Technische Universität Berlin

Institute of Software Engineering and Theoretical Computer Science
Database Systems and Information Management (DIMA)

Einsteinufer 17,
10587 Berlin

Master Thesis

Techniques for Hierarchical Product Classification

Marilyn Nowacka Barros

Matriculation Number: 365912
August 31, 2017

Advisors
Juan Soto (DIMA, TU Berlin)
Jan Anderssen (idealo Internet GmbH)

Reviewer
Prof. Dr.Volker Markl

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, August 31, 2017

. . . . . .

(Signature M. Nowacka Barros)

Abstract

As humans, we go through life categorizing objects: types of activities we can do, dog races, forms of life, departments in a company, sections in a supermarket, and we could keep naming examples for a while. Online shops (e.g. amazon.de), and in particular online tools for comparing prices of the online shops (e.g. idealo.de) are not the exception. In this domain (but not exclusively) the categories are presented in a hierarchy, where the upper levels are more general concepts, and the lower levels are more specific. A taxonomy of categories like this can contain thousands of categories, distributed among some levels. Further, the catalog of products that needs to be classified is growing every day. In recent years, multinomial product categorization in e-commerce domains has gained some attention ([SRS12, SRSS11, CJS11]). The need of classifying products automatically has increased, since manual classification costs can be high.

A solution for classifying products accurately into a hierarchical taxonomy is the main purpose of this thesis. In particular, different feature engineering strategies (e.g, word2vec, PCA) were explored. Also a diversity of supervised machine learning algorithms for classification, including SVM, kNN and multinomial logistic regression were investigated. From those the latter outperformed the others, a result that was not seen in any other work.

Different strategies for approaching in general the classification task are possible. We could basically just build a classifier that discriminates between all of the thousands of categories (i.e. a flat approach). However, we could also use the underlying hierarchy in our favor. The latter has been shown to work better in similar environments. In general the second approach was taken. However, a comparison with the flat one is made, in order to confirm the claims of previous works.

Due to the need for a scalable solution, Apache Spark was employed. Further, this allows our solution to be executed on a cluster, lowering execution times. To the best of our knowledge, this hasn't been studied before to tackle a similar problem setup.

Zusammenfassung

Es liegt in der Gewohnheit des Menschen Objekte in verschiedene Kategorien zu unterteilen: Alltägliche Aktivitäten, Hunderennen, Abteilungen einer Firma und Abschnitte eines Supermarkts, um nur wenige Beispiele zu nennen. Onlineshops (z.B. amazon.de) sowie insbesondere Preisvergleichsportale (z.B. idealo.de) bilden hierfür keine Ausnahme. In diesem Bereich sind Kategorien hierarchisch aufgebaut, wobei höhere Ebenen allgemeinere, und niedrigere Ebenen spezifische Konzepte darstellen. Solch eine Taxonomie kann tausende Kategorien, verteilt auf mehreren Ebenen, umfassen. Hinzu kommt, dass der zu klassifizierende Produktkatalog stetig wächst. In den vergangenen Jahren hat die multinomiale Kategorisierung von Produkten im Onlinehandel an Aufmerksamkeit gewonnen ([SRS12, SRSS11, CJS11]). Aufgrund des hohen Aufwands manueller Klassifizierung ist der Bedarf an automatischer Klassifizierung von Produkten gestiegen.

Diese Arbeit stellt eine Lösung für die genaue und automatische Produktklassifizierung in eine hierarchische Taxonomie vor. Insbesondere wurden sowohl verschiedene feature engineering Strategien (z.B. word2vec, PCA), als auch überwachtes maschinelles Lernen, unter anderem SVM, kNN und multinomiale logistische Regression, untersucht und ausgewertet. Es konnte festgestellt werden, dass letzterer Lernalgorithmus die anderen beiden übertrifft, ein Ergebnis welches in vorherigen Arbeiten nicht verzeichnet wurde.

Verschiedene Herangehensweisen bieten sich an, allgemeine Klassifizierungsprobleme zu lösen. Am naheliegendsten kann ein einziges Modell erstellt werden, das zwischen allen tausend Kategorien unterscheidet (sog. flacher Ansatz). Andererseits ist es ebenso möglich die zugrunde liegende Hierarchie auszunutzen. Aufgrund positiver Ergebnisse die in ähnlichen Anwendungsfällen erzielt werden konnten, verfolgt diese Arbeit in erster Linie den zweiten Ansatz. Nichts desto trotz wird ein Vergleich zum flachen Ansatz durchgeführt, um die Behauptungen vorheriger Arbeiten bestätigen zu können.

Aufgrund des Bedarfs einer skalierbaren Lösung wird Apache Spark für die Implementierung verwendet. Weiterhin erlaubt dies die Ausführung auf einem Cluster, was zusätzlich die Programmlaufzeit verringert. Nach bestem Wissen wurde dieser Aufbau für ähnliche Problemstellungen noch nicht verwendet.

Contents

List of Figures

List of Tables

1. Introduction

> "Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."
>
> — Lewis Carroll, *Alice in Wonderland*

Nowadays, consumers are increasingly opting to shop online. However, with so many retailers offering products online, arguably, consumers need better solutions to help them identify the specific product they would like to purchase at the lowest possible price. Commonly, retailers categorize consumer products according to varying schemes (e.g., based on a company-specific taxonomy) and performing classification using manual or automated means. Unfortunately, these methods are not entirely accurate. Therefore, in this master's thesis, we aim to use machine learning techniques for multiclassification of online products and map them to a taxonomy of categories. As a by-product, this would likely also help to improve the detection of product duplicates and discover complementary product recommendations [Koz15].

Despite the wide variety of existing eCommerce platforms, they all share one thing: the need to automatically classify their product offerings into a hierarchical taxonomy. Some research has already been done in this area by Yahoo! ([Koz15]), eBay ([SRS12, SRSS11, SRMS12]), and other more general work (e.g., performing hierarchical classification on web content [DC00]). As proposed in [SRS12, SRSS11, DC00], when a hierarchical taxonomy is employed for classification, we could vary the set of product features or the particular multiclassification algorithm depending on the level of the tree we are classifying into. However, to the best of my knowledge, combining the previous strategy with typical machine learning classification algorithms (e.g., SVM, kNN, Multinomial Logistic Regression) has not been tried using large-scale data processing engines, such as Apache Spark (e.g., to shorten runtimes and reduce training times). Indeed, there is a lot of room for improvement (e.g., increasing classification accuracy, reducing analysis times, and facilitating conducting benchmarks).

Germany's largest price comparison platform (idealo.de) offers consumers varying navigation options. Users are able to browse almost 2 million products that are labeled using approximately 2000 different categories. These categories are organized into a hierarchical taxonomy, where products can be compared at a glance. For example, a Coleman Tent is classified as Leisure & Outdoors - Camping - Tent, i.e., a Coleman Tent is classified as a Tent, which is a category that belongs to Camping, and at the same time is contained the Leisure & Outdoors' category. Designing and implementing a system that can automatically organize products into a broad set of categories accurately is a non-trivial task. Due to the large amounts of data handled by e-Commerce platforms

every day, we need a multiclassification algorithm that can run on a large-scale data analytics system, like Spark. This should enhance runtime efficiency and offer a scalable solution.

1.1. Motivation

Humans tend to classify everything. It's a cognitive task. This is a definition that is very related to *concepts* and how we give meaning to objects. "Classifications attempt to arrange a diversity of entities into sets of classes based on similarities possessed by the included individual entities" [MB02]. One of the ways people for example sort their ideas involves the use of hierarchies. We find hierarchy in many domains, such as biology (e.g. classification of organisms, diseases), classes in programming languages (e.g., Scala, Java, Smalltalk), drugs or medicines, etc.

Specifically in the domain of e-commerce, marketplaces as Amazon or eBay organise their products into a hierarchical schema. Each class is a category that can contain more specific categories underneath. Classifying items for an e-commerce site correctly, is a fundamental task for reasons such as assigning category specific rules, determining a reasonable shipping, handle fees or detecting duplicate products ([SRS12]). This, besides the direct impact on the user experience.

An item (i.e. a product) in this domain is basically a compound of information in the form of text (and maybe some pictures). This is why, when we talk about this specific type of problems, it's usually related to solving a text classification task ([SRSS11]).

The purpose of this work is to explore different approaches to classify items that are sold online into an already provided hierarchical schema of categories. The use of automatic classification methods hopefully will facilitate and decrease the need of doing it manually.

The amount of categories in the idealo corpus (≈ 1700) and the uneven distribution of products among them, makes this problem a very challenging one. Recent works on this field have been published, tackling the problem in different ways. Different common classification algorithms have been used for this, such as Naive Bayes ([KS97]), k-Nearest Neighbors (KNN) ([YZK03], [SRS12]), Neural Networks ([OM07]), but the most commonly used is Support Vector Machines (SVM) ([DC00, YZK03, LYW$^+$05, SRS12], [CJS11]).

