” the street or on the wrong side of the street.

Hard margin classification in linear SVCs is rarely realizable and has two big issues. First of all, the data given as input might not be linearly separable which results in the impossibility of finding a division. Second of all, this problem is very sensitive to *outliers.* Suppose that Figure 3.12 had a red dot at (2, -1), in the middle of the blue dots. It would have been completely impossible to linearly and completely separate the two instances. However, this wasn’t due to the fact that the data didn’t have a reasonable linear separation, rather it didn’t have a complete separation. Hard margin classification is rarely required in present times and most implementations of support vector machine classifiers default to solving soft margin classification. The balance between keeping the street as large as possible, while limiting the margin violations is calibrated through a parameter called ’C’ in the Scikit-Learn module in Python. A smaller C value results in a

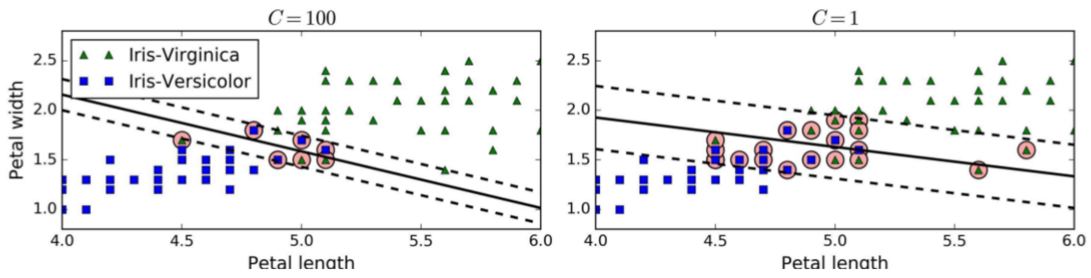


Figure 3.13: The influence of C on SVCs[28]

larger street, with more street violations, while a bigger values results in a tighter street. This parameter can be further used when the model is over-fitting the training data by giving as input a small C value. Figure 3.13, taken from *"Hands-On Machine Learning with Scikit-Learn & TensorFlow"*[28], visually shows the influence of the parameter C. The mathematical influence of the parameter will further be explained in the following subsection.

The linear SVC classifier used in this application has the following properties: its loss function is called *hinge*, it solves the dual problem and its strategy for multi-class classification is called ovr (one-versus-rest).

3.2.1 The Decision Function and the Hinge loss function

The linear SVC, just like a neuron in an Artificial Neural Network, has a set of weights with which the prediction is computed. The number of weights is equal to the number of features given as input plus one, for the bias. The bias term simply allows for a larger range of output values, irrespective of the input. For example, if we were to provide as input zeros for every feature, the weighted sum will always default to zero, irrespective of the value of the weights. Adding a bias term is a trick which will permit the system to output non-zero values in case of such occurrences.

Suppose we wanted to train an SVC to determine whether an apartment is family friendly. We have the following information about each apartment: the number of rooms of the apartment, the latitude and the longitude of the location. The weights vector will have a length of 4: w_0 will associated to the bias term $x_0 = 1$, w_1 will associated to $x_1 = num.\ rooms$, w_2 will associated to $x_2 = latitude$ and finally w_3 will associated to $x_3 = longitude$. The prediction(i.e. decision function) of a binary SVC is, as follows[28]:

$$pred(x) = \begin{cases} -1, & \text{if } w^T \cdot x + b < 0 \\ 1, & \text{if } w^T \cdot x + b \geq 0 \end{cases}$$

x holds the features of a training instance, w is the column vector of the weights and b represents the biases. The result of the prediction (-1 or 1) will classify the input into one of the two classes: -1 for the "negative" class (i.e. not family friendly) and 1 for the "positive" class (i.e. family friendly). Therefore the prediction of an instance is not heavy from a computational point of view: the linear SVC will compute the weighted sum of the input and will add the result with the biases. The question becomes: how do we choose appropriate values for the weights and the biases such that the predictions will be as accurate as possible? The method used is similar to the one in ANNs: minimize the results of a loss function, using the instances in a training set.

The loss function used by the Linear SVC model proposed in this application is called *hinge*[28].

$$hinge(t, y) = max(0, 1 - t * y)$$

y is the output of the decision function (i.e $pred(x) = y$), and t takes a value of -1 if the intended class of x was the negative class and 1 otherwise. When t and y have the same sign (i.e. x is classified correctly), the result of the hinge function is 0, indicating no mistakes whatsoever for that particular instance. However, if t and y have different signs, the hinge function will be a positive number equal to $1 - t * y$.

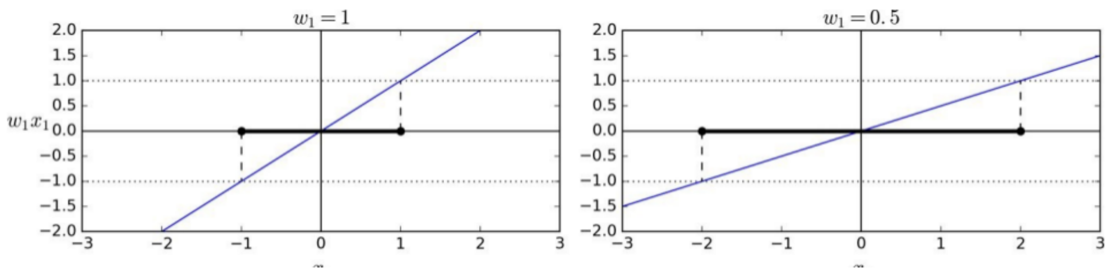


FIGURE 3.14: Smaller weights result in large margins, taken from "Hands-On Machine Learning with Scikit-Learn & Tensorflow"[28].

3.2.2 The dual problem

Let's remind the training objective of a linear SVC: it wishes to minimize the loss function. However, as opposed to the gradient descent algorithm proposed in ANNs, SVCs have a different mechanism in order to calibrate the weights such that the loss function has as small a value as possible.

Support vectors are instances of data which are exactly on the edge of the street, i.e. on the margin. Written mathematically, this is equivalent to $w^T x^{(i)} + b = \pm 1$. If we were to solve a hard-margin classification problem, we would obtain the following mathematical equation[28] for each data instance in the training set:

$$t^{(i)}(w^T x^{(i)} + b) \geq 1, \text{ for i = 1, 2 ... m}$$

The m above represents the number of instances in the training set, while w, x and t have the same meaning as before. However, finding a solution to this equation does not solve the hard margin problem: we need to find w and b such that the margins are as large as possible. The mathematical representation of this requirement is that we need to minimize $\frac{1}{2}w^T w$. Understanding why the previous expression is equivalent to finding the largest margin is best explained visually.

Figure 3.14 shows two scenarios: one where the weight is of size 1 and another where its value is 0.5. The bolded line represents the width of the street. The margins of the street are at -1 and 1 respectively. When the weight is smaller, it is easily noticeable comparing the two plots that the range in which x_1 performs a margin violation becomes smaller as the size of the weight increases. Therefore, the size of the weight is inversely proportional with the size of the margin, hence the requirement of minimizing the weights. While the requirement could have been expressed as “minimize $||w||$”, the previous value is not differentiable on its whole domain and therefore a similar requirement was used instead: $\frac{1}{2}\sum_{i=1}^{m} w^{(i)^2}$, which is equivalent to $\frac{1}{2}||w||^2$.

Combining the two equations, the mathematical objective of the hard-margin linear SVM classifier is the following[28]:

$$\textbf{minimize } \frac{1}{2}w^T \cdot w$$

$$\textbf{subject to } t^{(i)}(w^T x^{(i)} + b) \geq 1\textbf{, for i = 1, 2 ... m}$$

However, as not all data is linearly divisible, we also need to define the mathematical objective of the soft margin linear SVM classifier. A new variable is added to the equation: $\zeta^{(i)}$. It measures how much the i-th instance is allowed to violate the margins. This is a conflicting variable compared to our goal to maximize the margin. The aforementioned parameter C is the deciding factor for how much ζ is taken into account. The mathematical objective of the soft-margin linear SVM classifier is, therefore[28]:

$$\textbf{minimize } \frac{1}{2}w^T \cdot w + C\sum_{i=1}^{m}\zeta^{(i)}$$

$$\textbf{subject to } t^{(i)}(w^T x^{(i)} + b) \geq 1\textbf{, for i = 1, 2 ... m}$$

Both of these problems (i.e. the hard and soft margin classification problems) are actually *convex quadratic optimization problems with linear constrains. Quadratic programming* is the process of solving such problems and there are many off the shelf frameworks which provide solutions to such problems.

The duality principle states that optimization problems may be viewed from two perspectives: the primal problem or the dual problem. The solution to the dual problem provides a lower bound to the solution of the primal problem. In some particular case (the objective function being convex and the inequality constrains being both convex and continuously differentiable[28]), the solution to the dual problem is the same with the solution to the primal problem. The hard and soft margin classification problems of SVCs fulfill this requirement and, therefore, the optimal weights can be found either from solving the primal or the dual problem.

The soft-margin classification problem has the following dual problem[20]:

$$\textbf{find } \alpha_1, \alpha_2, ...\alpha_m \textbf{ such that}$$
$$\sum_{i=1}^{m} \alpha^{(i)} - \frac{1}{2}\sum_{i=1}^{m}\sum_{i=1}^{m}(\alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}x^{(i)^T}x^{(j)}) \textbf{ is minimized}$$
$$\textbf{subject to } \sum_{i=1}^{m} \alpha^{(i)}t^{(i)} = 0 \textbf{ and } 0 \leq \alpha^{(i)} \leq C\textbf{, for i = 1, 2 ... m}$$

The dual problem becomes faster to solve when the number of features is bigger than the number of instances. Incidentally, after pre-processing the book titles, the number of features resulted is higher than the number of instances in the training set. Therefore, since it fastens the training process, this application indeed chooses to solve the dual problem of the soft-margin classification problem instead of the primal problem.

3.2.3 OVR strategy for multi-class classification

There are two main strategies when it comes to classifying instances into more than two categories:

- *one-versus-one*, or *ovo*, where multiple binary classifiers are trained in order to differentiate between each pair of classes. Suppose we wanted to train a multi-class classifier in order to differentiate between N classes. We would need exactly $\frac{N(N-1)}{2}$ binary classifiers in order to achieve this. When trying to predict the class of a new instance, we would need to compute the predictions from all $\frac{N(N-1)}{2}$ classifiers and return the class which "won" the most duels.

- *one-versus-rest*, or *ovr*, where a binary classifier is trained to differentiate the first class from the rest, the second class from the rest and so on. Suppose we wanted to train a multi-class classifier in order to differentiate between N classes. We would need exactly N binary classifiers in order to achieve this. When predicting the class of a new instance, we would compare the decision score from each of the N classifiers and return the class which had the biggest decision score out of all.

This application uses the ovr strategy. The biggest advantage of this strategy is that is uses a considerably smaller number of binary classifiers, due to the fact that ovr scales linearly with respect to the number of classes, as opposed to ovo which scales squarely.

3.3 Stacking (or Stacked Generalization)

The method through which the results of the aforementioned classifiers are aggregated is called *stacking*. It was originally proposed in the paper "Stacked Generalization", in 1992[55]. Previous aggregation methods were using simplistic mathematical functions in order to output the final result. For example, *voting* is a method through which the result of the aggregation is the label which has been predicted (i.e. "voted") by the majority of the classifiers. Stacking, however, uses a more complex model: instead of manually defining the result of the aggregation, it uses a machine learning model and lets it figure out what is the best course of action when combining the results. The steps in constructing such a model, are, as follows:

1. Train the initial classifiers. In the context of this application, the initial classifiers are the ones which independently make predictions for cover images and book titles. These initial classifiers are often called *first-layer classifiers*.

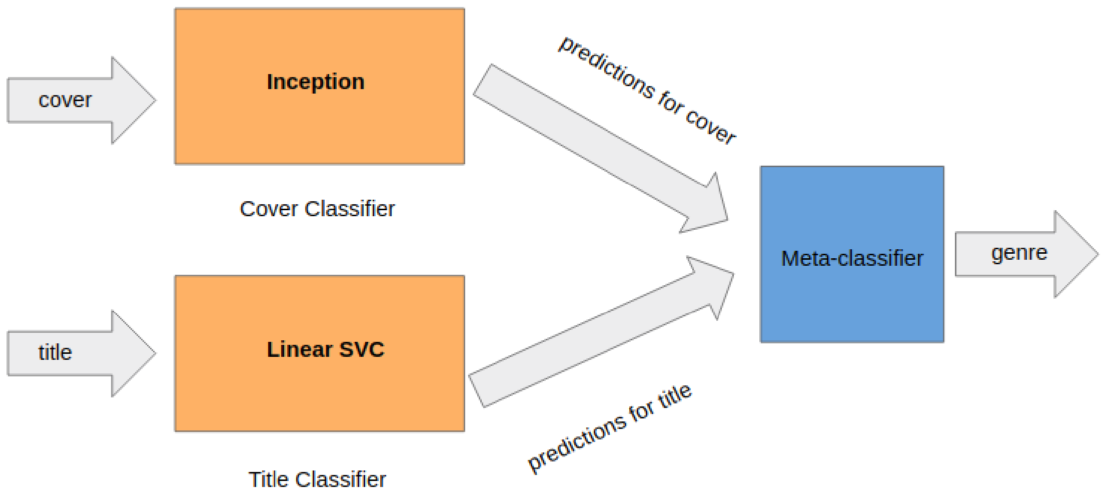


FIGURE 3.15: The stacking model proposed in this paper.

