University of Castilla-La Mancha
School of Computer Science and Engineering

Undergraduate Dissertation

Bachelor's Degree in Computer Science and Engineering

Major in Computing

Deep Learning techniques applied to prediction from images. Use case: pet's adoption

David Mora Garrido

July, 2021

UNDERGRADUATE DISSERTATION

Bachelor's Degree in Computer Science and Engineering

Major in Computing

Deep Learning techniques applied to prediction from images. Use case: pet's adoption

Author: David Mora Garrido

Supervisors: María Julia Flores Gallego

José Antonio Gámez Martín

July, 2021

To my parents and brothers

Autorship Statement

I, the undersigned, David Mora Garrido , with ID number 49427520Z, declare that I am the sole author of the Undergraduate Dissertation titled “Deep Learning techniques applied to prediction from images. Use case: pet’s adoption ”, that this work does not break the current intellectual property law, and that all the non-original material contained in this work is properly attributed to their legitimate authors.

Albacete, 14th July 2021

Signed by: David Mora Garrido

MORA GARRIDO DAVID - 49427520Z

Firmado digitalmente por MORA GARRIDO DAVID - 49427520Z
Fecha: 2021.07.14 23:45:22 +02'00'

Abstract

In this project, we conduct a data mining process in order to solve a domain-specific problem: predicting the adoption speed of online pet profiles given up for adoption in Malaysia. In particular, we divide the work in a common number of stages: domain knowledge acquisition, exploratory data analysis, feature engineering and transformations and eventually model creation and evaluation.

All those stages are properly developed, even though the main focus is on the feature engineering stage, and especially on the extraction of image features using Deep Learning techniques (training and using CNNs on different configurations in order to extract the best possible features).

Acknowledgements

I want to thank the supervisors of this project, Julia and José Antonio, for helping me whenever I needed it and for understanding my personal circumstances.

Also, thanks to my father and brothers for their continuous help and support, and also to my mother for all the received love even if she is no longer able to comprehend what I do.

Finally, I want to thank my two best friends during these four years of the Degree for their support, Pablo and Nuria.

Contents

List of Figures

List of Tables

1. Introduction

Machine Learning methods and techniques have been developed and applied over the years to solve different kind of tasks: predicting a disease on the basis of certain symptomatology, weather forecasting, finding the main topic or the presence of certain emotions in a particular text, detecting objects in an image, creating an artificial agent that plays chess, detecting fraud, clustering DNA patterns, etc. Most of these tasks or problems can be approached as one of three major paradigms: supervised learning (an algorithm receives examples of inputs and their outputs, and it creates a model or function that allows us to predict the outcome of new examples), unsupervised learning (an algorithm try to find different structures or patterns in the input data, which does not include any output) or reinforcement learning (an algorithm or computer program interacts with an environment to achieve some goal, adjusting its actions based on the feedback or rewards it receives).

In this project, our objective is to employ a set of different techniques to accomplish a supervised learning task: predicting the adoption speed of a pet (dog or cat) based on data in different formats, extracted from different sources. The data is posted in **Kaggle**, a well-known Data Science website that provides a platform to run code in a controlled environment. This platform is also a hub of different datasets and utilities that are uploaded by Kaggle, the users/practicioners or even companies, as they try to create challenges (with a price in cash or not) that eventually turn into profit, in the shape of models that allow them to achieve the proposed task or even to recruit people to work or collaborate with them.

In particular, the aforementioned task is proposed as a competition[1] (which was already over at the time we found it), and the data is provided by **PetFinder.my**, a website where profiles of pets are posted, mainly given up for adoption, including data such as the breed, age, sex, whether it is neutered or vaccinated, images of the pet, a description, etc. (see Figure 1.1 for an example). We will analyse all these data in the next chapter.

[1] https://www.kaggle.com/c/petfinder-adoption-prediction/overview

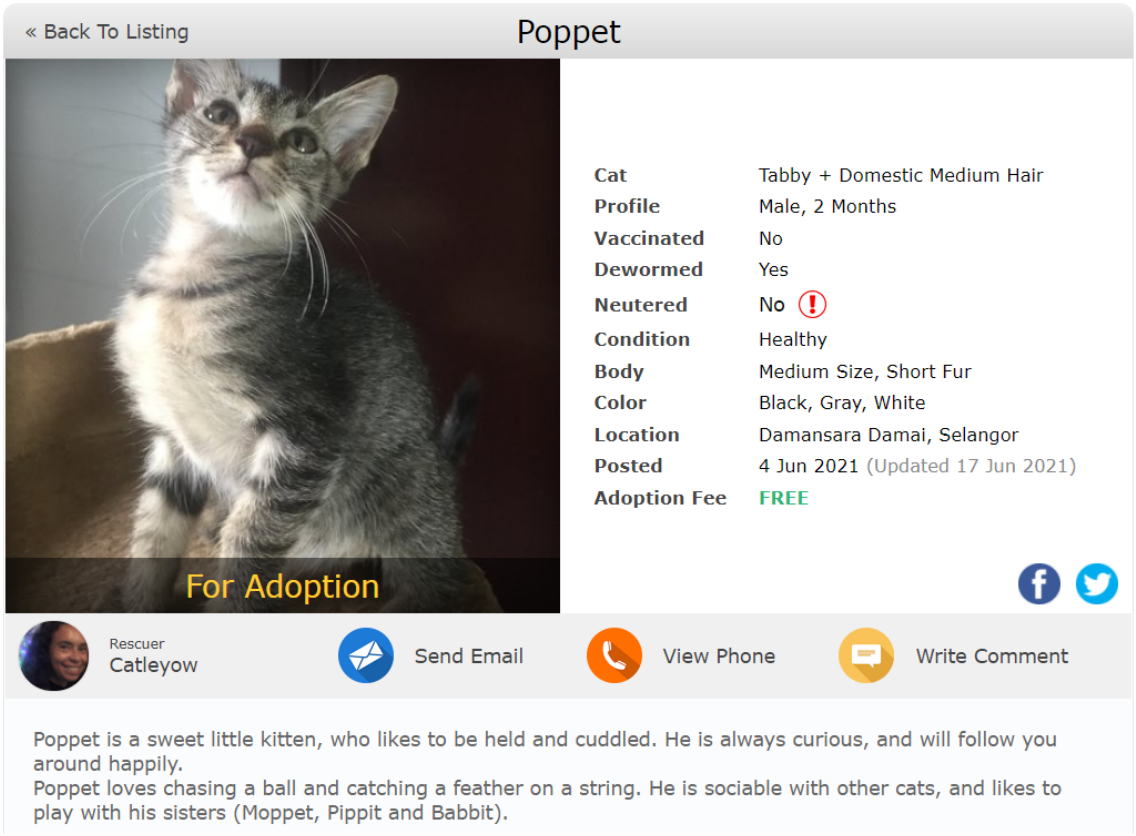

Figure 1.1: Profile of a cat given up for adoption in PetFinder.my

1.1. Context

Before diving into the objectives of this project and the successive sections, it is worth trying to understand the problem domain and the motivation to carry out the task of predicting the adoption speed of a pet.

One of the main goals of websites like PetFinder.my is to act as a centralized platform that allows rescuers and shelters to publish profiles of pets that are given up for adoption, so that they reach more users or potential adopters. We have to take into account that sheltering or temporarily adopting dogs or cats may be expensive; [Lord et al., 2014] shows estimations of these costs (food, health tests, neutering, vaccination, deworming, etc.) under United States conditions, and concludes after doing a number of simulations on those conditions that even if the main goal of a shelter is to increase the number of adoptions, increasing the number of sheltered animals is not sustainable if adoption fees or donations are not increased too. Thus, being able to estimate the expected outcome (that is, the adoption speed) can be crucial to in turn estimate the costs.

On the other hand, when the length of stay of the pets is high and the shelters are overcrowded, a sad but necessary decision has to be made on whether to euthanize a pet (for example, [Crawford et al., 2017] conducted a study on the effect of waiving adoption fees

of cats in Australia is described, concluding that offering free adoptions would save the cost of long-term housing for animals with lower adoption potential, in addition to the fact that free and normal-fee adopters were similar even across different demographic variables). Consequently, we should be especially focused on predicting those profiles that could be susceptible to not be adopted for a long time.

There are many studies on what characteristics makes people adopt a particular profile of pet rather than others. For example, [Vojtkovská et al., 2019] shows how different attributes like breed, age and sex influences the adoption of cats and dogs in Czech Republic and [Weiss et al., 2012] concludes that in the US receiving more information about the pet's health was more important than receiving information about the pet's behaviour or its life before entering the shelter, in addition to the fact that dogs at the front of their kennels were preferred over those at the back, while the length of stay of cats in shelters was proportional to the latency to approach an unfamiliar person. These last findings are particularly interesting, because they can be captured in images (in the latter case, let's think about a cat that is far from the camera), so this kind patterns can be extracted using Deep Learning techniques, as we will see in Chapter 3.

In fact, there are also studies on what attributes of images and descriptions of pets in online profiles affect their adoption speed. For example, [Nakamura et al., 2020] addresses some important findings like the fact that the image background was correlated with the length of stay of the animal (in particular, if the pet was photographed in a kennel rather than in an natural environment, then the probability to be adopted earlier was higher; this fact, in addition to others, like the higher probability of a shorter length of stay when the dogs did not have their mouths opened, led to the conclusion that potential adopters may favour dogs that seem to be in need of help). The aforementioned article is particularly interesting, because it studies a large amount of details and aspects of the images and the pets in them and their effect on the adoption speed.

As we will also see, the other source of information we are given in the shape of non-structured data is text, in particular names of pets and descriptions of them. The article [Rix et al., 2021] concludes that the name the cats/their profiles were given in the UK did not have any effect on the length of stay in shelters, while those profiles with a description written in third person were more probable to end up being adopted earlier than those written in first person ("impersonating" the pet). Another article, [Nakamura et al., 2019], concludes that the presence or absence of certain terms in the description of adoption profiles published in Australia had breed-specific associations with the length of stay of those pets in shelters.

Finally, as the data we will work with comes from profiles published in Malaysia, we have to consider that certain conditions will probably be different to those we may encounter in the Western world (mentioned before). If we search in repositories or journals like *Animals*[2] or more specifically the *Malaysian Journal of Animal Science*[3], there is little to no information about pet adoption. However, we can look at other sources, like the *Animal Protection Index*[4], which evaluates different indicators related to animal welfare in each country; in

[2] https://www.mdpi.com/journal/animals

[3] http://mjas.my/mjas-v2/rf/pages/index.php

[4] https://api.worldanimalprotection.org/country/malaysia [2020 report]

particular, Malaysia is rated as 'C' in a scale from 'A' (the highest score) to 'G' (the weakest score), which is a good score taking into account that countries like Spain, France or Germany also have that score and all the countries that surround Malaysia have worse scores. For example, in the indicator "Protecting companion animals", Malaysia gets a 'B', as the breeding, rescue and animal shelters are regulated, practices such as tail docking, animal abandonment and animal fighting are banned and the Government creates awareness campaigns on responsible pet ownership. On the other hand, stray dogs in Malaysia can be captured by local authorities and be euthanized if they are not claimed after a short period of time to prevent rabies. Moreover, dogs must be vaccinated against rabies.

Regulation on dog ownership in Malaysia is strict, which is logical to prevent the spread of rabies, but it goes further than we could expect. For example, in certain types of houses or apartments in some areas of Malaysia, neighbours consent is required to acquire a dog license ([Loh, 2017]). But there may be another reason which explains this kind of regulation: the predominant religion in Malaysia is Islam, so devoted Muslims won't even touch a dog, and of course won't adopt them. This finding was shared by the Kaggle user Nefeli Kousi in the discussion forum of the competition[5], which also shares other interesting information like the existence of a local breed of cats with short, twisted tails, which are not adopted as often as other breeds (pet rescuers try not to show their tail in images so that those profiles are not discarded earlier), very few dogs or cats are neutered in Malaysia, so there is overpopulation of stray animals, dogs are seen as tools rather tan pets (so bigger dogs may be more 'useful'), etc.

1.2. Objectives

Our main goal is to conduct a data mining process, using the Kaggle competition as an excuse, since it was already closed at the time we found it (it was opened in December of 2018 and closed in March of 2019). This way, we can focus on developing the different stages in the process as we should, because there is a gap between what is the correct thing to do and what people usually do to win the competition (Kaggle users mainly search Exploratory Data Analysis by other users to save time and focus on Feature Engineering, for example, or they may look into the test dataset to notice how different it is from the training dataset; this is not the case). Still, we have been compliant with the main competition rule: when we have used external data, it was already shared by other users in the competition's forum.

We can extend the aforementioned goal to some more specific objectives:

1. Understand the problem domain (already developed in the previous section).
2. Analyse the data we have been provided or that we have extracted from public sources.
3. Clean, transform and extract information from the data using different data mining techniques, especially focusing on image feature extraction with deep learning.

[5] https://www.kaggle.com/c/petfinder-adoption-prediction/discussion/86581, if you cannot see this post (Kaggle may prompt you to enter the competition), you can see it here: https://vetfuturist.com/stray-animals-malaysia-reality-i-saw-travelling-there-past-months

4. Use different models and paradigms of models and evaluate how well they perform.

These are actually the main stages of certain data mining methodologies like CRISP-DM (excluding the deployment of the model, as in this case it doesn't have to predict new data as it is sent by users, but just making a prediction on a closed test dataset).

1.3. Document structure and competences to be developed

This document is divided into five chapters (including this one). In the second one, we will describe the analysis we have conducted on the data we will use in later stages. That chapter is in turn divided into six sections, one for each different source or format of data.

The third chapter describes how the data we are given is transformed so that we can extract valuable information that could be useful to predict the adoption speed of each pet. This is the longest chapter, divided in a similar way to the first one, in which we especially emphasize on the extraction of features from images and state-of-the-art Deep Learning techniques.

The fourth chapter describe the machine learning models we have used in order to learn from the data and be able to make better predictions. We have covered several paradigms, obtaining different results that we try to explain.

So far, the order of the chapters coincides with the objectives we enumerated in section 1.2, as each chapter tries to develop them. In the last chapter we recapitulate to see how the following competences (which are specific to the Major in Computing) have been achieved, and whether some of them haven't or need further improvement:

[CM4] Ability to know the fundamentals, paradigms, and techniques of intelligent systems, and analyse, design, and build systems, services, and digital, applications which could use such techniques in any application context.

[CM5] Ability to acquire, formalise, and represent human knowledge in a computable form for the solution of problems throughout a digital system in any application context, especially the one linked to computational aspects, perception, and behaviour in intelligent frames.

[CM6] Ability to develop and assess interactive systems, and present complex information and its application in the solution of problems with the design of person-computer interaction.

[CM7] Ability to know and develop computational learning techniques, and design and implement applications and systems which could use them, including the ones for the automatic extraction of information and knowledge from great batches of information.

2. Exploratory Data Analysis

Probably the most important stage in the data mining process is to understand the data you are given, the anomalies there might be, the relationship with the target variable or between predictor variables, etc. All this information that we extract by looking at the data is crucial to apply the appropriate Feature Engineering techniques and build robust models.

First of all, we will introduce which kind of data we are given[1]; then, each section will be focused on a format or source of data, giving the main properties we have found and the anomalies that we have detected. The complete Exploratory Data Analysis (EDA) can be found in **Notebook 1** (see Annex), this chapter is a summary of that notebook.

The data we are given is divided in training data and test data; of course, the EDA is based on the training data only, there is no data leakage from the test data. Then, both training and test data is divided into different formats or sources:

- A main dataset in a .csv file with the base tabular data and also the text data of each pet profile, with the following variables:
  - *AdoptionSpeed*: this is the target variable (values: 0, 1, 2, 3 or 4). The lower value, the faster the adoption was. Thus, it is in fact an ordinal variable.
  - *PetID*: unique identifier of each pet profile.
  - *Type*: whether the pet is a dog or a cat (1 or 2, respectively).
  - *Age*: months of age of the pet when the profile was published.
  - *Breed1* and *Breed2*: primary and secondary breeds, respectively.
  - *Gender*: can be 'Male', 'Female' or 'Mixed' (the profile may represent a group of pets).
  - *Color1*, *Color2* and *Color3*: predominant colors in the pet's fur.
  - *MaturitySize*: estimated size of the pet at maturity ('Small', 'Medium', 'Large', 'Extra Large' or 'Not Specified'; 1, 2, 3, 4 or 0, respectively).
  - *FurLength*: 'Short', 'Medium', 'Long' or 'Not Specified' (1, 2, 3 or 0, respectively).

[1] https://www.kaggle.com/c/petfinder-adoption-prediction/data

- *Vaccinated*, *Dewormed* and *Sterilized*: each one can be 'Yes', 'No' or 'Not Sure' (1, 2 or 3, respectively).
- *Health*: 'Healthy', 'Minor Injury', 'Serious Injury' or 'Not Specified' (1, 2, 3 or 0, respectively).
- *Quantity*: number of pets represented in the profile.
- *RescuerID*: unique identifier of the rescuer.
- *VideoAmt*: number of videos included in the profile.
- *PhotoAmt*: number of images included in the profile.
- *Name*: name of the pet or the profile (text).
- *Description*: write-up included in the profile about the pet/s (text).

All the categorical or ordinal variables are represented with numbers (except the two ID variables which are strings of hashes). The mapping of some of them to a categorical value has already been mentioned; others are mapped to values included in external files.

- Three .csv files containing the mapping between the integer value specified in the base .csv and its categorical value or string for the variables *Breed1* and *Breed2*, *Color1*, *Color2* and *Color3*, and States (of Malaysia), respectively.
- A folder containing each pet profile's images (.jpg format), identified by the *PetID* and the image number.
- A folder containing .json files with the metadata of each image in the previous folder, obtained with Google Vision API.
- A folder containing .json files with the sentiment analysis of each pet profile's description, obtained with Google Natural Language API.

2.1. Base tabular data

In this section we will see the details of the training tabular data we are provided (there are 14993 pet profiles in this dataset). Some of the variables are very interesting and conclusions have been extracted for the Feature Engineering stage; for other variables, there is little we have extracted, so we won't emphasize on them. In this section and the next ones, plots or images will be included when there is something remarkable to say, we don't want to fill this document with images, even though we could as there are many variables in total. For the complete EDA and all the plots and images, again, see Notebook 1.

2.1.1. Target variable

As we said previously, our target is the ***AdoptionSpeed*** variable (so we have to assume that it represents a proxy of "adoptability"). It has 5 possible outcomes, each one meaning the following:

- **0** : adoption took place on the same day as the profile was published.
- **1** : adoption took place in the first week after the publication (between 1 and 7 days).
- **2** : adoption took place in the first month after the publication (between 8 and 30 days).
- **3** : adoption took place in the second or third month after the publication (between 31 and 90 days).
- **4** : **no** adoption took place 100 days after the publication (there are no pets that waited between 90 and 100 days in the data).

We can say that this is either a classification or ordinal regression problem (the latter case means regression and rounding between thresholds to get one of the previous integer values). The distribution of values is shown in Figure 2.1, which in relative terms means that there are approximately 2.7% of '0' cases, 20.6% of '1' cases, 26.9% of '2' cases, 21.7% of '3' cases and 28.0% of '4' cases. As we can see, among the '1' to '4' cases, they are more or less balanced; the main issue here is that there are few instances with a '0' outcome. However, given the meaning of each case, in fact this is not worrying, because those are pets that are adopted the same day their profile are published; the main problem in this domain would be the '4' cases (and it is the majority class, so this is an advantage), which are the pets that are not adopted after 100 days and, consequently, they would imply more expenses for the rescuer or shelter and probably a deterioration of the pet's health or behaviour (or eventually lead to be euthanized; PetFinder does not provide this information).

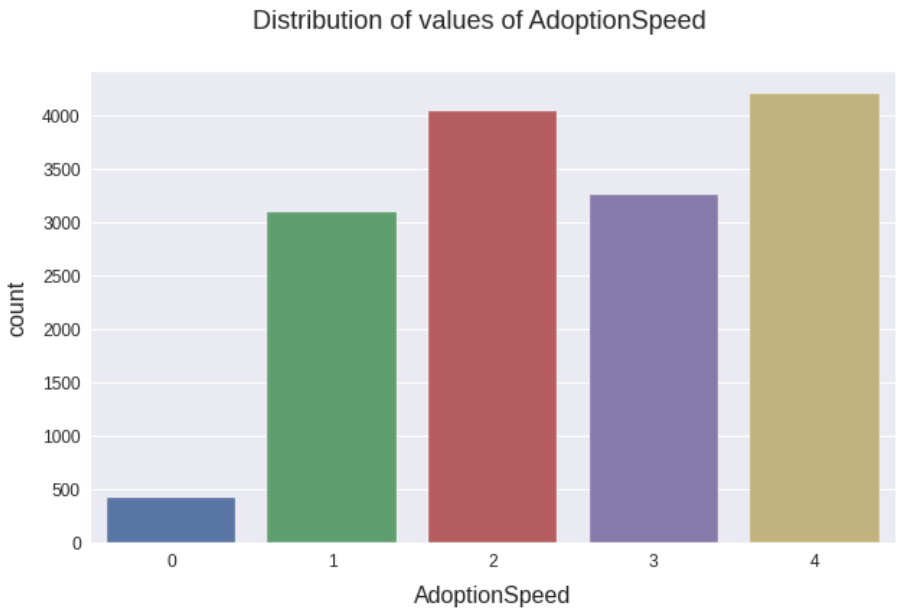

Figure 2.1: Univariate distribution of values of *AdoptionSpeed*

2.1.2. Predictor variables

The first predictor variable we will talk about is ***Type***, that is, whether the profile listed a dog or a cat. 54.2% were dogs, while 45.8% were cats; if we discriminate the outcome of *AdoptionSpeed* by *Type*, in general, cats were given up for adoption earlier than dogs (the percentage of '0' and '1' cases are higher than those of dogs; the percentage of '3' cases is higher in the case of dogs), while the percentage of '4' cases, that is, profiles that were not adopted, is higher in the case of dogs. This is consistent with what we saw in the Context section: in general, cats are preferred over dogs in Malaysia.

The next predictor variable is ***Age***, which we can already tell that is important to estimate *AdoptionSpeed*. The mean *Age* is 10.45 months, even though there are some extreme cases beyond 200 months, so this continuous variable is right skewed (the first quartile is 2 months, the third is 12 months). Moreover, there are peeks of values in multiples of 12, as people probably estimated the number of years instead of months. As we can see in Figure 2.2, when the *Age* was smaller than 6 approximately, the proportion of '4' cases is smaller than the rest of classes (except for '0'), but when the *Age* is greater, '4' is the most repeated outcome again, in a greater proportion than overall. This tendency change is probably due to the fact that around this age is when the pets starts puberty and the 'puppy' or 'kitten' appearance is lost.

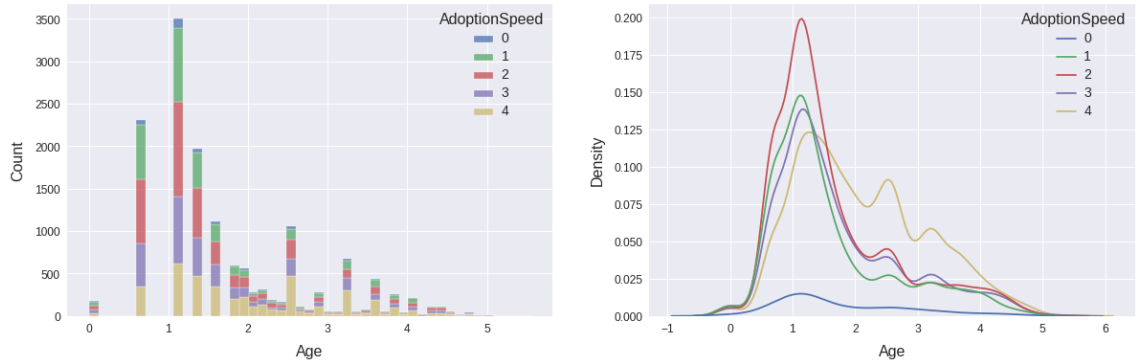

Figure 2.2: Distribution of *Age* and response of *AdoptionSpeed* (log scale)

The mean *Age* by *Type* is 12.9 months in the case of dogs and 7.5 months in the case of cats; given that there are already more dogs than cats, this imbalance is more evident when the *Age* is greater, reinforcing the preference of cats over dogs. Moreover, as we can see in Figure 2.3, it seems that *Age* can be more correlated with *AdoptionSpeed* when we dealing with cat profiles, but not that much in the case of dogs.

Another important variable is the breed, which is actually divided into primary breed, ***Breed1***, and secondary breed, ***Breed2***. The primary breed is present in almost every instance (only 5 cases don't have a value, but in those the secondary breed is present, so we should use it as primary), while the secondary breed has many missing (or simply non-existing)

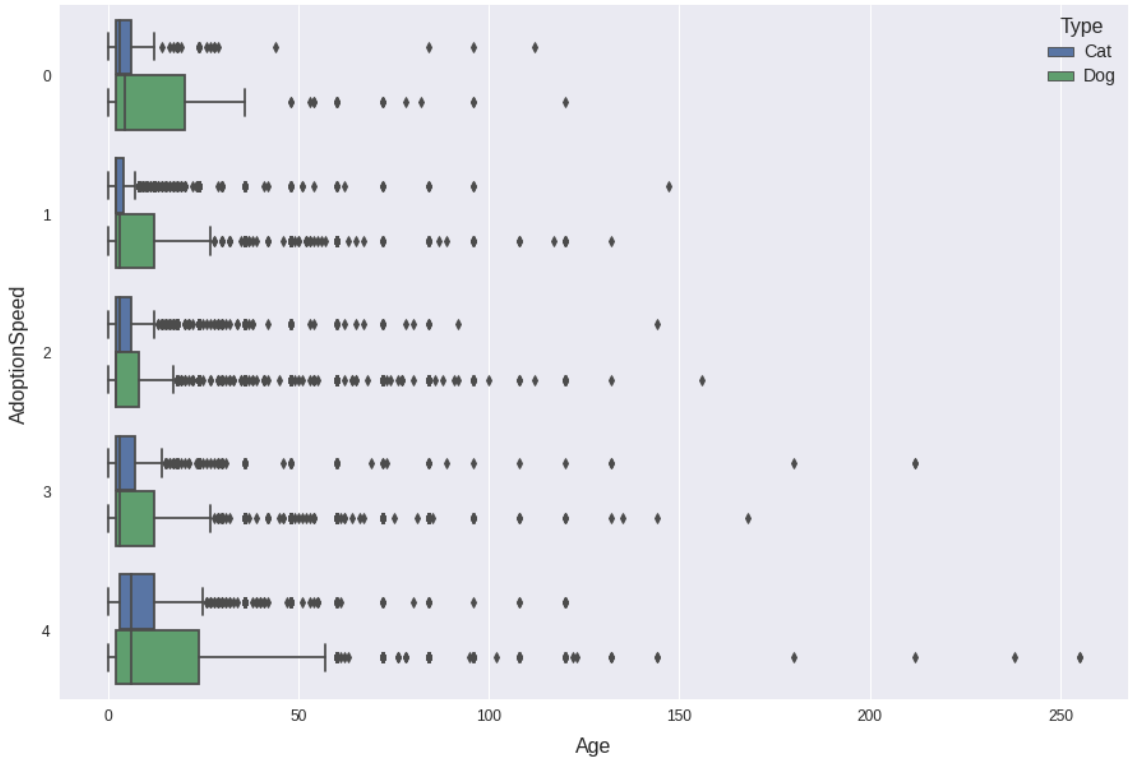

Figure 2.3: Usual range of *Age* values for each *AdoptionSpeed* value discriminated by *Type*

values, more than two thirds of the instances (no instance misses both values; moreover, over 10.1% of the cases have the same primary and secondary breeds). We could think that this is due to the fact that two thirds of profiles show pets with a 'pure' breed, but this is in fact not true: the most repeated primary (and secondary) breed in the case of dogs is 'Mixed Breed', and in the case of cats, 'Domestic Short Hair', and we can assume that both represent pets that cannot be identified as having a specific 'pure' breed (moreover, in the case of cats another two predominant values are 'Domestic Medium Hair' and 'Domestic Long Hair'). Consequently, and in contrast to what PetFinder.my says in the data description, **the number of actual 'pure' breed pets is, in fact, smaller than the number of instances with no secondary breed**.

The presence or not of *Breed2* had little effect on the *AdoptionSpeed* value (in fact, when it is not specified the percentage of '4' cases is a little bit greater), while having a 'pure' breed value (modelled as those instances with 'Mixed Breed' or 'Domestic x Hair' values as primary or secondary breed and those with both primary and secondary breeds) shows that **adoptions took place earlier than those with assumed mixed breed** (higher percentage of '0' and '1' values, smaller of '3' and '4').

Moreover, we detected **anomalies**: 4 instances where cats (this is indicated both by *Type* and looking at the images of those profiles) but the primary breed was 'Mixed Breed', and this value, according to the external dataset of breeds, corresponds to dogs. In order to

detect these cases, the image metadata could be required (to change either *Breed1* or the *Type*).

The **primary breed is an important variable** (after examining the data and according to the articles that we mentioned in the Context section, and also because it is intuitive). For example, in the case of the most repeated breed of dogs after 'Mixed Breed', which is 'Labrador Retriever', the proportion of '4' cases is smaller than overall, but this decrease seems to be balanced by '3' cases instead of '0' or '1', as the case of Shih Tzu dogs (which, by the way, are very common in Asia). Thus, some technique like frequency encoding by *Type* or target encoding is needed to keep as much information as possible of the breed while reducing the high cardinality. All these strategies will be described in the next chapter.

Another variable is ***Gender***: in the training dataset, the 48.5% of profiles are female (the profile shows a single female or a group of pets where all of them are female), 36.9% male (the profile shows a single male or a group of pets where all of them are male), and 14.5% are 'mixed', that is, at least one male and one female as the profile represents a group of pets (2 or more). In our case, it seems that **male pets are adopted earlier than female pets**; this may imply that people would not like to take care or give up for adoption more pets, and even more particularly male dogs might be chosen over female dogs as may be seen by Malaysians as a tool rather than a pet, as we discussed before. The case of 'mixed' pets is not that clear, because the percentage of '4' cases is higher than overall, and when the adoption took place it was more probable to be later; this might be just due to the fact that those profiles represent a group of pets, and less people can afford the adoption. If we look at the ***Quantity*** variable, this seems to be true, more profiles end up not being adopted if they represented a group of 2 or more pets; however, the *Gender* does also matter in those cases: if all the pets in the group have the same *Gender*, then the data shows that the percentage of '4' cases is higher than when the group of pets have different *Gender* (see Figure 2.4).