What the majority of these works have in common is the particular way they approach the problem: They use the hierarchical disposition of the classes in their favor ([SRS12, DC00]. In comparison to the conventional flat approach, the hierarchical approach achieves better results. Instead of treating every category separately (i.e. ignoring the underlying hierarchical structure), a top-down approach is taken: classifying first the items into the first level categories, and then into the lower levels (grained classification). The goal is basically to "divide and conquer", train a total of M models, where M = top-level categories + 1. Each of these models correspond to a multiclassification classifier. One serves for determining the first level category and after that, one per first level category is used for selecting to which of its more specific categories the item belongs to.

This hopefully will decrease training times, since the models we are training are thought to be smaller than a flatten model, and because it is possible to train them in parallel. Also, it is expected to perform better (than using the flatted approach), because we can tune independently every section of the problem. However, error propagation through levels could be a problem.

1.2. Objective

A solution for classifying products accurately into a hierarchical taxonomy is the main purpose of this thesis. Qualitatively, the proposed solution will need to be: (i) consistent, i.e., if the environment does not change, the products should always be classified into the same category (deterministically), (ii) efficient, i.e., the entire task including both training and classification should take a reasonable period of time and resources, (iii) precise and accurate, (iv) scalable, i.e., if the number of products and/or categories increases, the solution should be able to scale, and (v) adaptable, i.e., changes to the taxonomy should not drastically affect the classifier.

The goals we aim to tackle can be summarized as follows:

- Feature Selection: Determine the most significant features (e.g., which attributes are better to use).
- Feature Engineering: Transform features, using different techniques (e.g, Word2vector), in order to classify products accurately based on existing training data. This also includes cleaning the data. Basically preparing the data for the classifier.
- Training Data Selection: Choosing a representative training set that represents products from all classes, given that products may possess a skewed distribution across categories.
- Explore and Select Varying Techniques: Examine the literature behind varying algorithms (e.g., supervised learning algorithms, such as SVM or kNN), structural techniques (e.g., consider the underlying hierarchical taxonomy), and strategies for multiclassification (e.g., one-against-all or all-against-all), then select appropriate approaches for implementation.
- Implement and Compare Techniques: Implement and compare different algorithms and settings for multiclassification, based on the previously obtained results.
- Evaluation: Conduct experiments using the implemented approaches, evaluate their performance using well- known metrics, such as the F1 score, and then compare the obtained results against the current baseline.

1.3. What is contained in this work?

The thesis is comprised of three general stages. First, we employ data preprocessing and feature extraction techniques, in order to simplify the text (e.g., product description,

title), seek to detect and eliminate outliers that are classified wrongly in the training set, and make the data easier to use. This includes making the proper transformations to the data. Second, we perform multiclassification, i.e., explore different algorithms and engines for large-scale data processing. Lastly, we evaluate the results of the classification process.

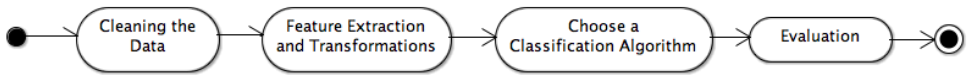


Fig. 1.1. Classification Pipeline

Basically, we will experiment with things that can be done in the two inner states of the diagram in Figure 1.1: Feature Extraction and Transformations, and Choose a Classification Algorithm (including how to tackle the problem considering the hierarchical organization of its classes).

1.4. Peek through next Chapters

This thesis is split into seven chapters.

Chapter 2: Fundamentals and Related Works. Here, we will describe shortly previous work. Which techniques have been tried, and which concepts have been exploited.

Chapter 3: The Problem. In this chapter, we will explain how the classifier is integrated into idealo's system. We describe "the big picture". Also we briefly discuss the requirements and settings.

Chapter 4: The Data. For a better understanding of the problem, first we need to make a deep exploration of the data. In this chapter we dive into it.

Chapter 5: Assembling the Pieces. We start this chapter talking about our classification task, and what we need to consider, in order to solve it. We describe the implementation of the considered algorithms, as well as some theoretical information about them (such as scaling characteristics, assumptions, and implications).

Chapter 6: Time to Evaluate it All. Here we discuss how we evaluated our results, the test environment and (of course) the obtained results. We also explain some implemented techniques we used for improving the accuracy of our models.

Chapter 7: Conclusions: This chapter summarizes our work, describes the problems that occurred and gives an outlook about future work and recommendations.

2. Fundamentals and Related Works

> "The only true wisdom is in knowing you know nothing."
>
> — Socrates

Classifying e-Commerce items is a task that has been approached using many different techniques. There are many variables that can be changed in order to affect the final outcome: Features to use, their representation, supervised or unsupervised learning, different machine learning algorithms, among others. All of these different variables can dramatically affect the result we obtain at the end, and we are still not considering algorithm specific parameters that we need to tune, in order to adjust them the best way possible for the model we are using.

In this chapter, we'll travel through the different concepts that in one way or another are related to this topic. Multiclassification is a topic that has been explored frequently in a diverse number of fields. Since we will discuss this concept repeatedly, in order to be sure we all are on the same page, let's be sure we have a common definition about what this is.

As pointed out in [JM07, chapter 4, p. 113], some people might make a distinction between the terms *text categorization* and *text classification*. The first one, refers to sorting documents by content, while the latter is a more general concept, that includes all kinds of matching of documents to a class (e.g. by language or author). We will treat both concepts as synonyms. Also, it is worth mentioning that the term *class* will be used interchangeably with *category*, as they also establish in [JM07, chapter 4]. Nevertheless, for us it makes even more sense, as our classes actually correspond to our *business* concept of category.

Another important difference noted by [JM07] is the (perhaps evident) discrepancy between classification and clustering. The main difference is that the first one requires a predefined scheme, to which our products will be assigned (i.e. already existing category hierarchy). The assignment to one or other category, will enforce similarity between products, depending of course on a well defined reason (usually mathematical). On the other hand, clustering doesn't have any preexisting scheme. Products will be grouped together, if they seem to be similar under a certain definition. The first one (classification), can be solved with a subset of machine learning algorithms (i.e. classification algorithms), and belong to the well known supervised learning branch. In contrast, the latter (clustering) is a unsupervised learning problem.

In this thesis, we will work with the first one. We will use different machine learning classification algorithms and techniques. But also we will suggest that a short exploration

into the clustering field might be appropiate, in order to see if our predefined schema is the most appropriate.

Next we present some of the work done in this some related areas of knowledge.

The majority of the features we deal with to classify e-Commerce items using text: titles, descriptions, and brands. The majority are represented with a textual description (except maybe some images, or quantities). In any case, our work is based principally on transforming text attributes into features that can be used for classification.

Specifically, word embeddings have gained some attention since they were introduced by Mikolov et al. ([MSC+13]) in 2013. This technique finds a map for each word to a k-dimensional real valued vector. They are typically learned in a totally unsupervised manner. The process for achieving this, includes learning which words appear surrounded by which other words. This is done in order to capture a rich variety information about that word ([DFU15]). At the end we would like to see words that have a stronger semantic relation closer together. An example is shown in Figure 2.1, taken from [MSC+13]. The k-dimensional space is projected down to a 2 dimensional space with PCA. Then, it is shown, how words that share a property: being a country. And words that represent other: being a capital, are closer between them. Also, the relation between a certain country, let's say German, and its capital Berlin is captured there.

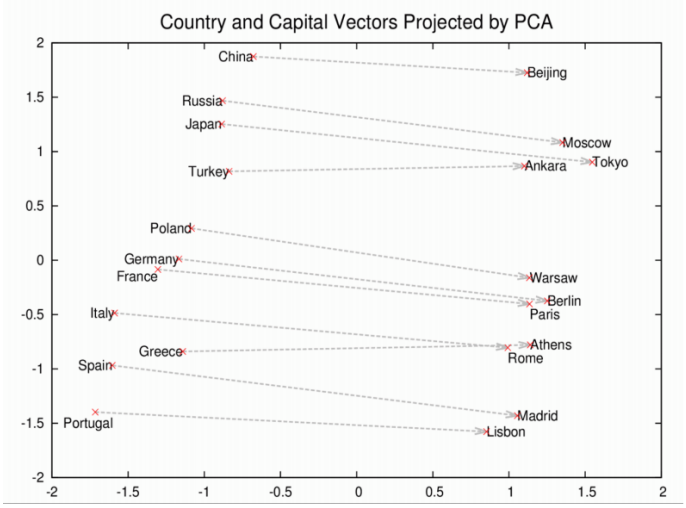


Fig. 2.1. Word2Vec PCA Projection example with countries and their capitals.

This works very well for finding synonyms for words. Also for making some nice conceptual analysis like substraction of words. A typical example for this is: *King - Man = Queen*. This type of algebraical calculations of words can be done, obtaining something that semantically would make sense.

In the field of classification, word2vec models have been used for creating the features, showing good results ([Kim14, ZWZ16]. In particular [Koz15] used this approach for classifying items from the Yahoo Shopping platform. It seems promising to take this

approach, since texts products with similar titles should, arguably, belong to the same category.

Classifying products into a hierarchy has been studied in many recent works. Most of them, favor the use of the hierarchy to ease the classification process. This is what some refer to as the "gates-and-experts" ([SRSS11]) methods, or the "coarse-and-grained" classification ([SRS12]). Basically, the strategy is based on classifying the top level categories (i.e. the coarse level, or the gates) first, and then into the lower level categories (i.e grained level or the experts). In contrast, the flatten strategy simply considers all the categories, as if the underlying taxonomy wasn't there, and tries to classify the items into them directly.