2. Using the trained classifiers from step 1, train the *meta-classifier*. The meta-classifier is the model used to aggregate the results from first-layer classifiers in order to produce the final result. The training is done on a different set other than the one used to train the first-layer, called *hold-out set*. Therefore, stacking introduces a bit of over-head when it comes to training, due to the need of two separate training sets.

Figure 3.15 illustrates the architecture of the stacking model proposed in this application. Inception and Linear SVC are the first-layer classifiers of this model. In order to aggregate the results, a *Random Forest* is used as the meta-classifier.

3.3.1 Decision Trees

In order to explain what Random Forests are, we first need to give a definition and describe the classification methods of *Decision Trees*.

A decision tree is probably one of the most intuitive and easy to grasp models in machine learning. Perhaps the best way of understanding the underlining of decision trees is through a visual manner.

Suppose we have the simple decision tree illustrated in Figure 3.16. The leaf nodes decide the classes in which the inputs will be classified: "Cookbooks", "History"

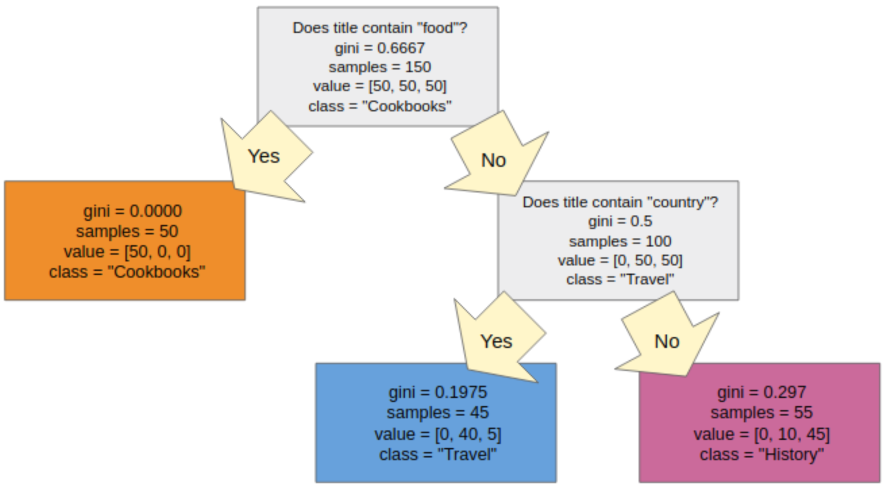


FIGURE 3.16: A simple decision tree for classifying book titles into three genres.

and “Travel”. If we were to classify a book with the title “Good food, good mood”, we would start from the root node and ask the following question: “Does the title contain “food”?”. The answer is ”Yes” so we move forward to the left child of the root node. There are no more questions left, therefore, the final prediction is the class of the leaf node, i.e., “Cookbooks”. What about the title “History of Japan”? It does not contain the word “food” so we move along the right child. We are then faced with another question: “Does the title contain “country”?”. The answer is yet again no. We finally reach a leaf node and the result is “History”.

The prediction process of a decision tree is very intuitive and easy to follow. The computational complexity of it is small, equal to $O(\textit{depth of the tree})$. Usually, decision trees tend to be close to balanced[28], therefore the computational complexity is $O(log_2(n))$, where n is the number of nodes in the tree.

Let’s move on to explaining what the values in each node mean:

- *a question (or equation with a True or False result)*, which is contained in all non-leaf nodes. It serves as the decision function for moving down the tree.
- *samples*, which measures how many instances in the training set apply to the current node. For example, in Figure 3.16, for the left-most node, a total of 50 titles from the 150 in the training set contain the word “food”.

- *value*, which measures how many training instances from each category apply to the current node. In the case of the aforementioned node, all instances that apply to it have the class “Cookbooks”, therefore *value* will be equal to [50, 0, 0].
- *gini*. The term *gini* comes from the name of the loss function used in training. It signifies the *impurity* of the node: the higher the gini value, the higher the impurity. For example, the left-most node has a gini value of 0 and is considered completely pure due to the fact that all samples it “contains” have the same category. The least pure node is the root node: it contains a uniform distribution of instances of all classes.
- *class*, which specifies the category of the node (in our case, category is equivalent to genre). In the case of leaf nodes, the class will be used as the final prediction. The class label in intermediary nodes is not equivalent to a prediction.

3.3.2 The Gini Loss Function

The loss function used by decision trees is called *gini*. It is applied per node and its mathematical formula for the i-th node is[28]:

$$gini_i = 1 - \sum_{k=1}^{n} {p_{i,k}}^2$$

$p_{i,k}$ represents the ratio of instances in the i-th node that are a member of the k-th class, while n stands for the total number of classes.

In order to better understand how the formula is applied, let’s compute the gini value of the right child of the root node, which is the third node in the tree (see Figure 3.16).

$$p_{3,1} = \frac{0}{100} = 0$$

$$p_{3,2} = \frac{50}{100} = 0.5$$

$$p_{3,3} = \frac{50}{100} = 0.5$$

$$gini_3 = 1 - (0^2 + 0.5^2 + 0.5^2) = 1 - (\frac{1}{4} + \frac{1}{4}) = 0.5$$

3.3.3 CART Training Algorithm

The training algorithm used in this application in order to build decision trees and minimize the gini loss function is called *CART (Classification and Regression Tree)*. The algorithm works as follows: find a feature k and a threshold t_k such that the weighted average between the gini value of the set to the left of the threshold and the gini value of the set to the right of the threshold is minimum. The mathematical representation of the aforementioned requirement is[28]:

$$J(k, t_k) = \frac{m_{left}}{m} gini_{left} + \frac{m_{right}}{m} gini_{right}$$

m stands for the total number of instances in the training set, while m_{left} and m_{right} stand for the number of instances in the training set that belong to the left part of the threshold and to the right part respectively. $gini_{left}$ and $gini_{right}$, quite predictably, represent the gini values of the “left” and “right” sets. This splitting operation is repeated until the depth has reached a limit (max_depth) or if a subsequent split would not reduce the impurity of the current node.

3.3.4 Random Forests

A Random Forest is an ensemble model based on decision trees which was proposed in an eponymous paper in 2001[16]. It trains a specific number of decision trees on different subsets of the training data and outputs the aggregation of the results. The training data is split using *bagging*. Suppose we had exactly 5 decision trees and needed to create 5 training subsets for each tree. The algorithm which we would following is:

1. Create a new training set which is initially empty.
2. Select a random instance from the training set and add a copy of it to the subset.
3. Repeat the second operation until the size of the subset is equal to the size of the training set.
4. Repeat the first operation until a total of 5 subsets have been created.

One might notice that a subset is likely to contain duplicates of instances. However, this isn't a consequence of bad design. The subsets are going to have a slightly different distribution with respect to classes, however, when aggregating the results from the each decision tree, the results are likely to be a better estimate of the true population as opposed to the results from using only the original training set[16].

The training algorithm for a Random Forest is pretty straight forward: each decision tree in its composition is trained on its allocated subset. However, one important change in the CART algorithm that is used in this application is the following: *"when splitting a node during the construction of the tree, the split that is chosen is no longer the best split among all features. Instead, the split that is picked is the best split among a random subset of the features."*[45]. This increases the bias of the sole decision tree, however, when aggregating the results, this leads to a better model.

The aggregation method used in the `scikit` package is to average the probabilities for each class resulted from decision trees. Decision trees can indeed return probabilities of membership instead of a final prediction. The way these probabilities are computed is the following. Let's take the previous example from Figure 3.16 and compute the probabilities of membership for the input title "History of Japan". First, we follow the same prediction algorithm which has been previously discussed and we reach the purple leaf node. The probabilities of membership which will be returned are exactly $p_{i,k}$ from the gini mathematical formula. Therefore, the probability of "History of Japan" belonging to the genre:

- “Cookbooks” is $\frac{0}{55} = 0.00$
- “Travel” is $\frac{10}{55} = 0.18$
- “History” is $\frac{45}{55} = 0.82$

Suppose we had another decision tree alongside the one in Figure 3.16 which had returned the following probabilities: 0.10, 0.00, 0.90. The final probabilities returned by the Random Forest model would be equal to: 0.05, 0.08, 0.87.

As a final conclusion of this chapter, we will re-iterate over the model used in this paper. The first-layer classifiers are Inception (a CNN architecture) and a Linear SVC. They are used for classifying cover images and book titles respectively. The probabilities from the first-layer are passed through a meta-classifier, which is an instance of Random Forests. The meta classifier will finally return as output the genre of the input.

Chapter 4

Application

The model presented in Chapter 3 was implemented using Python. Python is a high-level, interpreted programming language which has a lot of support for Machine Learning and Data Science, making it the obvious choice when deciding for the programming language of this application. The source code in this application can be divided into several categories:

- code used for gathering, organizing and splitting the data
- code used for training, calibrating and testing the machine learning models
- code used for gathering information about the performance of humans at classifying books into genres
- code used for making the ensemble system usable (through the CLI API and through a mobile app on Android)

This chapter will present the details of each item above, while also providing user guides and the evaluation results of the system.

4.1 Data

The database containing all training and test data used in this application originates from a GitHub repository[14] called *book-dataset*. I considered the size of the database to be adequate for the purposes of this application, therefore, the file `book32-listing.csv` became the main source of information for Genrifier.

4.1.1 Data Gathering and Preprocessing

The database is a CSV (comma separated value) file containing the following details about each book: the ASIN (Amazon Standard Identification Number), the URL of the front cover of the book, the title, the author and the category (genre) in which it has been classified on Amazon. In total, there are 207,572 books in this dataset.

The books in this dataset are classified in 32 genres. However, for the purposes of this application, not all genres have been taken into consideration. The reason behind this is that a book can generally be described using multiple labels from the 32 existing ones. A good example is "Harry Potter", a famous book written by J. K. Rowling. This book could be classified as "Children's Book", "Fantasy" or even "Young Adult Fiction". Therefore, it is crucial to select a specific set of genres such that the books in it are easily identifiable and non-interfering with other labels.

Following in the footsteps of the authors of the paper *Can you Judge a Book by its Cover*[47], I decided to limit the number of genres to 10. The previously cited paper decided to retain books from the following categories: *Children's Books, Comics & Graphic Novels, Computers & Technology, Cookbooks, Food & Wine, Romance, Science & Math, Science Fiction & Fantasy, Sports & Outdoors, Test Preparation* and *Travel*, considering them to be representative of the dataset and non-interfering. Some modifications were, however, done to this list. After some experimentation with the Guessing Game, it was observed that *Children's Books* and *Comics & Graphic Novels* were often confused and, thus, I have decided to remove the former from the dataset. *Test Preparation*, however crucial in the lives of students, I considered to not be popular enough for today's current trend in books and decided to removed it from the list as well. *History* and *Self Help* were subsequently added to the list of classes. The reasons behind choosing these two genres were that they are hardly confused with other genres in the list, while continuing to be popular enough today.

Another two changes were made to the aforementioned list of genres. After subsequent experimentation with the Guessing Game, it was later found that *Sports & Outdoors* and *Travel* were frequently confused with each other so I decided to merge them into an all encompassing genre entitled *Outdoor Activities*. *Computers & Technology* and *Science & Math* were in a similar position, therefore, they were merged into a class named *Science & Technology*. This leads us to a final list of 8 genres. The following table lists them in alphabetical order, along with the number of books in each class.

Genre	**No. books**
Comics & Graphic Novels	3,026
Cookbooks, Food & Wine	8,802
History	6,807
Outdoor Activities	24,303
Romance	4,291
Science & Technology	17,255
Science Fiction & Fantasy	3,800
Self-Help	2,703
Total	**70,987**

Table 4.1: `book8-listing.csv` - a distribution with respect to genres

While experimenting with the Guessing Game, some covers have been found to be missing and were deleted manually from the database. Some titles contained non-ASCII characters and I have decided to remove those instances in order to facilitate the process of text classification. The final dataset resulted from these changes has the following distribution with respect to genres.

Genre	**No. books**
Comics & Graphic Novels	3,011
Cookbooks, Food & Wine	8,693
History	6,697
Outdoor Activities	23,972
Romance	4,269
Science & Technology	17,078
Science Fiction & Fantasy	3,782
Self-Help	2,687
Total	**70,189**

TABLE 4.2: `book8-listing_no_unicode.csv` - a distribution with respect to genres.