The next variables are ***Color1***, ***Color2*** and ***Color3***. All the training instances have at least a value of *Color1*, while the 29.8% doesn't have a *Color2* value and 70.7% doesn't have a *Color3* value. These variables are consistent between them, that is, if *Color2* is not specified, then it is because the pet has only one fur color (so those cases are the 29.8% of the total), and there is no instance with a *Color3* value and without a *Color2* value (so 70.7 - 29.8 = 40.9% of pets have two predominant fur colors, even though it could also be that the profile represents a group of pets and the two main fur colors among all of them are those; PetFinder.my didn't give more information about this). Each of these variables can have 1 of 7 different values ('Black', 'Brown', 'Cream', 'Golden', 'Gray', 'White' or 'Yellow'). The most repeated combinations of colors are: Brown, Black-Brown, Black-White, Black-Brown-White and Black; some others, like Golden-Yellow-Gray and Yellow-Gray, rarely appear. **Different combinations of colors yield different distribution of values of *AdoptionSpeed***, and when *Color2* is not specified, that is, pets with just one fur color, the percentage of '4' cases is slightly higher (this is probably due to the fact that Black and Brown are the most repeated *Color1* values, and when the pets have only one of those colors, the proportion of '4' cases are higher than in the overall distribution of *AdoptionSpeed*; however, for example, when the only color is White, then the proportion of '4' cases decreases).

Now, we have to talk about three variables that can be used as categorical or **ordinal** (if

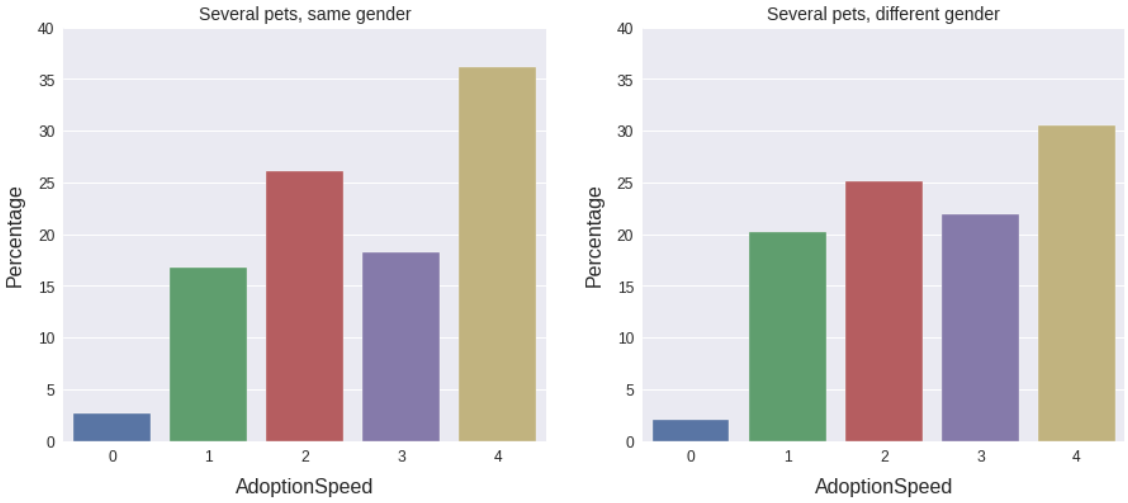

Figure 2.4: Comparison of the outcome of *AdoptionSpeed* given the *Gender* when *Quantity* is greater than 1

the 'Not Specified' values are replaced or imputed in order to maintain the order, which we already described in the introduction of this chapter).

The first one is ***MaturitySize***: in profiles of both cats, and dogs the most repeated value is 'Medium' (60.9% and 75.3% of the total of cats and dogs, respectively). 'Extra Large' is a very unusual value for both cats and dogs, so it is not representative, while 'Small' represents the double of listed cats in comparison to dogs. The proportion of 'Large' values is more or less similar (a little bit higher in the case of dogs). When we discriminate by *Type* and *MaturitySize*, the distribution of values of *AdoptionSpeed* when the dogs are 'Medium' is almost the same as in the entire set of dogs; **when the dog is 'Small' or 'Large', the probability of being adopted earlier is higher** (specially in the case of 'Small'), while the probability of ending up without adoption decreases more in the case of 'Small' values than 'Large' values. In the case of cats, something similar happens, even though the difference is more visible; it seems that Malaysian people prefer 'Small' or 'Large' pets to 'Medium'-sized pets.

The second ordinal variable is ***FurLength***. The predominant value is 'Short' (58.7%), while the pets with 'Long' fur only represent he 5.5% (the rest have a 'Medium' value). When we filter *AdoptionSpeed* by *FurLength*, **the probability of being adopted earlier is significantly higher for pets with 'Long' fur** than when the value is 'Medium' or 'Short', even though the sample of 'Long' cases is also significantly smaller than 'Short' or 'Medium' fur length pets. Another finding on this variable is that **sometimes, its value does not match with the primary breed when the breed is 'Domestic x Hair'** (where 'x' is 'Short', 'Medium' or 'Long' as the values of *FurLength*). In particular, there are 644 of such inconsistencies, being the most repeated one 'Domestic Short Hair' and 'Medium' fur length (in fact, we can see that the profile of Figure 1.1 has this inconsistency). If we model a variable to represent when these inconsistencies happen (just for cats, as 'Domestic x Hair' is not listed as a possi-

ble dog breed), the distribution of *AdoptionSpeed* when there is no inconsistency is almost the same as the original in the entire set of cat profiles, but when the fur length does not match the breed, the probability of being adopted earlier is slightly higher and that of ending up without adoption increases over 5%. This may be due to the fact that, in general, cats with 'Domestic x Hair' as primary breed are not that attractive to people as those cats with a more specific primary breed; or maybe this type of inconsistencies make some people lose confidence on the rescuer.

The last ordinal variable is ***Health***: 96.6% of the training profiles represent 'Healthy' pets, while 3.2% have a 'Minor Injury' and the remaining, which is just 0.2% have a 'Serious Injury'. As we would expect, the data shows that **if the pet is injured, it is more likely that it ends up not being adopted**, especially if the injury is serious. However, the proportion of '0' cases does not decrease (compared to that proportion on healthy cases) when the pet has an injury, in fact it slightly increases when there is a minor injury (maybe some adopters are prone to take care of these pets). In addition, the *Age* of non-healthy pets is mainly gathered between 0 and 40 months, there are no injured pets being more than 140 months old.

No instance in the entire training dataset have a 'Not Specified' value in any of the previous three variables.

The three next variables in the dataset are related with the health of the pets. The first one is ***Vaccinated***: in the training dataset, the 48.2% of the profiles are not vaccinated, while 39.3% are. The remaining are 'Not Sure' cases. If we discriminate the outcome of *AdoptionSpeed* by this variable, we find something surprising: **it is more probable that the pets that are not vaccinated are adopted earlier than those that are vaccinated**; furthermore, the probability of ending up without adoption is lower than in the complete dataset when *Vaccinated* is 'No' (and higher when it is 'Yes') . On the other hand, a 'Not Sure' value didn't affect how early the adoption was when it eventually took place, but it did affect the number of '4' cases, with a higher proportion. This is actually due to the effect of two other variables: *Type*, because while the 48.0% of dogs are vaccinated, only the 29.0% of cats are (and respectively, 37.1% and 61.3% are not vaccinated), and we know that dogs are not commonly preferred over cats in Malaysia, and the fact that the majority of pets that are not vaccinated have a small *Age* value (see Figure 2.5), and as we saw they are more likely to be adopted earlier.

The second variable related to the pet's health is ***Dewormed***: the most repeated value is 'Yes' (56.0%), while the 32.1% are not dewormed (the remaining have a 'Not Sure' value). Again, the probability of the pets that are not dewormed to be adopted earlier seems to be higher than those that are dewormed, but this difference is more subtle than that of *Vaccinated*. As with *Vaccinated*, when the value was 'Not Sure' the proportion of '4' cases was significantly higher. When we look at the values of this variable and the previous one, the most repeated combinations are Yes-Yes, No-No and No-Yes for *Vaccinated* and *Dewormed*, respectively, while the combination Yes-No is very uncommon. Different combinations yield different distributions of *AdoptionSpeed*.

The third and final variable related to the pet's health is ***Sterilized***: 67.2% of the pets are not sterilized, while 20.7% are, and the remaining are 'Not Sure' cases. When the value is 'Not Sure' or 'Yes', the percentage of profiles that were not adopted ('4' cases) is very high, over 40%, and the in the case of 'Yes' when they were adopted it was generally later. When

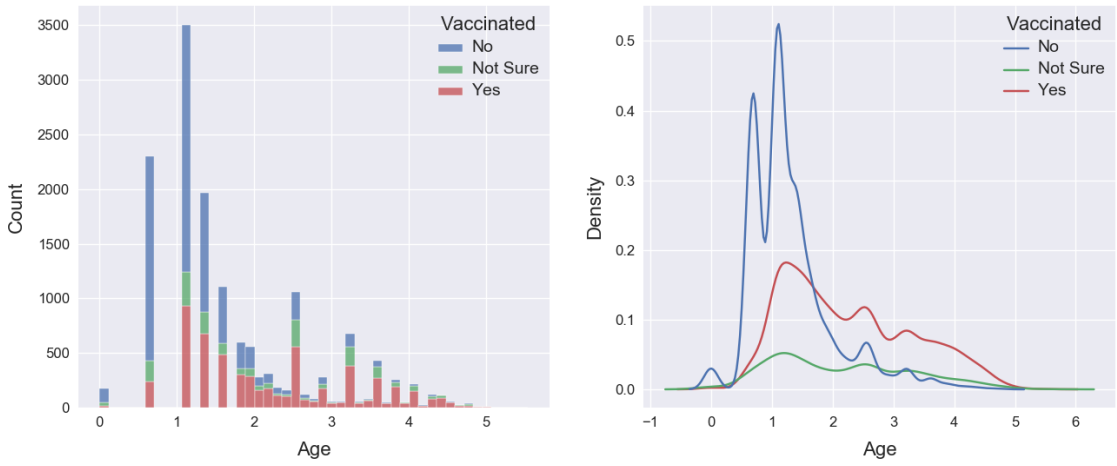

Figure 2.5: Distribution of *Age* and response of *Vaccinated* (log scale)

the value was 'No', the percentage of '4' cases decreased to 20% approximately. Moreover, in this case the *Gender* does not affect the aforementioned outcome too much, because even if the percentage of '4' cases when the value of *Sterilized* is 'Yes' is 37.8% in the case of Male pets and 43.1% in the case of Females, when the value is 'Not Sure' the percentage is 39.1% and 40.6%, respectively, and the *Gender* composition of 'Not Sure' is almost identical to that of 'No'.

All the remaining variables that are not identifiers are numerical. The first one is ***Quantity***: approximately the 77.1% of the training dataset are single-pet profiles, while the profiles with 2, 3, 4, 5 and 6 pets represent the 9.5%, 4.8%, 3.5%, 2.2% and 1.2% of the dataset, respectively. The remaining profiles list more unusual values, between 7 and 20 pets. As we would expect, **the higher the *Quantity*, the more likely the pets are adopted later or end up not being adopted**. In particular, the proportion of '4' cases increases to approximately the 32% when the number of pets is between 2 and 6 (both included), even though the balance between '1', '2' and '3' cases is more or less maintained; however, when the number of pets is 7 or more, both the proportion of '4' cases increases (almost to 40%) and the balance of the remaining values is also lost, decreasing considerably the proportion of '1' values and increasing that of '3' values.

The next numerical variable is the adoption **Fee**: 12663 profiles were free to adopt, while 2330 did have adoption fees. In particular, in the latter cases the mean Fee was 136.8 (we assume that it is in Ringgits, which is the Malaysian currency, but for us which currency is used is irrelevant), while the maximum amount was 3000. The difference between free adoptions and non-free adoptions in the outcome of *AdoptionSpeed* is not that big, as we could expect, because the percentage of '4' cases does only increase over 3-4%, but the percentage of '3' cases is a little bit smaller when the adoption is not free. In fact, the usual range of values of Fee in the profiles that were not free and ended up not being adopted is concentrated in smaller Fee values than the profiles that were adopted. One of the possible reasons is again the preference of dogs over cats and the fact that there are more cases of

the former than those of the latter; if we look at Figure 2.6, we can say that **a factor that could make cat profiles to not be adopted earlier is the Fee, while in the case of dogs the opposite happens**.

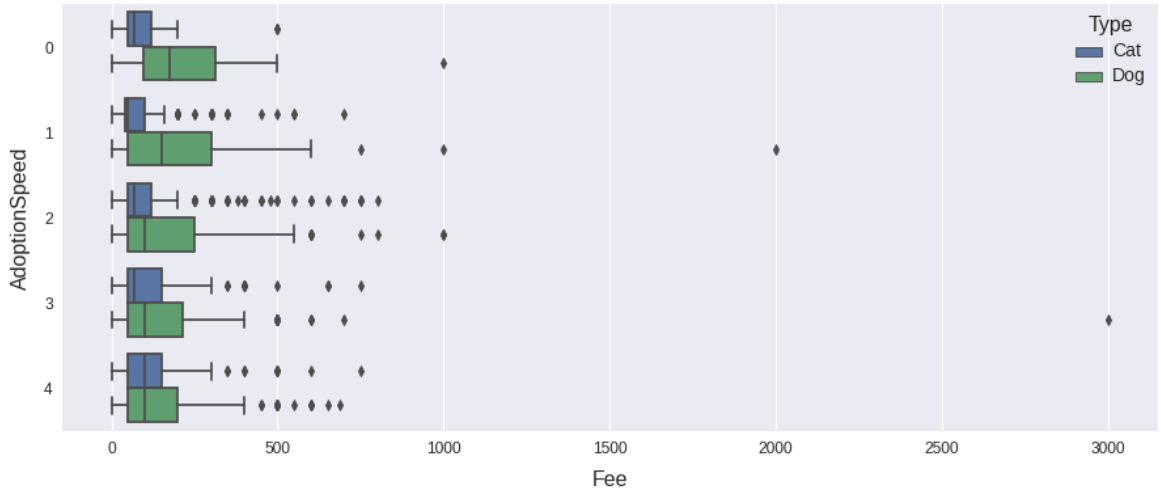

Figure 2.6: Usual range of Fee values for each outcome of *AdoptionSpeed* and value of *Type* (Fee > 0)

Moreover, as we can see in Table 2.1, the mean value of Fee is more or less consistent with the values of the health-related variables (only 13 of the possible 27 combinations of values are shown, which are the ones with 100 or more cases).

Then, another variable is **State**, but this one is not numerical, is categorical as it indicates the State in Malaysia in which the pet was given up for adoption. However, even if the data we are given comes from Malaysia, we would like to construct a model that could predict data from different sources (let's think that PetFinder.my would try to use it on a different country; the normal thing to do would be to re-train the model, but if it is not possible, we would like to maintain some information of this variable, as we may come up with data from Indonesia, let's say, and the information of each State of Malaysia would be useless). Thus, State is replaced by **StateGDP**, which is actually the Gross Domestic Product per Capita of each State[2]. When comparing the values of this new variable and the outcome of *AdoptionSpeed*, there isn't a clear tendency; if we look at the outcome by State, the original variable, there are some states where the shape of the distribution changes significantly, but mainly in those States that are not that common in the dataset.

Another variable that would be difficult to handle is ***RescuerID***, the identifier of each person or entity that posted the profile, because its cardinality is very high (5595 different rescuers, an average of 2.68 profiles published by each rescuer). One way to extract infor-

[2] 2019 data, extracted from https://www.dosm.gov.my/

Vaccinated	Dewormed	Sterilized	count	mean	std	max
No	No	No	4304	8.247444	38.481300	800.0
		Not Sure	181	10.320442	41.107678	300.0
		Yes	169	10.994083	39.756476	300.0
	Yes	No	2159	17.217230	65.693757	750.0
		Not Sure	100	6.250000	28.412323	250.0
		Yes	228	29.315789	78.044923	800.0
Not Sure	Not Sure	No	436	9.263761	52.019286	550.0
		Not Sure	1003	13.419741	83.976438	2000.0
		Yes	135	24.822222	93.614075	750.0
	Yes	No	140	18.650000	66.549373	500.0
Yes	Yes	No	2847	33.596769	99.547195	1000.0
		Not Sure	351	46.415954	189.366578	3000.0
		Yes	2436	36.633826	89.273043	700.0

Table 2.1: Fee variable properties filtered by health-related variables

mation from this variable is **creating another variable which replaces each *RescuerID* value by the number of times it appears in the training set** (frequency or count encoding). Moreover, this strategy can be meaningful in this context, as we would create an ordering based on the number of pets each rescuer has published, which by intuition could be a **significant factor for potential adopters to trust the rescuer**. Once we have modelled this variable, we can see that the 25.2% of rescuers published only one profile, while only 48 out of 5595 rescuers published more than 30 pets. The percentage of profiles that ended up not being adopted when they were published by 'rookie' rescuers increases to approximately 44%, while only over the 23% of profiles published by rescuers who had already posted at least one profile ended up not being adopted. We can also see how important this variable seems to be in order to discern probable '4' cases in Figure 2.7.

The two final numerical variables of the dataset are ***VideoAmt*** and ***PhotoAmt***. Only the 3.8% of the profiles had attached at least one video of the pets, but in those cases the proportion of the outcome '4' decreases to approximately 21%, but when the profiles with videos were adopted it wasn't necessarily earlier. Regarding the number of images included in the profile, the average is 3.89, and **the effect of not having at least a profile image (2.3% of the cases) is very clear: the percentage of profiles that ended up not being adopted is 62%**. Moreover, the percentage of profiles that were not adopted having more than 5 images decreases to 18.5%, while the percentage of '2' and '3' outcomes increases to 30.1% and 32.3%, and that of '0' and '1' is 17.4% and 1.7%, respectively. That is, it seems that **having more than 5 images** (which is 16.8% of the dataset) **increased the chances to be adopted** but it is also true that the increase of '2' and '3' of such cases may be caused by the fact that, as the pets stayed longer in the rescuer's care, more pictures of them were taken.

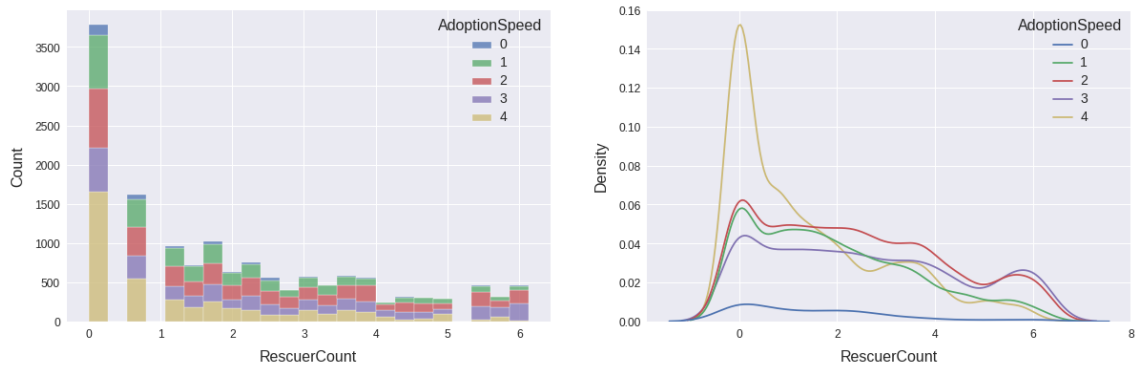

Figure 2.7: Distribution of *RescuerCount* and response of *AdoptionSpeed* (log scale)

2.2. Text data

In the previous section, we saw the details of the main tabular data we are provided. In this one, we will talk about two text variables: ***Name*** and ***Description***.

First, we have ***Name***, which in some cases actually shows the name that the pet was given and some other times are something like 'My name is...', 'Please adopt...', etc., or even some kind of codes or identifiers. 1311 out of 14993 profiles did not specify anything as name or it directly was 'No Name' (actually, we can consider that at least 1578 of them didn't have a proper name, that is, something with 3 or more letters), while some names that are repeated more than 50 times are 'Baby', 'Lucky' or 'Brownie'. The effect of having or not a proper name (text of 3 or more characters) is subtle, not having a name increased approximately a 3% the number of '4' cases. Moreover, no significant effect on *AdoptionSpeed* was detected checking the length of *Name* as it was, neither when *Name* had a single word. Thus, **the only worth information that we should extract from this variable is whether it is specified or not**.

The other text variable is ***Description***, and from this one, as we could expect, valuable information can be extracted. If we take a sample of the descriptions, there are different approaches: **speaking in first or third person, providing information about the context of the pet** or being redundant with the information that is already included in tabular data, etc. In a small sample, when there was redundant information, we checked that it was captured in the tabular data in the same way. The number of profiles that didn't have a description was very small, just 12 out of 14993, and from those remaining 14981 cases, the most frequent words by *Type* can be seen in Figure 2.8. There are words that are very recurrent, so more information can be extracted from those that are not that as common, like 'street', 'stray', 'abandoned', etc. The mean number of characters of characters is 339, approximately 63-64 words (75% of the profiles have 81 or less total words). **The effect of the description length is not very significant**, when the number of words is greater than 81 the percentage of '4' cases decreases just a little to 26.1%, but nothing remarkable.

If we use the Pandas Profiling tool[3], the 6.3% of the descriptions are repeated (some of them are short descriptions like '*For Adoption*', '*Dog/Cat for Adoption*', '*Friendly*' and also some template descriptions that some rescuers apply like '*Please feel free to contact us : Stuart*'); some of the descriptions have emojis (over 250 of such descriptions were found, but there are probably more because some of them were not detected as emojis), even though the effect on *AdoptionSpeed* is not very clear (the percentage of '4' cases slightly increases but the adoptions didn't necessarily take place earlier).

In addition to the information we could extract using TF-IDF or word embeddings, as we will see in the next chapter, the main information that can be extracted from the descriptions are described in the next section.

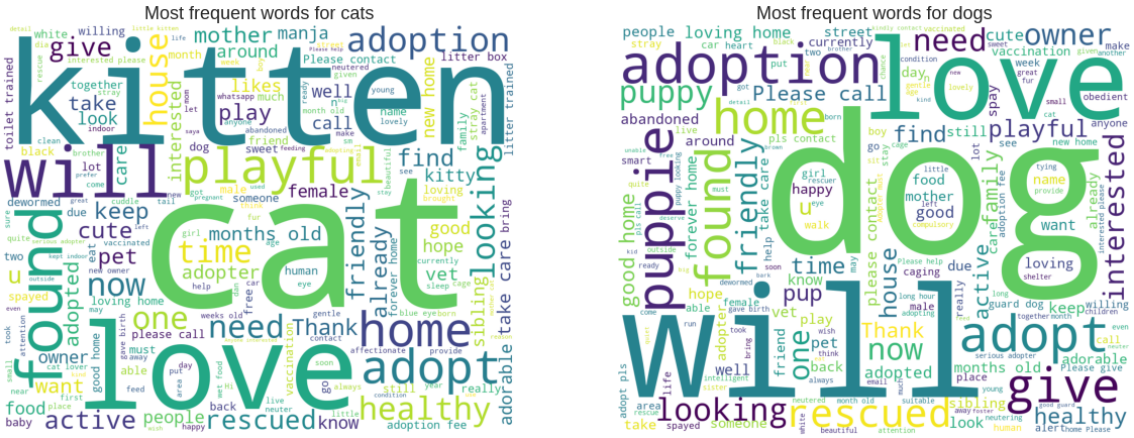

Figure 2.8: Most frequent words in *Description* by *Type*

2.3. Sentiment analysis

Each ***Description*** text was sent to the Google's Natural Language API[4] by PetFinder.my, and they have provided us with the JSON files containing the analysis of those texts. In particular, Google's API performed two types of analysis (there are more that wasn't requested) on the descriptions, which we will summarize:

- **Sentiment analysis**: each sentence in the description is given two measures of "sentiment", **magnitude** and **score**. The former gives an estimation of the sentence's "strength" of emotion, regardless of whether it is positive or not, so it is an absolute measure in the $[0, +\infty)$ interval; the latter is an estimation of the sentence's emotional leaning, between -1.0 (negative emotions) and 1.0 (positive emotions). An

[3]https://github.com/pandas-profiling/pandas-profiling
[4]https://cloud.google.com/natural-language/docs/basics?hl=en

example of sentence that is given a negative score is the following: "*I found Pup with 2 other brothers who were abandoned and were in terrible condition*". Then, a value for both variables is also given for the entire text. The clue of these variables may be the interaction between them, because a score of 0.0 could mean that the text is neutral in emotions if the magnitude is closer to 0.0 or that it has mixed emotions if magnitude is higher (positive and negative emotions cancelling out each other).

- **Entity analysis**: this is a classification of different entities including its **name** (words in the text), **type** (which can be a person, a location, other, etc.), **salience** (the relevance of the entity in the text), and whether it is a proper name or a common name. From all this information, we have only extracted the **number of detected entities** and the **maximum single salience of any of the entities in the text** (we cannot reliably extract the salience of the entity that represents the pet because sometimes it is labelled as a 'person' if its name is mentioned, and some other times as 'other').

  Moreover, the API also provided the **language** string code that it detected in the text.

The percentage of descriptions that the API could not analyse is the 3.6% (539 cases), and the reason was simple: they were written in Malay or in a mix of English and Malay. Thus, in the Feature Engineering stage **we have to impute the corresponding missing values** (because there's actually text, it is not the same case as a missing description). In particular, the 95.6% of the dataset's descriptions were written in English, and the remaining mainly in Chinese or Mandarin Chinese. As the English ones are the vast majority, no effect on *AdoptionSpeed* can be seen; **in the case of Malay, the proportion of cases with an outcome of '4' increases a little bit, but when the adoption took place they were generally adopted earlier**. The case of the profiles with Chinese descriptions is interesting, the small sample may not be very representative, but the proportion of '4' cases increases over 50% (if we conduct a quick search on Google, we can see that there exist/existed disputes between ethnic Malaysian people and Chinese Malaysians).

The mean **score** observed in the training set is 0.27 (let's say that **descriptions are in general more positive than negative in emotions**), and the proportion of '4' cases was smaller when the score was closer to 0.0 than when it was closer to 1.0 (it seems that very positive descriptions may be taken as an 'advantage' or as not needing earlier help). In general, there are few descriptions with a negative score.

The other variable, **magnitude**, didn't seem to give much information at least by itself (in fact is similar to the description length variable, they are moderately correlated, and even if we normalize the magnitude dividing by the description length it is still questionable). Moreover, the number of detected entities is even more correlated with the length, so this one is directly discarded as it won't provide additional information.

Finally, the maximum single **salience** of any entity in each description yields interesting results: when it is strictly between 0.0 and 1.0, the proportion of '2' and '4' cases is more or less the same, but when it exactly 0.0 or 1.0 the proportion of '4' cases increases. The main reason is that they represent very short descriptions without nouns (mainly adjectives or short sentences with verbs but not explicitly mentioning the subject) or with just one noun or distinguishable entity, respectively.

2.4. Image data

This section, and in fact also the rest of this chapter, have a common subject: the images attached to each pet profile, which is the source in which more computing time has been spent to extract information, as we will see in the next chapter. In Figures 2.9 and 2.10 we can see a sample of cats and dogs images, each one with the ***PetID*** of the profile (the image filename is identified by the *PetID* and the number of image, and always the first one is the profile image or the one that is selected to be displayed in the listings before accessing the profile). There are in total 58311 images, 14652 of them are the 'profile image' or the first image that is shown before accessing the profile.

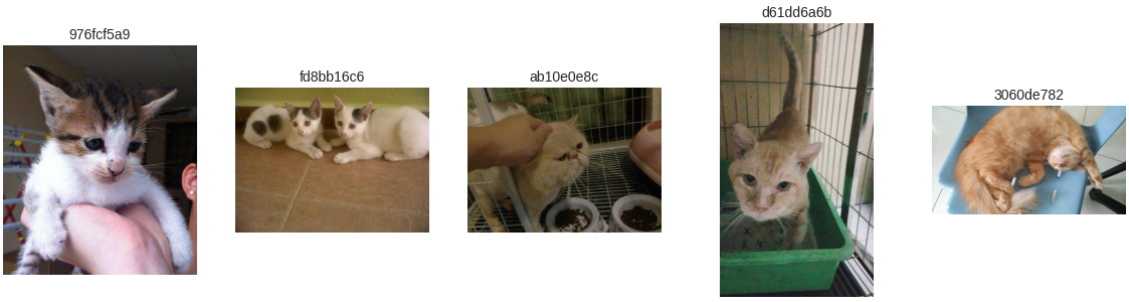

Figure 2.9: Sample of profile images of cats

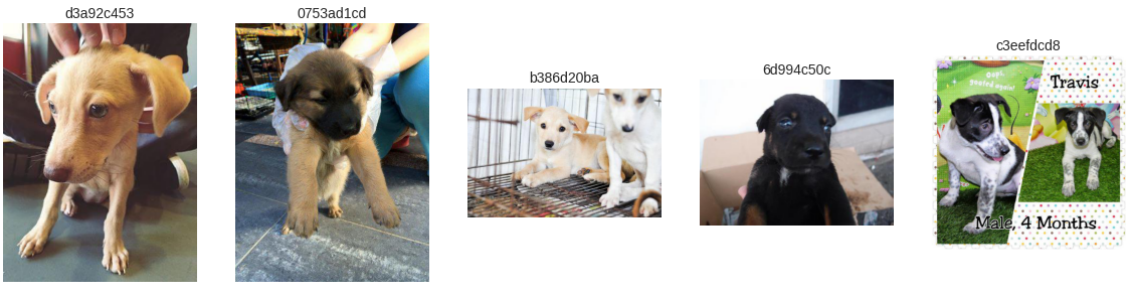

Figure 2.10: Sample of profile images of dogs

We can see how different images have different aspect ratios, lighting conditions (the third and fourth images in the sample of cats are very dull or muted, and they may not be very attractive to potential adopters), some of them are collages or have text, some others may not even show the pet's face or the pet may only occupy a portion of the image and not be the main 'protagonist'. All these conditions, however, may not be necessarily bad

to extract information, because we are not dealing with the task of predicting whether the pet is a cat or a dog, or the breed of the pet (because we already have that information), but rather **we want to extract, somehow, the presence of those conditions in the images (environmental, context or background properties may be important too)**.