One of the advantages that were found in some works ([SRS12] is that different algorithms can be used, depending on the level we are trying to classify to. For example, using a more general model for distinguishing between first level categories, and a more specific one for the lower levels. However, some works also employ the same algorithms (probably with different hyperparameters) in both levels ([DC00, SRSS11, CW13]). Independently of the approach (hierarchical or flatten) or regardless of the combination or not of algorithms, different common classification algorithms have been used. The most populars are Naive Bayes ([KS97]), k-Nearest Neighbors (KNN) ([YZK03], [SRS12]), Neural Networks ([OM07]), but the most used is Support Vector Machines (SVM) ([DC00, YZK03, LYW+05, SRS12], [CJS11]).

The main problem of using SVM is the high training time ([SRS12]) because it requires training many binary classifiers.

Another advantage of using an hierarchical approach is that we can use different features, depending on the granularity of our task. Dumais et al. in [DC00], explain this in a very illustrative way. They say, imagine we would like for example, to discriminate between the six categories placed in the leaves of the hierarchy in Figure 2.2. In a non-hierarchical way, a word like "computer" is not very discriminative: how could we distinguish between the three computer related categories? However, if we use the hierarchical model, we would be very discriminative in the first level. And for the second level classification, we would use some more specialized words. This way we use less amount of features per level. In contrast to maybe using all of the features for the whole classification. They say that "category specific features should improve accuracy".

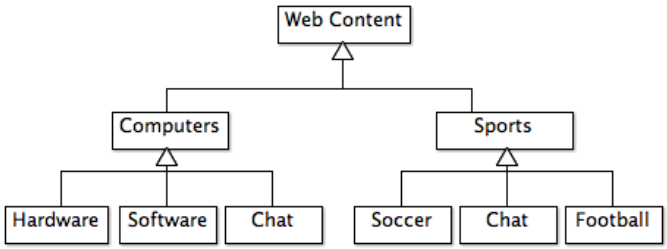


Fig. 2.2. Web Content Hierarchy Example

In the same way, Shen et al. in [SRSS11] improved a classifier on eBay items, by focusing in one of the branches (during the grained level classification phase). They showed how carefully designed domain-specific features, can improve the accuracy of the experts (or grained classification). For this, they explore the data in only one subtree (Electronics), domain knowledge and linguistic hints in order to achieve the improvement. This kind of enhancement is only possible when we take the hierarchical approach: As we are splitting the problem into many small ones, that we can tune at our disposal.

Dumais et al. ([DC00]) used mutual information between terms and categories, in order to create a binary vector that represents the existence or not of a word in a category. This vector was then used to feed the classifier.

Some other ways were implemented in order to create the features for the classifier, include latent group discovery. It is a very common problem the lack of objectivity (and even precision in the meaning of the categories) when we work in the domain of online shops. Latent group discovery introduces a way of solving this ([SRS12]). In particular, Shen et al. proposed creating a new graph of categories, using the confusion probability between every two, and selecting those edges (i.e. probability) that were higher than a threshold. With this, dense subgraphs were calculated. Every subgraph corresponds to a latent group. Latent Dirichlet Allocation is use for this matter too [CJS11, Koz15].

The majority of the studies use titles or descriptions as part of their features, because those are the most common, and informative fields that products have in general ([Koz15, LYW$^+$05, KS97]). However, Hsiang-Fu Yu et al. in [fYhHA$^+$] focused on the problem of classifying a product's title into categories. They stated that there is a difference between classifying text and classifying titles in particular. Titles are different from other text features, because they normally are short and contain (hopefully) the "key words" that describe the product. Based on this, they didn't remove any stop-words or performed stemming in order to improve performance. In general, for small sentences, arguably, doing this preprocessing steps is not useful. In our case, we have different text fields, where the longest is the description of a product (in average they contain 20 words).

In general, studies were made with similar data that included the *battle* between using bigrams or unigrams. In [Koz15], unigrams performed better than using both (unigram+bigram). Probably overfitting problems might come from this combination. In [fYhHA$^+$] both were proved to show good results.

The principal problem of the hierarchical approach is that errors at the first level are not recoverable.

For evaluating the results, the majority of the reviewed works use F1-score. Because we are in a multiclassification context, accuracy is not a good measure to use. This is true especially when we are dealing with uneven distributions of the classes. A weighted error measure should be employed in general. However, in [fYhHA$^+$] different types of errors in this setting are determined. For example: (1) When an instance is wrongly labeled. This refers to training data being wrongly labeled, maybe in a more general way (i.e. exists a category that is more specific or appropriate for the product). (2) The instance can be considered in more than one class. Handling this is a multilabeling problem, which we are

not trying to solve. We only want one category. (3) The instance is wrongly classified. The classifier made an error, basically. However, they don't propose any solution for differentiating automatically between these types of errors. In relation to this, Sun et al. in [SL01] proposed a way of weighting errors. They calculate similarities between categories to consider the degree of misclassification when measuring the performance of a classifier. It is not the same classifying a computer as a tablet, than classifying a computer as a toothbrush.

This is a very wide area of study in general.The selection and modeling of features play a very important role in having a good performance. However, it seems that considering using a hierarchy (independently of the existing underlying one, or one created on our own i.e. with latent topics) is the way to go.

In a parallel aspect, we need to deal with training times, testing times, and memory space (those models need to be somewhere). More than that, we expect our data to scale fast: online shopping is increasingly more common everyday: for simplicity users are opting to buy online what they need. An article in Fortune from last year ([Far16]), shows the results of a survey made in USA that state that traditional department chains, such as Macys, had a decrease of shoppers last year. On the other hand, Amazon rose 15.8% its purchases in the last year. Online shopping is increasing its popularity, in part thanks to smartphone users who make their purchases through their device.

This has a direct implication. Every day, more items are added to websites so they can be sold. Moreover, the quantity of online stores is also increasing. For pages like idealo.com, whose goal is to compare prices from a diverse selection of online retailers, this is also something that needs to be thought through. On the other hand, the quantity of categories in the domain of idealo.de will (probably) increase too. Of course in a slower pace than the products. It is important to come up with a scalable solution that performs well.

Lately large-data processing engines like Spark, Flink and Hadoop are becoming very popular. They ease the implementation of distributed and parallel solutions. They add a new level of abstraction, giving the user an easier (but still powerful) tool for scalable solutions. These frameworks enable scaling by providing "scaling-out" methodology [Par15].

They provide some benefits that include fault-tolerance, scalability (as already mentioned), distributed processing , and parallelism. Of course, for this we need machine learning algorithms that are presented for scaling. Some studies ([YZK03] having made about the complexities of scaling when having an hierarchical classification task. Moreover, some algorithm's implementations (such as kNN) have been already well studied and explained in [Par15], using already Spark and Hadoop for solving the problem.

Our goal is to explore solutions for classifying online products using a hierarchical taxonomy. But while doing so, we should always keep in mind that our solution needs to scale: the online world is changing, and we need to evolve with it.

3. The Problem

> "In general, we're least aware of what our minds do best. We're more aware of simple processes that don't work well than of complex ones that work flawlessly.".
>
> — Marvin Minsky *The Society of Mind, 1986*

"How would you conceptualize what a *table* is?". A professor once asked this, at a university course. At first glance, everyone thought she was joking. However, after a couple of minutes of reasoning, discussing, and debating mostly weak concepts, no one could make up a satisfying definition for something as simple as a table. It is not as easy as one might think to identify features that describe any table. Sometimes we take for granted the rich and powerful processes that our minds are capable of. Nevertheless, when we try to explain something we just can't.

Categorization is a cognitive process that we learn to do since we are children, to make sense of the world. Incrementally, we start building up concepts around objects. The study of it have been done in different subject areas including: linguistics, psychology, philosophy, artificial intelligence and information technology.

It was Plato who introduced the -classic- concept of categorization. According to him, categorization is the action of grouping objects based on their similar properties. After him, Aristotle established that a category is defined by a set of properties that are shared by the members of it. According to the classical view of categorization a category should be clearly defined, mutually exclusive and collectively exhaustive (see [LJ99]). This implies that any item in the classification universe has to belong unequivocally to one, and only one, of the existent categories. In the table example, this was a problem. We tried to find necessary and sufficient features that all tables have and that is difficult to do. Tables can be very different from each other.

However, in the 1970s a new theory was presented: the prototype theory by Eleanor Rosch (see [Ros75]). She proposed that categorization is the process of grouping things based on prototypes (or typical example for a certain category). From this, an item X can be more similar to the prototype of a category than an item Y, although both X and Y belong to the same category as the prototype. This means that category members resemble each other, but they don't represent their category equally well: a robin represents a *bird* category better than a penguin. A degree of category membership exists. Prototypes should be a clear example of the category they represent. They are supposed to combine all typical features. More than that, prototypes might change from person to person, depending on her/his cultural background, experiences, personalities, even context. Subjectivity is all around us.

Classification as we see, is not an easy task. The best way to classify our surroundings may depend on many facts. And once we find a good way (if so), it won't necessarily persist in time, or work for every situation. This has been a very complex field of study for many years, and still is. We are still having problems understanding our brain and its natural mechanisms.