The datasets were stored in 2 CSV files:

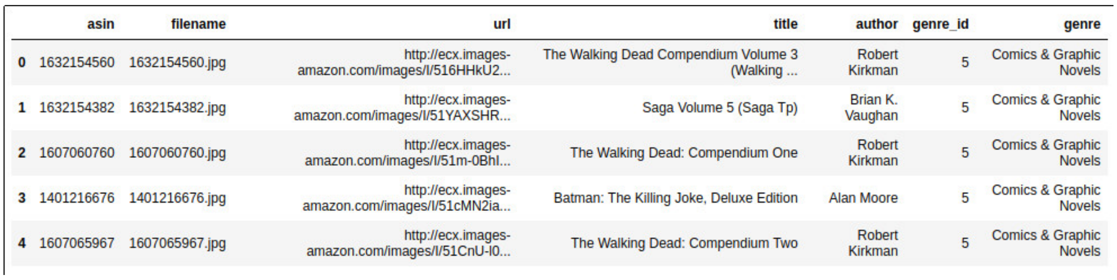
- `book8-listing.csv`
- `book8-listing_no_unicode.csv`

In order to process a dataset, one can easily load them in main memory using the `pandas` module in Python. For example, Listing 4.1 shows how one can load the `book8-listing_no_unicode.cvs'` dataset into a variable called `books`.

```python
import pandas
books = pandas.read_csv('book8-listing_no_unicode.csv')
```

LISTING 4.1: Load dataset in memory

The result of this command is a data structure called `DataFrame`. According to the official documentation from `pandas`, a `DataFrame` is a *"Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be*



	asin	filename	url	title	author	genre_id	genre
0	1632154560	1632154560.jpg	http://ecx.images-amazon.com/images/I/516HHkU2...	The Walking Dead Compendium Volume 3 (Walking ...	Robert Kirkman	5	Comics & Graphic Novels
1	1632154382	1632154382.jpg	http://ecx.images-amazon.com/images/I/51YAXSHR...	Saga Volume 5 (Saga Tp)	Brian K. Vaughan	5	Comics & Graphic Novels
2	1607060760	1607060760.jpg	http://ecx.images-amazon.com/images/I/51m-0BhI...	The Walking Dead: Compendium One	Robert Kirkman	5	Comics & Graphic Novels
3	1401216676	1401216676.jpg	http://ecx.images-amazon.com/images/I/51cMN2ia...	Batman: The Killing Joke, Deluxe Edition	Alan Moore	5	Comics & Graphic Novels
4	1607065967	1607065967.jpg	http://ecx.images-amazon.com/images/I/51CnU-I0...	The Walking Dead: Compendium Two	Robert Kirkman	5	Comics & Graphic Novels

FIGURE 4.1: The first 5 books in the dataset `book8-listing_no_unicode.csv`

thought of as a dict-like container [...]. The primary pandas data structure."[22]. In order to get a feeling of how the data is represented in a `DataFrame`, Figure 4.1 shows the top 5 books in the dataset `book8-listing.csv'`.

The next step in having the entire dataset stored on disk is to download the book covers. In order to accomplish this, I wrote a Python function which iterates through every item in a `DataFrame` and proceeds to download the contents of the book's URL. The file will be stored on disk with the exact `filename` which appears in the `DataFrame`. This operation is done in a parallel fashion, using Python's `ThreadPoolExecutor`. The operation took several hours to complete and the total size of the resulting directory is 3.1 GB.

4.1.2 Data Storage

For the main purposes of this application, the previously explained method of dividing and storing the data was enough. However, several additional modifications had to be made in order to fulfill the needs of third-party frameworks.

The `retrain.py`[41] script provided by TensorFlow developers requires the images to be divided into class-specific directories. The framework detects which images belong to which category by checking the name of the directory they are in. This was, yet again, done through a Python script. Listing 4.2 shows the exact function which was used to divide images in folders. First, the genre directories were created, in case they didn't exist already (lines 2-5). Afterwards, the books are iterated one by one: for each instance the source file path is computed, along with its destination folder. The module `os` is used to create directories, to check for file existence and to resolve file paths. `shutil` provides us with function `shutil.copy(src, dest)` which, with its easy to use API, helps us copy a file from a source to a destination.

```python
import shutil
import os

def organize_in_genre_folders(books):
    for genre in consts.GENRES:
        directory = os.path.join(consts.DIR_IMAGES, genre)
        if not os.path.exists(directory):
            os.makedirs(directory)

    for index in range(len(books)):
        book = books.iloc[index]
        filename = book.filename
        src = os.path.join(consts.DIR_IMAGES, filename)
        dst = os.path.join(consts.DIR_IMAGES, book.genre)
        shutil.copy(src, dst)
```

Listing 4.2: Python function which organizes images into class-specific directories

Facebook's FastText[24] library for text classification required a specific way of storing text alongside labels in order to use its API. The requirements are as follows: the input of the classifier is a CSV file; on each line of the CSV file, there is a data instance; the fist item on each line represents the class index; the second item of each line represents the text associated to the class index. Before feeding this CSV file as input to the classifier, the data has to be processed. The bash script (`prepare_input.sh`) used to generate the train and test sets for FastText is adapted from `classification-example.sh` on FastText's GitHub repository[25]. It helps us process the data so that several useless characters (mostly non-letter, non-digit characters) are removed, all letters are lower-cased, spaces are added where needed in order to facilitate the classification and the data is shuffled.

4.1.3 Data Consistency

Several functions in the `utils.py` script were created with the scope of assuring data consistency. Initially, all images from `book32-listing.csv` were downloaded, in order to experiment with the Guessing Game and figure out how good are humans at classifying books into a large set of classes. However, as modifications were made and books from a total of 22 genres were no longer needed, a clean-up of the directory containing all cover images was a good next step in order to retain only the necessary data for training and testing. Python functions were therefore written in order to delete unnecessary images. Similarly to Listing 4.2, the script iterated through all the books in a `DataFrame` and proceeded to delete the file associated to books which did not have appropriate genres. However, as opposed to the sequential manner in which Listing 4.2 was implemented, `remove_images_of_genre(books, genre)` made use of `ThreadPoolExecutor` in order to fasten the deletion process.

In order to make sure that all images have been downloaded, I wrote a simple `Bash` script (Listing 4.3) which counts the number of JPG images in each genre sub-directory and computes the total number of images. In this fashion, it is possible to make sure that all images have been downloaded and that the distribution of books with respect to genres is correct.

```bash
#!/bin/bash
counter=0
for d in */; do
  jpgs_in_dir=`ls -l "$d" | grep ".jpg" | wc -l`
  echo "Number of images in $d is: $jpgs_in_dir"
  counter=$(($counter + $jpgs_in_dir))
done
echo "Total: $counter"
```

Listing 4.3: The Bash script used to count the number of images in each genre directory

```
iulia@eagle $ ./counter.sh
Number of images in Comics & Graphic Novels/ is: 3026
Number of images in Cookbooks, Food & Wine/ is: 8802
Number of images in History/ is: 6807
Number of images in Outdoor Activities/ is: 24303
Number of images in Romance/ is: 4291
Number of images in Science Fiction & Fantasy/ is: 3800
Number of images in Science & Technology/ is: 17255
Number of images in Self-Help/ is: 2703
Total: 70987
```

Listing 4.4: Sample usage of the script from Listing 4.3

The downside to this approach is that, if an image is missing, it would be extremely hard to figure out which titles have a missing cover image. Therefore, a Python function was written with the purpose of finding the culprits. `sanity_check(books)` from `utils.py` iterates through all the books in a `DataFrame`, computes the file path of the cover of a book and checks, using a helper function from the `os` module, whether that specific path exists or not. In case of a mismatch (i.e. the cover of the book does not exist on disk), both the file path and the title of the book are printed on the screen.

4.1.4 Data Statistics

While the distribution of books with respect to genres was already discussed in a previous section, some additional information about the data was needed in order to better understand the dataset that we are working with.

4.1.4.1 Title statistics

I made use of the excellent API of the `pandas` module, and started exploring the titles through `pandas.Series.describe()`. This functions "generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values."[40]. The output for the dataset

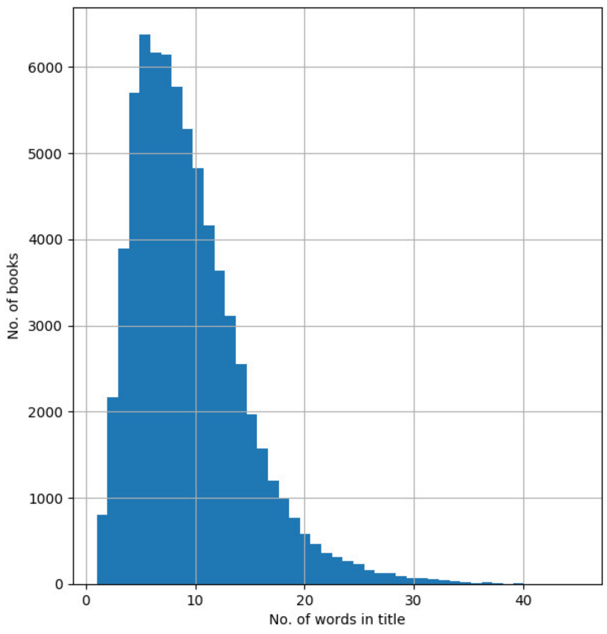


FIGURE 4.2: The distribution of titles with respect to the number of words.

`book8-listing_no_unicode` is found in Listing 4.5.

```
count    70167.000000
mean         9.137629
std          5.247917
min          1.000000
25%          5.000000
50%          8.000000
75%         12.000000
max         42.000000
```

LISTING 4.5: Statistics for the number of words in book titles

As shown above, the average number of words in a title is a bit over 9 words. The vast majority of words (75%) have under 12 words per title. There also exist some extremely long titles(see Figure 4.3), the biggest one having almost 5 times the number of words compared to the average-length title. The distribution of titles with respect to the number of words is shown in Figure 4.2.

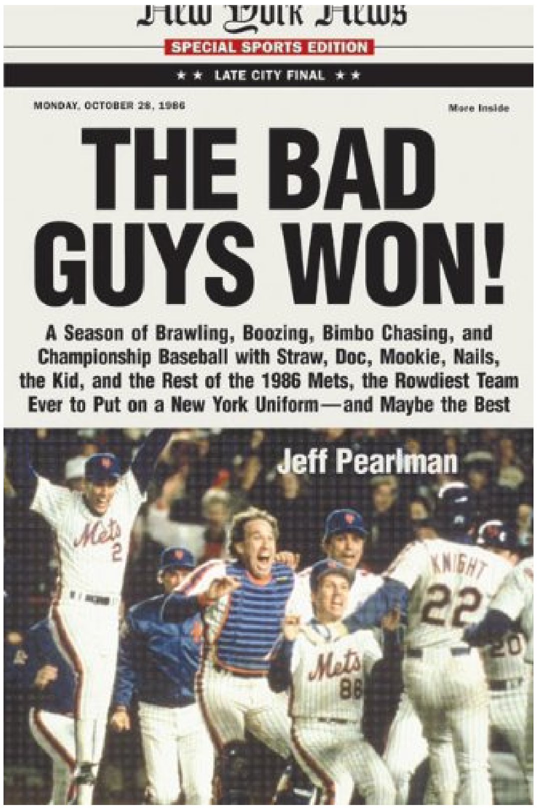

New York News
SPECIAL SPORTS EDITION
★ ★ LATE CITY FINAL ★ ★
MONDAY, OCTOBER 28, 1986
More Inside
THE BAD GUYS WON!
A Season of Brawling, Boozing, Bimbo Chasing, and Championship Baseball with Straw, Doc, Mookie, Nails, the Kid, and the Rest of the 1986 Mets, the Rowdiest Team Ever to Put on a New York Uniform—and Maybe the Best
Jeff Pearlman

FIGURE 4.3: The book with the longest title in the dataset.

4.1.4.2 Cover statistics

When exploring the images contained in the dataset, I decided to look for the distribution of images with respect to the number of pixels that they have. First, I iterated through all images and computed the number of pixels. Then added them in a `pandas.Series` object and used the same `describe()` function that I used before on titles. The results were concerning.

```
count    7.018900e+04
mean     1.719783e+05
std      5.460568e+04
min      1.000000e+00
25%      1.620000e+05
50%      1.665000e+05
75%      1.905000e+05
max      8.061440e+06
```

LISTING 4.6: Statistics for the number pixels in book covers in `book8-listing_no_unicode.csv`

As shown in the Listing 4.6, the tiniest image has one pixel. As opposed to titles, where it is perfectly acceptable for a book to have a one-word title, an image is completely useless with exactly one pixel in its composition. I decided that a good limit for "when a book cover would be identifiable and analyzable by humans" would be an image with at least 10,000 pixels and therefore deleted the instances that didn't meet this requirement. Fortunately, the number of books in `book8-listing_no_unicode.csv` that were deleted was exactly 22 so not a lot of books had this particular problem. After performing this change, I saved the resulting dataset in a file called `book8-listing-final.csv`. This file was the final, curated, dataset and was then used to train and test the Machine Learning model used in this application. The final distribution of images with respect to the number of pixels is shown in Listing 4.7.

```
count    7.016700e+04
mean     1.720277e+05
std      5.453961e+04
min      1.008000e+04
25%      1.620000e+05
50%      1.665000e+05
75%      1.905000e+05
max      8.061440e+06
```

Listing 4.7: Statistics for the number pixels in book covers in the final dataset

The final distribution of books with respect to genres is listed in Table 4.3.