When exploring other samples discriminating by *Age*, for example, we saw that some profile images showed more than one pet, which may be consistent with the fact that the profile represented a group of pets; however, there are some others that, for example, give up for adoption a puppy but in the image it is easier to spot the mother than the puppy. Some profile images show a single pet, but the profile represents a group of pets (and **this may be a greater disappointment than if the image already shows multiple pets**), and also the opposite happens: the image shows the same pet multiple times. In many images we can also see that the pet is in a cage or a cage is in the background, which happens more often when the rescuer has already published multiple profiles (again, the assumption of the fact that higher *RescuerCount* values may indicate that the rescuer is a shelter or someone linked to a shelter).

In general, **the first image (or simply the 'profile image') is the one that presents better lighting conditions, sharper quality and greater focus on the pet when there is more than one image**. However, other images can also provide useful information, and this is the case that we can see in Figure 2.11. In the profile image, there is nothing worrying or exceptional, but in the second one we could see a big, dreadful wound in the head of the dog (it has been pixelated due to obvious reasons). If we check the *Health* value of this profile, it was 'Healthy', but the second image clearly shows us that it wasn't 'Healthy' before, and even things that we already mentioned like the illumination are worse.

Figure 2.11: Example of how additional images may give more information

2.5. Image metadata

The next source of data is the analysis performed by Google Vision API[5] on each image included in a profile. This API provides so much information on the image:

- Possible crops of the image. This information was not extracted nor analysed.
- Whether a human face was detected or not, and for each one the detection confidence and a detailed report on the probability of different emotions like being in anger, joy, wearing something on the head, the bounding polygon of the face in the image, the estimated angles of the image, and even the detailed position of every part of the face or head.
- Possible text annotations in the image, their position, description or the raw text (which sometimes is not provided or certain parts of the string are modified to hide personal information like phone numbers and emails) and its language.
- Dominant colors in the image, in RGB, each one with its pixel fraction.
- Label annotations: this is similar to the entities in the text metadata, that is, a list of objects or things detected in the image, each one with a description, an opaque entity ID used by Google and its topicality (according to Google, the relevance of the entity in the image according to the context and the other labels).

The percentage of images with at least one detected **human face** is 1.1% of the total, and 1.4% of the profile images (the actual faces in the images are blurred for anonymity purposes). This is a small sample, but in fact it is actually smaller: from a sample of 20 profile images, we observed that 10 of them did not include any human face (see Figure 2.12). Thus, we extracted the detection confidence from each face, and there was a more or less clear threshold: below that value most of the images in the sample did not contain any human face. However, filtering by that threshold left only 74 profile images (still some of them didn't show any human face), and as the distribution of values of *AdoptionSpeed* in that sample was almost identical to the original, **this information was discarded** (even if it is a small sample, if there had been a huge difference in the response of *AdoptionSpeed*, we could have kept it).

Regarding the presence of **text** in the images, the 5.8% of the total have text, in particular the 7.3% of profile images have text. The presence of text may be related or not to the adoption, that is, sometimes the image is a **collage where information of the pet or the rescuer is given** (like the pet's name, contact information, etc.), but some other times is just an advertisement in the street, the serigraph of a T-shirt, etc. When discriminating the outcome of *AdoptionSpeed* by the presence or not of text, it seems that if there is text (in the profile image), the probability to be adopted earlier is slightly smaller and that of ending up without adoption slightly higher. It is not clear whether this is due to chance or not (at least the sample size should not be a problem in this case), but it could be due to the fact that **potential adopters may prefer to see a clean image of the pet** instead of a collage or an image overloaded with text.

[5] https://cloud.google.com/vision/docs/reference/rest/v1/AnnotateImageResponse

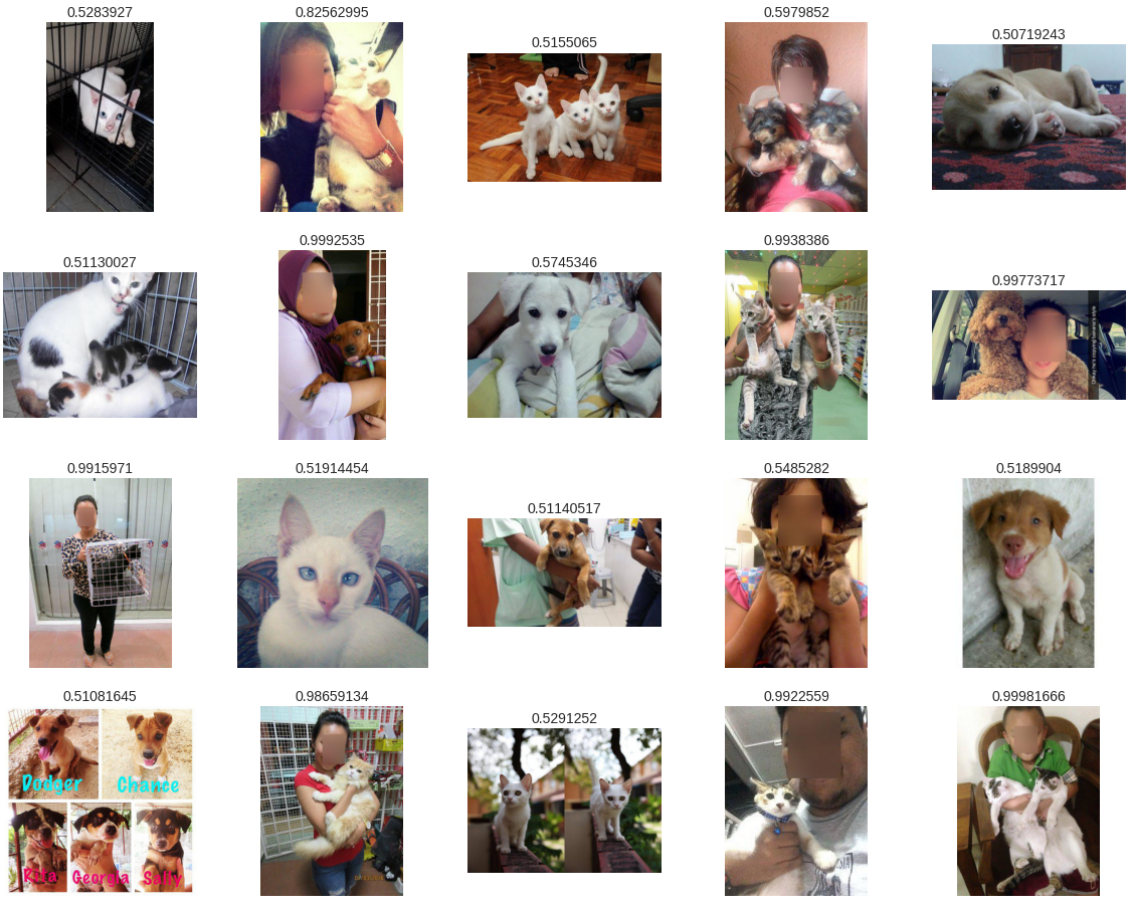

Figure 2.12: Face detection confidence in profile images (sample)

Other information that we have extracted from the image metadata is the presence of **dominant colors**, in particular the addition of the pixel fraction of all the detected dominant colors. This way, the variable will probably indicate how monotone is the image (closer to 1.0 would mean that there are few different colors in the image; if it is closer to 0.0, then there is a great variety of colors), even though this would be an estimation since there are 2^{24} different colors in RGB format, and the variation between some of the most dominant colors may not be detected as such by the human looking at the image. The response of *AdoptionSpeed* given the values of this variable is interesting: when the fraction of dominant colors is between 0.8 and 0.95 approximately (probably a **monotone image**), **the proportion of '4' cases is higher** than that of '2' cases, but when it is between 0.4 and 0.6 approximately, the opposite happens. Moreover, it seems that the usual range of values and the median of the pixel fraction is slightly higher when the earlier the adoption took place, but when they ended up not being adopted this observation does not hold (there are many of these cases with high sum of pixel fraction).

Finally, the main source of valuable information from the image metadata is the detected **entities** or labels. From these data, we have extracted three pieces of information:

maximum pet topicality (that is, the maximum topicality among the labels whose description contains 'cat' or 'dog'), the **number of entities** (regardless of whether or not they are related to the pet) and the concatenation of the **entities description**.

Regarding the **maximum pet topicality**, first of all, in 1.5% of the images no pet was detected (0.0 value), and in particular the 0.9% of the profile images (this is consistent with the fact that the first image is typically the most representative one). The effect on *AdoptionSpeed* **when no entity was detected in the profile image** is clear: **the percentage of '4' cases is higher** (almost 35% of those), which is mainly balanced out by a decrease of '2' cases (20%). Of course, the effect is not that significant as not having a profile image, because we assume that a human will probably detect the pet whenever it is actually in the image. When looking at a sample of such profile images, there are several reasons why Google's IA could not detect the pet:

- The pet cannot be seen at all, or it occupies a small fraction of the image.
- The pet's face is not shown.
- The pet or its fur color is camouflaged or can be confused with some elements of the environment.
- Bad image quality or illumination.
- The image is a collage and the pet is not the main focus.
- Mixture of the previous conditions.

When the maximum pet topicality is greater than zero, even if this variable is highly left-skewed (it is very unlikely that Google's IA don't detect the pet, some of the previous conditions must happen), the effect on *AdoptionSpeed* can be seen in Figure 2.13: **the higher the maximum pet topicality, the more likely the adoption takes place earlier**. Of course, in that view we cannot see the absolute density of each class (even if the previous statement holds, of course it is more probable that it is actually a '4' case instead of a '0' case because there are far more instances of the former than those of the latter), but with the interaction and filtering capability of other variables, this one could make it more clear.

Regarding the **number of entities**, in 73.2% of the profile images, Google's IA detected 10 entities, while it is very unusual that it detects 4 or less entities. For the different numbers of entities detected in a representative sample of profile images, **the smaller the number of detected entities, the more likely it was for the profile to end up not being adopted**. Even the balance between the proportion of '1' and '3' cases decreases in favour of the '3' cases as the number of entities decreases, at least for those with sufficient a representative number of instances.

The final variable that was analysed is the concatenation of the **entities description** of each image. If we create a cloud of words similar to that of Figure 2.8 for this variable (only profile images), we can see that there are many redundant words (different labels can be very similar, for example 'cat' and 'cat like mammal'), and also that some of the Breed values could have been extracted from this source, specially in the case of cats (the most repeated words are 'domestic', 'short', 'medium', 'hair', etc.). Thus, **this information is not very relevant. However, it is useful to fix a particular anomaly**: some of the profiles in the

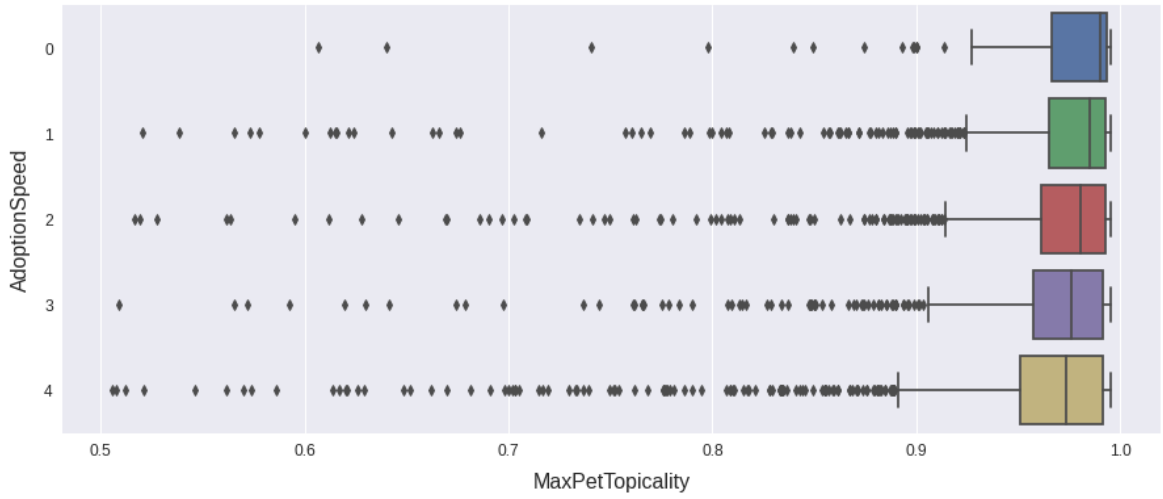

Figure 2.13: Usual range of maximum pet topicality values for each outcome of *AdoptionSpeed* (only > 0.0 cases)

dataset specify a *Type* of pet that was not detected by the IA in the profile image (in those cases where a pet was actually detected, of course, given the value of the maximum pet topicality). A sample of 20 of such cases was extracted, and in their profile images 15 were actually cats (and their *Type* was 'Cat', but the IA detected a dog), only one profile image was labelled as 'cat' but the *Type* was actually a dog, and there were 4 cases where the IA detected the correct *Type*, but those profiles had a wrong *Type* value. Thus, the concatenation of the entities description can be used to detect these anomalies and then extract the correct *Type* value from the *Name* or *Description*, check whether their Breed is correct according to their *Type* or just labelling them as Cat since in the sample the most of the IA's errors were labelling a cat as 'dog'.

2.6. Extra image properties

The last source of data comes from the analysis of different image properties, in particular the **dullness**, **whiteness**, **blurriness**, **size**, **width** and **height**. These features are not provided by PetFinder.my; we extracted them from the images using the code from a Kaggle user, who shared it in the notebook that we address in the Annex.

The **dullness** and **whiteness** are a measure of the proportion of dominant colors that are below or above an established threshold, respectively. In particular, the image is divided in two halves; then, a number of the most dominant colors is extracted from each one (at most 1000, for example) and we count the fraction of them that are below the RGB threshold of (20,20,20) and above the RGB threshold of (240,240,240) (these thresholds determine what is 'dark' and what is 'bright') in order to determine the dullness and whiteness of each half, respectively. Finally, the arithmetic mean of the score of each half is computed; by doing

this, the bias towards very dominant dark or bright colors is reduced.

Of course, depending on the value of the thresholds we will get different results; in our case, there are many profile images with a value of 0.0 in dullness, whiteness or even both, if all the most dominant colors are between the upper and lower bounds, which also implies that both values can be low, Figure 2.14 is an example. All the elements in the image may influence the value of both variables, as we can see with the black rag and the white object in the background.

Figure 2.14: Example of the dullness and whiteness values on a profile image

Within the instances with a profile image that is very dull (more than 0.7), the proportion of '4' cases increase, while the median of whiteness discriminated by the outcome of *AdoptionSpeed* seems to be slightly higher as the adoption takes place earlier.

The next variable is **blurriness**, which is obtained by convolving the Laplacian kernel (2.1) over one channel (grayscale) of the image. The actual value that we get is the variance in

the response of all those convolutions. This kernel is used to measure the second derivative of an image and, consequently, it estimates how rapid intensity changes are, which is higher as more defined edges there are in the image. Thus, small values indicates blurrier images. This quick method is described in [Pech Pacheco et al., 2000].

$$\frac{1}{6}\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{2.1}$$

As we can see in Figure 2.15, in the usual range of values, surprisingly, **the smaller the blurriness value, the higher the proportion of cases where adoption took place earlier**. If we take a sample of the supposedly sharpest images, all of them were taken in a natural environment or with the pet on grass or dirt (maybe the amount of edges there is higher). In fact, the 38% of the 1000 instances with sharpest profile images ended up not being adopted, so this may favour again the narrative on potential adopters seeking pets that seem to be in worse conditions.

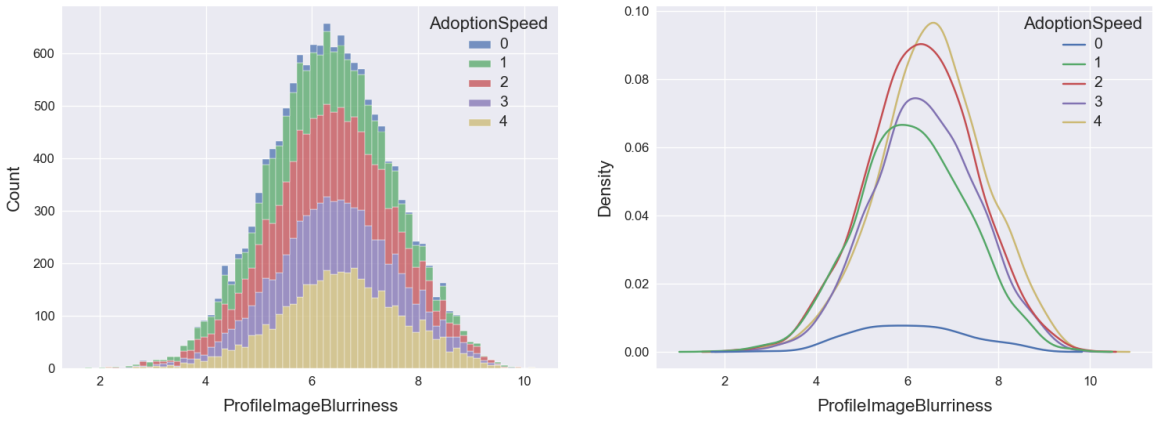

Figure 2.15: Blurriness values in profile images and response of *AdoptionSpeed* (log scale)

Another proxy measure of quality is the image **size**, which in fact has a similar distribution to that of blurriness: in its usual range of values, the smaller the size, the higher the proportion of cases that were adopted earlier. If we plot both variables, size and blurriness, there are certain areas where the number of '4' cases is smaller than that of '2' or '1' cases, so these two variables could work well together (even if they are similar they are not very correlated).

The two final variables are the image **width** and **height** in pixels: both variables are far from being evenly distributed, they have peeks of values mainly around the hundreds. The one that seems to give more information about *AdoptionSpeed* is the height: when it is around 300 or 400, the density of '4' cases is smaller than that of each one of the rest of majority classes ('0' class excluded), but when it is around 500 pixels, the density of '4' cases

is again the greatest among all the classes, and also the density of ’1’ cases decreases and that of ’3’ cases increases. A straightforward way to relate these two variables is the **aspect ratio** (width/height): it shows that instances with a profile image that has a greater height than width are more likely to be adopted earlier or to end up being adopted than those with profile images with a smaller aspect ratio.

3. Feature Engineering

Once we know the problem domain and we have carried out an Exploratory Data Analysis, we will use the knowledge we acquired from those stages in order to clean, transform and process the appropriate data in such a way that, given the requirements of each estimator, we can create robust models and increase the performance they would have without these steps. Moreover, some algorithms cannot handle certain characteristics of the input features, so this process is even more necessary.

Data Preparation or Data Preprocessing are similar terms that could be used instead of Feature Engineering; we use this one because the others are mainly used when we transform and encode the data, but this process goes further: encoding some of the features, creating new ones, discarding others or selecting subsets, extracting features from unstructured data, testing them, etc.

In particular, we will describe each particular technique when we actually use it in the pipeline[1], as many of these techniques are very different or varied among them, so it would not make sense to talk about those or the State of the Art of some of them at the beginning.

All the work that we will describe in this chapter can be found in **Notebook 2** (see Annex); this is actually a summary of that notebook, extended with the description of each technique and why we use it. Moreover, all the final transformers can be found in the **Transformers utility script**, also in the Annex, if the reader only wants to see the code of the subset of techniques that will be used in the next chapter (including the description of the hyperparameters).

3.1. Base tabular data

We start with the tabular data of each pet profile, directly extracted from PetFinder.my and provided by them. As we mentioned in the previous chapter, all the base tabular data is numerical (except the ID hashes, which are the *PetID* and *RescuerID* variables), regardless of whether certain variables are actually categorical or not. Thus, the first step in the pipeline is a simple one, it consists in **replacing the numerical values of the categorical variables to**

[1] https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

strings. This also implies loading the external .csv files containing the mapping between the number and the string for the variables *Breed1*, *Breed2*, *Color1*, *Color2*, *Color3* and State (the other mappings are manually extracted from the PetFinder.my data description, as we saw).

Another simple step in the pipeline is replacing the *Name* text variable by a boolean variable which simply tells **whether the profile had a significant name or not** ('significant' in this case means those *Name* values that are not null, have 3 or more characters and they are not 'No Name' or equivalents), as this is the only valuable information we could extract from it.

The first real burden that we have to overcome is **how to deal with** the variables ***Breed1* and *Breed2***. First of all, they are categorical, but some algorithms or rather some of their implementations can only handle numerical or continuous variables (for example, SVM, XGBoost, Linear and Logistic Regression, etc., and we will use some of them). This is a common problem, not only for these variables. There are two main strategies to overcome this problem:

- **Ordinal (or integer) encoding**: each unique category is assigned a different integer value, usually from zero to the number of categories minus one. However, this type of encoding establish an explicit order of the categories, which sometimes is not the case (certainly not for *Breed1* and *Breed2*). We can think of a decision tree trying to find a split in a categorical variable encoded like this: if its value is smaller than 2, then X, else Y. However, the 0 and 1 values may correspond to 'Mixed Breed' and 'Tabby', two different breeds with no relation between them. Thus, this encoding method is suitable for categorical variables in which there exists a natural ordering of their values, for example the variable *MaturitySize* in our case (because a value of 'Small' is smaller than 'Medium', which is smaller than 'Large', which is smaller than 'Extra Large', so they can be mapped to the values 0, 1, 2 and 3, respectively).
- **One-hot encoding**: each different unique value of the categorical variable is transformed into a new variable, such that all the instances who had a particular value will have a '1' in its corresponding new variable, and a '0' in the rest of variables corresponding to the rest of unique values. This way, no explicit or implicit order in the value is assumed (now, a decision tree would just check whether the breed value is 'Tabby', or not 'Tabby', and that for every other possible value). Thus, this could be a correct way to encode *Breed1* and *Breed2*.

However, the aforementioned variables also have another disadvantage: many different unique values. In particular, in the complete training dataset *Breed1* has 175 different values and *Breed2* 134. If we only one-hot encode *Breed1*, as more than two thirds of *Breed2* are null values, we would still get 175 new variables from just one original variable, with the additional disadvantage that some of them would be almost invariant (those variables that would correspond to the minority breeds with as few as 1 or 2 instances in the training dataset). If we don't transform them someway, this method is not very affordable. A way to overcome this problem could be using **clustering** techniques, such that different unique values which have similar statistics with the target variable are grouped. However, by doing this we could lose some intra-cluster important information.

Consequently, on the basis of the above we have implemented two different ways to encode the Breed. The first one is based on the method described in [Micci-Barreca, 2001], which is known as **Target encoding**. This encoding method is particularly useful for high-cardinality variables, because the number of new features only depends on the cardinality of the target (when it is multi-class, which is the case) and it is similar to clustering since values that have similar statistics with the target variable have similar encoded values, but in contrast to clustering, this method yields more granularity between categorical values. In particular, and using the target variable as categorical instead of numerical (that would also be acceptable for this problem), the encoding of the high-cardinality variable is done following these steps (assuming a binary target):

1. Obtain the estimation of the probability of positive cases given the value X_i of the variable as the proportion of positive cases over the total of cases discriminated by $X = X_i$. That is:

$$X_i \to S_i = \frac{n_{iY=1}}{n_i} \cong P(Y = 1|X = X_i)$$

2. Regularization: blend the prior ($P(Y = 1)$) and posterior ($P(Y = 1|X = X_i)$) probabilities in order to mitigate those values found in a small sample size that do not give reliable estimations.
3. Substitute the corresponding categorical values by their encoding obtained in step 2, or by the prior probability when they don't have a mapping. This is the way we can handle unknown (unseen in training) values.

A simple example is shown in Figure 3.1, with an additional step (the first one): as in our case the target has multiple categorical values, we have to one-hot encode it and repeat the previous steps for every value of the target (even though Figure 3.1 does not include the step 2). Blending the prior and posterior probabilities is necessary because the posterior probability is not reliable when the unique value of the variable is found in few instances, especially in our case (multi-class). Imagine that in a dataset of 14993 instances, which is the size of our training set, there is a breed that is only found in three instances: two have an *AdoptionSpeed* value of '0' and the other has a value of '3'. In this case, we would prefer the encoded values to be more similar to the prior probability, in order not to overfit to those minority breeds.

The way these probabilities are blended in the implementation we have used[2] is very similar to the one proposed in [Micci-Barreca, 2001], and is using the formula 3.1. As the value of $\lambda(n_i)$ has to be bounded between 0 and 1, it is computed using the formula 3.2, which is actually the sigmoid function. In the latter formula, k is called in the implementation `min_samples_leaf`, as it is half of the minimum sample size to "trust" the estimation of the posterior (if $n_i - k$ is greater than 0, considering that $f = 1.0$, then the posterior gets a greater weight than the prior in the formula, and vice versa if it is smaller than 0; if it is equal to 0, then we can see that the prior and posterior get the same importance, 0.5 each

[2] https://contrib.scikit-learn.org/category_encoders/_modules/category_encoders/target_encoder.html

one). The f value is called `smoothing` and must be greater than 0; it indicates how strong the regularization is.

Breed	AdoptionSpeed
Mixed Breed	1
Tabby	0
Tabby	4
Mixed Breed	2
Mixed Breed	3
Mixed Breed	2
Mixed Breed	4
Tabby	2
Tabby	1
Tabby	1
Tabby	3
Tabby	4

Breed	AdoptionSpeed_0	AdoptionSpeed_1	AdoptionSpeed_2	AdoptionSpeed_3	AdoptionSpeed_4
Mixed Breed	0	1	0	0	0
Tabby	1	0	0	0	0
Tabby	0	0	0	0	1
Mixed Breed	0	0	1	0	0
Mixed Breed	0	0	0	1	0
Mixed Breed	0	0	1	0	0
Mixed Breed	0	0	0	0	1
Tabby	0	0	1	0	0
Tabby	0	1	0	0	0
Tabby	0	1	0	0	0
Tabby	0	0	0	1	0
Tabby	0	0	0	0	1

Breed value	Mapping_0	Mapping_1	Mapping_2	Mapping_3	Mapping_4
Mixed Breed	0	0.2	0.4	0.2	0.2
Tabby	0.143	0.286	0.143	0.143	0.286
Prior	0.083	0.25	0.25	0.167	0.25

Breed	Breed_target_0	Breed_target_1	Breed_target_2	Breed_target_3	Breed_target_4
Mixed Breed	0	0.2	0.4	0.2	0.2
Tabby	0.143	0.286	0.143	0.143	0.286
Tabby	0.143	0.286	0.143	0.143	0.286
Mixed Breed	0	0.2	0.4	0.2	0.2
Mixed Breed	0	0.2	0.4	0.2	0.2
Mixed Breed	0	0.2	0.4	0.2	0.2
Mixed Breed	0	0.2	0.4	0.2	0.2
Tabby	0.143	0.286	0.143	0.143	0.286
Tabby	0.143	0.286	0.143	0.143	0.286
Tabby	0.143	0.286	0.143	0.143	0.286
Tabby	0.143	0.286	0.143	0.143	0.286
Tabby	0.143	0.286	0.143	0.143	0.286
Shih Tzu	0.083	0.25	0.25	0.167	0.25

Figure 3.1: Toy example of Target encoding without regularization

$$S_i = \lambda(n_i)\frac{n_{iY=1}}{n_i} + (1 - \lambda(n_i))\frac{n_{Y=1}}{n} \tag{3.1}$$

$$\lambda(n_i) = \frac{1}{1 + e^{-\frac{(n_i - k)}{f}}} \tag{3.2}$$

In particular, the transformer that we have implemented only encodes *Breed1* in this way, as *Breed2* has many null values and we also extract information from it in two previous transformers: one is used to create a **variable which indicates whether the pet has a pure breed or not** (we defined it in the EDA as those instances which have a non-null value in both *Breed1* and *Breed2*, or any of the two values is 'Mixed Breed', 'Domestic Short Hair', 'Domestic Medium Hair' or 'Domestic Long Hair'); the other transformer is used to **impute values of *Breed1*** using the value of *Breed2* if is not null, or just the most common Breed

discriminated by *Type* in the training dataset (as we saw that only 5 values of *Breed1* were missing, and in all those instances there was a value of *Breed2*, we won't use a more complex imputation strategy, as we will see in the next section). Moreover, in addition to Target encoding, we also add a new variable which indicates how frequent that *Breed1* value was in the training set (this is known as **Count encoding**, as we saw with the variable *RescuerCount*), and if it was not seen during training, then its value is set to the most repeated count seen in training.

The second way or option that we have implemented in order to encode the Breed variables is using **one-hot encoding** (which would yield at most, if we train with the entire training set, 175 new variables from *Breed1* + 134 new variables from *Breed2* + 1 new variable for the null, 'NaN', value in *Breed2*), and then, in order to reduce the number of obtained variables, we use the **truncated SVD** implementation of the Python package *scikit-learn*[3].

We have applied truncated SVD not only to reduce the dimension of the one-hot encoded Breed variables, but also to reduce the number of features of other data, as we will see in the next sections. Thus, it is worth to see some of the fundamentals of this technique (there are many mathematical details and complex transformations involved under the hood depending on the implementation of this technique; however, this is not the scope of our project, so a brief description is given).

SVD is a technique used to **decompose matrices**, like others such as eigendecomposition, even though the latter one is not used to compress the original matrix, but rather it is suitable to perform other operations. However, SVD and more specifically truncated SVD allows us to compress a matrix, which is what we want in this case (compress the number of variables associated with the one-hot encoded breed variables while keeping as much information as possible from all the instances).