3.1. What are we trying to find?: Zooming in

We classify everything by nature: animals, plants, objects, feelings, etc. Online (and offline) shops aren't the exception to this. Identifying the right category for any item in a shop, we could think about the prototypical categorization theory. In our brains, a sofa is faster classified as furniture than a lamp. Sofas better represent the furniture category. However, as mentioned before, this could be completely subjective.

The e-commerce company idealo, offers a price comparison tool for online shops. It contains more than 330 million items sold by shops like Amazon, eBay and Zalando. Retailers have their own categorization scheme for the products they offer. However, idealo defined its own category schema to include all. This schema consists of a taxonomy that includes all the categories in a hierarchical way. That means that more general or generic categories (such as fashion or computer and hardware) are in the upper levels of the category hierarchy. In the same way, more specific or grained definitions (such as costumes or USB sticks), are in the lower levels of the hierarchy. This way users have the option to find the item they are looking for through their categories to compare prices.

It is important to emphasize that everything, from the creation of the hierarchy of categories to the classification of products is biased. There is a team in charge of taking these decisions. Therefore, automating tasks is not only a matter of mathematical rules, but also includes understanding the context. We need to explore the data and have a good understanding of the distribution and properties of the underlying data. Classifying products in the hierarchy is a problem that could be difficult to solve even by humans whom, as we saw in last section, do this in a natural way since an early age.

In this thesis, we want to compare different ways to classify items into the already defined category hierarchical schema of idealo. Specifically, we would like to:

- Analyze idealo categories. Their organization, distribution, and possible problems (Chapter 4).
- Explore the data (i.e. the items or products). Their properties, distribution and features (Chapter 4).
- Based on the data structure, select and transform the products features in order to classify them (Chapter 5).
- Examine the literature behind varying algorithms (e.g., supervised learning algorithms, such as Logistic Regression or kNN) and structural techniques (e.g., consider the underlying hierarchical taxonomy), and then select appropriate approaches for implementation (Chapters 2 and 5).

- Conduct experiments using the implemented approaches. Evaluate their performance using well known metrics, such as the F1 score and then compare the obtained results against different baselines (Chapter 6).

The amount of data will grow in time (and not in a linear way). Therefore, the solution should be scalable w.r.t the amount of products.

The accuracy should be measured considering that we are classifying into multiple labels (i.e. we are dealing with multiclassification, and not binary classification). Knowing the distribution of the data, will give us a clue about if we should or shouldn't take into account that our classes are unbalanced. Even more, we need to keep in mind typical classification problems in machine learning like: over- and underfitting and the presence of outliers.

A solution that considers all of the mentioned criteria for classifying products accurately into a hierarchical taxonomy is the main purpose of this thesis. Qualitatively, the proposed solution also needs to be:

- consistent, i.e., if the environment does not change, the products should always be classified into the same category (deterministically),
- efficient, i.e., the entire task including both training and classification should take a reasonable period of time and resources,
- precise and accurate,
- scalable, as the number of products and/or categories increases,
- adaptable, to changes in the taxonomy which should not drastically affect the classifier.

3.2. Where does the Classifier fit?: Zooming out

The route a product follows since it is retrieved from an online shop till it is showed to the final user in the website could be a long way.

After a product is retrieved, it's determined whether it's a new product or something that has been seen before by a trusted classification process, e.g., through comparison of unique industry-wide identifiers such as the GTIN[1]. The products that move on to the next stage (i.e. haven't been seen yet), pass through a set of predefined internal methods (such as rule based classification). This reduces the amount of products that need to be labeled later by the classifier. The next step is the classifier. For the classifier's decision to be considered, the prediction should have a probability equal or higher to (an arbitrary) 80%.

[1] Global Trade Item Number (GTIN). "GTIN describes a family of GS1 (EAN.UCC) global data structures that employ 14 digits and can be encoded into various types of data carriers. Currently, GTIN is used exclusively within bar codes, but it could also be used in other data carriers such as radio frequency identification (RFID)". Source: http://www.gtin.info/

For products that aren't classified in any of this stages a variety of heuristics are tried, e.g., scanning the original shop category for keywords. Normally, about 0.2% of the total of products remain unclassified, these are manually classified by domain experts. A pipeline of the products' process is shown in Figure 3.1.

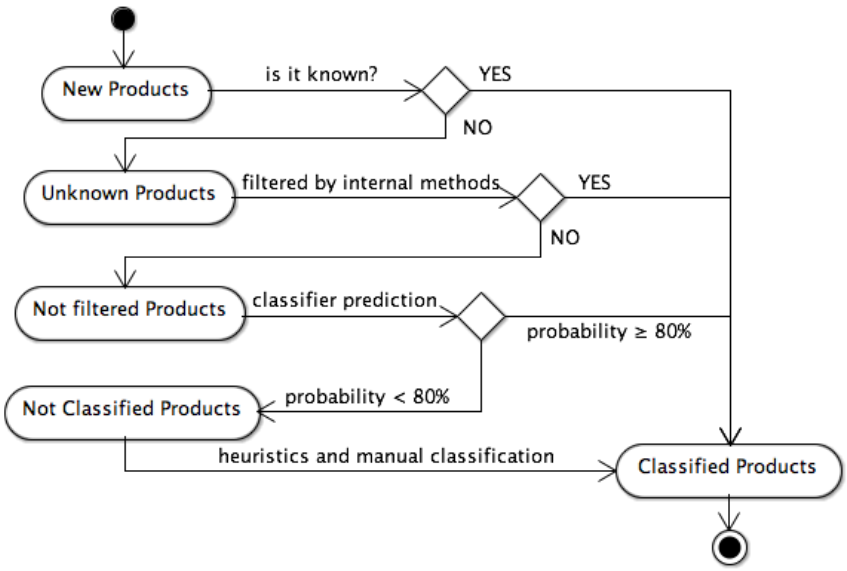


Fig. 3.1. Product's Activity Diagram

Our scope only includes the classifier phase of the products' process of categorization. The classifier is a supervised learning algorithm that assigns an unique label (category) to a given data point (product). That is $categoryOf : P \rightarrow C$, where P is the set of products, and C is the set of categories (see next chapter's section: 4.2 for more information). From now on, we will assume we are only inside the classifier context (i.e. products and categories are those that the classifier knows, and not the whole idealo's data).

Next chapter we will talk about the properties and distribution of the data, as well as some problems that need to be considered. After that, in Chapter 5 the design, decisions and implemented solutions will be presented. Finally, in Chapter 6 the results are presented, along with the respective discussions.

4. The Data

> "It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts".
>
> — Sir Arthur Conan Doyle, *A Scandal in Bohemia*

A good story can't be written without having clear which are the characters that will star it. The cast of the story that we will relate here is data. And it deserves a whole chapter to understand it well.

We will describe the facts about the involved parties (basically products and categories). This will allow us to understand better our problem, make appropriate assumptions, and derive into better conclusions

How the data is structured, distributed, and related is the main topic of this chapter.

4.1. The Categories

Let us start describing the categories. They will be the classes we want to classify into. Idealo, and in general online -and offline- shops, arrange their products by categories.

Idealo, like online shops such as Amazon or eBay, organises its categories in a tree structure (i.e. a hierarchical approach). Nodes and leaves of the tree represent categories that products can be classified into.

The already existing category taxonomy is designed and maintained by humans. Upper levels of the tree are related to broader definitions (e.g. Telecommunications, Computer&Hardware, Fashion&Accessories...), and leaves or categories in deeper levels represents more specific categories (e.g, Red Wines, Bank Benches, Footballs...).

Let us explain this with an illustrative example. *Pictonary* is a famous board game. When we find it on idealo's website, this is what we see (see Figure 4.1)

Fig. 4.1. Path of categories for a product (e.g. Pictionary)

As we can see in the category path of Figure 4.1, *Pictonary* -a *product*- is classified as a *Gesellschaftsspiel* (board game). It's worth mentioning, that we can see already in

the interface of the website that this category has *Spielwaren* (toys) as its direct parent in the category tree. The latter is a child of the *Spielzeug&Games* (toys and games), which is a first level category (*Startseite* is the root, i.e. level 0).

Products can belong to any category in the tree (i.e. classification is not exclusive among leaf categories, but also node categories). Nevertheless, not all categories are *target categories*: some nodes are just conceptual (i.e. they don't have assigned any product to it). Note that leaf categories are always target categories too. The opposite is not true. We call target category those categories that a product can be classify into.

When a product is assigned to one target-category, it means that this product -implicitly- belongs also to all the categories that form the path from this target category up to the root of the tree. Figure 4.2 shows a general -not precise- overview of the structure of the category tree. The category path of our *Pictonary* example can be seen there. The game was assigned to the *Gesellschaftsspiel* target category (which is a leaf category too). Implicitly it belongs to *Spielwaren* and *Spielzeug&Games* as well.