4.1.5 Splitting the Data in Training and Testing sets

Splitting the data correctly in machine learning is essential towards building a good system. For example, one perfectly valid way to split a 10k-instance dataset into a 70% training set and a 30% test set is to choose 7k random books and assign them to the training set and then save the rest into the test set. However, one major problem with this approach lies in the distribution of classes within each set. Consider, for example, the unlucky occurrence of assigning every single instance of a class to the test set. This is the worst case scenario and in most

Genre	**No. books**
Comics & Graphic Novels	3,011
Cookbooks, Food & Wine	8,693
History	6,697
Outdoor Activities	23,954
Romance	4,269
Science & Technology	17,075
Science Fiction & Fantasy	3,781
Self-Help	2,687
Total	**70,167**

Table 4.3: Distribution with respect to genres in `book8-listing-final.csv`

splits this would not happen due to the law of randomness. However, it would often be the case that the distribution of classes in the training set would wildly differ from the one in the test set, thus leading to a non-representative accuracy on the test set.

In order to assure ourselves of the fact that the distribution of classes is consistent across sets, we need to split the data taking into consideration the percentage of books from each class. Suppose that in a 10k-instance dataset, 20% of the books would be of a genre named *History*. The best way to split the original data would be to assure ourselves of the fact that the percentage of *History* books in each of the resulted sets is close to 20%. This is easily accomplished through a class provided by the Python module, `sklearn`. `StratifiedShuffleSplit` "returns stratified randomized folds. The folds are made by preserving the percentage of samples for each class" [51]. Listing 4.8 contains a function written in Python for splitting a `DataFrame` in two sub-sets while preserving the distribution of books with respect to genres.

```python
def split_set(books, second_set_size):
    split = StratifiedShuffleSplit(n_splits=1, test_size=second_set_size, random_state=42)
    for first_index, second_index in split.split(books, books['genre']):
        first_set = books.iloc[first_index]
        second_set = books.iloc[second_index]

        print('first set stats')
        print(first_set.genre.value_counts() / len(first_set))

        print('second set stats')
        print(second_set.genre.value_counts() / len(second_set))

    return first_set, second_set
```

LISTING 4.8: Splitting a `DataFrame` in 2 subsets using `StratifiedShuffleSplit`

For example, if we were to split `book8-listing-final.csv` into 70% and 30% sub-sets, the output generated from running this functions is in Listing 4.9.

```
first set stats
Outdoor Activities          34.138893
Science & Technology        24.333734
Cookbooks, Food & Wine      12.388786
History                      9.544557
Romance                      6.083433
Science Fiction & Fantasy    5.389173
Comics & Graphic Novels      4.291793
Self-Help                    3.829631
[...]
second set stats
Outdoor Activities          34.140896
Science & Technology        24.336136
Cookbooks, Food & Wine      12.388960
History                      9.543490
Romance                      6.085222
Science Fiction & Fantasy    5.386917
```

```
Comics & Graphic Novels          4.289582
Self-Help                        3.828797
[..]
```

Listing 4.9: Output from splitting a set in 70% and 30% stratified sub-sets

As seen in the output from Listing 4.9 the distribution of books with respect to genres is preserved across the sub-sets. Due to the fact that the application proposed in this thesis uses stacking as the ensemble method to unify the results from the cover and title classifiers, a total of 3 sub-sets need to be created:

- `training-ensemble.csv` consists of a stratified split equal to 70% of the original data; it will be used to train the title and cover classifiers
- `hold-out-ensemble.csv` consists of a stratified split equal to 18% of the original data; it will be used to train the stacking classifier
- `test-ensemble.csv` consists of s stratified split equal to 12% of the original data; it will be used to test the final system

4.2 Choosing the model

Choosing the model was one of the most time consuming tasks of this project. Due to the fact that the goal of Genrifier is to build a genre-classification system as accurate as possible, multiple models have been tested for each classifier. Each model was regularized and tweaked with in order to improve its accuracy before yielding the final test accuracy. In the following subsections, we will discuss how the model for each classifier was chosen, along with its final accuracy.

4.2.1 Book Cover Classifier

One huge problem in classifying book covers into genres lies in their diversity. Image classification usually consists of identifying a certain entity in a photo: be

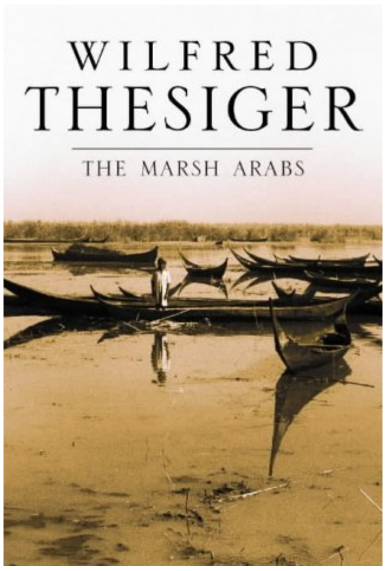


FIGURE 4.4: A book in the genre "Outdoor Activities"

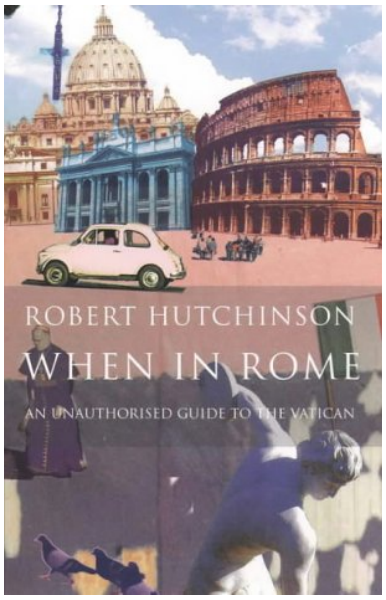


FIGURE 4.5: Another book in the genre "Outdoor Activities"

that a dog, a plane or a specific person. However, when it comes to classifying book covers into various genres, this becomes a fairly hard thing to accomplish due to the fact that they do not necessarily have a pattern. Taking a look at Figures 4.4 and 4.5, we find almost no similarity between the 2 covers: the colors are different, the landscape is different, etc. The hardship of classifying book covers into genres is noticed in the paper "Can you Judge a Book by its Cover?": the highest accuracy that they were able to obtain was 59%, with an extremely complicated, state of the art, model[47]. Therefore, when it comes to book cover classification, I decided to follow suit and use a preexisting system instead of building a less performing model from scratch. The aforementioned paper used a variation of something called *ResNet50*, one of the best ANNs in the field of image recognition. This paper proposes Inception V3, as discussed in Chapter 3. There are a total of 4 complex architectures which are re-trainable on TensorFlow: Inception, versions 1 to 3, and a combination between Inception and ResNet called "Inception-ResNet"[31]. Out of all 4 architectures, Inception V3 was the best performing one and was therefore used in this application.

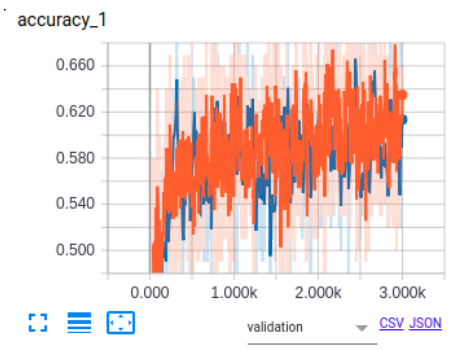
There are several parameters which can be tweaked in order to regularize Inception. `num_iterations` is probably the most influential parameter that the retrain script has: it specifies the number of iterations of the training algorithm. The default value is 4000. Several values have been tested, in order to figure out the best possible outcome. The following table lists the validation accuracy along with the training time for each number of iterations:

Num. iterations	**Validation accuracy**	**Training time**
3000	56.5%	5m37s
4000	58.0%	7m37s
5000	57.6%	8m6s
6000	57.9%	10m16s
7000	57.9%	11m37s
9000	56.6%	13m59s

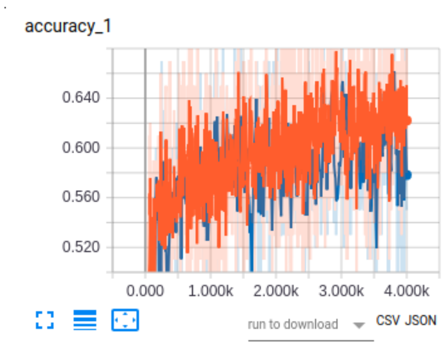
Table 4.4: Validation accuracy with respect to the number of iterations.

4000 is, indeed, the optimal number of iterations judging from the table above. While for 6000 and 7000 iterations the model reaches an accuracy close to 58%, it does not last: the 9000 iterations accuracy clearly shows that further increasing the number will lead to over-fitting instead of a better model. This is more clearly seen in Figure 4.6. The orange lines represent the training accuracy, while the blue lines represent the validation accuracy.

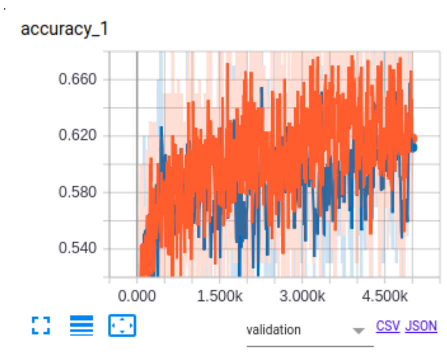
The orange lines have a clear upwards tendency: as the number of iterations grows, the training accuracy reaches higher values. However, as the training accuracy gets higher, the validation lines remain relatively consistent. They do not follow the same upward trend. A visual cue of this contrast between upwards and constant is best seen comparing the figure for 4000 iterations with the figure for 9000 iterations. In the first figure, the blue lines are barely visible, being in trend with the training accuracy. However, in the second figure, the blue lines are clearly visible, almost appearing to be a "shadow" of the orange lines. This is the result of over-fitting: the model gets better and better at classifying instances from the training set, meaning that it will fail to generalize and will perform worse on unseen instances.



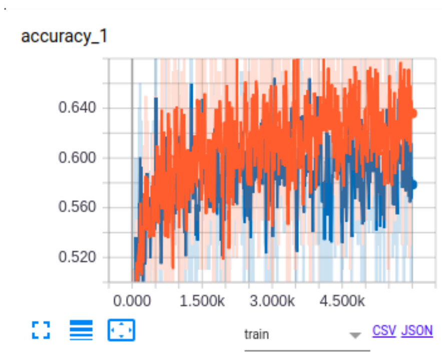
(a) 3000 iterations



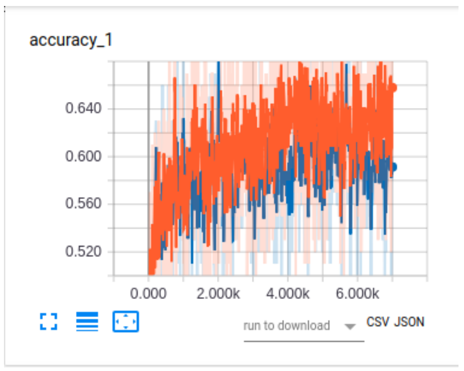
(b) 4000 iterations



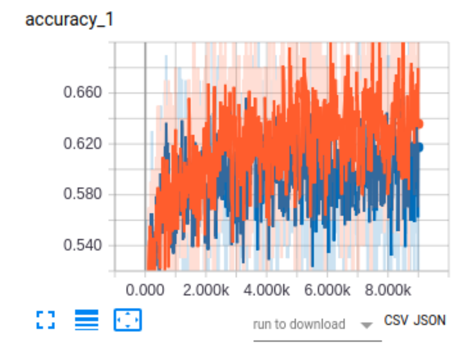
(c) 5000 iterations



(d) 6000 iterations



(e) 7000 iterations



(f) 9000 iterations

FIGURE 4.6: The training and validation accuracy with respect to the number of iterations.

This application uses the 4000 iterations model due to the fact that it has the highest validation accuracy. Running the model on the test set, we obtain **58.2%** accuracy, which is very close to the validation accuracy.

4.2.2 Book Title Classifier

As opposed to classifying book covers in genres, classifying book titles is a far more realizable task. *"Can you Judge a Book by its Cover?"*[47] obtained a relatively high accuracy, on a much smaller training subset than the one used in this application: 76%. In total, 4 different models have been tested: Multinomial Naive Bayes, Linear Support Vector Machines, FastText and Word2Vec with Convolutional Neural Networks. We will give a brief explanation of each classifier along with their results in the following subsections.

4.2.2.1 Multinomial Naive Bayes

Multinomial Naive Bayes is a classification method from statistics which is frequently used in text classification. It computes the probabilities of membership for the input, relative to the frequency of appearance of each word in each class. For example, suppose we wanted to classify a book title consisting of a single word: "Japan". "Japan" frequently appears in "Outdoor & Activities" compared to the other genres and therefore the title will be classified in the aforementioned genre. The full computation of the probabilities is not as simple as previously discussed, however, explaining the underlings of this method is beyond the purpose of the this paper, given that this model was not chosen to be part of the final ensemble system.

Using `sklearn`'s `GridSearchCV`[49], the best parameters for the model were identified. According to its documentation, `GridSearchCV` performs an *"exhaustive search over specified parameter values for an estimator"*. Not only does this class test how the model performs with each combination of parameters, but it also performs *cross-validation*. Cross-validation is a method through which we compute a model's validation accuracy without having to create a separate set. The algorithm for performing cross validation is the following:

1. Split the training set in 2 different subsets: the bigger one will be used as the training set and the other will be used as the validation set.
2. Train the model using the aforementioned set and test it on the remaining subset.
3. Merge the subsets in order to recreate the original training set.
4. Repeat operations 1-3 for a certain amount of times.
5. Compute the final validation accuracy by averaging all test accuracies obtained before.