In particular, SVD, Singular-Value Decomposition, decomposes an original matrix A of size (m x n) in 3 new matrices, as shown in equation 3.3, where the matrix U has a size of (m x m), Σ is a (m x n) matrix (or n x n, which is extended to m x n by adding m-n rows filled with zeros) and V^T is the transpose of a matrix of size (n x n). U and V^T are known as 'singular vectors' while Σ contains the 'singular values' in its main diagonal (it is a rectangular diagonal matrix). All matrices, rectangular or square, have a singular-value decomposition, even though some of them could involve complex numbers.

$$A = U \cdot \Sigma \cdot V^T \tag{3.3}$$

Where do we find the compression of features? To get such reduction we need to compute the SVD of the original matrix and then, we truncate the U matrix to k columns, Σ to k columns and rows, and V^T to k rows, as shown in Figure 3.2[4], where $k << n$. The multiplication of U_k, Σ_k and V_k^T would be an approximation of A of size (m x k). The key here is selecting the top k singular values of Σ to get the best possible approximation of A (the

[3] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html

[4] Extracted from https://medium.com/analytics-vidhya/master-dimensionality-reduction-with-these-5-must-know-applications-of-singular-value-777299940b89

implementation already sorts an places them in the diagonal of Σ). In fact, in the implementation we only need to store V_k (the transpose of V_k^T), since if we multiply the original matrix by that one we will get a summary or projection of it. This way, we can **fit the SVD decomposition (that is, create the decomposition and store the V_k matrix) and obtain the transformations of any matrix with the same original features** (of course, as with any other stateful or data-dependent transformation, if the distribution of each feature is not similar to the one seen at fitting or training time, the approximation won't be very accurate, or in other words, some important information could be lost).

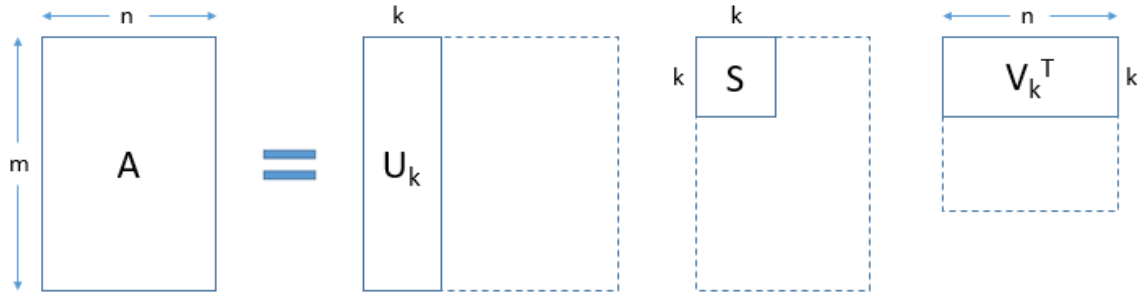

Figure 3.2: Truncated SVD for dimensionality reduction

Of course, the hard part of SVD is obtaining that decomposition. The *scikit-learn* implementation uses by default a randomized algorithm which computes the decomposition (in fact, an intermediate matrix to get the actual decomposition) iteratively. This algorithm is described in [Halko et al., 2010] and [Szlam et al., 2014], and mainly the iterations consist in applying other matrix decompositions to transform the multiplication of an initial matrix of randomly valued vectors (from a Gaussian distribution) and the matrix to be decomposed, and then the same process but multiplying by the transpose of the matrix to be decomposed. After a number of such iterations, we get an orthonormal matrix with approximately the same range of the matrix to be decomposed, which is then multiplied by the matrix to be decomposed; the result of this multiplication, which has a smaller number of features (in fact, k plus an additional small number as oversample to get better results), is the one actually decomposed using SVD based on a Divide and Conquer strategy (algorithm 4 in [Dongarra et al., 2018]).

Finally, the actual reason why we have implemented this way of encoding the breed variables is the fact that the *scikit-learn* implementation of truncated SVD can work with sparse matrices very efficiently, and the **one-hot encoded breed variables is a sparse matrix** (in the complete training dataset of 14993 instances with one-hot encoded *Breed1* and *Breed2* variables, there would be at most 29986 values equal to 1, if all the pet profiles had non-null *Breed1* and *Breed2* values, which is not the case, and 14993 · (175+134+1) - 29986 =

4617844 values equal to 0), so the **breed-variables dimensionality reduction takes a short time**.

Once we have encoded the breed variables, the other ones are encoded in the following way:

- One-hot encoding: *Gender*, *Color1*, *Color2*, *Color3*, *Vaccinated*, *Dewormed* and *Sterilized*. All of them have 7 or less different unique values, so in this case this type of encoding is acceptable.
- Ordinal encoding: *MaturitySize*, *FurLength* and *Health*, since the values of these variables can be arranged in an intuitive order. However, even if in the complete training dataset no instance have a 'Not Specified' value in any of these variables, we would change that value at transformation time by the most repeated one (for each variable) seen at training time, since it breaks the order.

The lasts transformers that are implemented as part of the first pipeline are used to **replace the State variable by the GDP per capita** of each Malaysian State (or the overall Malaysian GDP per capita if a State was not found in any instance at training time, extracted from the same source), **replace *RescuerID* by its Count encoding** (those unknown values at transformation time are replaced by the most repeated count value at training time), **remove the variables that have already been encoded** or that are not useful (for example, *PetID*) and a transformer to **scale all numerical features so that they have the same mean (0) and standard deviation (1)** (by subtracting the mean and dividing by the standard deviation of the variable). This last transformation is particularly important for some models which are sensitive to the scale of the features, especially those based on similarities or distances.

As we will see in the first section of Chapter 4, we have evaluated with several models each pipeline extension starting from the previous transformers and then adding the ones described in the following sections, in order to find which learners perform better for this problem and save computing time (limited by Kaggle).

Every transformer is fitted on the corresponding training set it receives; no transformation is done before the pipeline, and surely not on the complete training dataset (which could imply a significant data leakage especially in the case of those transformers that depend on frequencies like the encoding of *RescuerID*).

3.2. Additional tabular data

In order to add the supplementary data extracted from the texts of the *Description* variable and the images included in the profiles, all of them provided by PetFinder.my, we have implemented a number of transformers.

3.2.1. Description metadata

First of all, we add the tabular data obtained from the *Description* variable. The first transformer creates a new variable which indicates the ***Description*'s text length**. The second one

adds the variables **score**, **magnitude**, **language** and **maximum salience** (of any detected entity in the text) that we mentioned in the EDA, extracted from the JSON file corresponding to each profile *Description*. This data must be loaded outside the pipeline, before we use it, but it is included inside the pipeline. As we saw, there are few missing *Description* values, so the previous variables are set in those cases to 0 (which makes sense with the meaning of them and the fact that they are missing) and the language is left as null ('NaN'). However, the *Description* texts written in Malay language could not be analysed and consequently they also miss those values; this condition is found in the 3.6% of profiles in the complete training dataset, so it is a large proportion of cases to impute those values with a simple strategy.

There are many imputation strategies. The most straightforward is imputing the missing values using the mean of the variable in those instances which do not miss it. This strategy, however, is not suitable when the distribution of the variable is very skewed, as the mean is not a value as representative as, for example, in a normal distribution (we have to impute several variables that are very right-skewed, one of them is the *Description* **magnitude**; other such variables are image properties as we will see later), and we would also create a peek of values that could be noisy. A more complex strategy could be using k-Nearest Neighbours (kNN) to impute each missing value. This strategy consists in searching the k instances ('neighbours') in the training dataset with the smallest distance (usually the euclidean distance) to the one which has a missing value, so that this missing value is replaced by the computed average of the same variable from the k nearest neighbours (which can be the arithmetic mean or be weighted by the inverse of the distance to each neighbour). However, we have used another method, which is experimentally implemented in *scikit-learn*, **Iterative imputation**.

This strategy is based on the MICE algorithm (Multivariate Imputation by Chained Equations), originally implemented as a package in the *R* language environment, and it is described in [van Buuren and Groothuis-Oudshoorn, 2011]. The following is a summary of the steps involved in this type of imputation extracted from [Azur et al., 2011]:

1. The missing values of each variable are replaced by a temporary placeholder value, such as the mean of the non-missing values.
2. One of the variables to impute is rolled back to its previous state (that is, with missing values); the rest of variables to be imputed are not modified (still have their placeholder value).
3. A regressor is fitted on the instances with no missing values in the variable selected in the previous step, using as predictor the rest of variables.
4. The missing values of the selected variable are predicted using the regressor model.
5. Steps 2 to 4 are repeated for the rest of variables to be imputed, fixing the already imputed values of the predictor variables to either the initial placeholder or the already imputed value in a previous step or cycle.
6. When all the variables with missing values have already been imputed once, that constitutes a cycle, which is repeated a number of times.

A toy example of this process is shown in Figure 3.3[5], where the initial imputation strategy is filling the missing values with random values, instead of the mean of the non-missing values of each variable, and the regressor is a Random Forest.

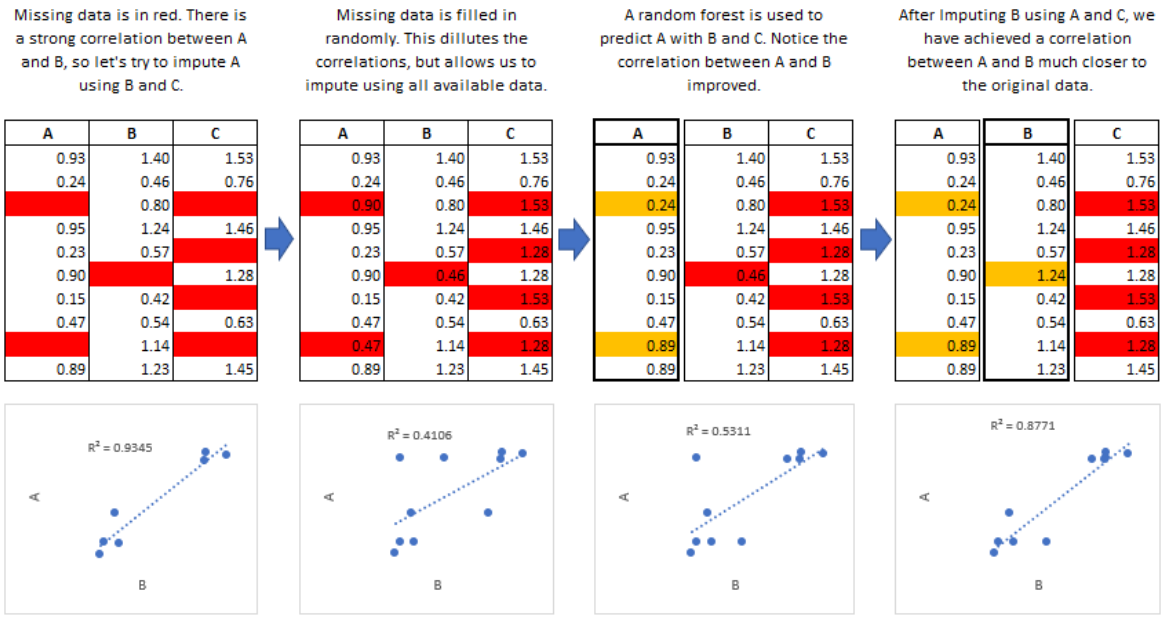

Figure 3.3: Toy example of (almost) an iterative imputation cycle

Nevertheless, there are several differences between the R package and the Iterative imputer implemented by *scikit-learn*. The most evident is that MICE creates several copies of the dataset, and then perform the entire process on each one, obtaining multiple imputations (as many as the number of processes that are performed) for each missing value. This is why in Figure 3.3 random numbers are used as initial placeholders, so that in each different process the results are different. The reason is that MICE actually conducts a statistical analysis on the different copies, and then pool the appropriate imputation based on that analysis. However, the *scikit-learn* implementation returns a single imputation (the process is performed once, in particular with a default number of cycles of 10), so it does not perform any kind of statistical analysis (in order to perform the imputation, the chain of regressors are used in the order they were created, starting from the initial imputation strategy that is using the mean).

In our transformer, we have used as regressor a decision tree limited to a maximum depth of 5, and only the data from the main database and the supplementary metadata (*Description*, images and image properties) are used as predictor variables (the data extracted using the strategies described in sections 3.3 and 3.4 are not used in this process).

[5]Extracted from https://cran.r-project.org/web/packages/miceRanger/vignettes/miceAlgorithm.html

Moreover, in order to identify the descriptions that were written in Malay after the imputation, whose **language** value is null, we have implemented another transformer to identify such cases using the ISO language code of Malay, 'ms', which will be used to translate them to English (section 3.4). In this transformer we also group those languages that are very uncommon (by default, the minimum group size to be considered as a particular language is 10 instances at training time, otherwise their language value is labeled as 'others'). Then, this variable is also one-hot encoded.

3.2.2. Image metadata and properties

The third and fourth extensions of the base pipeline are obtained by adding the Image metadata (extracted from the JSON files provided by PetFinder.my) and properties (extracted by us), respectively. As with the *Description* metadata, this supplementary data is prepared in CSV files outside the pipeline, and then they are included using the corresponding transformers in the pipeline. Moreover, as there may be more than one image included in each profile, and we saw in the EDA that additional images could provide important information, the data that is added can be just the one extracted from the profile image (the first one) or an aggregation of the data extracted from all the images included in each profile. In particular, the variables are:

- **Metadata**: sum of the pixel fraction occupied by the predominant colors, maximum topicality of any entity whose description includes 'cat' or 'dog', number of detected entities, the concatenation of their descriptions (let's simply call it the 'image description') and whether the image contained text or not. The aggregation of this data would be the mean, sum and variance of each numerical variable, the concatenation of the images description and the number of images of the profile that contain text (addition).
- **Properties**: dullness, whiteness, blurriness, image size, width and height. Each one would be aggregated using the mean, sum and variance of the values found in all the images included in the profile.

Whether to add the metadata and/or properties from the single profile image or the aggregation of all the images is evaluated in the hyperparameter tuning process of each pipeline, described in the next chapter. They are included using two different transformers in the pipeline, so we can have a combination of single profile image metadata and aggregation of images properties, for example.

Once we have included the data, it is checked to carry out a number of transformations. First of all, we need to find those **profiles whose *Type* value ('Dog' or 'Cat') does not coincide with any of the detected entities**, in order to fix that value if necessary. This is done by checking whether the *Type* value is not included in the image description, only when there is actually an image description (or concatenation of several images description if we have included the aggregated data) and the maximum pet topicality is not 0 (that is, a 'cat' or 'dog' was detected). If these conditions are met, then:

1. If any of the equivalents of 'cat' or 'dog' is found in the *Name* variable (we remove it at the end of the pipeline), then the correct *Type* value is set. The equivalents are 'cats', 'dogs', 'puppy', 'kitten', etc.
2. Otherwise, we sum the number of appearances of any 'cat' equivalents and any 'dog' equivalents, separately, in the profile *Description* text, and set the *Type* value according to the number of appearances.
3. If the number of appearances of 'cat' and 'dog' equivalents is the same, then we extract the *Type* value from the Breeds CSV file, containing the correspondence between the original integer code and the Breed value. If this value is the same as the profile's one, then we set *Type* to 'Cat' (as mentioned, we found in a sample that the majority of anomalies corresponded to the IA labelling a cat as a 'dog'); if not, then the extracted *Type* from the Breeds file is used.

Finally, regardless of the outcome of the previous process, if the final *Type* value does not correspond to the specified *Breed1* value (the *Breed2* value is not checked), then it is set to the most common *Breed1* value seen at training time in the subset of profiles with the same (final) *Type* value. Moreover, the image description variable is only used in order to detect the previous anomalies, it is eventually removed after this process.

The second transformation is a simple one, it rounds the values of the image width and height to the nearest hundred and then with those it computes the **aspect ratio** (width/height, zero if any of those are rounded to that value or there is no image included in the profile).

The only thing left to deal with is the **values of all the included variables when the profile doesn't have any image**. We have to take into account that the Iterative imputation is a costly process, a single regressor has to be constructed for each variable to be imputed, in each cycle or iteration. Thus, we have to keep the number of variables to be imputed as small as possible, only including in the imputation process those in which it wouldn't make sense to assign a pre-defined value. In particular, all the numerical image metadata is set to 0 in those cases (it makes sense to say that the number of detected entities is zero, for example), the image description is left as an empty string and the variable that indicates whether there is text or not in the image is set to false (if the aggregation is included instead, the number of images that contain text is set to zero too). Regarding the image properties, the size, width, height and aspect ratio are also set to zero; however, this is not the case for **dullness, whiteness and blurriness**, which are left as null so that they can be **imputed** iteratively.

Why those variables? First of all, they are very right-skewed, so using the mean of the non-null values is definitively not an option. We may think that this is also the case of the variable maximum pet topicality (even though that one is left-skewed), for example, but in that case grouping these profiles at zero still makes consistent our assumptions and the behaviour of this variable with respect to *AdoptionSpeed*, because we also saw in the EDA that the vast majority of profiles without images ended up not being adopted (so the distribution shown in Figure 2.13 would be maintained). Moreover, setting them as zero does not make much sense: we can say that no entity was detected because there is no image, but saying that it is zero in our Blurriness variable would indicate that it is extremely blurry, so it could negatively influence the final prediction.

3.3. Extracting features from image data

In the previous sections, we talked about extracting certain properties from the images and we are also provided with metadata about the contents of them. However, the extraction of those features is independent from our objective, which is predicting the adoption speed, in the sense that obtaining the image blurriness or detecting the entities in that image was not a process conducted taking into account our objective (rather, we have explored the data and used the data which we think provides useful information on *AdoptionSpeed*, but the Google's Vision API would have conducted the same process for any data). What we are looking for now is to extract more abstract or high-level features from the images and create a model that learns which features to extract based on our target variable.

For this purpose, the main state-of-the-art technique and almost a paradigm itself that we have used is Deep Learning. In the two next subsections, we will talk about the foundations of Deep Learning, from the concept of perceptron to the concept of Convolutional Neural Network; then, we will describe the different models and techniques we have used in order to extract higher-level features from the images and include them into our pipeline.

3.3.1. Foundations of neural networks

The building block of the majority of neural networks is the **perceptron**. A perceptron is a model that was first described by [Rosenblatt, 1958] as an abstraction of the basic function and structure of a neuron. A neuron or nerve cell is composed by a number of inputs and output ramifications (called dendrites and axon, respectively); the nucleus of the neuron receives the electrical impulses from other neurons in its dendrites, and then processes those signals in order to generate new electrical impulses to be sent to other neurons through its axon (the electrical impulses can be received or be inhibited based on chemical processes produced in the synapses, which is the join between dendrites and axons).

As we can see in Figure 3.4[6], the perceptron is a simplification of a biological neuron as a model that receives a number of inputs, then computes the weighted sum of them, uses an activation function and produces an output. The weights are used in order to inhibit or give more importance to certain input values, basically obtaining a linear function of the input; then, the activation function is used to transform that sum and produce an output. The selection of this function depends on several factors, like the value to be estimated, the type of neural network, etc. For example, if this simple model was used for a classification task, then the main choice would be the sigmoid function, as it is bounded between 0 and 1 and the transition between predicting the positive and negative class would be smooth, while if our target is continuous, we would choose the identity function (that is, no transformation, $f(x) = x$).

Then, the next natural step was stacking and connecting perceptrons among them, following the analogy of the brain, in order to create more complex models that are able to find approximate solutions to more complex, non-linear, problems. The most simple structure of perceptrons is the concept of **Multilayer Perceptron (MLP)**, the first Artificial Neural

[6] Original image extracted from https://inteligenciafutura.mx/english-version-blog/blog-06-english-version

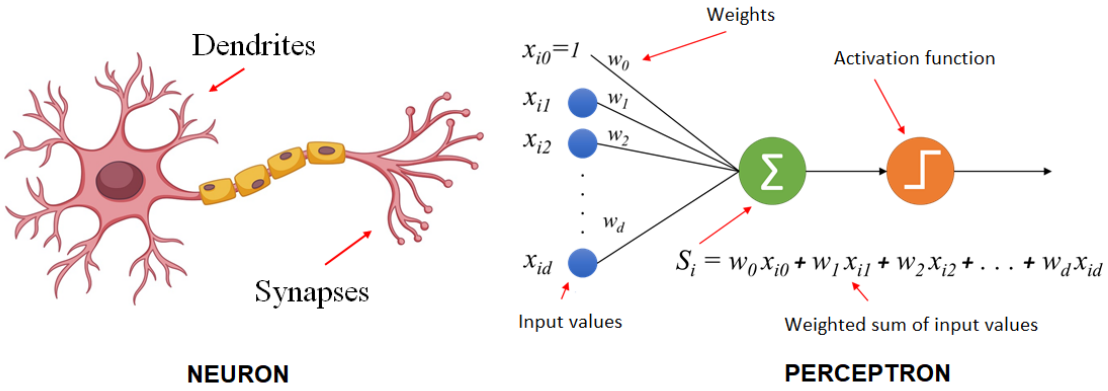

Figure 3.4: Analogy between a biological neuron and a perceptron or artificial neuron

Network (ANN), which consists of 2 or more layers of perceptrons, so that each perceptron receives as input the outputs of all the perceptrons in the previous layer[7] (this is also known as feed-forward neural network, see example in Figure 3.5[8]). In particular, a MLP with a single hidden layer (that is, a layer of perceptrons between the input and output layers) can approximate any measurable function under the conditions described in [Hornik et al., 1989]. Moreover, they conclude that the reasons that do not allow it are structural, related with how the neural network is trained or by a poor relationship between inputs and target.

Using a MLP to **predict an output** based on certain input values is straight-forward: starting from the input data, each perceptron in the input layer computes the weighted sum of those and transform them using the activation function; then, their outputs are sent to all the perceptrons in the next layer, which repeat the same process, creating more complex combinations of features (as non-linear 'nested' expressions or polynomial expressions are constructed). This propagation of outputs to the next layer of perceptrons is repeated until the output is reached and a prediction is obtained (in regression, a single unbounded output value; in binary classification, a single value in the $[0, 1]$ range; in multi-class classification, as many outputs as possible classes, where each one gives the estimated probability of the input instance to belong to a particular class, the positive class for that neuron, so that the selected class is the most probable one). The previous process is known as **forward propagation**.

Thus, the predictions depend on three elements: input values, weights between neurons and activation functions. Once we create a neural network, the activation functions are fixed, so the way we have to learn from the input data and minimize the error between the actual outcomes and our predictions has to be done by **adjusting the weights between**

[7] Actually, there is one neuron that does not receive the outputs of the previous layer: the bias neuron, similar to the intercept term in linear regression, whose output is always 1

[8] Original image extracted from http://ufldl.stanford.edu/wiki/images/thumb/4/40/Network3322.png/500px-Network3322.png

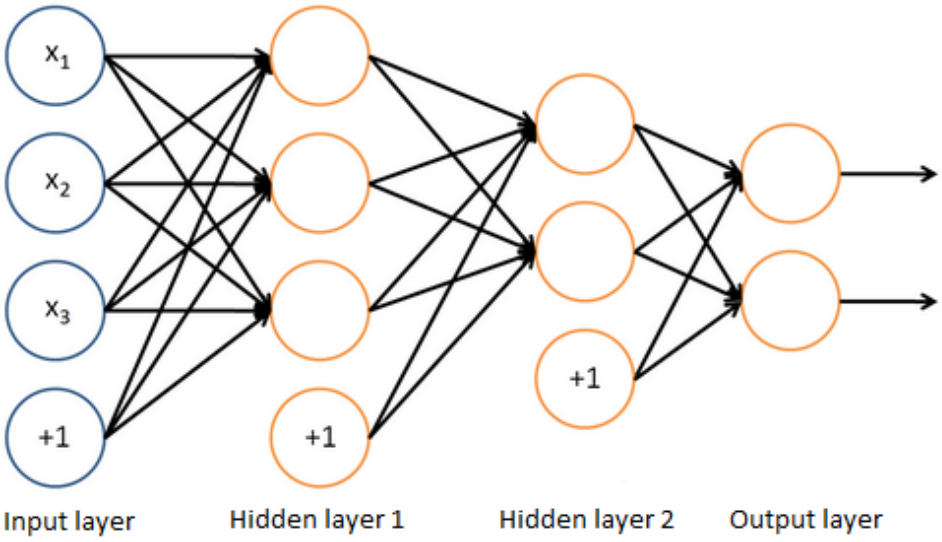

Figure 3.5: Example of a MLP with two hidden layers

neurons. This is done using **Gradient Descent**, an iterative algorithm used to find a local minimum of a differentiable function (see Figure 3.6[9]) by adjusting the weights in a proportional way to the negative of the gradient at the current point (first-order partial derivative with respect to each input dimension, each weight in this case). As we follow the negative gradient, we come closer to a local optimum; by doing it proportionally, we ensure that, as the magnitude of the slope is greater, the descent can also be greater, but as the slope starts to decrease (when we get closer to the aforementioned local optimum), the updates of the weights are more subtle. The proportion of the gradient's magnitude used to update the weights is known as **learning rate**, which is used in order to avoid fluctuations in the gradient descent (smaller steps are usually better to eventually converge to a minimum).

In particular, given a set of input values and their actual target value, the updates of the neural networks' weights are computed after the forward propagation process once we have the output values of every neuron and the prediction error, used to compute the gradients. With this data, the update of the weights between the last and penultimate layers is computed, but the actual weights are not modified in order to update the whole network at the same time. Then, going one layer back, the weights between a neuron in a layer and other neuron in the following one, given that the later one does not belong to the output layer, is done in a similar way. However, as we cannot compute the prediction error in the later neuron since it is not an output, what we compute as the gradient of that neuron is the influence that it had on the final error, which involves computing a weighted sum of the gradients of its successors in the next layer (if they belong to the output layer, then they were directly influenced by the prediction error) and also its output value stored after the forward propagation. Then this process is repeated for all the weights, from one layer to the previous one, and is known as **backpropagation**.

[9]Extracted from https://www.kdnuggets.com/2020/05/5-concepts-gradient-descent-cost-function.html

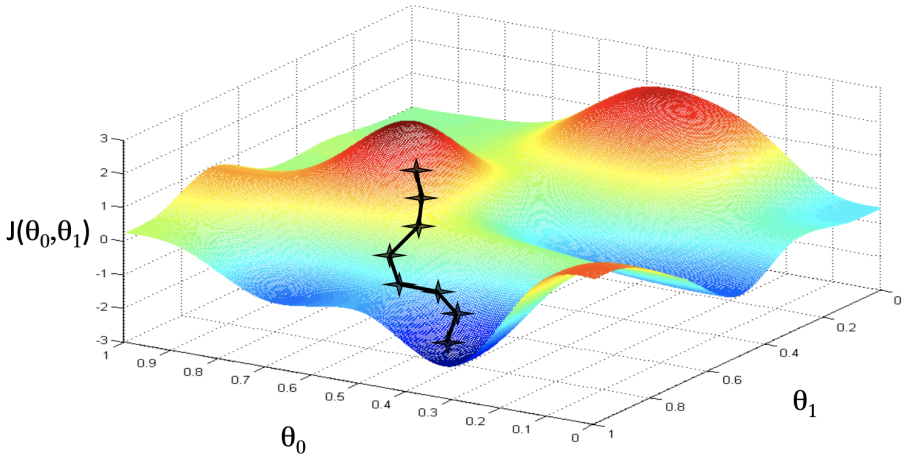

Figure 3.6: Gradient descent for a particular differentiable function with two input parameters

Usually, the update of the weights is not done after each forward and backwards propagation process (per instance), but grouping the updates from several instances in what is known as a **batch**, so that the updates are smoother. Sometimes, however, as gradient descent can converge to a local optimum, Stochastic Gradient Descent (SGD) can be used, as it updates the weights after each forward and backwards propagation, so that the updates are more volatile and hence there exists a higher probability of escaping a local minimum and reaching the global one.

3.3.2. Foundations of Deep Learning

The concept of **Deep Learning** emerged in the second half of the 2000's decade in order to solve different problems that common neural networks (MLP) have when dealing with non-structured data, such as images or text (the concept of 'deep' neural networks was first mentioned by [Hinton et al., 2006] in their approach to solve the classification of images of hand-written digits). One of such problems when dealing with the classification of images with a MLP, for example, is that it can only extract features from the base or lower-level input element, which is the pixel, while higher-level features that rely on spatial information may be needed (if we were trying to classify an image that has a hand-written '9' at the left part, then if the MLP has only been trained using centered '9's in the images, the MLP will probably fail because it can only learn very restrained patterns built from low-level features). Instead, a deep neural network would extract higher-level features, such as the existence of a vertical line and a small circle on top. Another problem of a MLP would be the number of parameters (weights) to train, since the input image has to be flattened in a very long vector of pixels and be fully-connected to the next layer (for a low resolution greyscale image, for

example 256x256, and a first hidden layer of 500 neurons, the number of weights in between those layers would be 32768000).

A **deep neural network** it's not only a neural network with many hidden layers, but a network structured in such a way that it can learn how to extract the most appropriate higher-level features from lower-level ones, and then use those more abstract features to perform a particular task. This task can be classifying images, text, audio, generating a segmented image or a description of an image, etc. In our case, we need to extract features from images in order to predict an output value, so we will focus on **Convolutional Neural Networks** (CNN).

A CNN is a particular type of deep neural network with two distinguishable parts: one is used to **extract high-level features** from the input, which is a 2-D or 3-D array of data, such as an image with three input channels (3-D data: one 2-D array for each RGB channel) or a 2-D array containing the word embeddings of a text (as we will describe in the section 3.4.3); the other part is a **fully-connected network** (or simply, a MLP added on top of the feature extractor network) which receives the extracted features and performs the intended task. We can see a simple example of the main blocks of a CNN used for classification of images in Figure 3.7[10].