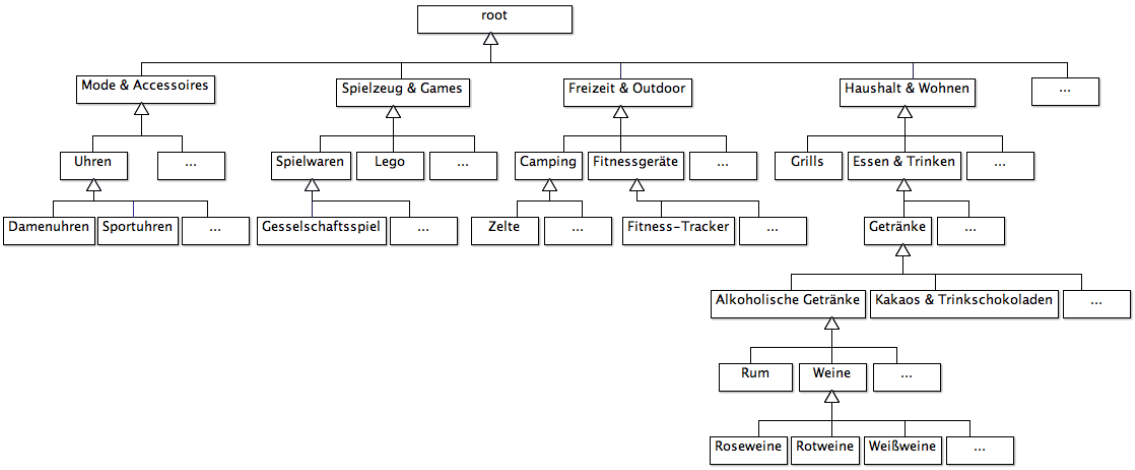


Fig. 4.2. Category Tree

Now that we have understood intuitively this concepts, let us build a mathematical definition around it.

Let C be a set containing all categories, $C = c_1, c_2, ..., c_n$, $C_T = \{c_1, c_2, ..., c_t\}$ the set of target categories, where $C_T \subseteq C$, and P the set of products $P = p_1, p_2, ..., p_k$. Let $R = r_1, r_2..., r_t$ be a set that contains the paths of all categories in C_T to the root. That is $\forall_{r_c \in R} : r_c \subseteq C$. r_c is a set that contains all the categories that belong to the path from the root of the tree to the category $c \in C_T$. When a product p is classified into c ($categoryOf(p) = c$), implicitly p belongs to all categories in the path r_c too (i.e. the path from the root to c).

We say that category $c_j \prec c_i$ if and only if, c_i is a child (direct or not) from c_j in the tree (i.e. the tree that contains c_j as root, contains c_i as a node or leaf). We say then, that c_j precedes c_i. We can say that a path r_c is totally ordered under $\prec$. Furthermore,

from $c_j \prec c_i$ we can imply that $r_{c_j} \subseteq r_{c_i}$. Also, if c_j is direct parent of c_i in a tree, $r_{c_j} + c_i \equiv r_{c_i}$.

Then, the classifier $categoryOf$ is a function $categoryOf : P \to C_T$. It selects the target category that is located in the deepest level of the tree, that matches better with a product.

$$
\begin{aligned}
(\forall_{p \in P} : & \exists_{c \in C_T} : categoryOf(p) = c \wedge && \textit{product } p \textit{ is assigned to category } c \\
& \exists_{r_c \in R} : c \in r_c \wedge \forall_{a \in r_c - \{c\}} : a \prec c \wedge && r_c \textit{ contains the path from } c \textit{ to the root} \\
& \neg(\exists_{x \in C_T} : categoryOf(p) = x \wedge x \neq c)) && c \textit{ is unique}
\end{aligned}
$$

That is, for all products exists one and only one category, to which a product belongs, and this category doesn't have any (direct or indirect) child to which the product could belong.

In the next subsections, we will talk about the distribution of the data. All calculations are based on a data set containing 100k products (i.e. 250MB). This data set was selected randomly, therefore it is an approximation of the real data distribution.

The Taxonomy

The taxonomy is a very wide tree containing 6 levels. It presents a very uneven distribution of the categories among its levels (see Table 4.1). The third level not only contains more categories, but more target-categories and leaves from all the levels. Because of the great amount of leaves in this level, it should come as no surprise that the next level (i.e. level 4) contains fewer categories. The first level of the tree contains 13 categories. Non of them are target categories. Table 4.2 shows the average branching factor per level (Avg B.F.). Also the average out-degree, the minimum and maximum amount of category children that the categories have in the respective level. For example, a_i is the average branching factor for level i, which was calculated using the number of categories that levels i and $i+1$ have. That is, if we say m_i is the number of categories in level i (this information can be taken from Table 4.1):

$$a_i = \frac{m_{i+1}}{(m_i - l_i)},$$

where l_i is the number of leaves that level i has. For example, in level 2 the average (or average branching factor) would be calculated as: $a_2 = \dfrac{1024}{138 - 90} = 21.3$.

The average is calculated between the out degree of node categories in the level.

We have a total of 1870 categories distributed across the 6 levels of the taxonomy. 1749 of those categories are target categories, 726 are leaves, and 1023 are target categories, but also nodes of the tree.

Finally, in Table 4.3 we find the top 5 categories with more direct children.

Level #	**X-Level Categories**	**Target Categories**		
		Total	**Leaves**	**Nodes**
1	13	0	0	0
2	138	90	90	0
3	1024	979	322	657
4	587	573	252	321
5	105	104	59	45
6	3	3	3	0
Total	1870	1749	726	1023

Table 4.1.: Distribution of Categories in the Category Tree

Level #	**Out degree of Level-X Categories**			
	Avg B.F.	**Avg.**	**Max**	**Min**
1	10.6	10.61	15	5
2	21.3	10.34	44	1
3	0.83	7.15	35	1
4	0.31	4.77	10	1
5	0.065	3	3	3

Table 4.2.: Children Categories in the Category Tree

Category	**Children**	**First Level Category**
Spielwaren	44	Spielzeug & Games
Schuhe	40	Mode & Accessoires
Küchenzubehör	35	Haushalt & Wohnen
Taschen	31	Mode & Accessoires
Sportartikel	30	Freizeit & Outdoor

Table 4.3.: Top 5 Categories with Mode Direct Children

The First Level

Since we are tackling our problem considering the hierarchical organization it has, let us take a closer look to the first level of the categories tree.

As mentioned before, there are 13 categories in the first level. Let $B = \{b_1, b_2, ..., b_{13}\}$ be the set of 13 different subtrees (or branches) of the category tree, created out of the first level categories. Every branch b_i, is represented as a set that contains the categories that exist in this subtree (including the respective first level category). These sets, fulfill the following properties:

1. The intersection of all sets of branches is empty, i.e. $\cap_{i=1}^{13} b_i = \emptyset$. Further, $(\forall\, b_i, b_j \in B : b_i \cap b_j = \emptyset)$
2. The union of all sets of branches is the set of existing categories, except for the root of the tree, i.e. $\cup_{i=1}^{13} b_i = C - \{root\}$, where C is the set with all the categories of the category tree, and $\{root\}$ is a singleton that contains only the root of the tree.

Table 4.4 summarizes the distribution of subcategories and products across the first level. It also shows the amount of target and rare categories (rare are those with less than 10 products) in the first level. Target categories in general represent $93 \pm 9\%$ of the total of categories in each level (in average is 93%). This also holds for the overall amount, where the target categories represent 94% of the total.

Category	**Categories**	**Target Cats.**	**Rare Cats.**	% **Products**
Freizeit & Outdoor	339	309	164	15,66%
Haushalt & Wohnen	308	293	156	17,09%
Heimwerken & Garten	301	285	134	8,03%
Mode & Accessoires	193	190	29	**27,49**%
Computer & Hardware	172	168	74	6,72%
Auto & Motorrad	151	146	28	13,86%
HiFi & Audio	96	84	66	0,72%
Spielzeug & Games	70	70	34	2.17%
Gesundheit	55	50	20	1,92%
Wellness & Beauty	51	43	21	1,03%
TV, Video, DVD	44	38	21	3,66%
Fotografie	39	38	22	**0,53**%
Telekommunikation	38	35	20	1,08%
Total	1857	1749	789	100 %

Table 4.4.: Distribution of Categories and products in the First Level of the Hierarchy. The highest and lowest percentage of products in the categories were highlighted. There is a great difference between them.

4.2. The Products and their features

The provided data set contains 100K products (i.e. 256.9MB), distributed over the already explored 1857 categories. From the uneven distribution of categories w.r.t. tree levels and branches, it's already expected that the products will also show an uneven distribution across levels in the tree. This is confirmed by Figure 4.3. As expected, the majority of the products can be found in level 3: it is the level with more target categories too, and more specifically leaf categories (Table 4.1). Remember that the first level does not have any target category, and therefore any product is assigned into it.