The reason behind using a validation set is that it is bad practice to regularize a model judging from its test accuracy. If we were to tweak parameters and base our decisions on how good they perform on the test set, we will be "over-fitting" on the test set. The resulting accuracy will no longer be a good indicator of how the model performs on unseen data instances, due to the fact that it will present an optimistic value which is lower in reality. Using cross-validation we can regularize and tweak the parameters on a separate set while still getting an accurate value of how well the system performs on completely new data.

After several tweaks, the best model that I could find using `sklearn`'s `MultinomialNB` classifier has a cross-validation accuracy of **82.5%**. This value can be obtained through calling `run('mnb')` in the script `find_best_title_classifier.py`.

4.2.2.2 Linear Support Vector Machines

Linear SVCs have been previously discussed at large in Chapter 3, however, no concrete results of how well this model performs have been given. Using yet again `GridSearchCV`, we have determined what the best parameters are and what the cross-validation accuracy is for `sklearn`'s `LinearSVC`: **83.5%**. This value can be obtained through calling `run('svc')` in the script `find_best_title_classifier.py`.

The final accuracies of these two models are presented below. `LinearSVC` is the winner in all categories and is the current favorite in the quest for finding the best title classifier.

Model	**Training acc.**	**Validation acc**	**Test acc.**
`MultinomialNB`	96.2%	82.5%	84.3%
`LinearSVC`	99.6%	83.5%	85.9%

Table 4.5: The accuracies of Multinomial Naive Bayes and Linear Support Vector Machine.

4.2.2.3 FastText

FastText is a library created by Facebook's AI Research Lab[24]. It is an unsupervised model which is used for obtaining vector representations and classifying text. It is a very easy to use framework. First we need to prepare that data, just like it was explained in the previous section. Then can train a classifier without writing any code whatsoever. We just run a command in the terminal and we are good to go:

```
./fasttext supervised -input data/genrifier.train -output genrifier
```

This will produce a file named `genrifier.bin` which can further be used to compute new predictions. The training algorithm is, as the name of the library promises, fast. It takes only 3.2 seconds! Testing the model is done, yet again, through a command:

```
./fasttext test genrifier.bin data/genrifier.test
```

The results vary due to randomness. Five subsequent runs gave the following test accuracy: 84.3, 84.0, 84.2, 83.9, 84.1. Therefore, we can compute the average of these values and use that as the final test accuracy: **84.1%**. `LinearSVC` is still the favorite in the battle of text classifiers.

4.2.2.4 Word2Vec with CNN

The last model I investigated was a complex, state of the art method which uses *word embeddings* and CNNs. The embedding of words is achieved using *Word2Vec*, which was created by several Google engineers[38]. It is two-layer ANN which has as input a large corpus of text. Its output is a set of vectors representing the *embedding* of each word in the input[56]. These vectors are then fed in a CNN architecture which classifies them into genres. The implementation of the CNN along with w2v contained in this application was heavily based on a similar script which performs sentiment analysis on Twitter[17]. The implementation of this architecture can be found in the file `w2v_cnn.py` and the results for training and testing can be found in `output_training_w2v`. I tried to regularize the CNN in different ways: change the number of filters in the convolutional layers, increase the number of neurons in dense layers or apply l2 regularization. However, the biggest test accuracy that I could get was **84.7%**. This was less than the value obtained using `LinearSVC` so I decided that the final model which will be used in order to classify book titles was `LinearSVC`.

The table with the final test accuracies for each model tried is presented below.

Model	Test accuracy
`MultinomialNB`	84.3%
`LinearSVC`	85.9%
`FastText`	84.1%
`w2v & CNN`	84.7%

Table 4.6: Test accuracies for all four models.

4.2.3 Meta Classifier

For the final classifier, which will be used to aggregate the predictions from the cover and title classifiers, I decided to test several classifiers from `sklearn`. We will present the results of each in the following subsections.

4.2.3.1 `LinearSVC`

Given that `LinearSVC` gave such good results on titles, I decided to investigate whether it will serve as a good meta classifier as well. Using, yet again, `GridSearchCV`, I tried out several parameters for it. The best validation accuracy that I could obtain had a value of 86.2%. The test accuracy using the best parameters has a value of **86.7%**. These values can be re-created by calling the function `grid_search_linear_svc()` in the script `find_best_meta_classifier.py`.

4.2.3.2 `SGDClassifier`

Another model that I investigated was `SGDClassifier`. This a simple model which, depending on the problem, can work surprisingly well. It uses Stochastic Gradient Descent to minimize a loss function. The difference behind a Stochastic Gradient Descent and a normal Gradient Descent is that it uses a *random batch* of instances from the training set per iteration, instead of using the whole training set. The reasons behind doing this is that on large training sets, a normal Gradient Descent might become computationally costly. Using a subset of the training set leads to approximately the same result while streamlining the algorithm. The best validation accuracy I could obtain with this model was 86.0% and the test accuracy using the best parameters was **86.7%**, which is exactly the same as the one given by `LinearSVC`. The values can be re-created by calling `grid_search_sgd()` in the script `find_best_meta_classifier.py`.

4.2.3.3 `RandomForestClassifier`

The last model that I investigated was `RandomForestClassifier`, which was thoroughly discussed in Chapter 3. The best validation accuracy that I obtained using `GridSearchCV` was 87.0%, while the test accuracy was **87.8%**. These values can be re-created by calling the function `grid_search_random_forest()` in the script `find_best_meta_classifier.py`. A comparison between the accuracies of each model is presented in Table 4.7.

Model	Validation accuracy	Test accuracy
`LinearSVC`	86.2%	86.7%
`SGDClassifier`	86.0%	86.7%
`RandomForestClassifier`	87.0%	87.8%

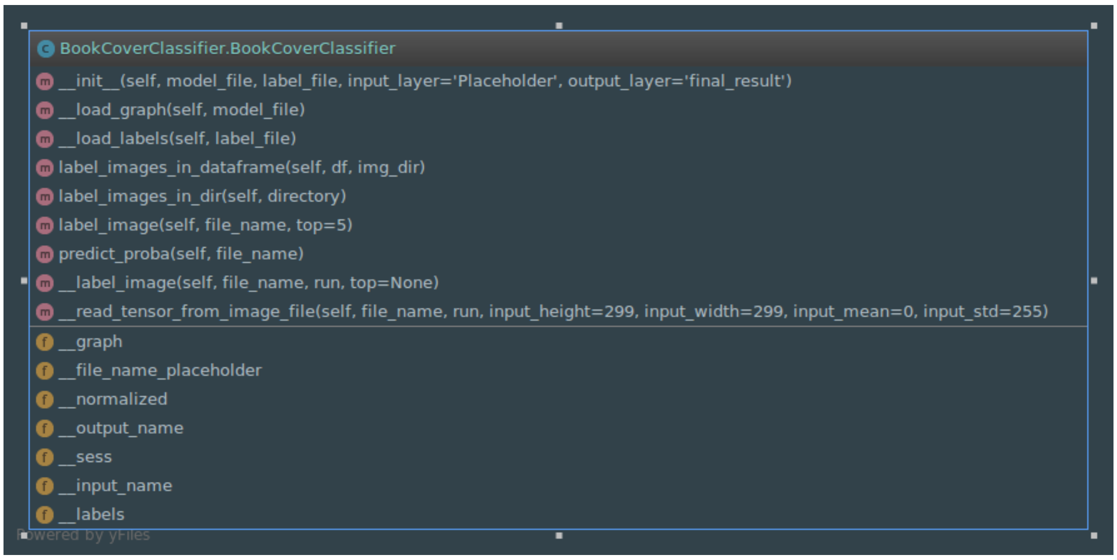
Table 4.7: The validation and test accuracies for all three models.

While all three models improved the test accuracy of the system, Random Forests performed the best and have, therefore, been used in the final model of this application. One way of making sure that the final test accuracy of the ensemble system is the value presented above, one can run the script `Genrifier.py` with the flag `--test_set`, while also specifying the directory containing all images.

4.3 `BookCoverClassifier`

`BookCoverClassifier` is the name of the class which performs the classification of cover images. This class heavily uses TensorFlow for most of its operations: reading images, making predictions, loading the CNN in memory etc. In this subsection we will describe what each of its fields represents and explain the implementation details for each method. Figure 4.7 visually presents all methods and fields of the class.

The constructor (`__init__`) receives several parameters as input: `model_file`, which is the name of the file containing the retrained Inception V3 architecture, `label_file`, which is the name of the file containing all the categories of the CNN, and lastly `input_layer` and `output_layer` which represent the name of the input layer and output layer of the CNN respectively. The constructor performs several important actions. First, it loads the graph of the retrained Inception in memory using the function `__load_graph(model_file)`. The "private" member `__graph` is the reference to the aforementioned graph. Second, it loads the category labels using the function `__load_labels(label_file)`. All labels will be added to a list, referenced by `__labels`. One very important field of the class is `__sess`. It holds



```
BookCoverClassifier.BookCoverClassifier
__init__(self, model_file, label_file, input_layer='Placeholder', output_layer='final_result')
__load_graph(self, model_file)
__load_labels(self, label_file)
label_images_in_dataframe(self, df, img_dir)
label_images_in_dir(self, directory)
label_image(self, file_name, top=5)
predict_proba(self, file_name)
__label_image(self, file_name, run, top=None)
__read_tensor_from_image_file(self, file_name, run, input_height=299, input_width=299, input_mean=0, input_std=255)
__graph
__file_name_placeholder
__normalized
__output_name
__sess
__input_name
__labels
```

FIGURE 4.7: The class diagram of `BookCoverClassifier`.

a reference to the current `Session` which executes the computations. TensorFlow works in a lazy manner. First, a graph of operations is built, then a session is created which will *perform* the actual computations. At each public operation exposed by this class, we will need to create a new `Session` in order to process the input. Due to the fact that several functions need a reference to the `Session` in order to perform some operations, I decided to hold the reference as a member of the class.

First, let's discuss the public functions `label_image` and `predict_proba`. These functions both make prediction for a single input image. The output of these functions is different: the former returns the `top` labels for the input image, along with the probability associated to them, while the latter returns only the probabilities of membership to each category. Listing 4.10 shows the returned values for each call on the same image.

`label_image` is more "user-friendly" due to the fact that it lists the name of the genres next to the probabilities and sorts them in descending order. `output_proba`, however, is used in CLI app of Genrifier: it returns the only necessary information for the ensemble system, i.e. the probabilities for each category.

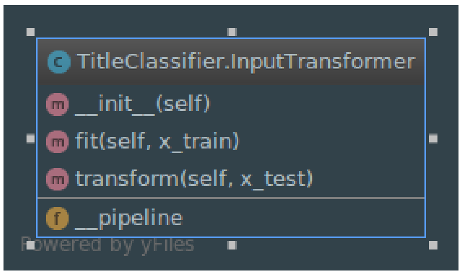
```
print(classifier.predict_proba('./trial-dir/harry_potter.jpeg'))
[[0.13144718 0.02122935 0.09364622 0.04917717 0.1547562  0.00547396
  0.52915645 0.01511344]]

print(classifier.label_image('./trial-dir/harry_potter.jpeg'), top=5)
[('Science Fiction & Fantasy', 0.52915645), ('Romance', 0.1547562),
 ('Comics & Graphic Novels', 0.13144718), ('History', 0.09364622),
 ('Outdoor Activities', 0.04917717)]
```

LISTING 4.10: The outputs for `predict_proba` and `label_image`

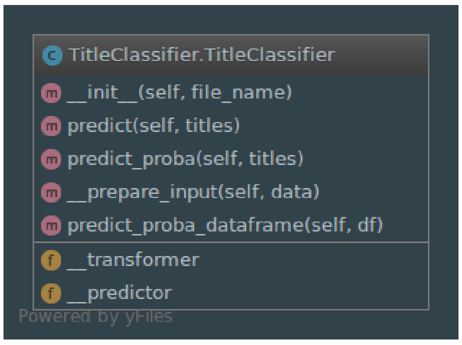
Both functions use the save private method `__label_image`, which can be seen as the core of the classifier. It does the following: first, it loads an image in memory, using `__read_tensor_from_image_file`. Then, it feeds to the input layer the values extracted from the image and runs the prediction algorithm of the CNN. The results are extracted from the output layer and are changed in order to fulfill the needs of `label_image` and `output_proba` respectively.

`__read_tensor_from_image_file` is another important method of the class. It decodes the JPG file given as input and reads all 3 channels of the images: the red, green and blue values of each pixel. The input is cast to float and normalized (i.e. each of the pixel values will be divided by 255 in order to fit each value in the range [0, 1]).

`label_images_in_dir` and `label_images_in_dataframe` make predictions for multiple images. As hinted by the name of the function, the former will process all JPG files in the given directory and return their associated probabilities. The method will return a similar output as `label_image` (i.e. the name of the genre and the probability associated to it). The latter processes all images in a `DataFrame`, a data structure previously discussed in section 4.1. The output of it is similar to `output_proba`. Both functions sequentially process each image and add the results for each instance in a list. For example, suppose that `result` contains the output of `label_images_in_dataframe` and we wish to find out what is the probability of the i-th book to be of the j-th genre. The probability could be accessed through `result[i][j]`.