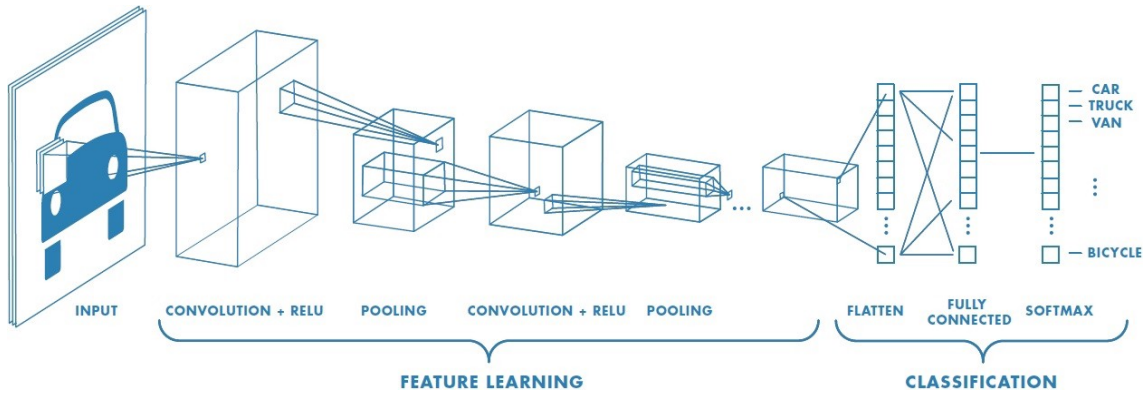

Figure 3.7: Example of the constituent parts of a CNN used for the classification of images

In order to process the input image and extract the most suitable information while trying to keep a reasonable number of trainable weights, a CNN uses the concept of **convolution**. A convolution is the operation by which a **kernel** is convolved over an input matrix, obtaining what is called a **feature map**. The result of each convolutional step is actually a value in the feature map which captures information from several input elements (in the input layer, pixels), thus extracting (local) spatial relationships. In fact, if we assume a RGB input image, each kernel in the input layer is actually composed of 3 matrices convolved over the three input channels, respectively, plus the bias element for each input channel. A group of 2-D kernels is known as **filter** (which, consequently, is 3-D).

[10]Extracted from https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html

Different filters extract different features[11], so the key issue is how to extract the most suitable ones for the intended task. Precisely, **what is updated in the feature extraction part of the network is the values of those matrices or kernels that are convolved over the input channels** (their values are randomly initialized; however, there are studies like [Glorot and Bengio, 2010] which proves that a random initialization in a particular range of values yields faster convergence). This has a remarkable implication: **the number of weights to be updated** in this part of the network **does only depend on the size, number of filters and number of input dimensions**, and not on the actual size of the original image, according to the assumption that if a filter is useful over a portion of the input space, then it should be useful over any another portion of the input space. Nevertheless, as the output volume of features must be flattened to a vector of features so that it can be handled by the fully-connected network at the end of the CNN, the number of weights in between these parts of the network does depend on the size of the feature maps.

We can see an example of how convolutions are applied over an input volume in Figure 3.8[12]. Each output value is computed as the sum of the convolution results over the respective input channels plus the bias value of the filter (given the step that we are computing in the image, the highlighted one, the first kernel yields a value of -2, the second one a value of -1, the third one a value of 0 and the bias is 0, so the sum of them is -3, which is the value mapped to that convolutional step). This Figure also serves us to explain two hyperparameters of a convolutional layer that determines (along with the depth or number of filters) the dimensions of the output feature maps:

- **Stride**: it is the number of positions the kernels are slid at each step or convolution. The greater the stride, the smaller the total number of convolutions and consequently the size of the output feature map.
- **Padding**: border of zeros that is added to the input matrix in order to get more spatial information from the border elements (if the padding is not added, then each border element is captured in a smaller number of convolutions than the rest of elements, assuming a stride of one).

For example, if we add padding and set the stride to one, then the output feature map will have the same dimension as the input feature map; if we don't add padding and the stride is still one, the output feature map will be smaller than the input one (see Figure 3.9)[13]. In Figure 3.8, the convolutions are applied over padded input features with a stride of two, obtaining a smaller output feature map. In particular, the output size given the size of the input feature map (W, assuming that its shape is a square), the filter size (F, also a square), the padding size (P) and the stride (S) is shown in equation 3.4. Of course, this is the output size of the first two dimensions (again, assuming that they are equal, otherwise W should be replaced by the height to compute the second dimension size), the third dimension is equal to the number of filters applied over the input volume.

[11] Visualize transformations obtained by convolving different kernels: https://setosa.io/ev/image-kernels/

[12] Extracted from https://cs231n.github.io/convolutional-networks/. The image is actually an animated gif, we encourage the reader to watch it.

[13] Extracted from https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

$$W_{output} = \frac{W_{input} - F + 2P}{S} + 1 \tag{3.4}$$

Thus, the output size of each filter given the conditions in Figure 3.8 is:

$$W_{output} = \frac{5 - 3 + 2 \cdot 1}{2} + 1 = 3$$

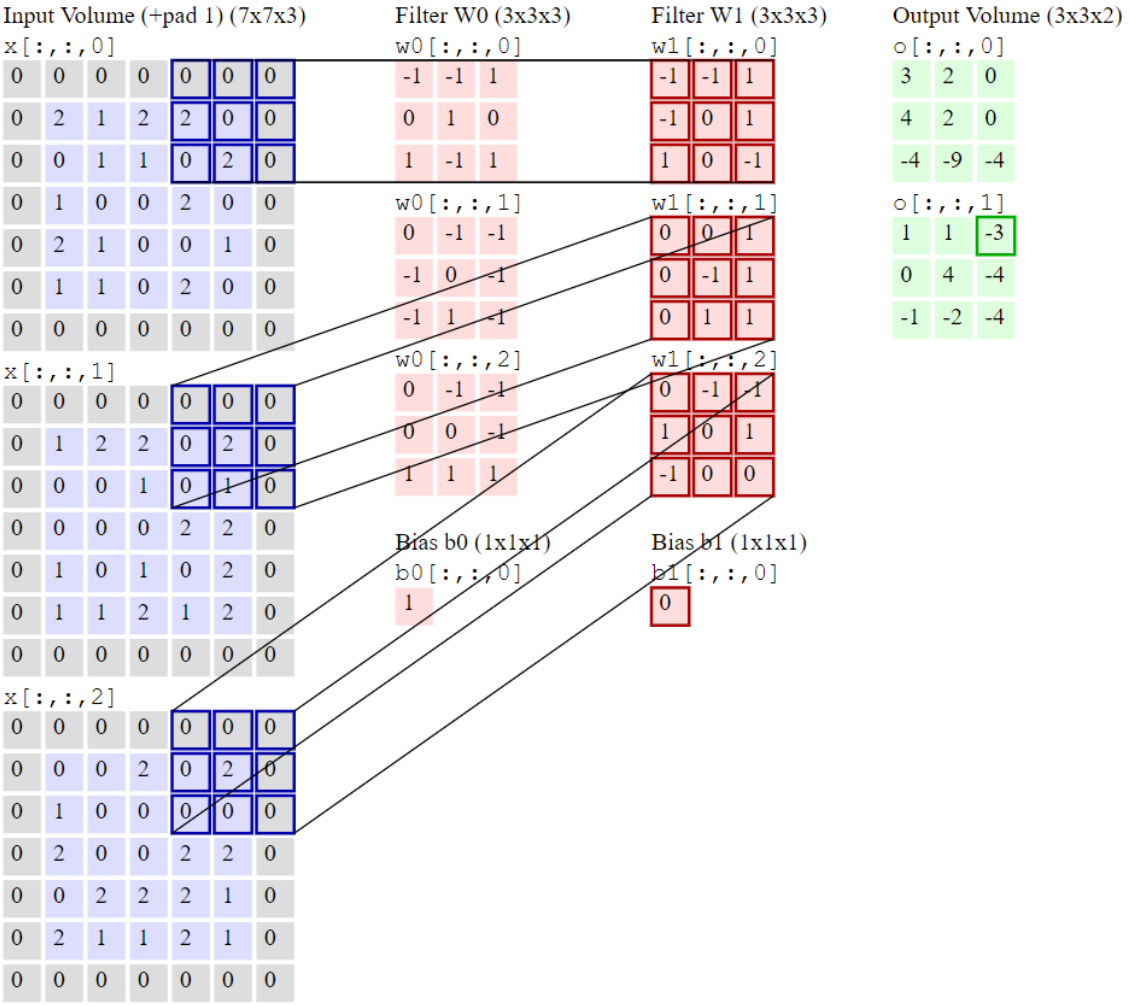

Figure 3.8: Example of two filters convolved over the input channels (with stride=2 and padding)

Finally, there are other operations typically applied in a Convolutional Neural Network to improve performance:

- **Pooling layer**: it is used between convolutional layers in order to reduce the dimen-

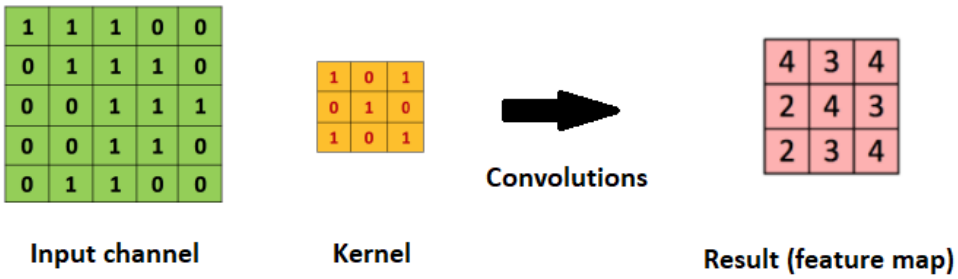

Figure 3.9: Example of a single kernel convolved over an input channel

sionality of the feature maps. The most common transformation to do so is getting the average or maximum value of the values captured by the pooling filter (Figure 3.10[14] shows an example). This way, we reduce the dependence on particular features obtaining a more general view of them, and also a reduction in the number of required convolutions in the next layers.

- **ReLU activation function**: one of the main problems in deep neural networks is the fact that the propagated error is diluted as it is backpropagated through the network, especially given the gradients of some activation functions like sigmoid or tanh (when their value is closer to one of their bounds, the derivative gets closer to 0). This is known as the **vanishing gradient problem**. One of the solutions to this problem is using the Rectified Linear Unit activation function, which has many of the advantages of a linear activation function while it is still non-linear. In particular, this piecewise function is defined as:

$$g(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

This way, as the derivative of the function is constant to 1 when the input is greater than 0, the backpropagated gradients are proportional to the activations, so the vanishing gradient effect is avoided as shown by [Glorot et al., 2011], even though later architectures still needed to use other approaches to solve this problem for very deep neural networks, as we will see in the next subsection. Moreover, ReLU is easy to implement and easy to optimize. However, it has a drawback: if the input is always smaller than or equal to zero, the gradient value in those cases is always zero, so no update of weights would take place for the neurons that suffer this problem, they are 'dead' neurons. The solution is using a slight modification of the function in which instead of a 0 output when the input is smaller than or equal to zero, it is a small fraction of that input, so that weights can still be updated. This modification is known as

[14] Extracted from https://cs231n.github.io/convolutional-networks/

Leaky ReLU, and studies like [Xu et al., 2015] show that they outperform the original ReLU.

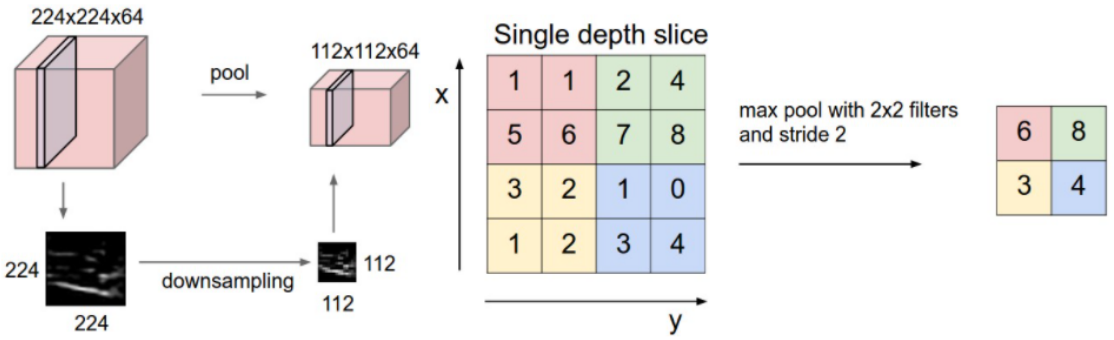

Figure 3.10: Result of applying max pooling over an input feature map volume

Different CNNs have been created over the last years using different building blocks, introducing new elements, etc. These novelties and the fact of being able to create even deeper architectures is possible thanks to the progress of parallel computing, mainly in the field of **GPUs** (Graphic Processing Units) and the more recent **TPUs** (Tensor Processing Units). Training neural networks or convolving filters over input channels mainly involve two simple operations: additions and multiplications, which can be done by simple processors (linear and ReLU activations are also easy to perform on those processors; however, sigmoid and tanh activation functions require more computing time or using especial processors), different filters can be applied at the same time over the input channels by different processors, input matrices can be partially transferred to faster memory units and then the results be merged, and many other optimizations that can be done using the aforementioned architectures, which considerably speed up the training and prediction processes.

3.3.3. Pre-trained CNNs and preliminary feature extraction

In order to extract features from the profile images, we would need to create a CNN, train it so that it constructs the best possible features and then get the output of the feature extraction part of the network (which is what the fully-connected network on top receives), so that it can be added to the tabular data to feed the final estimator in the pipeline. However, this is not an easy task, for many reasons. First of all, the classification task in *AdoptionSpeed* values is not an easy one: we don't really know what image features could explain the actual outcome (in the case of a classification of hand-written digits, it is very clear), we don't know if background conditions affect the outcome more than the pets, or whether the 'cuteness' of the animal influences it (and what features define the 'cuteness'). This in turn implies that the architecture should be more complex that the one we would use to classify hand-written digits, which also would imply much more training time and more training samples

to obtain good results.

The solution to the above problem is using more complex CNNs that have already been trained on significant amounts of images. For example, the *Keras* API (which is the one we have used to create, train and extract features using CNNs) provides a number of CNN architectures that were trained on ***ImageNet***, a database of millions of labelled images belonging to 1000 different categories. Thus, we could take advantage of some of those already trained CNNs in order to extract image features using them (this is also the base of Transfer Learning, as we will see in the next subsection).

There are many architectures (even though some of them are similar or use the same building blocks with the only difference of how deep they are), so we have used a few to obtain preliminary results and focus on the most suitable one. We will briefly describe each one, as this also serves us to see the evolution of CNNs and the novelties that were introduced in later, state-of-the-art architectures. In particular, the trained architectures that we have used are:

- **VGG16**: this is one of the oldest architectures provided by *Keras*, and the one that best fits the base definition of what is a CNN: it consists on a succession of blocks composed of convolutional layers with ReLU activations and max pooling layers. This architecture was created by [Simonyan and Zisserman, 2015], the name indicates that it has a total of 16 weight layers (a weight layer refers to the connection of weights in between neuron layers), including those of the fully-connected network. The novelty of this architecture was its greater depth at that time and the use of small a kernel size (3x3), which significantly outperformed previous architectures.
- **InceptionResnetV2**: this architecture consists of two main building blocks. The first one, is the **Inception block** (see Figure 3.11), created by [Szegedy et al., 2014], which is based on the idea of applying convolutions with different sizes on the same input features maps to extract both global and local information, splitting the pipeline in several paths and then concatenating the output feature maps. Moreover, as applying different convolution sizes could be very computationally expensive, each path reduces the number of input channels by applying 1x1 convolutions (this way, the number of input channels of each path is determined by the number of filters that apply those 1x1 kernels). The other building block is the use of **residual connections** in order to mitigate the vanishing gradient effect by combining the outputs of a block with the outputs of another previous block or layer. This way, gradients from the last layers can flow not only to the corresponding immediately precedent layer but also to another, earlier layer in the pipeline, allowing the possibility to create deeper networks as the weights updates are more significant in early stages. In particular, InceptionResNetV2, as described by [Szegedy et al., 2016], uses sequences of Inception blocks, with slight modifications in the split paths between different sequences, and the input of each such block is combined with the input of the previous block using residual connections[15].

[15]The following blog shows the complete architecture: https://ai.googleblog.com/2016/08/improving-inception-and-image.html

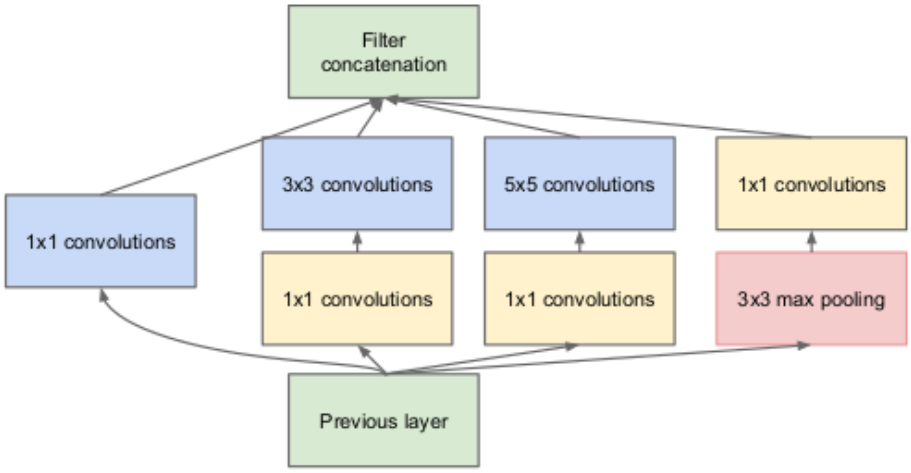

Figure 3.11: Structure of an Inception block

- **Xception**: this architecture is based on the same assumptions that motivated the creation of the Inception block, but it goes a step further: cross-channel correlations and spatial correlations (within the same channel or input feature map) can be handled separately. The implementation of this idea is named by [Chollet, 2016] as an **'Extreme Inception' block**, shown in Figure 3.12: first, a 1x1 convolution is applied over the input volume to obtain a projection (reducing the number of channels); then, each channel or single feature map is convolved in a separate path, eventually concatenating all the output feature maps. The architecture is mainly composed of this type of blocks, and it also uses residual connections.
- **DenseNet121** and **DenseNet201**: as shown by [Huang et al., 2016], these architectures are based on the succession of layers whose output is forwarded as input to all the subsequent layers in the convolutional network. This has many advantages: feature maps are reused without the necessity to increase the number of filters (the only thing that changes is the number or volume of feature maps each layer receives, but the number of parameters to train does not change); the vanishing gradient effect is mitigated, since every layer receives the gradients from later stages of the convolutional network, and the concatenation of feature maps is very straight forward (residual connections combined the output feature maps summing them, obtaining the same number of feature maps, but in this case the number of feature maps increases). The only restriction is the fact that concatenated feature maps must have the same dimensions, so instead **Dense blocks** are built. Within them, the feature maps have the same dimensions, so the layers are connected in the described way. Between Dense blocks, there are transition layers that reduce the dimension of the obtained feature maps.
- **NASNetLarge**: the architecture of this network was constructed by Google in a rather

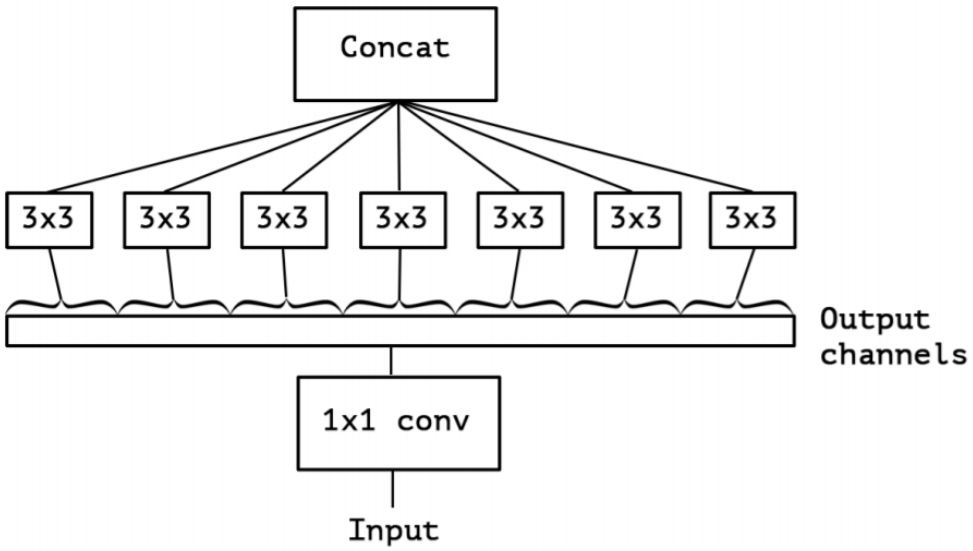

Figure 3.12: Structure of an Xception block

unusual way, in particular as a reinforcement learning problem were the reward was the accuracy on a dataset (smaller than ImageNet). [Zoph et al., 2017] describes this search problem, where a Recurrent Neural Network (RNN) is used as the controller; the building blocks were constructed recursively, selecting hidden layers outputs as inputs of the chosen sample of operations (the most common ones in state-of-the-art models, such as 3x3 average pooling, 3x3 max pooling, 5x5 max pooling, 1x1 convolution, etc.). This search process yielded an architecture that was then used as building block to be used with a larger dataset (*ImageNet*), obtaining better accuracy results than all the previous state-of-the-art CNNs.

In order to extract features from these CNNs, some simple **preprocessing** has to be done first on the input images. This involves resizing the images to have the same dimensions, padding them inside the image so that the original aspect ratio is maintained and using the corresponding preprocessing function (different CNNs need the input images to be in BGR instead of RGB format, others normalize and scale the input values in a certain way).

Another transformation that has to be made in order to extract a reasonable number of features is the **global average pooling** operation, which obtains the average value of each output feature map, so that we obtain as many features as the volume or number of output feature maps. This is an affordable alternative to just flattening all the output volume to a one-dimensional vector, which would yield an unmanageable number of features (by averaging each feature map, we may lost some important features, but flattening small output feature maps of size 5x5 would increase by 25 the number of extracted features).

Then, in order to estimate how suitable the extracted image features are in combination with the rest of tabular data we need to create several models, each one using the features of one of the previous networks, and then evaluate their performance. As we will see in the

first section of Chapter 4, Random Forest and **XGBoost** (Gradient Boosting implementation) are two estimators that give good preliminary results when evaluating their performance using the original and supplementary tabular data (that is, without CNN image features and *Description* text features). As XGBoost didn't improve as much as Random Forest when extending the pipeline with the supplementary tabular data and its transformations, we have evaluated the extracted image features with this estimator (under the assumption that if XGBoost improves the performance with these new features, Random Forest will probably improve too).

Given that with the base and supplementary tabular data we obtain a total of 73 variables, adding more than a thousand new variables would be too much (this is the case when using all the networks but VGG16, from which we extract 512 features, still too many), not only in terms of computing time (even for a fast and very optimized estimator like XGBoost it takes too much time) but also because we would implicitly give much more importance to those raw features than to the base and supplementary tabular data (especially for stochastic algorithms that select a random subset of features, such as Random Forest). Thus, we have used SVD to obtain a reasonable number of features, varying the number of truncated components (to 16, 32 and 64 for the features of each CNN).

The evaluations have been done using a cross-validation process of 5 iterations (5-CV) on the complete training dataset. As we will also mention in Chapter 4, we have done this in order not to spend too much computing time (Kaggle gives 30 weekly hours of GPU and each session has a time limit of 9 hours). This way we can filter and focus on the best options (in this case, on the CNN which extracts better features under certain conditions), even if the evaluation should not be done in this way, since we know that by evaluating models and hyperparameters several times on the same dataset we eventually obtain information of that dataset. A proper scheme for hyperparameter tuning and model evaluation is described in Chapter 4, as well as the metrics we have used; the following results and the ones of the next subsection are used to fix hyperparameters values in the pipeline, which wouldn't be computationally affordable to include in that search.

The evaluation results are shown in Table 3.1. These results were obtained with the 73 variables from the original and supplementary tabular data plus the features extracted from the first profile image (if there was no profile image, then the CNN was fed a black square). In order to choose one CNN, we have considered several aspects: **average accuracy** score in 5-CV (proportion of correctly predicted instances over the total number of predicted instances), **average Quadratic Weighted Kappa** score in 5-CV (QWK, the metric used to evaluate submissions in the competition, measures the level of agreement of the predictions with respect to the real outcomes taking into account the distribution of the target variable; a deeper description will be given in Chapter 4) and the **size of each network** (structure and number of weights), which also affects the storage space needed to save each trained model, as we will see. The average fit time is affected by the number of final image features after applying SVD (it takes more time for XGBoost to construct the model), and also by the decomposition itself and the number of original 'raw' image features. Taking all of these into account, the CNN that we have selected is **DenseNet121**: the configuration with 16 SVD components is the second best configuration in average accuracy and QWK. The best configuration corresponds to that with NASNetLarge image features and 16 SVD com-

ponents too, but in that case the number of original features is larger, so it takes more time to apply SVD and each saved model occupies approximately 343 MB (while a DenseNet121 model 33 MB). This last criteria also affects VGG16, which provides good features with 16 SVD components, but each model occupies more than 500 MB.

The reader can see Table 3.1 and all the results of the image feature extraction process in **Version 76** of Notebook 2 (see Figure 3.13), in this document we will show some of them. The last version does not include it since all the evaluations and operations in that Notebook take too long to be executed within the limit of 9 hours that Kaggle imposes.

CNN	SVD	Average fit time (s)	Average accuracy	Average QWK
DenseNet121	16	32.302502	0.422797	0.371695
	32	45.863883	0.409790	0.352098
	64	73.497283	0.412392	0.345691
DenseNet201	16	37.643561	0.417061	0.368443
	32	54.230856	0.410992	0.363692
	64	88.931732	0.413326	0.364016
InceptionResNetV2	16	35.070499	0.416261	0.361871
	32	50.612532	0.419395	0.362506
	64	81.601580	0.412459	0.355913
NASNetLarge	16	51.264992	0.423198	0.373589
	32	76.237476	0.416928	0.368467
	64	125.166793	0.416928	0.363931
VGG16	16	28.980260	0.421264	0.369433
	32	40.491534	0.415994	0.363529
	64	64.557840	0.412659	0.352828
Xception	16	38.603064	0.416194	0.364396
	32	55.707300	0.412393	0.357952
	64	91.317940	0.404855	0.350671

Table 3.1: Evaluation results varying the source CNN and number of SVD components

Then, as we have seen that a small number of SVD components gave good results, we have also tried different, small number of components (4, 8, 12, 16, 20, 24 and 28). 4 and 16 SVD components gave a better balance between average accuracy and average QWK (even though compressing the original 1024 extracted features to 4 features may be too risky or at least more unstable). Needless to say that the SVD V_k matrix has always been obtained using just the training set of the corresponding 5-CV iteration (it would be easier to apply the decomposition to the entire set of training profile images since it could be done outside the pipeline and its evaluation would take less time, but of course we haven't done it as it

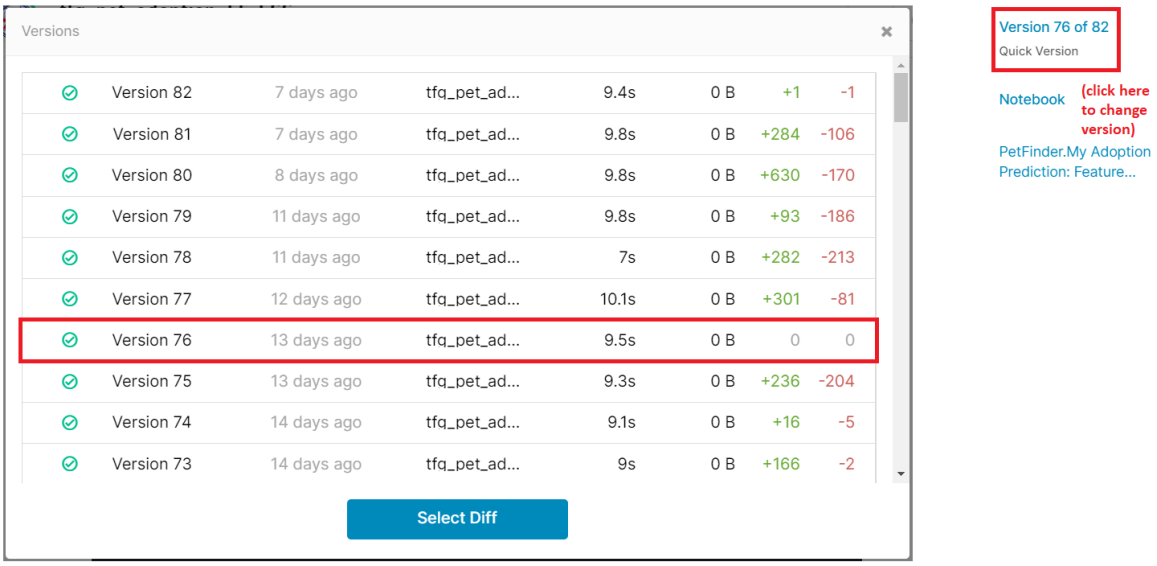

Figure 3.13: How to see previous versions of a Kaggle notebook

would be a data leak).

We have also tried to resize the input image to different values (the previous evaluations were done on 256x256 images, in this case we also checked 224x224, 384x384 and 512x512), but the difference in the evaluation metrics were not very significant, so we have maintained the 256x256 size. We have also averaged groups of 4 original features, obtaining 256 total variables instead of 1024, and then applying SVD as before. However this aggregation made the evaluation metrics to decrease a little bit, so probably we were losing some important features. As we can imagine, checking all these hyperparameters in a search grid would imply a huge number of possible combinations and also the necessity to store too many different files with the extracted features from the training profile images using different configurations.

3.3.4. Selecting the appropriate features by training

In the previous subsection, we described how we have used SVD in order to reduce the number of originally extracted features while maintaining as much information as possible. However, SVD does not take into account the target variable, so it does not necessarily choose the linear combinations of features that gives more information about *AdoptionSpeed*. However, we can extract the most suitable image features in order to predict the value of *AdoptionSpeed* by training a fully-connected network using that variable as output and the extracted image features as input. This is known as **Transfer Learning**: using the feature extraction part of an previously trained CNN in order to obtain a set of (pressumably useful) image features and then, instead of using the original fully-connected network, which was trained for other purpose (in the case of DenseNet121, to predict instances of the *ImageNet* dataset), we replace it by our own fully-connected network to predict other

variable (or to carry out other task).