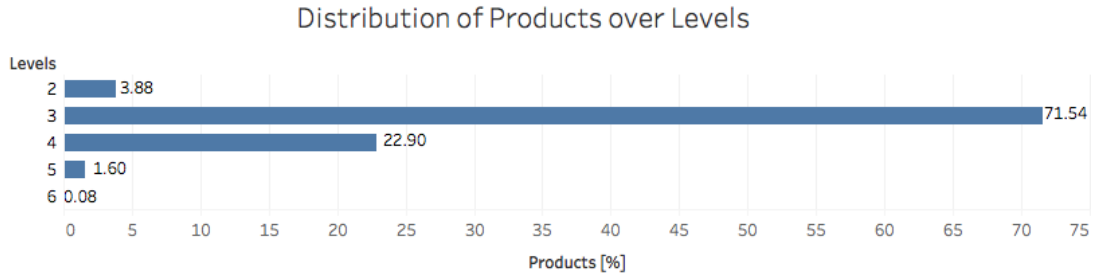


Fig. 4.3. Distribution of Products across levels of the category tree

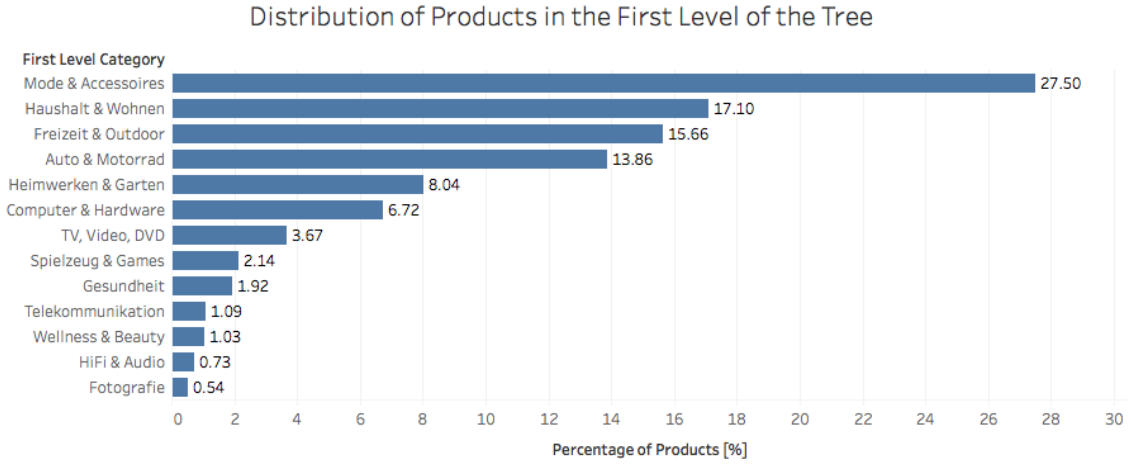


Fig. 4.4. *Mode und Accesoires* is the category with more products in its branch: 27%. It is followed by *Haushalt und Wohnen*, *Freizeit und Outdoor* and *Auto und Motorrad* which contain between 13 and 17% of the products. All of the other categories have less than 10% of the products.

More important, in Figure 4.5 it is shown how products are distributed over target categories. The highly skewed distribution is evident in the figure. There are 179 (out of 1749) categories with only 1 product. In general, many categories have a small inventory. And few categories have a lot of products.

Target Categories present a long-tail distribution: There are a few categories with many products, and a lot with a very small amount of them. This is a severely imbalanced data.

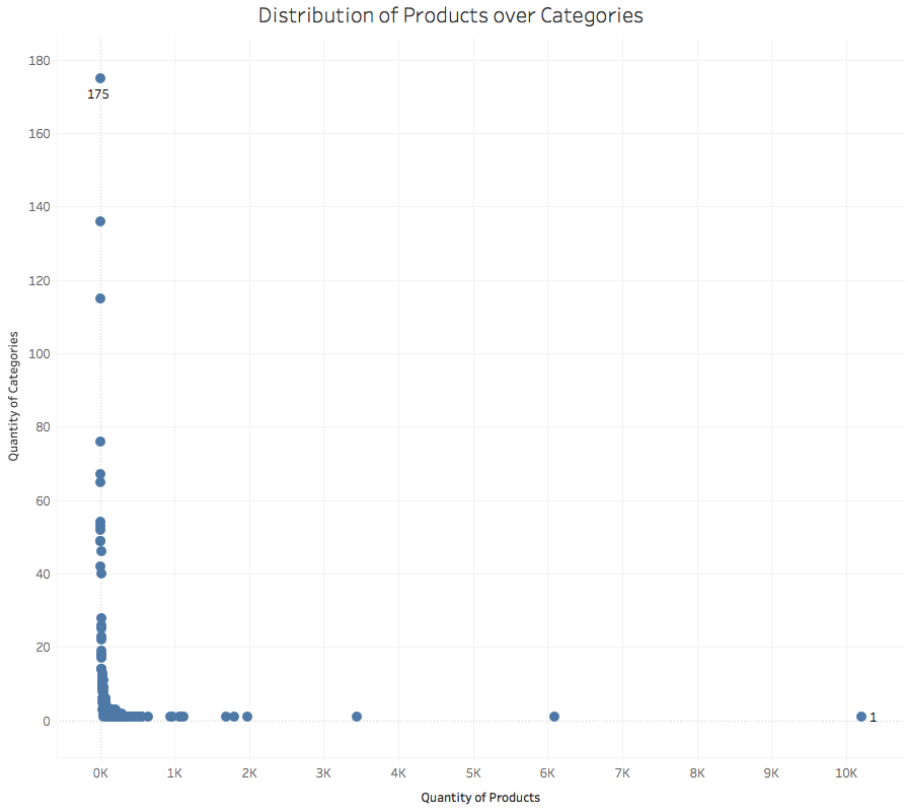


Fig. 4.5. There is a high skewness with the majority of the categories that have a small amount of products

Table 4.5 shows the top 5 categories with the more density of products assigned to them. One column was added to it with information about the branch (i.e. first level category) to which this category belongs to.

Category	**% Products**	**First Level Category**
Bücher & Zeitschriften	10.9	Freizeit & Outdoor
Handytaschen & Cover	6.5	Mode & Accessoires
Wanddekoration	3.69	Haushalt & Wohnen
Fun Shirts & Fan Shirts	2.12	Mode & Accessoires
Audio – CDs	1.93	TV, Video, DVD

Table 4.5.: Top 5 Categories used by products

As mentioned before, products are the items that we are going to classify. They have several features that can be used for this purpose.

A subset of the text features was chosen, under the assumption that they are more clarifying and information rich. This was determined empirically, by looking at the data directly.

- Title: This feature is prevalent in (almost) all products. It's the title of a product that is retrieved from the retailers. It's a text field that on average has 11 words.
- Description: This text field describes what the product is.
- Brand: Manufacturer provided by the shop.
- Product Search Text: It can be basically treated as a title. Sometimes it contains more information.
- Shop Category Name: Name of the category in the shop the product comes from.
- Category Path: Category path of the product in the original shop.

In Figure 4.6 we can see in which proportion this features are present in products (i.e. they don't consist of an empty field). Notice that not always all features exist in all products.

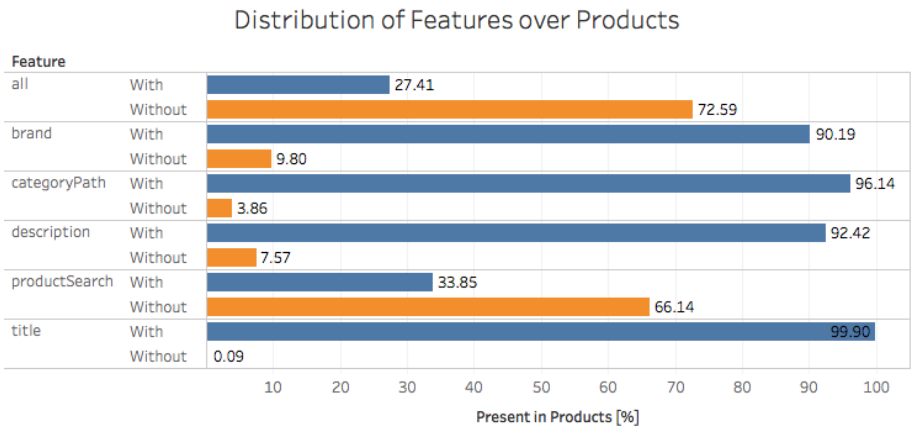


Fig. 4.6. Existence of Features in Small Dataset

In average, titles have 7 words. Category paths normally have between 1 and 10 words. The others are very heterogeneous.

4.3. Problems with the Data

In any task where great quantities of data is processed to predict something, the quality of the data plays a fundamental role. This section will show some of the first problems found while analysing the data.

Orthography

As already said, the used data comes from several sources. This translates into having not only a diversity of schemes (i.e. the one that each online shop chose), but also it is -very- possible to find different errors made while inserting the data. Including of course human-related ones, such as orthographic errors (on titles, descriptions of the data, etc).

Alphabet

Even though all the data is supposed to be in German, a couple of cases were found where products contained text with Asiatic alphabets instead of the Latin alphabet (used by the German language). These cases were simply ignored and not used for any purpose.

Also, small considerations where taken into account considering some particularities of the German language (more about it in next Chapter: 5.3).

Unbalanced

Having balanced data in markets (no matter if real life or online) is something uncommon. Think about the distribution of items in a supermarket, or an electronic store: there is a big chance, you won't find the same diversity you find in the marmalade section, as in the milk section. Or the quantity of different models and brands of Laptops (even tho is very high), probably won't be higher than the diversity of CDs and DVDs. It is a natural thing. Life is not balanced.