(a) `InputTransformer`


TitleClassifier.TitleClassifier
__init__(self, file_name)
predict(self, titles)
predict_proba(self, titles)
__prepare_input(self, data)
predict_proba_dataframe(self, df)
__transformer
__predictor
Powered by yFiles

(b) `TitleClassifier`

FIGURE 4.8: The class diagrams of `InputTransformer` and `TitleClassifier`.

`label_image` and `label_images_in_dir` are more "user-friendly" and their best use case is for when we want to utilize `BookCoverClassifier` independently (i.e. out of the ensemble system context).

`output_proba` and `label_images_in_dataframe` are mostly used in the ensemble context, as they output raw probabilities for each input image.

4.4 `TitleClassifier`

`TitleClassifier` is the name of the class which performs the classification of book titles. It heavily uses the `sklearn` package in order to process the input data and make predictions. The input is processed using a custom made class, `InputTransformer`. In this subsection, we will describe the fields and implementation detail of both classes. Figure 4.8 contains the class diagrams of the aforementioned classes.

`InputTransformer` prepares the titles for `TitleClassifier`. It makes use of 2 classes from `sklearn`: `CountVectorizer` and `TfidfTransformer`. The first class transforms each title in a list of sparse-matrix embeddings and removes the *stop words* from the input. A stop word is a commonly used word which can be safely ignored without losing any information relevant to the classification, while saving space. In order to understand how words are embedded, let's illustrate the process on a simple example. Suppose that the we wanted to use `CountVectorizer` in order to embed the following text: “Mary has an apple. An apple a day keeps the doctor away. Mary, the doctor, has an apple”.

```
vectorizer = CountVectorizer(stop_words='english')
vectorizer.fit(['Mary has an apple. An apple a day keeps the doctor
    away. Tommy, the doctor, has an apple'])
print(vectorizer.vocabulary_)
=>
{'doctor': 3, 'tommy': 6, 'day': 2, 'apple': 0, 'mary': 5, 'away': 1,
    'keeps': 4}
```

Listing 4.11: The vocabulary resulted from using `CountVectorizer` on a simple example.

“the”, “has”, “an”, “a” do not appear in the vocabulary due to the fact that they are stop words. Each non-stop word in the input has been associated to a number, i.e., it has been *tokenized*. From now on, when `vectorizer` will be faced with the word “Tommy”, it will transform it in its token value: 6. Suppose we want to use `vectorizer` to embed the following sentence: “Tommy keeps an apple”. The output is show below, in Listing 4.12.

```
print(vectorizer.transform(["Tommy keeps an apple"]))
 =>
   (0, 0)   1
   (0, 4)   1
   (0, 6)   1
```

Listing 4.12: The output from a transformation using `CountVectorizer`.

The tuple on each line has 2 elements: the index of the transformed instance (in our case we are transforming only one instance, “Tommy has an apple”, therefore the only index which appears is 0) and the word tokens of the vocabulary that are contained in that particular instance. A total of 3 different words were identified: the word with token 0, “apple”, the word with token 4, “keeps” and the word with token 6, “tommy”. The value next to the tuple represents the number of times that particular word has appeared in the instance. For example, if the input were to contain an extra “apple”, the number next to the tuple (0, 0) would be 2.

The `CountVectorizer` used in this application has an additional modification, which is represented in the value of its parameter `ngram_range`. This parameter specifies the range of the number of words in each entry in the vocabulary. For example, in this application `ngram_range` is set to (1, 2). The entries in the vocabulary therefore contain groupings between 1 and 2 consecutive words from the input set. Listing 4.13 better illustrates how the vocabulary is constructed with `ngram_range` set to (1, 2).

```
vectorizer.vocabulary_
 =>
{'apple day': 2, 'doctor apple': 8, 'apple': 0, 'mary': 12,
 'doctor': 7, 'tommy doctor': 15, 'keeps doctor': 11, 'keeps': 10,
 'day': 5, 'mary apple': 13, 'away': 3, 'apple apple': 1,
 'tommy': 14, 'doctor away': 9, 'away tommy': 4, 'day keeps': 6}
```

Listing 4.13: The vocabulary resulted from using a `CountVectorizer` with `ngram_range` set to (1, 2).

As seen in the Listing above, the vocabulary contains groupings of 2 consecutive words as well as groupings of one word. This modification has improved the accuracy of `TitleClassifier` and was therefore used in the application. The total size of the dictionary for Genrifier is 206,825.

`TfidfTransformer` also helps with data preparation. The name is short for "Term Frequency (times) Inverse Document Frequency Transformer". This class is used for normalizing the term frequencies (i.e. the values next to the tuples in Listing 4.12) and for transforming them such that some words will be more relevant than others when performing the classification. The new value is computed as follows:

$$term\_frequncy(w_i, d) = \frac{num_{i,d}}{num_d}$$

$$inverse\_document\_frequency(w_i) = log_2 \frac{n+1}{num\_docs(w_i)+1} + 1$$

$$tfidf(w_i, d) = term\_frequency(w_i, d) \cdot inverse\_document\_frequency(w_i)$$

w_i stands for the i-th word in an instance and d represents the d-th document (i.e. instance) in the input. $num_{i,d}$ stands for the number of times the i-th word appears in the d-th document, while num_d stands for the total number of words in the d-th document. n represents the total number of documents and $num\_docs(w_i)$ counts the number of documents that contain the i-th word.

Suppose that "apple" would appear in every book title in the training set. The importance of the word is diminished due to $inverse\_document\_frequency$ and the word will have a smaller impact during the training. This is one of the main goals of `TfidfTransformer`. The other is to normalize the term frequencies (i.e. to restrict the range of values). Normalization will help the training algorithm as it will lead to a faster convergence.

`CountVectorizer` and `TfidfTransformer` are coupled together in a `Pipeline`, referenced through a private member `__pipeline`, in `InputTransformer`. The `fit` function is used in order to create the vocabulary and normalize the term frequencies. The `transform` function is used to embed new data, using the already fitted `InputTransformer`.

`TitleClassifier` is the wrapper for the text classifier. At construction time the file name of the serialized and trained classifier is given as input. The classifier is loaded into memory and a reference is kept to it through `__predictor`. `predict` and `predict_proba` work for any number of titles. The former returns the most probable genre, while the latter returns the probabilities of membership to each genre. The input can be a string (if only one title is given), a Python list or a `Numpy` array. `predict_proba_dataframe` receives as input a `DataFrame`. All three functions make use of `__prepare_input` which passes the input data through a fitted `InputTransformer`. The result is then fed to `__predictor` for processing.

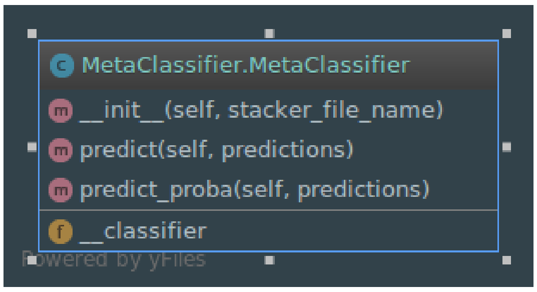

MetaClassifier.MetaClassifier
__init__(self, stacker_file_name)
predict(self, predictions)
predict_proba(self, predictions)
__classifier
Powered by yFiles

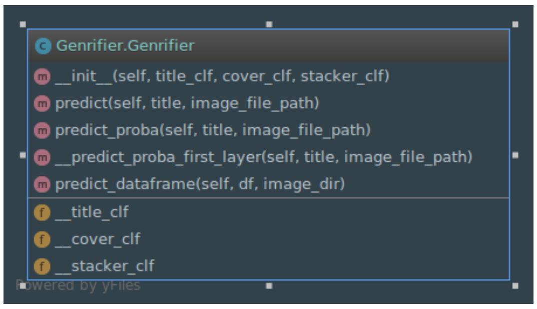
FIGURE 4.9: The class diagram of `MetaClassifier`.

4.5 MetaClassifier

`MetaClassifer` is the name of the wrapper class for the meta-classifier in the stacking ensemble system. It has a simple composition: only 2 methods and only one private field. The constructor receives as input the file with the serialized, pre-trained classifier, loads it into memory and saves a reference to it in `__classifier`. The first method, `predict`, receives as input a matrix with a fixed number of columns (16; 8 for the predictions from the cover + 8 for the predictions from the image) and a volatile number of rows. Each row will contain the probabilities associated to a single book. The output of this function will be a list containing the predicted genre, for each instance in the input matrix. `output_proba` receives the same input. The output, however, is different: it is a matrix with a fixed number of columns (8) and a fixed number of rows (equal to the number of rows of the input). The value on the i-th row and the j-th row represents the probability that the i-th input book has the j-th genre.

4.6 Genrifier

`Genrifier` is the name of the whole ensemble system. It contains references to the first-layer and to the meta-classifier. It also coordinates how the prediction is done for the input title and image. The constructor receives as input the three classifiers needed for prediction: the cover, title and meta classifier will be referenced using `__cover_clf`, `__title_clf` and `__stacker_clf` respectively. There are 3 public functions, `predict`, `predict_proba` and `predict_dataframe`. All



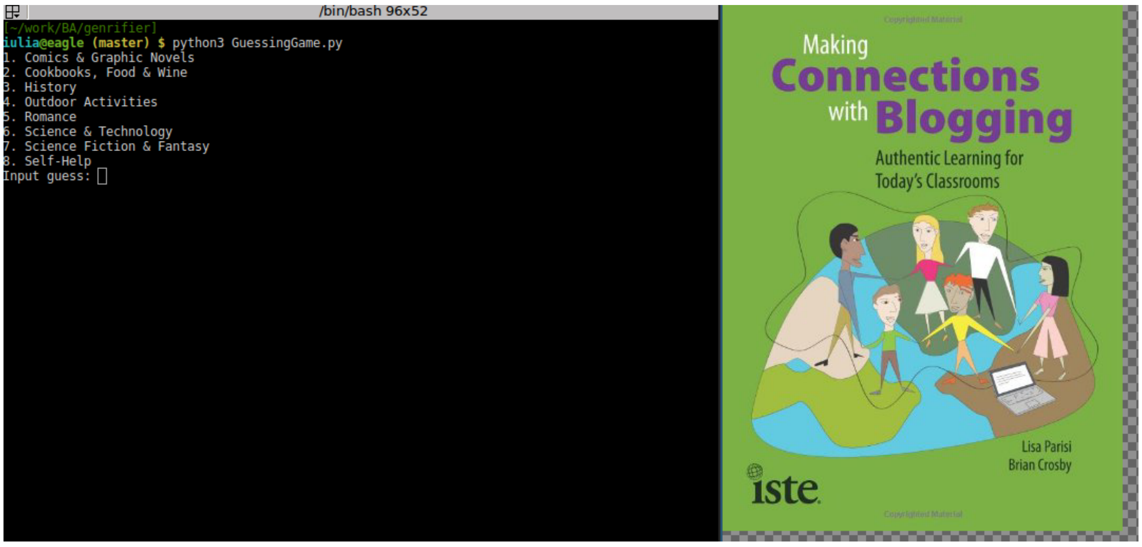
```
Genrifier.Genrifier
__init__(self, title_clf, cover_clf, stacker_clf)
predict(self, title, image_file_path)
predict_proba(self, title, image_file_path)
__predict_proba_first_layer(self, title, image_file_path)
predict_dataframe(self, df, image_dir)
__title_clf
__cover_clf
__stacker_clf
```

FIGURE 4.10: The class diagram of `Genrifier`.

three functions have a similar running mechanism: first, they compute the probabilities passing the input data through the first layer (i.e. `BookCoverClassifier` and `TitleClassifier`); then, they aggregate the results, creating a 16-column matrix which contains the concatenation of the probabilities; lastly, the resulted matrix is passed through the meta-classifier in order to obtain the final prediction. A visual representation of this process was previously given in Figure 3.15.

4.7 The Guessing Game

The Guessing Game is a CLI game which aims to understand how good are humans at classifying books into disjointed genres. The game is quite simple: the player is presented with the front cover of a book (which also contains the title) and they have to guess from which genre the book is most probably from. This action is repeated for a certain amount of times (the default number is 5), the player being presented with different books at each iteration. At the end of the game, the accuracy of the player is outputted, along with the correct answers, had the player guessed incorrectly. The results are stored in a file called `statistics.csv`, where the total number of iterations is written, along with the total number of bad guesses. A *confusion matrix* is also persisted. The confusion matrix is an 8x8 matrix, where the value from the i-th row and the j-th column represents the number of times a book from the i-th genre has been miss-classified as a book from the j-th genre. It is through this file that we can experimentally compute what is the “test accuracy” of humans when classifying books into genres. The game has another option (`vs_AI`), where the player is going to test his/her skills against the



```
/bin/bash 96x52
[~/work/BA/genrifier]
iulia@eagle (master) $ python3 GuessingGame.py
1. Comics & Graphic Novels
2. Cookbooks, Food & Wine
3. History
4. Outdoor Activities
5. Romance
6. Science & Technology
7. Science Fiction & Fantasy
8. Self-Help
Input guess: 
```

FIGURE 4.11: The beginning of the Guessing Game.