The objective is to train the fully-connected network, not the feature extraction part again (or at least not a big part of it), since that would require too much computing time, so we **freeze the weights** of those layers so that they are not modified (gradients are not backpropagated to them). This also implies that a pre-trained CNN used to detect cancer cells should not be used to predict whether a pet is a cat or a dog, for example, because **the most suitable image features for one task may not be very useful for a very different task**. In our case, we could search on the Internet for a CNN which was trained on a significant amount of images of dogs and cats, in order to predict the breed, the age, etc. However, as we mentioned before, we don't exactly know what we are looking for: the pet age seemed to give valuable information about *AdoptionSpeed*, but *RescuerCount* also did, and in this case maybe the difference between an archetype of rescuer and another (which is the information that *RescuerCount* seems to give, as we have seen in the EDA) could be detected by the elements on the background, such as the presence of a cage or kennel, grass, etc. Thus, as the DenseNet121 backbone was trained on the *ImageNet* dataset, it could give information not only about the pet but also about other elements of the image (*ImageNet* includes a number of different breeds of dogs and cats as possible class values[16], so we can be sure that many images of dogs and cats were also used in its training process).

Once we train the fully-connected network, we extract the outputs of one of its layers between the DenseNet121 backbone and the output layer in order to extract a combination of image features that gives more information about *AdoptionSpeed* than the original extracted features. However, it is not possible to evaluate the extracted features using a 5-CV strategy, since that would imply 5 different training processes if we want to obtain non-optimistic estimations, but that isn't feasible either due to the limit of 30 weekly GPU hours of Kaggle. Instead, we have evaluated the extracted features after training the CNN using a **single training-validation split** on all the training images that we are provided, not just the first one of each profile, in such a way that **all the images of a profile are either used in the training dataset or in the validation dataset**, but they are not divided between training and validation. This way, we ensure that we validate the model (XGBoost) using features of images that were not used to train the CNN, or that were not similar to those (because in many profiles with more than one image, some of them are very similar). The split is done in a stratified manner, so that the proportion of profiles belonging to each class is maintained as the original in both datasets. 46617 images (from 11994 profiles) were used to train the CNNs, while 11694 images (from 2999 profiles) were used for validation.

Then, in order to get better results and be able to generalize better to unseen input images, we have used **data augmentation**, which consists on applying different transformations on the input images in order to artificially expand the size of the training dataset. *Keras* implements a class[17] that does this function on-demand for each batch of input images, and we can specify which transformations are used. These transformations are randomly applied to each input image, so there is still the possibility that no transformation is applied, or several are applied on each image, in each epoch (the number of **epochs** determines how many times the entire dataset is used to train the CNN).

[16] https://github.com/anishathalye/imagenet-simple-labels/blob/master/imagenet-simple-labels.json
[17] https://keras.io/api/preprocessing/image/#imagedatagenerator-class

Before using different methods to train the CNN and extract image features, we need to define how many features we will extract (i.e., how many neurons will have each fully-connected layer that we will add, and how many layers), and use always that configuration to compare different methods. Thus, we have trained 6 different CNNs, each one with a different number and size of fully-connected layers (with ReLU activations), using as target variable *AdoptionSpeed* as a classification problem. This implies using an output layer of five neurons, one for each possible class, and the **softmax** activation function, a generalization of the sigmoid activation function with which we obtain as output a multinomial probability distribution (the outputs sum is 1.0, it is computed as shown in equation 3.5, where K is the number of classes, in this case 5, from 0 to 4, and $\sigma(x)_c$ is the actual output of the neuron used to indicate how likely the class c is). The loss function is the most commonly used for multi-class classification, **categorical cross-entropy**, which is used to estimate the difference (error) between two probability distributions: the known outcome (probability of 1.0 for the known class value and 0.0 for the rest of classes, which is actually the class variable one-hot encoded) and the predicted one, computed for a single prediction as equation 3.6 shows, where y is the vector corresponding to the one-hot encoded known class value and $\hat{y}$ is the vector of predicted values (softmax layer).

$$\sigma(x)_c = \frac{e^{x_c}}{\sum_{k=0}^{K-1} e^{x_k}} \tag{3.5}$$

$$H(y, \hat{y}) = -\sum_{k=0}^{K-1} y_k \cdot log(\hat{y}_k) \tag{3.6}$$

Each combination of number of layers and their size was trained with a small number of epochs, 5, and the only data augmentation transformation that was applied was the horizontal flip (Figure 3.14 shows an example), since it does not affect the original input image too much for these early evaluations but still we can double the total number of possible input images. The results are summarized in Table 3.2, which shows the average evaluation metrics values over the 5 training and validation epochs. As we can see, in validation the best configuration (looking at all the metrics) is a 64-neuron layer and then a 16-neuron layer (in this order) between the output of the feature extraction stage and the output layer for *AdoptionSpeed*. Thus, we have used this fully-connected network in every CNN model we have trained (with slight differences in some of them). Moreover, this configuration allows us to extract two different sets of features, one with a smaller size that uses combinations of features from the first, bigger, set.

As predicting the *AdoptionSpeed* can be a classification or regression problem, due to the ordinal relationship of its values, the first three models that we have trained using the selected configuration have different outputs: the first model has 5 output neurons as a classification problem, the second one just one output neuron with linear activation function as a regression problem and the third one with 5 output neurons as an ordinal regression problem using the approach that [Cao et al., 2019] describes[18]. The main issue of setting

[18]It is not implemented in *Keras*, but there is an external package that does it:

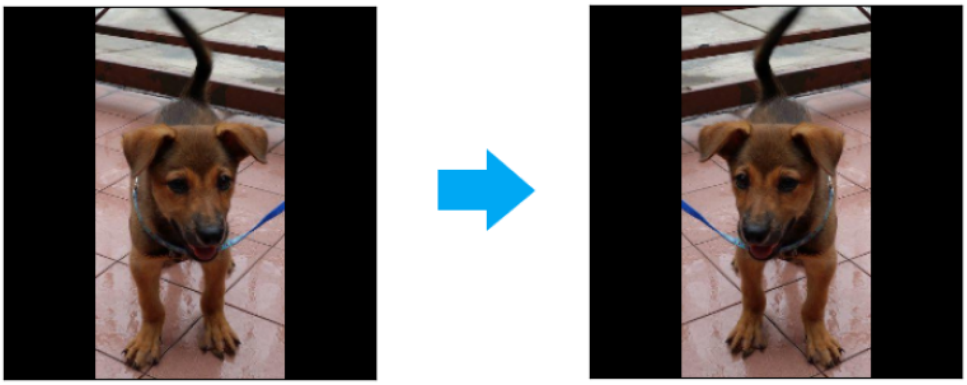

Figure 3.14: Example of horizontal flip of an input image

Combination	Train Loss	Train Accuracy	Train QWK	Val Loss	Val Accuracy	Val QWK
1 - 16	1.390981	0.350756	0.268567	1.409662	0.333590	0.254701
1 - 32	1.393694	0.348753	0.256746	1.415729	0.328733	0.233493
1 - 64	1.388128	0.351344	0.271971	1.410872	0.333419	0.249035
2 - 32, 16	1.392067	0.349396	0.255622	1.411968	0.333487	0.251432
2 - 64, 16	1.390928	0.346882	0.254302	1.405881	0.338840	0.260938
2 - 64, 32	1.386175	0.353305	0.272064	1.414989	0.329639	0.236081

Table 3.2: Averaged training and validation results on 5 epochs for different combinations of fully-connected layers

out the prediction of *AdoptionSpeed* as a classification problem is that the predicted probabilities of each class may not be consistent given the ordinal relationship among them, thus [Cao et al., 2019] solves this by forcing all the output neurons to share the same weights with the neurons of the penultimate layer, but each one having its own independent bias. Then, a custom loss function is also provided to achieve the rank-monotonicity in the output probabilities.

These three models using different output strategies were trained each one on 5 epochs, with a batch size of 32 input images with horizontal flip as the only data augmentation transformation. Once we trained these models, we extracted the image features from both the 64-neuron and 16-neuron layers and evaluated the performance using again XGBoost (each set of features independently), in the single split training-validation. From the classification model we extracted better features than from the regression and ordinal regression ones (from this last model we obtained worse features according to the obtained validation accuracy and QWK). However, looking at the regression features from the 16-neuron layer, only 3 of them were actually useful (the rest were all zeros), so the rest of CNNs that we have trained sets out the prediction as a regression problem, because there is probably more

https://github.com/ck37/coral-ordinal

room for this strategy to improve using regularization methods such as adding **dropout** layers. A dropout layer randomly erase (with a certain probability) some of the inputs received by a layer at each training step (not at prediction time); this way, we force the neurons to use a greater number of different inputs and not very specific ones, with the aim of reducing the probability of overfitting to the training data.

Several models with different configurations of dropout layers and more data augmentation strategies (we add the ones we can see in Figure 3.17 to the horizontal flip) were trained, the reader can see their results in Notebook 2. We ended up using a regression CNN with two dropout layers with a drop rate of 0.25, one immediately before the 64-neuron layer and the other before the 16-neuron layer (the number of useful or non-zero features extracted from the latter layer increased with respect to the model without dropout layers, as we expected). This structure was then trained on a greater limit of epochs with early stopping (when the validation loss, the Mean Squared Error, did not improve after 5 epochs), obtaining a model trained on 6 epochs whose validation loss were better than the training loss (which wasn't very common in other models trained on a similar number of epochs), as we can see in Figure 3.15. Then, we continued its training unfreezing the last Dense block, in order to train some of the last filters applied in the feature extraction part of the CNN, using a smaller learning rate (from 0.001 to 0.0001). All the models were trained using the Adam optimizer, which adapts the learning rate in a per-weight manner, based on the average of recent magnitudes of gradients and also their variance; [Kingma and Ba, 2017] describes it in more detail.

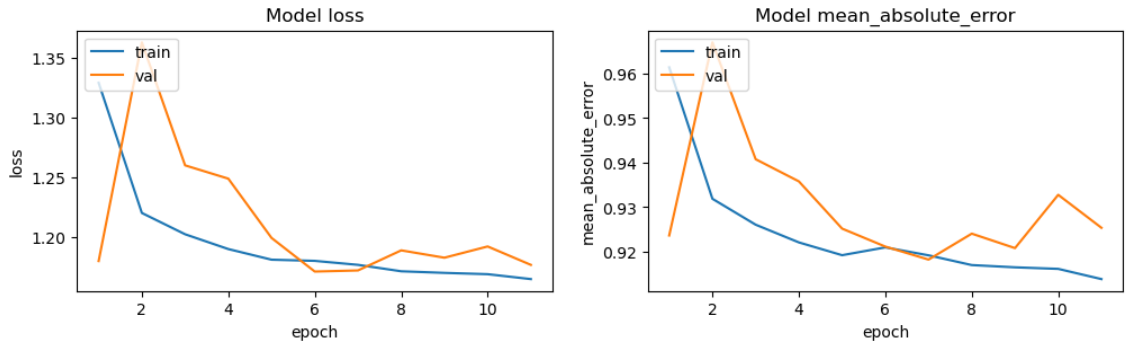

Figure 3.15: Training and validation loss (MSE) and MAE of a CNN model for regression

The evaluation results after unfreezing the last Dense block and training the CNN didn't improve the single split validation results of the 6 epochs model with the features from the 16-neuron layer, but the 64-neuron did improve. The two last useful steps that we have taken to improve the extracted features are:

- **Single input ensemble**: this is a CNN with a single input image that is forwarded to three different DenseNet121 backbones. The first one has all the feature extraction layers frozen, the second one the same but the last Dense block is trainable and in the

third one the last two Dense blocks are trainable. Then, we concatenate two layers to obtain the average and maximum of those 3 groups of output features. This implies that the number of output features grows from 1024 to 2048, so we constructed two different models: one in which the second and third DenseNet121 backbones had a final dropout layer with a drop probability of 0.1 and 0.2, respectively, and another model in which instead of adding dropout layers, the one placed before the 64-neuron layer has its drop rate increased from 0.25 to 0.5 (Figure 3.16 shows the final layers of this model).

- **Aggregation of image features**: so far, we have only used the image features extracted from the profile image (the first image of each profile) to evaluate the CNN models using XGBoost. Another option is aggregating the values of each extracted feature over all the images included in the profile, using the mean, sum and variance. In general, the validation results improved using this approach.

After training the aforementioned ensembles, the obtained evaluation results are better than using a single-backbone model at least using the 64-neuron features (with the 16-neuron features from the second ensemble we obtained worse results). However, when aggregating the features from all the images, both the single split validation accuracy and QWK using the 16-neuron output of the second ensemble were significantly higher than any other model. Thus, in the final extended pipeline we have used the second ensemble and its 16-neuron layer outputs, then aggregating the features of all the images of each profile. The results after extending the pipeline are shown in section 4.1.

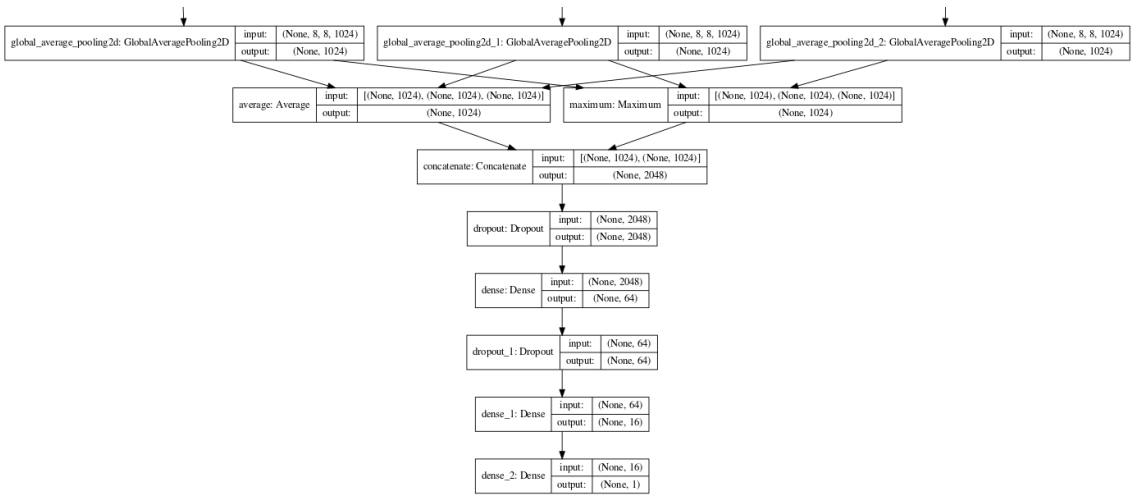

Figure 3.16: Last layers of the second single-input ensemble

Finally, there are some different strategies that we used but didn't bring better results:

- **Multi-output model**: instead of using *AdoptionSpeed* to train the fully-connected network, we constructed another CNN with three output variables: *RescuerCount*, *Age*

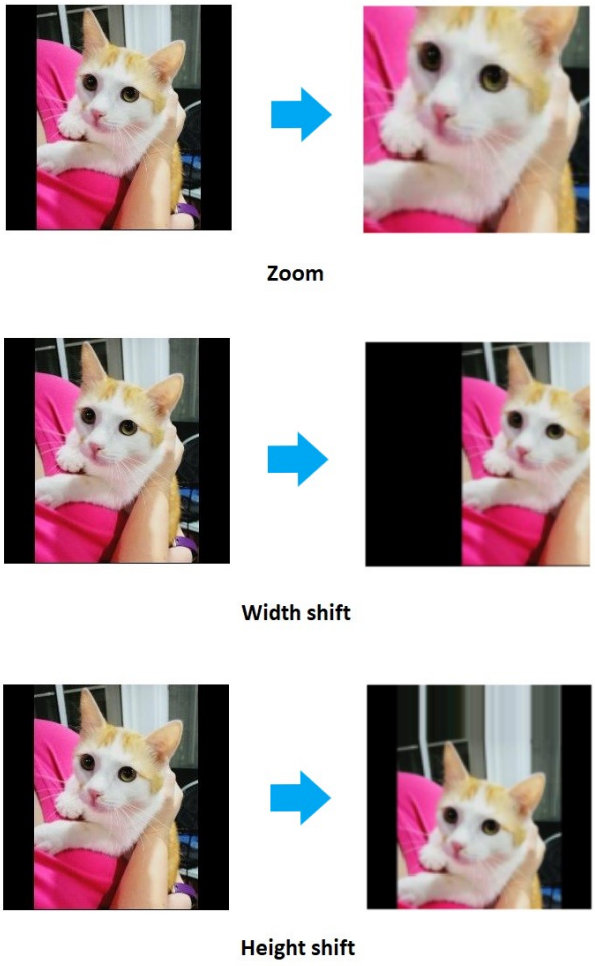

Figure 3.17: More data augmentation transformations

and *Breed1*. Those variables were selected by computing the mutual information between the predictor variables and *AdoptionSpeed* in a 5-CV process, since they had the greatest values. Then, in the training process, the loss function was set as the weighted sum of the different loss functions used by each output (for *RescuerCount* and *Age*, the mean squared error, for *Breed1* the categorical cross-entropy), were the weights were the mutual information values with *AdoptionSpeed*. It was trained on more epochs (with early stopping) than the first CNN models using *AdoptionSpeed* as output, but the results didn't improve those already obtained with our target variable, so this strategy was discarded as the training process was also a little bit slower.

- **Multi-input model**: instead of modifying the target variable to train the fully-connected network, this model uses several input images, each one being forwarded to a different DenseNet121 backbone. Then, the outputs of all the backbones were aggregated using the mean, sum and maximum values per feature, using the concatenation of those operations to feed the fully-connected network. The idea was to use different

images from each profile at the same time, so that instead of feeding the CNN with individual images and then computing the aggregated features outside the CNN it was done inside the CNN. The third quartile of the variable *PhotoAmt* is 5, so the number of inputs and DenseNet121 backbones were 5 (there are profiles with more than 20 images), in order not to make the forward propagation too costly. Then, for each profile, we selected random images to be the input for the CNN; if the profile had less than 5 images, then some of them were randomly chosen again to feed the remaining inputs; if it had more than 5, then only 5 random images were selected, but the profile image (the first one) was always used in every case as one of the inputs. Then, we created our own generator of inputs by wrapping 5 data augmentation generators, then creating random permutations of the selected input images in a per-batch manner so that the same input image can be forwarded to a different backbone in each epoch. However, we didn't even reach to evaluate the features since both the training loss and validation loss were significantly greater than those of the rest of CNNs we had already trained. There exist examples using a similar approach, like [Sun et al., 2017], although in this case what they concatenate is the feature maps obtained from different inputs at an earlier stage in the CNN feature extraction stage, instead of at the end (which makes more sense there since the input images are always similar, they are images of the same flower captured from different angles).

Just the training and extraction of image features using CNNs took us 120 GPU hours (4 weeks given the 30 hours limit of Kaggle), more time than the development and implementation of any other Feature Engineering technique.

3.4. Extracting features from text data

Text can be a very useful source of data, since we can extract information about its structure, words frequencies, semantic relationships between words, etc. In particular, we have used two different types of models in order to extract information from the text of the *Description* variable: TF-IDF and CNN with pre-trained word embeddings.

3.4.1. Cleaning and transformations

Before trying to extract information from the text, we have to clean and transform it first so that the models that we will implement are more robust to different conditions or properties that we have found in the corpus of *Description* texts, especially given that those texts are human-written and they may have spelling errors, contractions, etc.

First of all, the vast majority of *Description* texts were written in English, but not all: there are descriptions written in Malay and Chinese. Thus, this is the first obstacle, because if we want to use a model that is based on word frequencies like TF-IDF, which is the case, we would need to create one such model for each different language found in the training dataset (and then establish a good strategy to handle other possible unknown or unseen languages at transformation time). This is not feasible, so instead, we have to **translate** all

the non-English descriptions to English (or the majority language in the training dataset). As we mentioned in the subsection 3.2.1, the descriptions that couldn't be analysed by the Natural Language API of Google were those written in Malay (or a mix of Malay and English), so their language value was 'ms', while other very uncommon languages (less than 10 training instances of such) were grouped as 'others'. This is important, because depending on the group size, we will indicate to the translator the corresponding source language instead of telling it to detect it (if there is a mix of English and non-English languages in the text and we tell the translator to detect the source language, we may end up without a translation since it could detect that it is written in English). In our case, we have used the *googletrans* package[19] in order to translate those texts, even though this is a free API and is limited, so the translations were obtained using the transformer we have implemented but outside the pipeline, in order to store them in an external file and then include that data inside the pipeline.

Before translating the non-English texts, we also **removed the emojis** (detected in the 1.8% of the descriptions, without a clear influence on *AdoptionSpeed*), in order to avoid possible decoding errors when the models handle the texts. This was done using the *emoji* package[20]. Some of the emojis still were not detected, but the *sklearn* tokenizer did not return them.

The next transformation is a more complex one, it consists on **expanding English contractions**, such as "can't" by "cannot", since the tokenizer would truncate that word and return just "can" (words with a single character are deleted), obviously changing its meaning. In order to do so, as some contractions can be expanded in several ways depending on the context (for example, "he'd" could be expanded to "he had" or "he would"), we have used the *pycontractions* package[21]. This tool expands the simple contrations with a single rule (for example, "don't" by "do not") and then, if there is at least one contraction in the text that could be expanded in several ways, first all such contractions are replaced by any possible permutation of the multiple rules that can be applied (for example, from the sentence "I'd like to play chess right now if I'd brought it" we would obtain 4 versions: "I had ... I had...", "I had... I would...", "I would... I had..." and "I would... I would"). Then, they are sorted by the number of grammatical errors detected using a grammar checker[22], and, in order to break ties if more than one option have the same minimum number of errors, the package uses the **Word Mover's Distance** (WMD) to estimate how much it would cost to move all words from the original text to each possible hypothesis. In order to understand this, we have to know first what word embeddings are.

Word embeddings are a way to represent words as real-valued vectors in a pre-defined vector space (with a certain number of dimensions, usually 25, 50, 100, 200 or 300). Each such vector is learned based on the usage and context of each word, the main basis of word embeddings is the assumption that similar words are used in similar contexts (so similar words should have similar representations), which in turn give information about the words

[19] https://github.com/ssut/py-googletrans

[20] https://github.com/carpedm20/emoji

[21] Original source code: https://github.com/ian-beaver/pycontractions. This package has some compatibility errors and is outdated, so we actually used this modification: https://github.com/Martin36/pycontractions

[22] https://github.com/jxmorris12/language_tool_python

they wrap. One of the most common methods to create these word representations is *Word2Vec*, which captures not only syntactic similarities but also semantic (the example that is always given and the one used by [Mikolov et al., 2013] to describe this method is the following: if we subtract the vector representation of the word "Man" to the vector of "King" and then sum the vector of "Woman", the resulting vector is the closest to that of "Queen"; this type of relationships can be seen in Figure 3.18). *Word2Vec* used two models to learn word embeddings (see Figure 3.19[23]):

- **Continuous Bag-of-Words (CBOW)**: it is equivalent to a neural network with a hidden layer, a number of words as inputs and a single word as output. The objective is to predict a word given a context.
- **Skip-gram**: the inverse of the previous model, it predicts a context (a number of words before and after) given an input word.

Once these models were trained on millions of words, the word embedding is simply the output values extracted from a hidden layer (that is, the shared projection of words). In the previous models, each word is represented using Huffman encoding instead of one-hot encoding, since the last method is very inefficient for a vocabulary of millions or billions of words (Huffman encoding still yields binary representations, but now based on element frequencies, with the property that no encoding of a single element can be the prefix of any other element encoding).

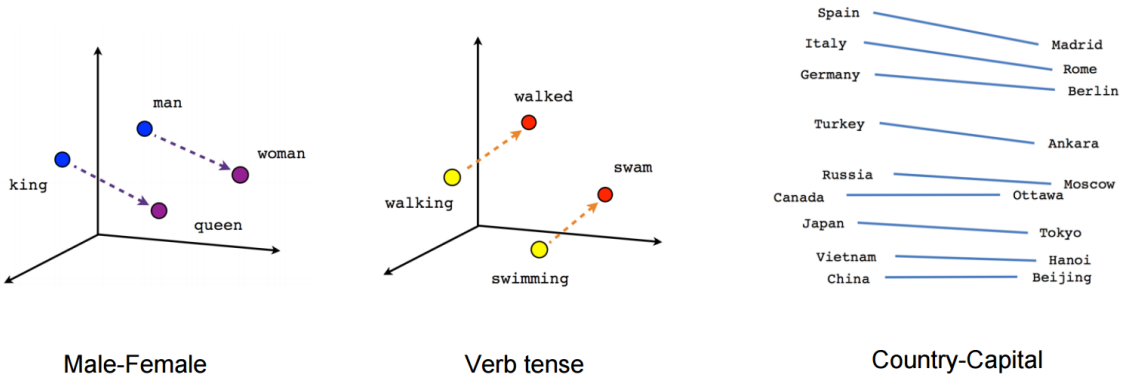

Figure 3.18: Syntactic and semantic regularities projected in low-dimensional word embedding spaces

There are many other methods to represent words as real-valued vectors, but most of the state-of-the-art ones have as basis some of the concepts used by the *Word2Vec* method. The word embeddings that we have used to expand the contractions and also in a CNN to extract text features, as we will see, are the Global Vectors for Word Representation (*GloVe*),

[23] Extracted from [Mikolov et al., 2013]

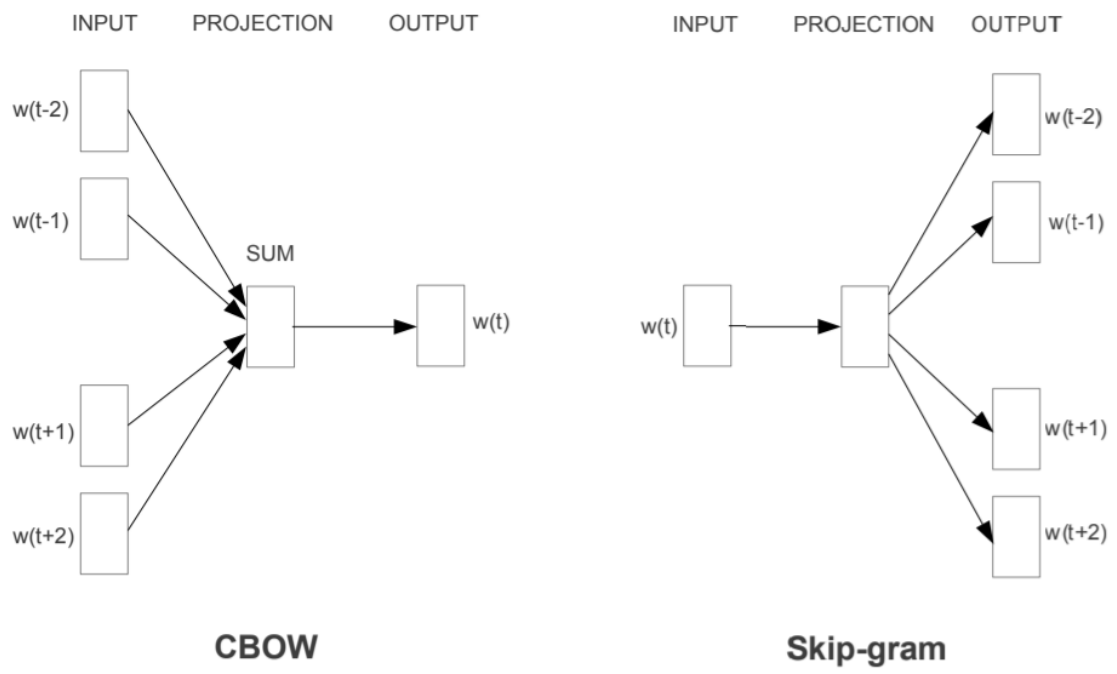

Figure 3.19: Models used by Word2Vec to learn word embeddings

and in particular the ones trained on a corpus of texts from Twitter, since their colloquial language will probably be more similar to that used in the *Description* texts than other sources. *GloVe* is an extension of *Word2Vec*, but according to [Pennington et al., 2014], instead of predicting words it extracts global statistics (word co-occurrence probabilities, that is, how frequently words co-occur with one another) from the corpus and then use them in the training process.

Going back to the contractions, we mentioned that in order to break ties among the possible hypothesis that had the smallest number of grammatical errors, the package *py-contractions* used the Word Mover's Distance (WMD), described by [Kusner et al., 2015]. This relates to word embeddings in that WMD uses the euclidean distance between the word embeddings of pairs of words in order to estimate how close two text documents are. Of course, this needs the two documents to have an almost identical syntactic structure to obtain a realistic value, which is the case, because the original document and the hypothesis only differ in a number of expressions (the expanded contractions).

After expanding the original contractions, the last transformation consists on **removing punctuation symbols**, except the single quote, since this is done after expanding contractions and consequently if it is not removed is either due to a spelling error or because it is the genitive form to indicate possession.

3.4.2. Text feature extraction models

After we have applied the appropriate transformations on the *Description* texts, we still have to represent it in a numerical way, with a fixed number of features as we did with the image data.

The first model that we have implemented in order to do so is **TF-IDF**. This is an example of a **Bag-of-Words** model, a type of models that extract information about the occurrences of words rather than the semantic, syntactic or order relationship among them. Given a known vocabulary of words, these models measure the presence of them (this measurement could be, for example, a simple count representing how many times a word appears in a text), so that each instance is represented by a vector with the same length as the total number of different words found in the corpus of documents. Thus, we obtain a matrix where the rows identify each text or document in the corpus and the columns identify each word. One of the main properties of this kind of matrices is that in a sufficiently large corpus they are sparse, because each text will contain just a few words of the vocabulary. In order to decrease the size of the vocabulary, different transformations can be applied, like cleaning the text, as we have done, but there are others: removing very frequent words like "the", "a", "of", etc. (obtained from a list of stop-words for that language or by establishing a threshold, for example the words that appear in more than the 90% of documents), applying techniques like Lemmatization (replacing words by their dictionary term, for example "books" by "book" or "played" by "play"), etc. We didn't use techniques like the last mentioned one because we think that the difference between present and past tenses may be meaningful in the *Description* of each pet.

The main characteristic of TF-IDF as a Bag-of-Words model is that it scores or measures each word in a text taking into account two frequencies:

- **Term Frequency**: how many times the word appears in a document.
- **Inverse Document Frequency**: it measures how rare a word is in the entire corpus of documents.