This is normally not a problem: We don't need 100 types of milks. But is nice to have a broad selection of marmalade. Nevertheless, in the world of machine learning and classification this could be source of a problem.

As you may already infer from last sections and Table 4.4, the data used for this thesis is unbalanced. This means we should select a loss function that considers this. We will contemplate this again in the results section (Chapter 6).

Just misclassification

The whole data comes from real idealo's offers that passed a classification process (explained in Section 3.2). This doesn't necessarily imply that an accurate category label was assigned to the product. Actually, a lot of label errors (mislabeling errors) in the data were found.

We can differentiate between three types of misclassification.

1. Severe: This is the easiest type of misclassification that can be spotted. It refers to those errors that are obviously that. Errors. For example, seeing a tent in the *MP3-Player* category.

2. Subtle: This type of misclassification, requires more work in order to identify it. This could happen when we find the tent in the middle of its way to its *right category*. There is a *Zelte* (Tents) category, which belongs to the *Camping* category.

If we find a Jack Wolfskin's Tent classified into the *Camping* category, instead of into *Zelte*, we found a subtle misclassification.

3. Ambiguous: Should a sport smart watch be classified as sport watch or into smart watches (both categories exist). Some items simply can belong to more than one category. More about this in next subsections.

Figure 4.7 shows an example of a severe misclassification. The cause of this is unknown: it might be because of the wrongly spelled word *Stiefelette*, but the reason could have been something else. It's a fact that a boot certainly shouldn't be in the *Elektrotechnik* category.

Fig. 4.7. *Stiefelette* misclassified

Subjective Classification

There are some items that, by nature, could belong to more than one category. Sadly, this is not allowed to happen: As we mentioned in past chapters, we are not trying to solve a multilabeling problem.

Let us say we want to classify an item that could be correctly assigned into category X or Y. If we take the training data (where our item is classified into X) and our classifier says should go into Y, we will mark that as an error.

These type of situations are very hard to detect, as the original data has only one label. Further, when a classification error occurs, it is complicated to distinguish in general if we're facing an instance of this situation or a severe error.

Hierarchical Problems

If you were told to classify Tennis Shoes, where would you put them? Into which first level category? Probably you would choose *Mode & Accessories*, as this is a *fashion* item: something you can wear. Nevertheless, you could opt too for choosing *Freizeit & Outdoors* category, as you basically go to play tennis in your free time.

Actually, *Tennisschuhe* is a category that belongs to this path: *Mode & Accesoires* $\prec$ *Schue* $\prec$ *Tennisschuhe*. It seems nice.

Now, you need some shorts and tshirt to match with your new tennis shoes. Where do you look for it? *Freizeit & Outdoors*? or *Mode & Accessories* again? If you chose *Mode & Accesories*, you were wrong, because it belongs to *Freizeit & Outdoors*: *Freizeit & Outdoors* $\prec$ Sportbekleidung $\prec$ Tennisbekleidung.

This seems like a subjective way of choosing where to place a category in the tree. The problem is that tennis shoes and tennis outfit, can be put in either of both first-level categories[1]. This could be a problem we can face, if we are trying to classify general things at the first instance, and more grained items next. Seems like we will be already classifying grained items, during the first level classification phase.

[1]This won't be a problem for the user of the website, because of the way it is presented to her/him -it is solved in the frontend-.

5. Assembling the Pieces

> "You can't connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future."
>
> — Steve Jobs

The reader at this point should be intrigued: She/he should know by now about the related theory and experiments (Chapter 2), our problem (Chapter 3) and the particularities of our data (Chapter 4). But the possible solutions, how well they work and the conclusions we can derive are still unknown. In this chapter, we will elucidate the possible solutions. Followed by this, in Chapter 6 the results will be shown along with some explanations for them. Finally, we'll close with all the conclusions and some recommendations for the future. Let us dive in.

5.1. All The Pieces Together

Two phases can be distinguished in the whole classification process, namely offline classification and online classification. The former refers to the whole process needed for building the model, including training, validation and also accuracy measures (see Figure 5.1). Once we have a well trained model, we want to use it to predict unknown information such as categories. That's the second phase (see Figure 5.2).

For now, we will talk about the first phase. The second one will be approached during the final experiments in Chapter 6. But in general, online classification refers to what happens in production, when the labels are really unknown.

The offline classification phase includes four marked steps that can be seen as a pipeline (see Figure 5.1). First, we need to clean the data. After that, we should select which features are better for our task. Then we need to model the features so they are "input material" for the classifier. And finally, we train and test our classifier. Later in this Chapter we will talk about each one of these steps, modifications and possibilities.

As we already mentioned, approaches to the classification that take the underlying hierarchical structure of our classes (i.e., categories) into account have shown better results in the past ([SRS12, SRSS11, DC00]) with similar settings. This means that we should first focus on finding the best first level category for our products, and then we should classify them into the corresponding category that belong to the selected branch (see online classification pipeline as reference, i.e. Figure 5.2). Then, instead of executing a global classification among all the categories of our categories' universe

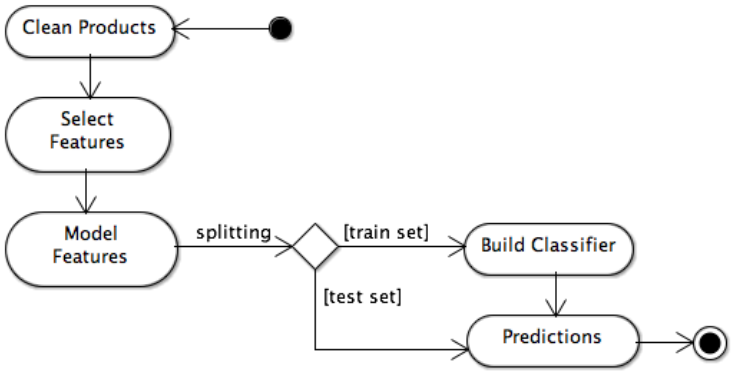


Fig. 5.1. Offline Classification Process

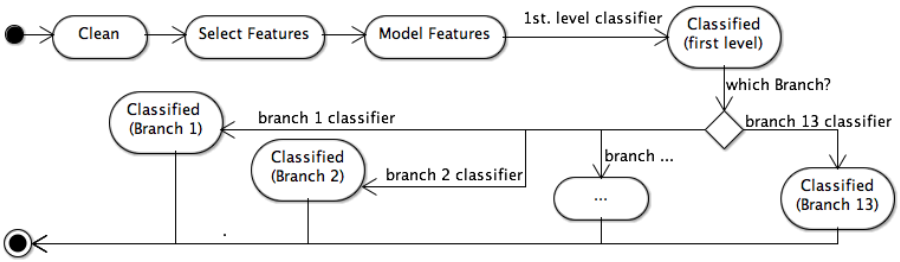


Fig. 5.2. Online Classification Process

(i.e. flat classification strategy), we will carry out two *flat classification* procedures: First between the 13 categories that represent the first level, and then among the X_c categories that belong to the subtree formed by the first selected category c.

Categories that belong to the same branch are considered to be more similar between them, than the ones belonging to different branches. That being said, it is natural then to assume that problems related to subjectivity will be more common in our grained classification phase. These types of problems occur for example when we have a men's t-shirt with a (subjectively) funny message, and we (as humans) don't know whether to classify it as a men's t-shirt (*Herren-Shirts*'s category) or as a *Funshirts und Fanshirts*'s category. It should be equally correct to classify it as either.

Our first level classifier should be then a very powerful one (i.e. very accurate), because the fine level one might not: these types of errors are hard to detect without any human checks.

Notice that we could *simply*[1] find those categories whose products are related in a way, and treat them as *similar*. The problem is these relationships are not the same, nor symmetrical among all products. For example: *Funshirts und Fanshirts* category could share a lot of its products with *Herren-Shirts*. But it could also share them with

[1] It's not that simple. It would require some clustering techniques. And then take some considerations, as we wouldn't like to change the categories' taxonomy.

Damen-Shirts. Using the given categories, how can we know if a product was wrongly classified into *Funshirts und Fanshirts* should be in a category of shirts for men or for women? Maybe it's unisex?

However, if an item is classified as *Damen-Shirts*, when its real label says it belongs to *Funshirts und Fanshirts*, we could check for our classifier's second option for classifying it, and test if it is actually *Funshirts und Fanshirts*. In other words, this would be like asking the classifier: "Ok, you told me it is a shirt for women. But what would you say if you were to propose two categories (or even three) for it, which would be your next guess?" Shen et al. in [SRS12] showed how the precision of their classification improved by 8 and 14 percentage points, when they returned the top 2 and top 5 categories respectively, with respect to the top 1 (i.e. first prediction). Later in Chapter 6 we do these kinds of experiments (see Section 6.4).

Choosing the Right Algorithm

When we talk about choosing the right algorithm for our task, we should consider the type of input and output we have and want, respectively. In any of the two phases (first and grained level classification) we have the same type of input: labeled input. The output in both should be a category (i.e. a class). We then need to explore supervised machine learning algorithms suited for classification. Moreover, we need to explore multiclass classification algorithms.