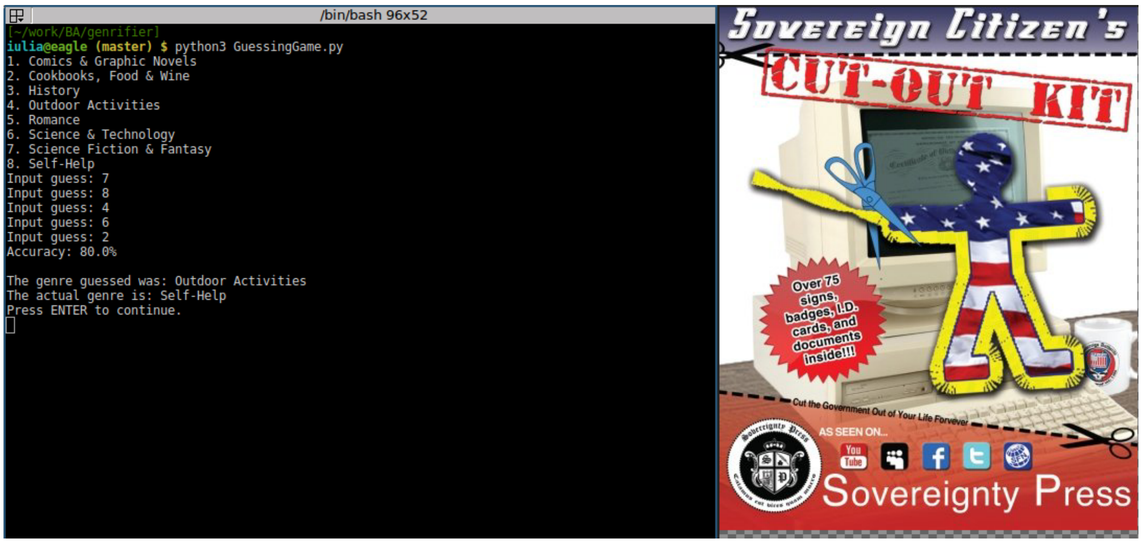
AI system proposed in this paper. If enabled, the game will output at the end the accuracy of the AI and the wrong guesses that it has made.

When running the script `GuessingGame.py`, the list of possible genres is printed to the screen, along with “Input guess:”, which is the input prompt. Figure 4.11 presents how the screen looks on a fresh start of the game: on the left hand side we can see the list of genres along with the input prompt. On the right hand side is the cover of the book from the first iteration. The user will have to input a number from 1 to 8, associated to the genre he/she believes the book is from. One possible end state of the guessing game is presented in Figure 4.12. The final accuracy was printed to the screen along with the mistake that the user made. For each mistake, the program will output the cover image of the corresponding book.

The game can easily started on the terminal by writing the following command:

```
python3 GuessingGame.py
```

There are 2 parameters with which the user can configure the behaviour of the game: `vs_AI` and `num_iterations`. The former can have 2 values: *True* or *False*. If enabled, the user will test their skill against the AI. The latter controls the number of books that will be shown during the game. Its value needs to be an



```
/bin/bash 96x52
[~/work/BA/genrifier]
iulia@eagle (master) $ python3 GuessingGame.py
1. Comics & Graphic Novels
2. Cookbooks, Food & Wine
3. History
4. Outdoor Activities
5. Romance
6. Science & Technology
7. Science Fiction & Fantasy
8. Self-Help
Input guess: 7
Input guess: 8
Input guess: 4
Input guess: 6
Input guess: 2
Accuracy: 80.0%

The genre guessed was: Outdoor Activities
The actual genre is: Self-Help
Press ENTER to continue.
```

Sovereign Citizen's
CUT-OUT KIT
Over 75 signs, badges, I.D. cards, and documents inside!!!
Cut the Government Out of Your Life Forever
AS SEEN ON...
Sovereignty Press

FIGURE 4.12: The end of the Guessing Game.

integer value between 1 and 8421 (the size of the test set). The images that are presented are taken from the test set of the ensemble system. Therefore, the AI will not have any advantage whatsoever due to the fact that the instances it will be presented with have not been used as training material.

4.8 Genrifier CLI API

One way of using Genrifier is through the Command Line Interface API. There are 3 main ways of using it:

1. Use it to predict the genre of a single book by giving as input the title and the file path to the cover.
2. Use it to predict the genres of multiple books stored in a CSV file.
3. Use it to recreate the test accuracy by predicting the genres of all the books in the test set.

The first use case can be achieved by running `Genrifier.py` with the following parameters: `--cover_image_path` and `title`. Suppose we wanted to classify the popular computer science book *"Introduction to Algorithms"*[19] and we had the image of its cover stored in the file `./trial-dir/clrs.jpeg`. The command we need to use in order to categorize the book is:

```
python3 Genrifier.py --title="Introduction to Algorithms"
                     --cover_image_path=./trial-dir/clrs.jpeg
...
Science & Technology
```

LISTING 4.14: How to categorize "Introduction to Algorithms" using `Genrifier`'s CLI API.

The output of the command above is the the most probable genre, according to the AI. There is an addition option that we can apply: outputting the probabilities associated to each genre instead of the final genre alone. The additional parameter that we need to add is `--output_proba=True`. The results are presented below, in Listing 4.15.

```
python3 Genrifier.py --title="Introduction to Algorithms"
                     --cover_image_path=./trial-dir/clrs.jpeg
                     --output_proba=True
...
The probability for genre Comics & Graphic Novels is 0.0%
The probability for genre Cookbooks, Food & Wine is 0.0%
The probability for genre History is 0.0%
The probability for genre Outdoor Activities is 0.0%
The probability for genre Romance is 0.0%
The probability for genre Science & Technology is 100.0%
The probability for genre Science Fiction & Fantasy is 0.0%
The probability for genre Self-Help is 0.0%
```

LISTING 4.15: The probabilities of "Introduction to Algorithms" using `Genrifier`'s CLI API.

The book given as an example is the training set, this being the reason behind the "unanimous" decision and the 100% probability.

One other way of using the Genrifier CLI API is to predict the genres of multiple books. The input method is through a CSV file and the structure of the file is, as follows:

- the first row of the CSV needs to specify what are the main components of the books; the needed information about each book is: the title and the file name of the front cover image. Therefore, the only possible first rows in the input file are “title,filename” or “filename,title”
- the following rows contain the required information about each book that is to be classified (i.e. the title and the file name), separated by commas.

Listing 4.16 shows one possible input file. As can be seen below, titles which have commas included in their name are enclosed between braces.

```
filename,title
1249225264.jpg,"Puerto Plata, Dominican Republic: Including its
    History, San Felipe de Puerto Plata, Guananico, Villa Montellano,
    and More"
0500252106.jpg,"Naturalists in Paradise: Wallace, Bates and Spruce in
    the Amazon"
0983639736.jpg,Many Are Called
1598638688.jpg,UML For The IT Business Analyst
```

Listing 4.16: The contents of a possible input file for `Genrifier`.

Let’s suppose that the above contents would be in a file called `example.csv` and that all files from the Listing above would in the the directory `./example/`. The call that we would have to make in order to label every book in the input file is:

```
python3 Genrifier.py --data_set=example.csv
                     --img_dir=./example/
```

Listing 4.17: How to categorize `example.csv` using `Genrifier`’s CLI.

The results will be saved in file called `example.out`, which will contain, on each row, the predicted genre for each book.

The last way of using the Genrifier CLI API is to run it on the test set. Suppose that the images contained in the test set were in the directory `./test/`. The call we would need to make in order to run the Genrifier CLI API on the test set is presented in Listing 4.18.

```
python3 Genrifier.py --test_set=True
                         --img_dir=./test/
```

LISTING 4.18: How to categorize the test set using `Genrifier`'s CLI.

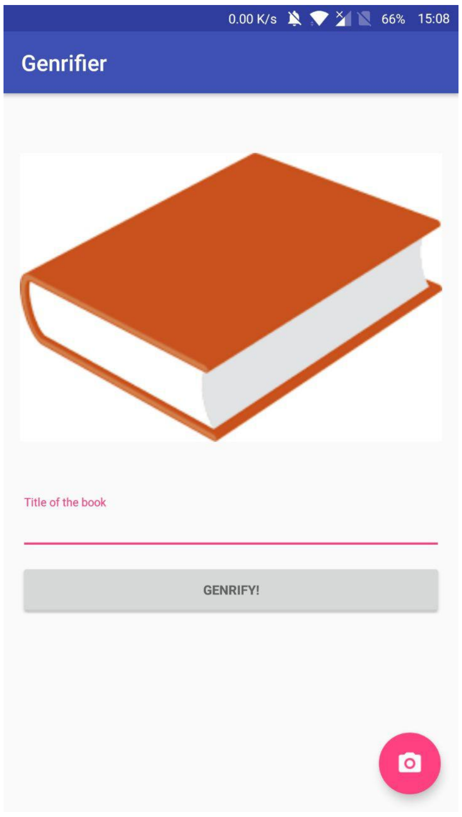
The call above will output on the Command Line the test accuracy and the results will not be saved to a file. If one wishes to save the labels to the test file, this can easily be done by giving the test file as input through the `data_set` parameter.

4.9 Genrifier Android App

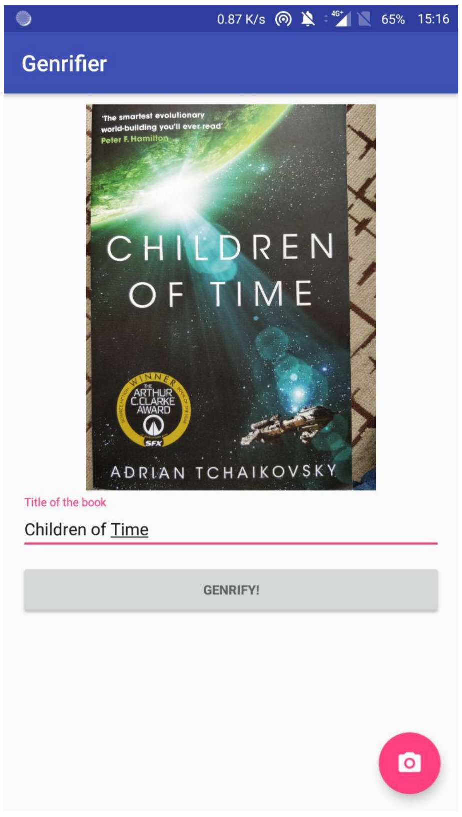
This application comes with an Android App in order to make the system more easier to use and more accessible. The app is used to classify a single book at a time. The user is required to take a picture of the book cover using their smart phone camera, input the title using the phone's keyboard and finally press a button ("Genrify") in order to start the classification process. The application will send the encoded image and the title to a custom made server. The image will be decoded and `Genrifier` will process the input in order to obtain the most probable genre. The result is then sent back to the user's phone and a new message containing the genre is outputted to the screen. Screen-shots illustrating the aforementioned flow are presented in Figure 4.13.

The Android mobile app is written in Java, which is the most used programming language for writing Android apps. In order to build this mobile app, I used Android Studio, which is an IDE (Integrated Development Environment) provided by Google[12] in order to build, develop and install Android apps

For testing I used my personal smart-phone, which is a OnePlus 5. The OS installed on this phone is a wrapper for Android Oreo 8.1, called OxygenOS. The minimum SDK version needed to run this application on an Android phone is 24 (i.e. Android Nougat 7.0[46]).

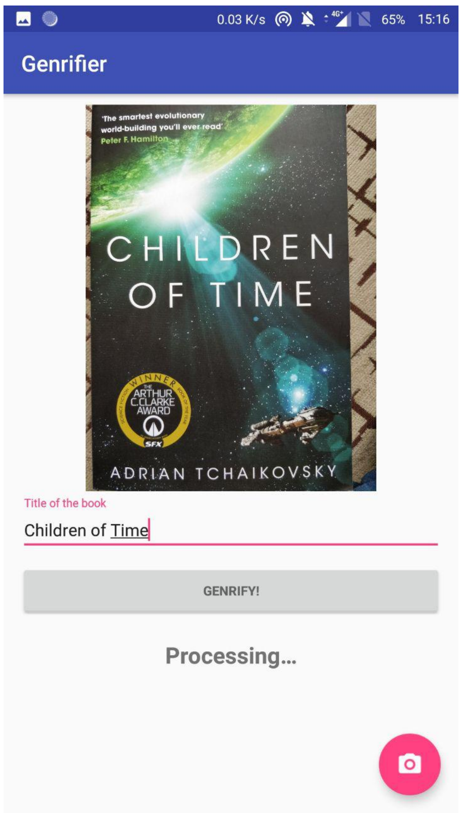


(a) Fresh start-up of the application.


0.87 K/s 65% 15:16
Genrifier
'The smartest evolutionary world-building you'll ever read' Peter F. Hamilton
CHILDREN OF TIME
WINNER THE ARTHUR C.CLARKE AWARD
ADRIAN TCHAIKOVSKY
Title of the book
GENRIFY!