By combining these measures, we decrease the importance of those words that are very common across different texts but that do not provide useful information about the current document (for example, "the", "a", "is", etc., but also more specific words like "pet" in our case, since it probably appears in many different descriptions). The *sklearn* implementation uses the formula 3.7, where $\mathsf{tf}(t,d)$ is the number of times the word t appears in the document d, $\mathsf{df}(t)$ is the number of documents in which t appears and n is the total number of documents (the idf formula explicitly assumes that every word appears at least once, this way we will not divide by zero when we see an unknown term at prediction time). Then, it applies the Euclidean or L2 norm on each vector (the sum of the squared values of each document vector is 1).

$$\mathsf{tf\text{-}idf}(t,d) = \mathsf{tf}(t,d) \cdot \mathsf{idf}(t) = \mathsf{tf}(t,d) \cdot \left(1 + log\frac{1+n}{1+\mathsf{df}(t)}\right) \tag{3.7}$$

Moreover, even if TF-IDF cannot really extract semantic information as a Bag-of-Words model, we can create a vocabulary where not only individual words or terms are examined,

but also groups of 2, 3 or even more sequential terms. This is known as **n-gram**; if we maintain a bigram (2-gram) vocabulary, from the sentence "Pet given up for adoption" we extract four vocabulary elements ("pet given", "given up", "up for" and "for adoption"). Thus, this is a way to capture more semantic meaning from the texts.

Once we obtain a TF-IDF matrix, as the number of columns (vocabulary size) would be unmanageable, we have to apply truncated SVD in order to reduce its size while maintaining as much information as possible. As with all the transformers that we have implemented, both the TF-IDF technique (obtaining the idf values of each vocabulary term[24]) and the SVD (obtaining the V_k matrix) are fitted on the training data the pipeline receives, it is not done outside the pipeline or on the entire training dataset before.

The second model that we have created to extract text features is a Convolutional Neural Network with **1-D convolutions over the pre-trained word embeddings** of the input text. 1-D convolutions allow us to find sequential relationships in the data, as the sliding kernels move along a single axis, covering up different groups of words at each convolution, while spanning the entire length of the word embeddings as we can see in Figure 3.20[25] (in the case of 2-D convolutions, like the ones used to extract image features, the filters capture spatial relationships instead of sequential ones). Then, the feature maps that we obtain are vectors, one for each kernel that we apply, that are concatenated so that the same convolutional operation can be applied (the kernels applied over them span all the input channels).

Using pre-trained word embeddings gives us the advantage of not having to train vector representations of each input word from random values, which would require many training instances that we don't have in order to obtain good representations. For example, [Kim, 2014] implements several CNNs to extract text features from both static (pre-trained) and non-static (learned) word embeddings. In fact, 1-D CNNs are more widely used in the analysis of signals and time series forecasting than for text feature extraction.

We trained this model over the same instances used to train the CNNs to extract image features (but now, we use their *Description* text instead of their images; in fact, we use the text obtained after the previously described transformations). As the input text must always have the same length (in order to obtain feature vectors with constant length in the CNN), we fixed the maximum number of input words to 200 (if the number of words of a text is smaller, then it is padded with vectors filled with zeros). The structure that we used is the one used as example in the *Keras* documentation[26], where the text features are extracted from the fully-connected layers and it was trained using *AdoptionSpeed* as target variable. However, the model overfitted too much to the training data, so we conducted a little search of hyperparameters inside the training dataset (the one used to train the CNN) in order to test different kernel sizes, number of filters, number of layers and dropout rates. Nevertheless, the selected combination of hyperparameters did not really improve the validation loss.

When evaluating the new set of features with XGBoost (extending the previous pipeline

[24] If the vocabulary does not include a term seen at transformation or prediction time, that term is simply not included in the matrix. The returned matrix will always have the same number of columns, corresponding to the training vocabulary.

[25] Extracted from https://cezannec.github.io/CNN_Text_Classification/

[26] https://keras.io/examples/nlp/pretrained_word_embeddings/#load-pretrained-word-embeddings

in which we added the image features), the validation results improved using TF-IDF, but they were significantly worse using the CNN features (the reader can see the results in Notebook 2). Thus, we use TF-IDF in the final pipeline for the hyperparameter search, as we will see in the incoming Chapter.

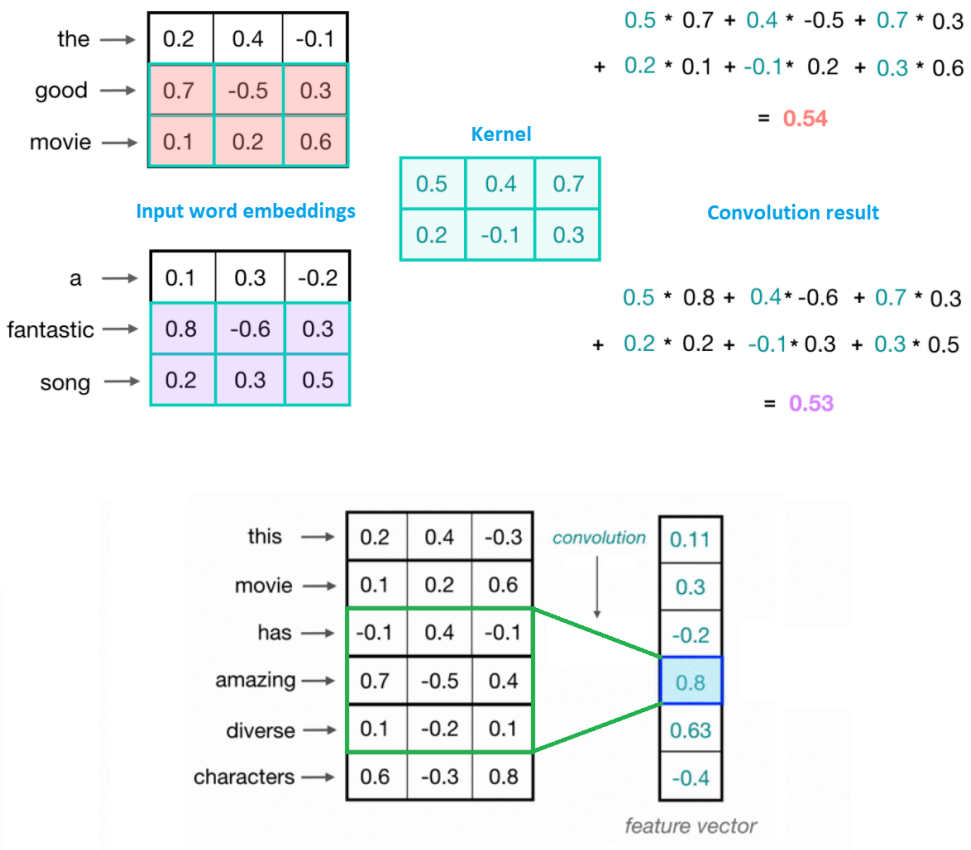

Figure 3.20: Graphic description of how 1-D convolutions are applied over word embeddings

4. Model Creation and Evaluation

So far, we have created a pipeline of transformers that include new variables, discard some others, clean and encode the corresponding ones in the most suitable way, etc. Thus, at the end of this Feature Engineering pipeline what we get is a data table that meet the appropriate conditions to be passed to an algorithm that learns from those data, constructing a model with which we can predict the outcome of new input instances.

In this Chapter, we will describe the models we have obtained, how we have evaluated them and what their performance is. All the results are on Kaggle, as always, even though they are divided between **Notebook 2** and **Notebook 3** (see Annex).

4.1. Preliminary results

In the previous Chapter, we described how the pipeline was extended several times with new data which implied new transformations or a special way to obtain that data (for example, training a CNN before to do so). As the pipeline gets longer, the number of variables and the different **hyperparameters**[1] of the transformers also do. In turn, since we want to get the best model, this implies a larger hyperparameter search space (as we will see in section 4.3), which directly affects the required computing time to conduct that search, or how exhaustive it can be. Moreover, some models may not perform very well for this prediction problem, so it would be pointless to conduct such search using them if we know that others perform significantly better. For example, this is why we used XGBoost to evaluate the image features and whether to use TF-IDF or extract text features from a CNN, and in this section we will see the results that motivated that choice.

Thus, in order to estimate on which models or paradigm of models we should emphasize in the hyperparameter search and what hyperparameters we could fix, **we have evaluated 4 different models after each pipeline extension**. As we are given a training dataset and a test

[1] A hyperparameter is an external value used to configure how the model learns from the data, to impose limits, stronger regularization, etc. It is different to the concept of model parameter in which the latter one is intrinsic to the model, for example the weights of a neural network or the coefficients of a linear regression model.

dataset, we could obtain early evaluation results by splitting the training dataset into two smaller datasets (after shuffling the original dataset just in case the instances were ordered by some variable, especially if it was the target variable): one actually used to train the models and another, the remaining instances (what we we call validation dataset), used to evaluate how far our predictions are from the outcomes that we know those instances had. However, even if we select the validation dataset randomly, or after shuffling the dataset, we could obtain a biased evaluation result, since we could get data with some condition that is not found in the training dataset, and also because we are not using all the original training data to evaluate the model (because with the previous strategy it would imply a **data leakage**: evaluating the model by trying to predict the outcome of some instances that we already used to train that model, or that were used to extract some training information, leading to optimistic evaluation results).

Hence, the way we have evaluated the models to obtain early results is using a **cross-validation** strategy, which consists in dividing the training dataset in k non-overlapping sets or folds (k-CV), such that in each of k iterations, $k-1$ folds are used as training set, and 1 fold is used as validation set. The difference between each iteration is which folds are used for training and which one for validation (every fold is used as validation once, this is why this strategy gives more reliable results than using a single training-validation split), as we can see in Figure 4.1[2] (the green folds would be used for training and the blue one for validation). Then, the score is obtained as the average of the validation score of all the iterations. We have set k to 5, and the folds are obtained from the training dataset in a **stratified** manner (that is, each fold has approximately the same proportion of instances belonging to each class as the complete or original training set; this is especially important in classification problems where there is a class imbalance, which happens here with the small proportion of '0' cases). As we mentioned in the previous Chapter, every transformer in the pipeline would also be fitted just on the training folds of each iteration; no transformation is done before the pipeline, and surely not on the complete training dataset (for example, if we applied TF-IDF on all the *Description* values of the original training dataset, we would create a data leak since we would obtain the validation TF-IDF matrix using frequencies that also included the ones of the validation instances).

As a remark, we are aware of the fact that we were implicitly getting information about the training dataset by already evaluating the models after each pipeline construction, but it is solely to obtain preliminary results on the entire training dataset that allow us to save computing time by discarding some models that may not perform very well later on. Moreover, we also needed this information for the work that we described in sections 3.3 and 3.4. Of course, these evaluations are performed once for each pipeline, without changing any hyperparameter of the transformers or the models; that is done later, as described in section 4.3.

The four models that we have evaluated in order to obtain preliminary results are Logistic Regression, SVM, Random Forest and XGBoost (Gradient Boosting implementation). We will briefly describe each of them.

The first one, **Logistic Regression**, is the equivalent for classification of Linear Regression,

[2] Extracted from https://scikit-learn.org/stable/modules/cross_validation.html

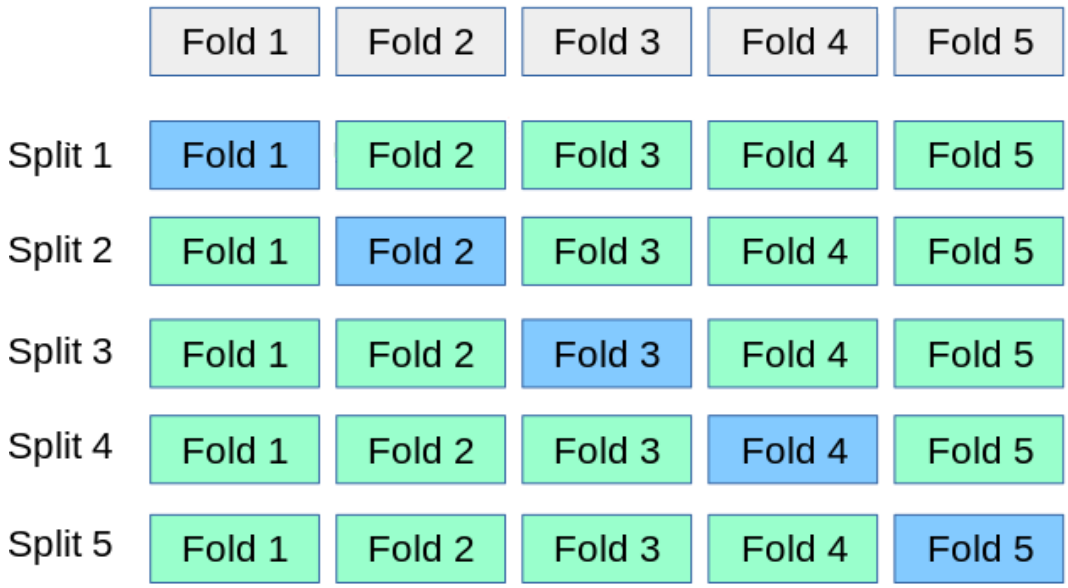

Figure 4.1: Example of cross-validation strategy with 5 folds

which tries to fit a linear model to the data by minimizing the average distance between the straight line and the actual observations. This model can be applied to a classification task, so that when the estimation is above a particular threshold, we predict the positive class, while if it is below we predict the negative class. This strategy has two disadvantages: the linear model is unbounded, and we would like to always get a probability, that is, a value in the $[0, 1]$ range indicating the probability of the positive class given the input values; besides, the linear model would be very weak as the aforementioned threshold could vary too much as we learn from more training instances. Instead, we apply a transformation: the sigmoid function, (equation 4.1), so that output value is always bounded between 0 and 1 and the transition between those values is smoother. In this case, the linear regression cost function (Mean Squared Error) cannot be used as it would not be convex with the sigmoid function. Thus, it is redefined to the expression in equation 4.2 (were y is the prediction rounded to 1 or 0), known as logarithmic loss.

$$h_\theta(x) = \frac{1}{1 + e^{-(\theta_0 + \sum_{i=1}^{N} \theta_i x_i)}} \tag{4.1}$$

$$J(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \left[y^{(i)} \cdot log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \cdot log(1 - h_\theta(x^{(i)})) \right] \tag{4.2}$$

As *AdoptionSpeed* is a multi-class variable, and Logistic Regression is used to predict a binary variable, we can convert *AdoptionSpeed* into 5 different target variables, in what is known as one-versus-rest prediction: a model will be used to predict the '0' class, which is the positive class, and the rest are the negative class, another where the '1' class is the

positive one and the rest are the negative class, etc. Then, we select as prediction the model which yields a higher probability estimation. This strategy is the default one used in the *scikit-learn* implementation for multi-class prediction. Moreover, it also uses by default **L2 regularization**, a proportional penalization to the sum of the square of the coefficients, so that a model which fits very well the data by using very large coefficients, which imply relying more on specific input variables, substantially increase the cost; on the other hand, a model that does not use that many large coefficients may not fit the training data that well, but could possible generalize better and be more robust to noise. This is the **bias-variance** trade-off.

The next model we have used is Support Vector Machines (SVM), created in the mid 90s by [Cortes and Vapnik, 1995]. The basis of this algorithm is the use of a Margin Classifier, a hyperplane that separates the instances in the input subspace (assuming a binary classification problem). If the number of input dimensions is two, that hyperplane would be just a line, as we can see in Figure 4.2[3], whose coefficients and intercept would be found by the learning algorithm. In order to find that line, what actually influences it is the margin or distance to the closest data points belonging to the positive and negative classes; these points are the **support vectors**.

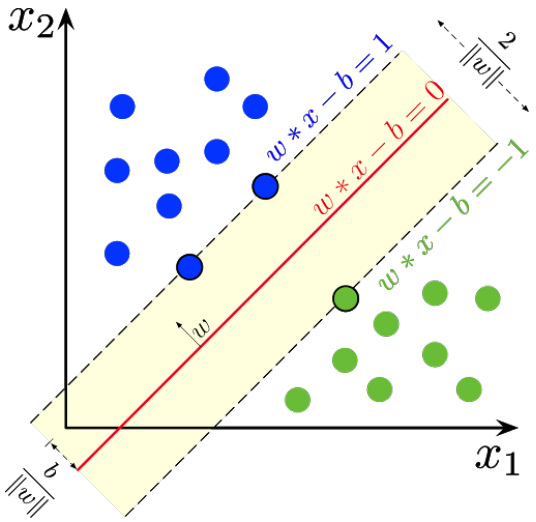

Figure 4.2: Maximum-Margin Classifier in a 2-D subspace

In practice, it could be impossible to separate the input instances so that all the ones that belong to the same class fall on one side of the hyperplane, so instead we allow a certain amount of violation of the hyperplane, obtaining a Soft-Margin Classifier. However, in some classification tasks the amount of violation could be very high; in such cases the data points or instances must be mapped to a higher dimension to find an appropriate hyperplane, as

[3]Extracted from http://albertotb.com/curso-ml-R/Rmd/07-svm/07-svm.html#1

we can see in Figure 4.3[4]. This process is known as **kernelling**; a kernel actually computes the inner-product of any pair of observations. There are many different kernels (linear, polinomial, radial, etc.). For example, the polynomial kernel with a degree higher than 1 would allow curved soft-margin classifiers, as the radial basis, but the latter one maps the input subspace to infinite dimensions[5] (even though it actually creates non-linear combinations to map data points into higher dimensions and then use a linear boundary).

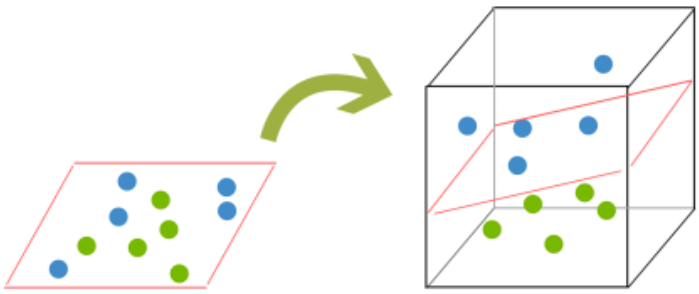

Figure 4.3: Mapping input 2-D subspace to 3-D to find the best hyperplane

The two following models are **ensembles of decision trees**. A decision tree is a model in which the data is recursively partitioned into subsets in a greedy manner, based on the reduction of an impurity measure. Normally, the recursive process stops when all the data in the current partition belongs to a single class (in the case of classification), the variance in the target variable falls below a threshold (regression) or because there is a single instance. One of the disadvantages of this learning algorithm is that it can easily overfit to the training data (in the worst case, we may end up with as many leaves as the number of instances), so different ways to reduce the complexity (size) of the tree can be applied: pruning after growing the tree, establishing a maximum depth, a threshold of impurity decrease, etc.

However, by creating an ensemble of trees we can reduce the probability of a wrong decision and also the variance, since the final prediction is the aggregation of the individual predictions of all the trees. The first ensemble of trees that we have used is **Random Forest**, created by [Breiman, 2001], which uses two methods to grow each tree:

- **Bootstrap sample**: instead of using all the instances in the training dataset, we select a random sample of them with replacement, so that at the end each tree uses a training dataset with the same size as the original, but in which a number of instances

[4] Extracted from https://www.kdnuggets.com/2016/07/support-vector-machines-simple-explanation.html

[5] Its decomposition results in an infinite sum of polynomial kernels, as we can see in http://pages.cs.wisc.edu/ matthewb/pages/notes/pdf/svms/RBFKernel.pdf

will probably be included several times. This way, each tree will emphasize more on different subspaces of the input data.

- **Random subset of features**: at each partition step in each tree, only the possible splits of a subset of features is checked (for example, a random subset whose size is the square root of the total number of features).

The combination of these two techniques allows the model to decrease the variance not only due to the aggregation of predictions but also as the built trees are less correlated with each other.

The last model that we have used to obtain the preliminary results that we are about to see is XGBoost, which is an efficient implementation of a **Gradient Boosting Machine**. This type of ensemble is an evolution of **Boosting**, in which each model tries to correct the prediction errors of the previous one, by assigning greater weights to the incorrectly predicted instances so that they can be sampled with a higher probability. Gradient Boosting is similar in that it tries to amend the prediction errors of the previous models, as the prediction is built sequentially. The target variable of each base learner is actually the negative gradient of the loss function. In a regression task, the prediction is straighforward as the negative gradient of the most common loss function (squared error divided by two) is directly the **residual** (prediction error); thus, from an initial prediction, like the mean of the target variable which is the value that minimizes the initial error with respect to all the training instances, we simply add the predicted residual of each individual model in a proportional way to a learning rate (to avoid fluctuations). In a classification task the most usual choice is using the logarithmic loss, as the loss function needs to be differentiable, so what the actual models predict is the logarithm of the odds, that is, the probability of the positive class over the probability of the negative class; then, the final prediction (the addition of the outcomes of each model from an initial prediction) is passed to the sigmoid function in order to obtain a probability between 0 and 1. [Friedman, 2001] gives a complete description of Gradient Boosting.

Then, **XGBoost** is a particular implementation of a Gradient Boosting Machine with the following characteristics and optimizations (this is a summary of the complete description of [Chen and Guestrin, 2016]):

- The splitting criteria is based on the **gain of a quality score**, which involves the calculation of the gradients (first-order partial derivatives of the loss function) and hessians (second-order partial derivatives) of the previous predictions and two regularization parameters: λ (L2) and γ (minimum gain to consider a split).
- Instead of evaluating all the possible splits of each variable, it can also use an alternative method, the **Weighted Quantile Sketch**. It consists on creating a histogram of each variable where the quantiles are selected based on the sum of the weights in each partition (the weight is the hessian). This way, for example, as the hessian in the classification case is $\hat{p}_{i-1} \cdot (1 - \hat{p}_{i-1})$, those instances that had a previous probability estimation closer to 0 or 1 will have small hessian values, while those closer to 0.5 will have greater hessian values; then, the former cases will be more likely to be grouped

together than the latter cases in order to obtain more or less evenly weighted quantiles. In other words, some of the quantiles will be smaller in number of instances as we have a lower confidence on their previous predictions.

- **Parallel processing**: several different processes can extract non-overlapping samples of the total number of training instances in order to create approximate histograms in parallel and finally merge them. The result is then used to compute the final weighted quantiles for each variable.
- **Column subsampling**: instead of evaluating all the possible variables, we can just select a subset of them to be evaluated for each tree, or doing that at the node level (like Random Forest) or even for each tree level (XGBoost builds the tree in a breadth-first fashion).
- XGBoost **can handle missing values** by assuming a default path at each split for them (in each possible split, we assume that all the instances with a missing value in the variable goes to the partition on the left or on the right, and then select the split that gives a higher gain, as always).
- **Cache-aware access**: basically, gradients and hessians are transferred to cache memory so that they can be retrieved more quickly.

Before seeing the preliminary evaluation results, it is worth to describe the main metric used in the Kaggle competition to evaluate the submissions: the **Quadratic Weighted Kappa** (QWK). This statistic is based on another one: Cohen's Kappa coefficient, which measures the level of agreement between two raters that classify a number of items into a number of mutually exclusive categories. Then, we can assume that one rater gives the true or ground labels, and the other one is our list of predictions. The estimated level of agreement takes into account the expected agreement by chance (considering the actual distribution of values); [Grandini et al., 2020] gives a nice description of Cohen's Kappa and compares it to other metrics. Summing up, the range of values of this metric is between -1 and 1, where negative values mean a worse agreement than what we could get by chance, 0 means the agreement we would get by chance and 1 means total agreement.

Then, as this is a multi-class problem and there is an ordinal relationship among the labels, QWK allows disagreements to be **weighted differently** (for example, if the true label is '1' and we have predicted '4', that would be a worse disagreement than if we predicted '2'). In order to compute its value, we have to create three different matrices: the matrix of observed scores, the matrix of expected scores and the matrix of weights, whose diagonal is zero, as it corresponds to a total agreement, and then further cells from the diagonal gets higher values, proportional to the squared difference between labels as we can see in equation 4.3.

$$w_{i,j} = \frac{(i-j)^2}{(N-1)^2} \tag{4.3}$$

The preliminary evaluation results of the first pipelines in a simple 5-CV are summarized in Table 4.1 (Notebook 2 gives more metrics and visualizations; we will see some of them later). Each pipeline means the following:

1: Base tabular data

2: 1 + *Description* metadata

3: 2 + Image metadata

4: 3 + Image properties

In this case, all the hyperparameters are fixed to their default value that we established in the transformers (breed is encoded with target and frequency encoding, only the profile image metadata and properties is added). We can see how every model except the XGBoost one significantly improves as the pipeline is extended with new data and their transformations, this is why we selected it to evaluate the extracted image features. In pipeline 4, the best performing model is Random Forest, plus it is the one whose average fit time is smaller, since different trees can be grown in parallel, and in every node only a subset of features is evaluated (so adding new variables does not increase too much that time). In contrast, SVM fit time is too high, since the *sklearn* implementation fits $n_{classes} \cdot (n_{classes} - 1)/2$ models (that would be 10 models in our case), in a one-versus-one fashion. Moreover, adding the profile image metadata decreased the average accuracy and QWK of the ensemble models, while the simpler models improved.

Pipeline	Model	Average fit time	Average accuracy	Average QWK
1	XGBClassifier	7.027791	0.424397	0.356105
	RandomForestClassifier	2.654937	0.415927	0.342006
	SVC (rbf kernel)	91.539028	0.389114	0.298415
	Logistic Regression	4.030118	0.378176	0.268038
2	XGBClassifier	9.140597	0.427733	0.360878
	RandomForestClassifier	3.255018	0.441072	0.370197
	SVC (rbf kernel)	105.852106	0.391982	0.303153
	Logistic Regression	4.802423	0.383245	0.274153
3	XGBClassifier	11.879944	0.426465	0.357981
	RandomForestClassifier	4.904582	0.435936	0.366991
	SVC (rbf kernel)	97.261596	0.395050	0.310946
	Logistic Regression	6.232931	0.383378	0.280119
4	XGBClassifier	15.188047	0.425731	0.368089
	RandomForestClassifier	5.962002	0.438337	0.379648
	SVC (rbf kernel)	111.837167	0.399452	0.323938
	Logistic Regression	7.300400	0.386113	0.290413

Table 4.1: Preliminary evaluation results of each pipeline extension in 5-CV

After we got these preliminary results and determined to use XGBoost to evaluate the

extracted image and text features, we included those as the two final extensions of the pipeline, yielding the evaluation results that we can see in Table 4.2, in which pipeline 5 is pipeline 4 + the aggregated image features extracted from the CNN we selected and pipeline 6 is pipeline 5 + TF-IDF on the *Description* text (fixed to 16 SVD components). Since the CNN to extract image features was trained on a single split of training and validation, the 5-CV evaluation gives optimistic results, because in each validation fold there will be features extracted from images that were used to train that CNN, as we mentioned in section 3.3. If we compare the results of pipeline 4 with the ones of pipeline 5 (even if the 5-CV ones are optimistic), adding the image features was a huge improvement, while we can see that adding the text features in pipeline 6 did not improve the single split results of Random Forest and Logistic Regression. In fact, the results of SVM and Random Forest in the last pipeline extension are similar, but the average fit time of SVM is very prohibitive.

According to these results, we discarded conducting a search of hyperparameters for the SVM and Logistic Regression algorithms, and we will focus on the tree-based ensembles.

Pipeline	Model	Avg. fit time	Avg. accuracy	Avg. QWK	1 split accuracy	1 split QWK
5	XGBClassifier	22.974383	0.441806	0.426829	0.438146	0.402173
	RForestClassifier	6.793313	0.445006	0.425277	0.437479	0.401443
	SVC (rbf kernel)	112.041703	0.431934	0.414075	0.425475	0.387992
	Log. Regression	8.006603	0.425797	0.411592	0.413471	0.387436
6	XGBClassifier	40.896393	0.452677	0.435379	0.464155	0.417941
	RForestClassifier	15.361865	0.456546	0.434955	0.441147	0.391791
	SVC (rbf kernel)	134.074037	0.445340	0.425769	0.434478	0.390644
	Log. Regression	16.713814	0.427864	0.412946	0.405802	0.372089

Table 4.2: Preliminary evaluation results of pipelines with image and text features in 5-CV and single split validation

4.2. Classification or Regression?

All the learning algorithms that we have used so far set out the prediction of *AdoptionSpeed* as a classification problem (the only exception is the CNN with rank-consistent outputs, which didn't work very well). A simple way to set out this problem as an ordinal regression one consists in **fitting a regression model on the training data and then rounding up or down the predictions** depending on a number of thresholds, so that continuous values are converted to the *AdoptionSpeed* classes. We could use pre-established thresholds like 0.5, 1.5, 2.5 and 3.5, so that the predictions below 0.5 are rounded to 0, those between 0.5 and 1.5 to 1, etc. However, as we may expect, this is a very naïve approach and the results

are not very good (they can be found at the end of Notebook 2).

Instead, another way is optimizing those thresholds, so that we train another model after the regressor than rounds the predicted values. This is the approach that some users shared in the competition forum[6], and it is based on the Nelder-Mead optimization method. This technique was named after [Nelder and Mead, 1965], and the following it's a summary of how it works[7].

The **Nelder-Mead method** is a heuristic optimization technique, which does not solve the optimization of an objective function analytically but by searching different configurations of parameters from an initial guess, which does not guarantee to reach the global optimum. In particular, the way this method conducts the search is by applying a number of transformations to a 'simplex', that is, a n-dimensional version of a triangle, whose number of vertices is equal to the number of search space dimensions plus one. In particular, given an initial guess, a point in the n-dimensional space, a number of random guesses not too far from the initial guess are generated so that in total we can have n+1 points that define the simplex. Then, we sort those points by the value of the objective function at each one of them and we keep as reference the two worst values and the best one. This way, several transformations can be applied, which are the ones shown in Figure 4.4[8], based on pivoting the simplex over all the vertices minus one or two or reducing or increasing its size, according to the transformation that is applied based on whether the values of the new vertices allow us to get closer to a local optimum.

In our case, the objective function is a wrapper of the QWK statistic, since it first receives a number of thresholds and then use them to round the predicted values of a regression model, which now can be used as input by the function that computes the QWK with respect to the true *AdoptionSpeed* labels. Thus, the initial guess can be the naïve thresholds (0.5, 1.5, 2.5 and 3.5) and eventually what we get with the Nelder-Mead method is a list of optimized thresholds (the best vertex of the last simplex), which should be near a local optimum (the method stops after a number of iterations). The code was extracted from the aforementioned competition's forum; the reader can found it in the Annex.

After evaluating a XGBoost model for regression using this optimization method, the average QWK result in 5-CV increased compared to the classification model (from 0.435379 to 0.448707). However, the accuracy decreased significantly (from 0.452677 to 0.394117) and the f1-score of each class also did. Thus, we concluded that the classification version had a better balance and we wouldn't search for the best hyperparameters of regression models, even though we will come back to this topic in section 4.4 for a good reason (related to the expected distribution used in the QWK statistic).

[6] https://www.kaggle.com/c/petfinder-adoption-prediction/discussion/76107

[7] The following post by Sachin Joglekar may help the reader to better understand this method: https://codesachin.wordpress.com/2016/01/16/nelder-mead-optimization/

[8] Extracted from [Dang et al., 2012]

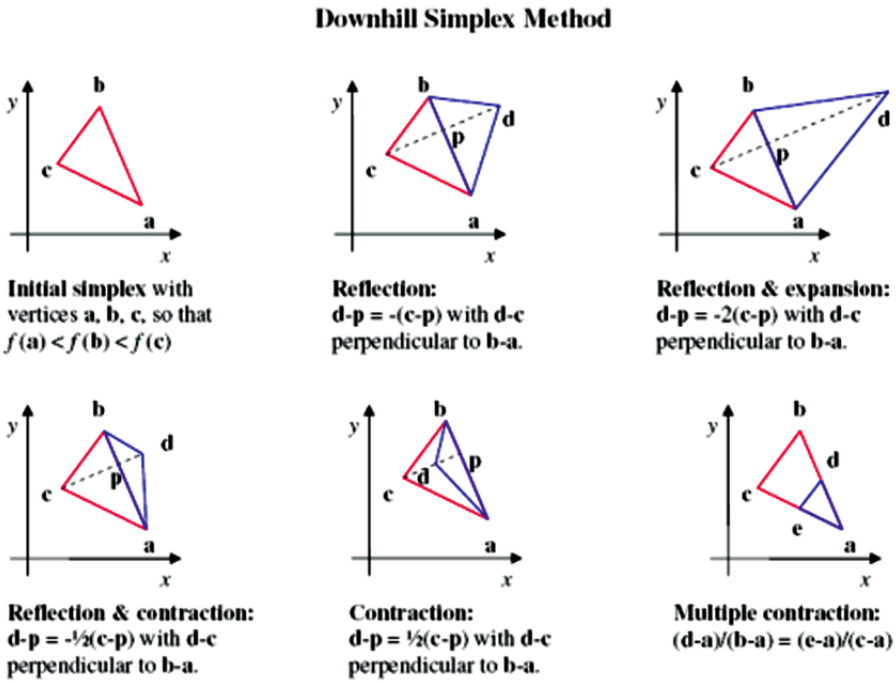

Figure 4.4: Transformations applied to explore the search space using Nelder-Mead method

4.3. Hyperparameter tuning

In order to evaluate different combinations of hyperparameters of the transformers and the final estimator in the pipeline, we should not repeatedly use the same 5-CV approach as with the preliminary evaluations for each combination, since we would implicitly get information about the entire dataset after multiple evaluations and the hyperparameters could overfit to that data. In other words, if we use the same dataset to evaluate these models, we would probably get biased (optimistic) estimations.

Consequently, we have to **decouple the hyperparameter search from the evaluation of the selected model** (the best performing one among the search space according to some metric). In particular, the approach we have followed is shown in Figure 4.5: first of all, we 'inherited' a partition of the original training dataset used to train the CNN, so we should not mix those instances in order to obtain the less possible biased estimation (since that validation set does not include images used to train the CNN, while the training set does). Then, the hyperparameter search has to be conducted in that training set. In order to achieve the aforementioned decoupling, we evaluate the models at two different levels: an **outer 5-CV** (on the CNN training set), used to evaluate the selected model after the hyperparameter search, and then an **inner 3-CV** used to estimate how good a hyperparameter combination is and select the best such combination according to the QWK statistic value (the inner evaluation is done inside the training folds of the current outer iteration). All the folds in the

outer and inner cross-validation processes are obtained in a stratified manner. The outer 5-CV gives us an estimation of how well the pipeline performs if we conduct an unbiased hyperparameter search, but we know that it is still an optimistic estimation (due to the training image features). Thus, the last 'filter' consists on training the pipeline and evaluating it on the CNN validation set.

As each outer 5-CV iteration corresponds to a different hyperparameter search (due to the different data, but the combinations that we have checked are always the same across the iterations), we could end up with 5 different models, so we have to select one of them to be trained and validated on the CNN sets. This is done by computing the average rank (according to the QWK score) that every combination that ended up being selected had on the other search processes (that is, first we get the selected models that were eventually validated in each outer iteration, and look the average rank they had in the other 4 search processes). The one with the greatest average rank (i.e., closer to 1) is selected.

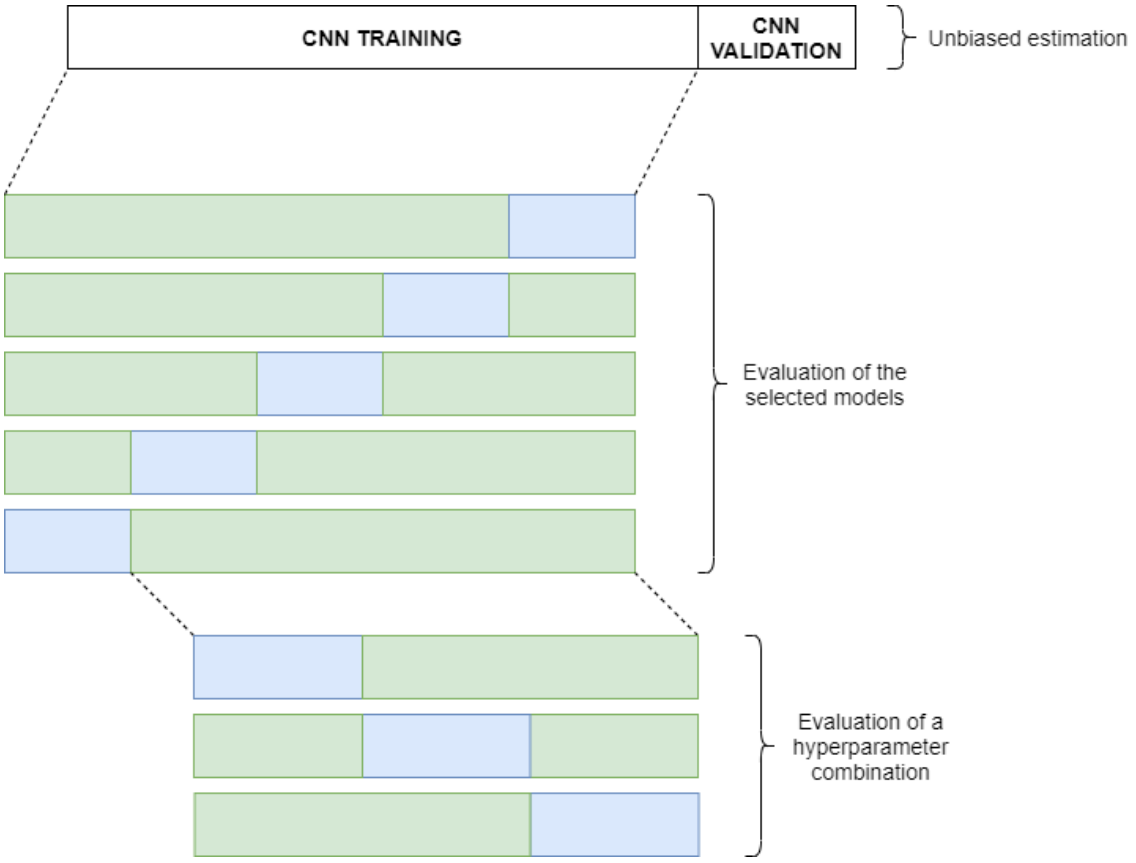

Figure 4.5: Strategy to decouple the evaluation of hyperparameters and the evaluation of the selected model

The hyperparameter tuning results can be found in **Notebook 3** (see Annex). As we have

a limit of 9 hours in Kaggle for each run, we could not conduct an exhaustive search of hyperparameters given the possible values we specify (that would be a 'grid search'). Instead, we select a number of random combinations (we can obtain the same ones in each search using the same seed). The possible hyperparameter values of the pipeline transformers that we have specified are the following:

- Breed encoding: target and frequency encoding or one-hot encoding + SVD (fixed to 10 components).
- Image metadata: only profile image or aggregation of all images.
- Image properties: only profile image or aggregation of all images.
- TF-IDF n-gram range: (1,1), (1,2), (1,3). This implies that all the vocabulary terms detected using a value of (1,1) are included in (1,2) and (1,3), but the second one includes bi-grams and the third one both bi-grams and three-grams.
- TF-IDF sublinear term frequency: whether to use this approach[9] or not.
- TF-IDF maximum document frequency: use all the possible vocabulary (1.0) or discard those terms or n-grams that appear in more than the 90% of documents.

  (SVD after TF-IDF is fixed to 16 components)

The number of possible combinations of the previous hyperparameters is 96; this implies that even if we include a small number of hyperparameters of the final learning algorithm, we end up with a large search space. This is why even if some transformers have more possible hyperparameters, their values were left fixed to the default ones.

As we mentioned, we have conducted the search using Random Forest and XGBoost as final estimators. Moreover, we also included another implementation of Gradient Boosting, **LightGBM**. According to [Ke et al., 2017], LightGBM introduces two new techniques to increase the efficiency:

- **Gradient-based One-Side Sampling** (GOSS): it consists on downsampling the number of data instances used to generate the quantiles that determine the splits. This is done by selecting for each tree a number of instances with the largest gradients, and then selecting a random subset of the remaining instances with smaller gradients (since the training error in those cases is smaller). Then, in order to give more importance to the instances with larger gradients, their weights are increased (the weights determine how large a quantile is, so by increasing them it will be more likely to include those instances with larger gradients in smaller quantiles where their contribution is also greater).
- **Exclusive Feature Bundling** (EFB): this technique aims to reduce the number of features by merging those which are mutually exclusive, that is, they cannot be zero at the same time. We could think that this would be similar to reverting a one-hot encoded variable and then applying ordinal encoding (one-hot encoded variables are

[9] https://nlp.stanford.edu/IR-book/html/htmledition/sublinear-tf-scaling-1.html

suboptimal in decision trees, as we may have to expand the tree very deeply to evaluate all the possible values of a single original variable). In order to group or bundle features, a weighted graph is constructed (each weight measures the fraction of instances whose value in the pair of variables is non-zero at the same time), and then new bundles are created as long as the number of conflicts in the same bundle after adding a new variable exceeds a threshold.

Random Forest doesn't have a great number of hyperparameters, so we only changed the number of trees (80, 100 or 120) and the minimum number of instances used to create any leaf (1, which could possibly lead to overfitting, or 5). We searched the 31.3% of the possible number of combinations. In the case of XGBoost, we could change the number of estimators (80, 100 or 120), the maximum depth of each tree (4, 5 or 6), the learning rate (i.e., the proportion of change the trees can induce from the initial prediction, 0.1 or 0.3), the γ value (which measures the minimum gain to split a node, 0.0 or 1.0) and λ (strength of L2 regularization, 0.0, 1.0 or 5.0). In this case, even with the histogram method running on the GPU, we could only search 180 out of 10368 combinations (1.7%). Finally, for LightGBM we varied the number of trees (same values as before), the number of leaves (LightGBM grows the trees in best-first fashion, in contrast to XGBoost that does it breadth-first, so limiting the maximum depth could be counter-productive; 15, 31 or 63), the learning rate (0.05 or 0.1), the minimum number of samples each child must have in order to split the current node (20, 30 or 40) and λ (here it also corresponds to the L2 regularization, 1.0 or 5.0). As we didn't use random subsampling of features in XGBoost, we use it in this case at the tree level; this way, the training time is reduced significantly and we can fit approximately the same models than XGBoost but without using the GPU in this case (in particular, 200 out of 10368 combinations, 1.9%).

The results are shown in Table 4.3. The first group of metrics corresponds to the outer 5-CV results, while the other corresponds to the single split validation using the best average ranked model in each case. We can see that even if the Random Forest classifier seems to perform better than the rest after tuning its hyperparameters, its capacity to generalize is worse than that of XGBoost, according to the single split results. On the other hand, the LightGBM model did not perform very well. The exact combination of hyperparameters after ranking them can be found in Notebook 3. In the case of the best model, XGBoost, we found that stronger regularization values were selected (both L2 regularization and minimum split gain), with a greater number of estimators, smaller maximum depth and smaller learning rate than the default.

Model	Avg. accuracy score	Avg. QWK score	S. split accuracy	S. split QWK
RandomForest	0.460147	0.447716	0.444481	0.412748
XGBoost	0.457562	0.440089	0.453151	0.419098
LightGBM	0.450976	0.436128	0.440480	0.397288

Table 4.3: Hyperparameter tuning results

Moreover, we also built two ensembles of those three models, using majority voting in the first one and stacking in the second one (that is, using the predictions of those models as input to a final estimator, which in this case was Logistic Regression). However, neither of them improved the accuracy or the QWK score of XGBoost, so they were discarded.

Finally, in order to visualize how much the different pipeline extensions and the final hyperparameter tuning can improve our predictions, we can see Figure 4.6, where the first plot corresponds to the preliminary evaluation results of pipeline 1 and the second one corresponds to the selected XGBoost model in the same 5-CV (they are precision-recall curves, which are more suitable than other visualizations like ROC curves when the distribution of the classes is imbalanced)[10].

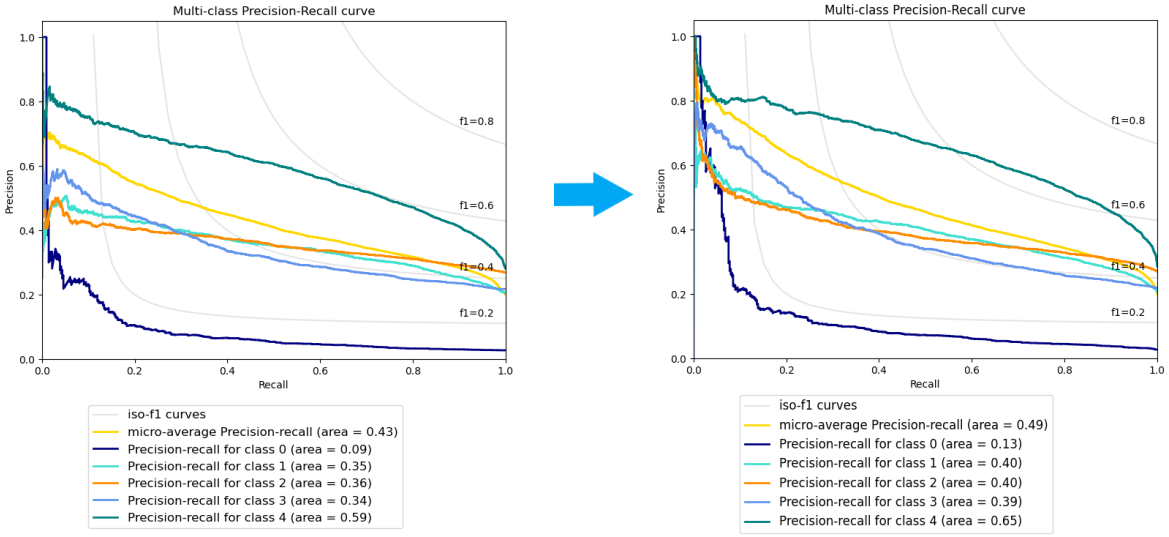

Figure 4.6: Precision-recall curves from pipeline 1 to final pipeline after hyperparameters tuning (XGBoost)

4.4. Final models

The last step is to train the selected model (pipeline with XGBoost) on the complete training set and then predict the instances of the test dataset. However, there are some issues that we noticed and that makes the final performance of the model worse than we expected.

The first issue is that the proportion each class represents in the prediction of the test instances is very far from the observed distribution in the training dataset. In general, we could expect the test cases to be 'trickier', but the difference is huge (for example, from

[10] More information about precision-recall curves and the code used to obtain the plots can be found in https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html

a percentage of approximately 28% '4' cases, the model predicted more than 60% of '4' cases), also given that the size of the training set is not relatively small. Thus, we trained another model: the regression version of XGBoost (with the same hyperparameters as the classifier version, even if they are not probably optimal for this case), rounding the final predictions with the Nelder-Mead optimization. We have used this approach because when we first used it we saw that the predictions were more evenly distributed, even if the accuracy was not very good. Indeed, with the regression model the main classes ('1', '2', '3' and '4') were more evenly distributed (for example, from more than 60% of '4' cases, we reduced it to approximately 35% and the difference in the distribution of the rest of classes is also reduced; the reader can check this at the end of Notebook 3, Version 11), obtaining more feasible results.

After solving this issue, **our final model would be the regression one**. However, once we submitted the predictions in the competition, the obtained score was far from our expectations: the QWK score of those predictions was 0.36761, while we expected around 0.42 given the single split validation score (of the XGBoost classifier case). One of the reasons, and let's point out that this is the first time we extract information from the test dataset, is that none of the *RescuerID* values in that dataset coincide with any of the training ones, and *RescuerCount* is the third most important variable of the entire dataset by average gain in the XGBoost regression model, as we can see in Figure 4.7[11]. This implies that **all the *RescuerCount* values assigned to the test cases is 1**, and we saw in Chapter 2 that with that value the probability of being a '4' case was significantly higher; moreover, as *RescuerCount* was not an useless variable at training time (that is, its variance was greater than zero), of course at prediction time it is not removed. If we remove the transformer that includes *RescuerCount* and we predict the *AdoptionSpeed* of the test instances, first of all the class distribution in the predictions is very similar to that of the training dataset (see last part of Netbook 3, Version 12), and secondly the QWK score increases from 0.36761 to 0.39003 (which would be even closer to the single split validation if we consider that it would have probably been smaller than the one we obtained if we didn't use *RescuerCount*).

We think that this change in the distribution of the predicted classes is due to the fact that the function to be optimized by the Nelder-Mead method relies on the QWK score after rounding the predictions with the thresholds, which in turn is a statistic that takes into account the distribution of the true labels (this would explain how similar the distribution of the predictions is to that of the training set after removing the noisy *RescuerCount* variable) and also the ordinal relationship which is not captured by a classifier (greater differences between the integer *AdoptionSpeed* labels had greater weights, as we mentioned).

Of course, this information would be used to improve our score in the competition, but as our objective was conducting a proper data mining process without data leaks, and not necessarily obtaining the highest possible score, we can assume that the QWK score of our final pipeline on new data is 0.36761. However, trying to explain the results if we can is another important step in the process.

[11]Moreover, the reader can find in the dataset in which we stored the intermediate results (see Annex) the file *xgb_tree_4.pdf*, which contains the graphical representation of the first tree created by the XGBoost regressor for the class '4': *RescuerCount* is the only variable that is used in all the levels (excluding the root).

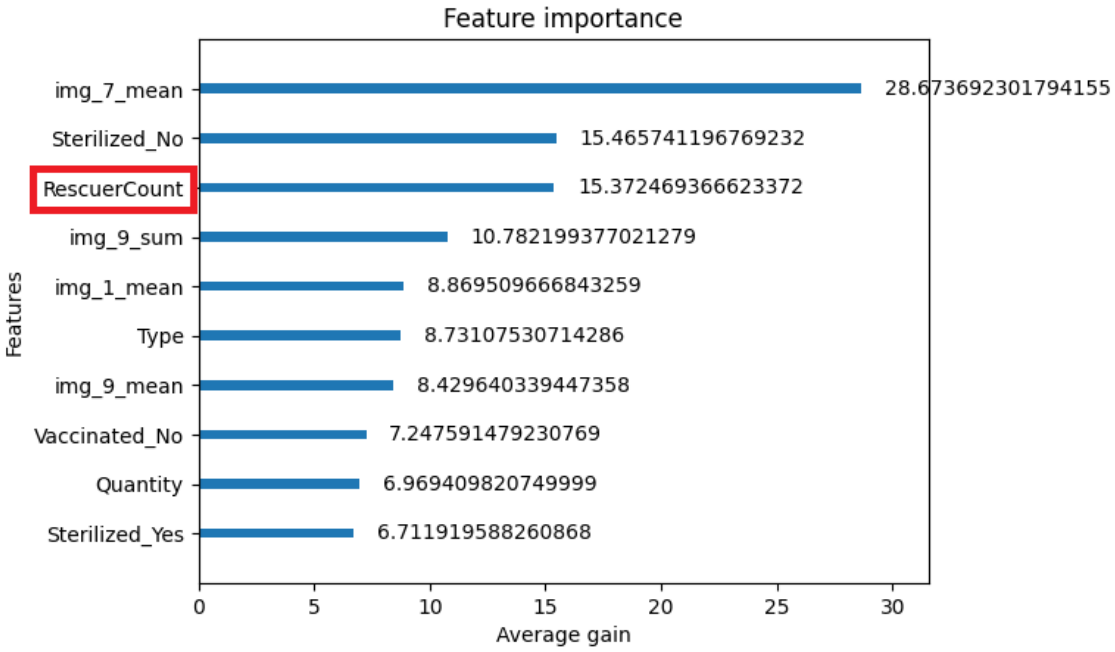

Figure 4.7: Top 10 features by average gain on XGBoost (regression)

5. Conclusions and Future Work

Throughout the development of this project, our main objective was conducting a data mining process divided in several stages where a variety of analyses and techniques are applied to extract information from a set of data in order to solve a particular problem: predicting the adoption speed of an online pet profile.

Consequently, this work aimed to review and use different machine learning methods, applying the most suitable ones given the format and properties of the data, with a special focus on Deep Learning techniques to extract image features (thus developing competence **[CM5])**. We have used a number of learning algorithms that belong to different paradigms and include them in different pipelines that preprocess the data they eventually use (competence **[CM7]**). Some of these techniques and models were not previously used by the author (and maybe the reader is not familiar with all of them either), hence we tried to explain them in the best possible way in order to widen our knowledge (competence **[CM4]**).

More specifically, we have faced some challenges and also different problems; for example, how a single variable that seemed to be among the ones that provided more information about the target could cause our final model to be less robust and not generalize as much as we would like. In this case an alternative validation strategy like adversarial validation could be a better choice, for example. We mention this to show that, of course, the work we have done can be improved.

Precisely, there are other techniques that we have not explored and could help us giving a better solution. In particular, the main area that can be improved is the extraction of text features, using state-of-the-art Natural Language Processing techniques, such as Recurrent Neural Networks (RNN) or more specific models like Long Short-Term Memory networks (LSTM). Nowadays, natural language processing is undergoing rapid progress, so there may arise new techniques besides the previously mentioned ones that allow us to extract more information from this format of data.

Finally, the only competence that we could not develop as we would like is **[CM6]**, due to the environment in which this work was carried out (Kaggle), so a good way to extend it would be actually deploying the model (for example, using the tools provided by platforms like AWS or Azure). This deployment would require Software Engineering and DevOps skills in order to define an infrastructure to obtain the different data from each source (each time

a new request or pet profile is published), training and deploying the actual models, automatically testing them, setting up the storage, data visualization, user interface, etc.

A. Annex

The following are the hyperlinks of all the notebooks and scripts of code that we have implemented or used in Kaggle to develop our project:

Notebook 1 (EDA):
https://www.kaggle.com/davidmora/tfg-pet-adoption-eda
Utilities of Notebook 1:
https://www.kaggle.com/davidmora/utils-tfg-pet-adoption-eda
Functions to extract image properties:
https://www.kaggle.com/shivamb/ideas-for-image-features-and-image-quality/notebook
Notebook 2 (Feature Engineering):
https://www.kaggle.com/davidmora/tfg-pet-adoption-fe-fss
Transformers extracted from Notebook 2:
https://www.kaggle.com/davidmora/transformers-tfg-pet-adoption
Nelder-Mead optimization to round predictions:
https://www.kaggle.com/c/petfinder-adoption-prediction/discussion/76107
Notebook 3 (hyperparameter tuning and final models):
https://www.kaggle.com/davidmora/tfg-pet-adoption-hpt-models
Generated data and intermediate results of all the stages:
https://www.kaggle.com/davidmora/tfg-pet-adoption-data

Bibliography

[Azur et al., 2011] Azur, M., Stuart, E., Frangakis, C., and Leaf, P. (2011). Multiple imputation by chained equations: What is it and how does it work? *International Journal of Methods in Psychiatric Research*, 20(1):40–49.

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

[Cao et al., 2019] Cao, W., Mirjalili, V., and Raschka, S. (2019). Consistent rank logits for ordinal regression with convolutional neural networks. *CoRR*, abs/1901.07884.

[Chen and Guestrin, 2016] Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754.

[Chollet, 2016] Chollet, F. (2016). Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357.

[Cortes and Vapnik, 1995] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3):273–297.

[Crawford et al., 2017] Crawford, H. M., Fontaine, J. B., and Calver, M. C. (2017). Using free adoptions to reduce crowding and euthanasia at cat shelters: An australian case study. *Animals*, 7(12).

[Dang et al., 2012] Dang, Q., Nguyen, T., Bae, W., and Nguyen, H. (2012). Advanced reservoir management to maximize hydrocarbon recovery in mature and geologically complex reservoirs. *Energy Sources Part A-recovery Utilization and Environmental Effects*, 34:1288–1304.

[Dongarra et al., 2018] Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Tomov, S., and Yamazaki, I. (2018). The singular value decomposition: Anatomy of optimizing an algorithm for extreme scale. *SIAM Review*, 60:808–865.

[Friedman, 2001] Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256. PMLR.

[Glorot et al., 2011] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In Gordon, G., Dunson, D., and Dudík, M., editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323. PMLR.

[Grandini et al., 2020] Grandini, M., Bagli, E., and Visani, G. (2020). Metrics for multi-class classification: an overview.

[Halko et al., 2010] Halko, N., Martinsson, P.-G., and Tropp, J. A. (2010). Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions.

[Hinton et al., 2006] Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554.

[Hornik et al., 1989] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366.

[Huang et al., 2016] Huang, G., Liu, Z., and Weinberger, K. Q. (2016). Densely connected convolutional networks. *CoRR*, abs/1608.06993.

[Ke et al., 2017] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[Kim, 2014] Kim, Y. (2014). Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882.

[Kingma and Ba, 2017] Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

[Kusner et al., 2015] Kusner, M. J., Sun, Y., Kolkin, N. I., and Weinberger, K. Q. (2015). From word embeddings to document distances. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 957–966. JMLR.org.

[Loh, 2017] Loh, I. (2017). No dog until neighbours agree. *The Star*.

[Lord et al., 2014] Lord, E., Olynk Widmar, N., and Litster, A. (2014). Economic impacts of adoption and fundraising strategies in animal shelters. *Preventive Veterinary Medicine*, 113(4):423–429.

[Micci-Barreca, 2001] Micci-Barreca, D. (2001). A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problems. *SIGKDD Explor. Newsl.*, 3(1):27–32.

[Mikolov et al., 2013] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space.

[Nakamura et al., 2020] Nakamura, M., Dhand, N., Wilson, B., Starling, M., and McGreevy, P. (2020). Picture perfect pups: How do attributes of photographs of dogs in online rescue profiles affect adoption speed? *Animals*, 10:152.

[Nakamura et al., 2019] Nakamura, M., Dhand, N. K., Starling, M. J., and McGreevy, P. D. (2019). Descriptive texts in dog profiles associated with length of stay via an online rescue network. *Animals*, 9(7).

[Nelder and Mead, 1965] Nelder, J. and Mead, R. (1965). A simplex method for function minimization. *Comput. J.*, 7:308–313.

[Pech Pacheco et al., 2000] Pech Pacheco, J. L., Cristobal, G., Chamorro-Martinez, J., and Fernandez-Valdivia, J. (2000). Diatom autofocusing in brightfield microscopy: A comparative study. volume 3, pages 314–317 vol.3.

[Pennington et al., 2014] Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

[Rix et al., 2021] Rix, C., Westman, M., Allum, L., Hall, E., Pockett, J., Pegram, C., and Serlin, R. (2021). The effect of name and narrative voice in online adoption profiles on the length of stay of sheltered cats in the uk. *Animals*, 11(1).

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408.

[Simonyan and Zisserman, 2015] Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition.

[Sun et al., 2017] Sun, Y., Zhu, L., Wang, G., and Zhao, F. (2017). Multi-input convolutional neural network for flower grading. *Journal of Electrical and Computer Engineering*, 2017:1–8.

[Szegedy et al., 2016] Szegedy, C., Ioffe, S., and Vanhoucke, V. (2016). Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261.

[Szegedy et al., 2014] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. *CoRR*, abs/1409.4842.

[Szlam et al., 2014] Szlam, A., Kluger, Y., and Tygert, M. (2014). An implementation of a randomized algorithm for principal component analysis.

[van Buuren and Groothuis-Oudshoorn, 2011] van Buuren, S. and Groothuis-Oudshoorn, K. (2011). mice: Multivariate imputation by chained equations in r. *Journal of Statistical Software*, *Articles*, 45(3):1–67.

[Vojtkovská et al., 2019] Vojtkovská, V., Voslářová, E., and Večerek, V. (2019). Comparison of outcome data for shelter dogs and cats in the czech republic. *Animals*, 9(9).

[Weiss et al., 2012] Weiss, E., Miller, K., Mohan-Gibbons, H., and Vela, C. (2012). Why did you choose this pet?: Adopters and pet selection preferences in five animal shelters in the united states. *Animals*, 2(2):144–159.

[Xu et al., 2015] Xu, B., Wang, N., Chen, T., and Li, M. (2015). Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853.

[Zoph et al., 2017] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2017). Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012.