Children of Time

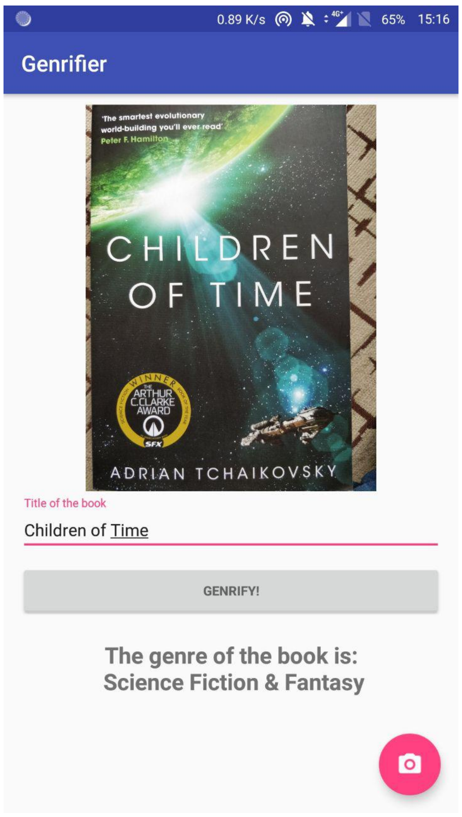
(b) The cover and title are loaded.



Children of Time

Processing...

(c) The book info is sent to the server.



Children of Time

The genre of the book is:
Science Fiction & Fantasy

(d) The genre is outputted to the screen.

FIGURE 4.13: The flow of the mobile app.

The Android app is organized in 4 different classes, each serving a different functionality:

- `Book` which is a simple object used to represent a book. It stores the title and the file path to the cover image of the book.
- `ImageEncoder` which is used to encode the images before sending them to the server. The encoding is done through `Base64`[13], a utility class in the Android framework which can be used to transform the bytes of the image in a base 64 number.
- `HttpClient` is the only class in the system which has direct contact with the server. It prepares the data, makes the HTTP requests and unpacks the results from the server.
- `GenrifyActivity` is the "main" class of the system. It loads the UI elements, gathers input data and coordinates the operations of the whole application.

The source code and the resources used to build the mobile app are in the directory `mobile_app`.

The custom server used in this application is written, yet again, in Python. It heavily uses `flask`, which is a framework used to host the server and process requests. It has a single route, `/genrify` and accepts only POST requests. The information (i.e. the encoded image and the title) "arrive" wrapped in a JSON object. First, the server unpacks the JSON object, decodes and saves the image to the disk. Then, the title alongside the image path are fed to `Genrifier`. The output (i.e. a string representing the genre) is then sent back to the client. The code for the server can be found in `server.py`. In order to start the server, the following commands need to be written in a terminal.

```
export FLASK_APP=server.py
python3 -m flask run --host=0.0.0.0
```

Listing 4.19: The commands needed in order to start the server.

Chapter 5

Conclusion

This thesis has introduced the problem of book genre classification and has managed to successfully categorize 87.8% of the test books, using as sole information the image of the front cover and the title. As previously mentioned in Chapter 4, the Guessing Game gathers statistics regarding the accuracy of humans at classifying books into genres. The current values, at the time of writing this chapter, are the following: out of 143 iterations, humans have made a total of 32 miss-classifications. This leads us to believe that a possible value for human accuracy is around 77.6%, which is less than the value obtain through this system. However, I consider that no comparison between human and AI skills can be made, due to the fact that the number of iterations is small compared to the size of the test set, which contains a total of 8,421 books. A good experiment with reliable results would also involve a large variety of people, with different backgrounds and cultures. Therefore, this paper does not wish to declare that the system provided is a better classifier than humans, rather to prove that intelligent systems can be built in order to extract valuable information about the contents of books judging by their title and cover.

There are several areas of improvement of the current system. First of all, we could incorporate more genres into the system. Fiction sub-genres are underrepresented compared to the non-fiction ones. A possible addition of more fiction sub-genres would lead to a bigger book coverage, as the current eight genres definitely do not cover the vast variety of literature.

However, with additional genres, the classes start to become less and less mutually exclusive. This is another area of improvement. We could build a system which outputs the two or three most probable genres for a book and therefore minimize the number of occurrences where a book is perfectly suitable for multiple genres. The model also has room for improvement: the image classification system proposed in this paper had a final test accuracy of 58.2%, a value which definitely can be improved with the rapid growth in the state of the art.

One final area of improvement that this paper proposes is to build a way of automatically gathering new data, while maintaining the already existing dataset. Perhaps one could make a crawler which updates the information of the books in the dataset (if, perhaps, the cover has changed). Upon new data instances, the crawler would save the information of the new books and periodically retrain the system using the updated data-set.

Bibliography

Print Resources

[15] Boser, Bernhard E., Guyon, Isabelle M., and Vapnik, Vladimir N. “A Training Algorithm for Optimal Margin Classifiers”. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory.* COLT ’92. 1992, pp. 144–152.

[16] Breiman, Leo. “Random Forests”. In: *Mach. Learn.* 45.1 (2001), pp. 5–32.

[19] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms, Third Edition.* The MIT Press, 2009.

[20] Cortes, Corinna and Vapnik, Vladimir. “Support-Vector Networks”. In: *Mach. Learn.* 20.3 (1995), pp. 273–297.

[28] Géron, Aurélien. *Hands-On Machine Learning with Scikit-Learn & Tensorflow.* O’Reilly Media, 2017.

[33] Iwana, Brian Kenji and Uchida, Seiichi. “Judging a Book By its Cover”. In: *CoRR* abs/1610.09204 (2008).

[35] Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. “ImageNet Classification with Deep Convolutional Neural Networks”. In: (2012), pp. 1097–1105.

[36] Lecun, Yann, Bottou, Léon, Bengio, Yoshua, and Haffner, Patrick. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE.* 1998, pp. 2278–2324.

[37] McCulloch, Warren S. and Pitts, Walter. "Neurocomputing: Foundations of Research". In: (1988), pp. 15–27.

[38] Mikolov, Tomas, Sutskever, Ilya, Chen, Kai, Corrado, Greg S, and Dean, Jeff. "Distributed Representations of Words and Phrases and their Compositionality". In: *Advances in Neural Information Processing Systems 26*. 2013, pp. 3111–3119.

[39] Nielsen, Michael. A. *Neural Networks and Deep Learning*. Determination Press, 2015.

[42] Rumelhart, David E., Hinton, Geoffrey E., and Williams, Ronald J. "Neurocomputing: Foundations of Research". In: 1988. Chap. Learning Representations by Back-propagating Errors, pp. 696–699.

[43] Russakovsky, Olga, Deng, Jia, Su, Hao, Krause, Jonathan, Satheesh, Sanjeev, Ma, Sean, Huang, Zhiheng, Karpathy, Andrej, Khosla, Aditya, Bernstein, Michael, Berg, Alexander C., and Fei-Fei, Li. "ImageNet Large Scale Visual Recognition Challenge". In: *Int. J. Comput. Vision* 115.3 (2015), pp. 211–252.

[48] Simonyan, Karen and Zisserman, Andrew. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014).

[50] Springenberg, Jost Tobias, Dosovitskiy, Alexey, Brox, Thomas, and Riedmiller, Martin A. "Striving for Simplicity: The All Convolutional Net". In: *CoRR* abs/1412.6806 (2014).

[52] Szegedy, C., Liu, Wei, Jia, Yangqing, Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. "Going deeper with convolutions". In: (2015), pp. 1–9.

[54] Vapnik, V and Lerner, A. "Pattern Recognition using Generalized Portrait Method". In: *Automation and Remote Control* 24 (1963).

[55] Wolpert, David H. "Stacked Generalization". In: *Neural Networks* 5 (1992), pp. 241–259.

Online Resources

[1] URL: http://image-net.org/challenges/LSVRC/2014/results. (accessed: 24.06.2018).

[2] URL: https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/. (accessed: 24.06.2018).

[3] URL: https://www.frontiersin.org/research-topics/4817/artificial-neural-networks-as-models-of-neural-information-processing. (accessed: 24.06.2018).

[4] URL: http://cs231n.github.io/neural-networks-1/. (accessed: 24.06.2018).

[5] URL: https://www.researchgate.net/figure/The-basic-components-of-an-artificial-neuron_fig1_228395588. (accessed: 24.06.2018).

[6] URL: https://hackernoon.com/gradient-descent-aynk-7cbe95a778da. (accessed: 24.06.2018).

[7] URL: https://www.jeremyjordan.me/nn-learning-rate/. (accessed: 24.06.2018).

[8] URL: https://github.com/PetarV-/TikZ/tree/master/2D%20Convolution. (accessed: 24.06.2018).

[9] URL: https://towardsdatascience.com/multi-label-image-classification-with-inception-net-cbb2ee538e30. (accessed: 24.06.2018).

[10] URL: https://stackoverflow.com/questions/48133830/why-is-svm-a-black-box-learning-algorithm. (accessed: 24.06.2018).

[11] *Amazon Book Section*. URL: https://www.amazon.com/books-used-books-textbooks/b?ie=UTF8&node=283155. (accessed: 24.06.2018).

[12] *Android Studio*. URL: https://developer.android.com/studio/. (accessed: 24.06.2018).

[13] *Base64 - Android Developers*. URL: https://developer.android.com/reference/java/util/Base64. (accessed: 24.06.2018).

[14] *Book Dataset - Task 2.* URL: https://github.com/uchidalab/book-dataset/tree/master/Task2. (accessed: 24.06.2018).

[17] *Capstone_part11.ipynb - twitter sentiment analysis.* URL: https://github.com/tthustla/twitter_sentiment_analysis_part11/blob/master/Capstone_part11.ipynb. (accessed: 24.06.2018).

[18] Chiang, Holly, Ge, Yifan, and Wu, Connie. *Classification of Book Genres By Cover and Title.* URL: http://cs229.stanford.edu/proj2015/127_report.pdf. (accessed: 24.06.2018).

[21] *CS231n Convolutional Neural Networks for Visual Recognition.* URL: http://cs231n.github.io/convolutional-networks/. (accessed: 24.06.2018).

[22] *DataFrame.* URL: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html. (accessed: 24.06.2018).

[23] *Deep Neural Network Learns to Judge Books by Their Covers.* URL: https://www.technologyreview.com/s/602807/deep-neural-network-learns-to-judge-books-by-their-covers/. (accessed: 24.06.2018).

[24] *FastText.* URL: https://github.com/facebookresearch/fastText. (accessed: 24.06.2018).

[25] *FastText; classification-example.sh.* URL: https://github.com/facebookresearch/fastText/blob/master/classification-example.sh. (accessed: 24.06.2018).

[26] Franklin, Bruce H. *Science Fiction: The Early History.* URL: http://andromeda.rutgers.edu/~hbf/sfhist.html. (accessed: 24.06.2018).

[27] *Genre - Literature, Encylopedia Britannica.* URL: https://www.britannica.com/art/genre-literature. (accessed: 24.06.2018).

[29] *How to Retrain an Image Classifier for New Categories.* URL: https://www.tensorflow.org/tutorials/image_retraining. (accessed: 24.06.2018).

[30] *ILSVRC 2012 Results.* URL: http://image-net.org/challenges/LSVRC/2012/results.html. (accessed: 24.06.2018).

[31] *Image Modules.* URL: https://www.tensorflow.org/hub/modules/image. (accessed: 24.06.2018).

[32] *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)*. URL: http://www.image-net.org/challenges/LSVRC/. (accessed: 24.06.2018).

[34] Jordan, Emily. *Automated Genre Classification in Literature*. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.864.9862&rep=rep1&type=pdf.

[40] *pandas.DataFrame.describe - pandas documentation*. URL: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.describe.html. (accessed: 24.06.2018).

[41] *Retrain script for Image Classification*. URL: https://github.com/tensorflow/hub/raw/r0.1/examples/image_retraining/retrain.py. (accessed: 24.06.2018).

[44] Schlag, Imanol. *Important ILSVRC achievements from 2012-2015*. URL: https://ischlag.github.io/2016/04/05/important-ILSVRC-achievements/. (accessed: 24.06.2018).

[45] *scikit learn documentation - Ensemble methods*. URL: http://scikit-learn.org/stable/modules/ensemble.html#forest. (accessed: 24.06.2018).

[46] *SDK Platform release notes*. URL: https://developer.android.com/studio/releases/platforms. (accessed: 24.06.2018).

[47] Sigtryggur, Kjartansson and Ashavsky, Alexander. *Can you Judge a Book by its Cover?* URL: http://cs231n.stanford.edu/reports/2017/pdfs/814.pdf. (accessed: 24.06.2018).

[49] *sklearn.model_selection.GridSearchCV*. URL: http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. (accessed: 24.06.2018).

[51] *StratifiedShuffleSplit*. URL: http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedShuffleSplit.html. (accessed: 24.06.2018).

[53] Tugce, Tasci and Kyunghee, Kim. *ImageNet Classification with Deep Convolutional Neural Networks presentation.* URL: `http://vision.stanford.edu/teaching/cs231b_spring1415/slides/alexnet_tugce_kyunghee.pdf`. (accessed: 24.06.2018).

[56] *Word2Vec, Doc2vec & GloVe: Neural Word Embeddings for Natural Language Processing.* URL: `https://deeplearning4j.org/word2vec.html`. (accessed: 24.06.2018).