This could be approached in different ways. We could just select an algorithm whose goal is multiclass classification. Another solution could be to consider one of the algorithms that are designed for binary classification and implement together with it, one multiclassification technique, such as one-versus-all, all-versus-all or error-correcting codes [HPK11].

Before we jump into our analysis concerning which classification algorithms we used, we want to summarize a couple of things that we should consider when we select our algorithms.

Two important concepts we have to keep in mind are:

- Underfitting: This happens when the created model is unable to capture how the data and its classes are related (i.e. the input and the output). This can easily be checked by running the model with the training data. If it performs poorly, we are underfitting. The reason for underfitting could be because the features (input) aren't expressive enough, or maybe the selected model is too simple for the given data. This is related with having a high bias and a low variance (see Figure 5.3)

- Overfitting: This happens when the model is basically memorizing the data. If a model performs well in the training data, but poorly on the testing data we are overfitting our model. This is related with having high variance, and low bias (see Figure 5.3).

Then, it is necessary to find a trade-off between bias and variance: Not having our model too complex to overfit, but neither too simple for it to underfit ([FR]).

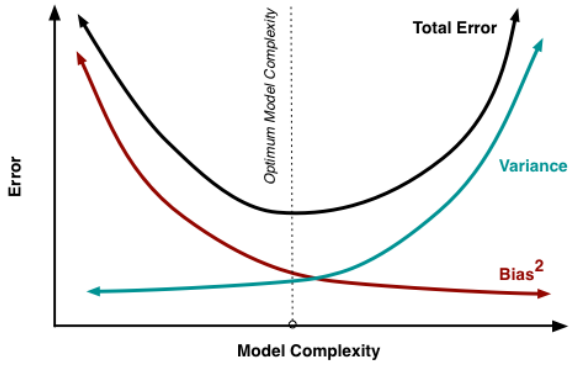


Fig. 5.3. Bias Variance Tradeoff. (Source: [FR])

If we are facing underfitting problems, we might need to add more features that can represent better the label for an item. On the other hand, if we are overfitting we should consider using fewer features and adding some regularization that forces (or penalizes) parameters for having large values.

The number of training examples is also important. Specifically for us, where this could be critical at the grained classification level. As we saw in Chapter 4, our branches are not well balanced (neither from the categories perspective, nor from the quantity of products point of view). Therefore, we need to always keep in mind how many samples are we using to train and test our models. It is also important to always have a representative selection for every category in the training set.

Problem Generalization

Independent of the algorithm we use, when we solve a classification problem we are basically trying to to find the posterior probability distribution $p(y|x)$ for every y. Here $y \in L$, and $L = y_1, y_2, y_3, ...$ are the possible classes (e.g. categories) for that classifier. We should assign the y_i that corresponds to the largest posterior probability. Let x be the feature vector representing a product, and $\hat{y}$ the predicted label for it, then we want to calculate:

$$\hat{y} = argmax_{y \in L} \{p(y|x)\} = argmax_{y \in L} \{p(x|y)\, p(y)\}, \tag{5.1}$$

where $p(x|y)$ is the observed model and $p(y)$ is the prior distribution on the discrete labeled.

5.2. Working Environment

For the implementation of this work, Scala 2.11.8 was used with Apache Spark 2.0.0 and Java Virtual Machine Version 1.8. Unit tests were done with jUnit 4.12. Everything

was programmed using IntelliJ IDEA IDE version 2016. With Maven 3.3 as the package manager.

More information about the test environment is discussed in Section 6.2.

Next we will explain about the tools we used, a bit more.

Apache Spark

In order to develop our solution, we used Apache Spark. However, in data science or data analytics problems, this is still not one of the favorites tools out there: Normally for these types of tasks, Python's libraries like scikit-learn are more popular. So, why make it more complicated with Apache Spark?

Apache Spark eases the development of tasks that are thought to be run in a distributed and parallel way. Algorithms implemented for example in scikit-learn (in Python), aren't specially built for use in a distributed environment. Spark offers an easy way to combine machine learning algorithms or data engineering tools with big amounts of data that may not fit into memory. Another reason to use Spark over other frameworks or libraries, is its speed. Scaling and running algorithms on a cluster is very straightforward. Parallelizing your tasks, may result in having a solution calculated faster. In-memory computations also lower latencies (in the network) significantly, with respect to the use of tools like Hadoop (which writes the partial results to disk after every operation). However, we need to consider that not all tasks can be run in parallel, because some just need time.

Spark is an engine for large-scale processing ([Spaa]). It is a framework that supports MapReduce framework's functionality (such as Hadoop) through its API. M. Parsian in his book [Par15] mentions some benefits we should expect by using Spark including: reliability (i.e. fault-tolerance: if a node goes down, results won't be lost), scalability (i.e. it supports large clusters of servers), distributed processing (for input data and processes), parallelism (on cluster of nodes).

Most machine learning algorithms involve iterations (i.e. for estimating parameters until convergence or arriving to a tolerance). The benefit of using Spark for their execution is then clear: If we have every iteration running in parallel with in-memory computations (i.e. without I/O every sub-result to disk), it should be faster than sequentially running the same task (However, notice that the result should be the same).

When we run a Spark job, its main program is executed on the specified driver. The driver program is the one in charge of scheduling tasks on the cluster. It uses a cluster manager (such as YARN or MESOS) if set and if not, it uses Spark's own standalone cluster manager. The cluster manager is in charge of the allocation of the resources, such as memory and nodes across applications. Every worker node executes in parallel the tasks that the driver sent to them. The tasks are also run with multiple threads in every executor ([Spab]).

An overview of Spark's component interaction can be seen in Figure 5.4 taken from [Spab].

There are some variables we can set when running a Spark job that will determine the program's execution on the cluster. Three important ones that we want to consider

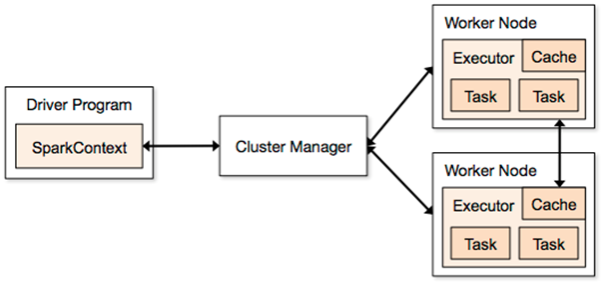


Fig. 5.4. Spark's Architecture (Source: [Spab])

are: (1) the number of executors (per node), (2) the amount of memory per executor, and (3) the number of cores. This can be set dynamically at the moment we run the job with the flags *–num-executors*, *–executor-memory* and *–executor-cores*, respectively. However, the first one (number of executors) can only be set this way, when using a Yarn cluster manager (which is not our case). They of course have their limit set by the underlying hardware where is run.

In Section 6.2 we will introduce the properties of the cluster and local environment we are using. Later on the same chapter, in Section 6.4, we will show some experiments related to this.

"Scala is a Java-like programming language, which unifies object-oriented and functional programming" [MO]. Spark's functions are very similar from its API point of view to Scala's collections. When we create a Spark rich pipeline for a job, our way of thinking is very Scala oriented (i.e. functional). This is why Scala was chosen as the preferred programming language to use with Spark. However, of course some radical differences occur under the hood.

Resilient Distributed Datasets (RDD) are the primary data abstraction in Apache Spark ([Las]). It is a distributed collection of data, across the nodes memory. RDD are similar to immutable lists in Scala. All usual high order functions, such as map, reduce, filter can be used in this context too. However, the execution will be in a distributed way. Normally, the driver program (with the Spark's context) will send to every worker node the function to execute, such as the map function. Then, every worker executes it on the part of data that they have.

Even though we need to specify the type of the objects that reside inside the RDDs we are using, Spark is not able to do any type of optimization for it. For example, if we want to do a computation with only one of the elements of the objects in the RDD (such as the age of a Person, where Person is the object's type in our RDD, i.e. RDD[Person]), Spark would serialize each of the Person objects in the RDD in order to perform this computation. This might not be necessary, as we only need the age for achieving it. Spark is unable to optimize this a priori.

DataFrames are another type of abstraction of Spark that, in contrast to RDDs are structured (like tables in relational databases, they require a schema). However, this is like a shell around RDDs. With this new API (belonging to the Spark SQL module),

we can "load, query and persist structured and semi-structured data, using structured queries that can equally be expressed using SQL" ([Las]). In this way Spark is able to see the breakability of the *native* DataFrame structure "breakable", and produce some optimizations such as only serializing those parts that are needed for computations, (e.g., only the age of the Person in our last example). A lot of optimizations can be easily done by Spark with DataFrames (instead of the users as is the case when using RDDs).

Basically the Spark SQL includes two specialized optimizers. The first one is the Catalyst query optimizer (i.e. optimizes the code), and the second one is the Tungsten off-heap serializer (i.e. encodes Scala objects in a optimal way off-heap).

As users, the down side of using DataFrames is basically that keeping track of the type in the DataFrame is difficult. The problem is that they are weakly typed (i.e. types in their schema are not checked by the Scala compiler). Further, if we transform an RDD of type Person to DataFrame, and afterwards try to bring it back, the Person type is not recoverable. We will have an RDD of type Row, instead of the original RDD of Persons (for more information about this topic, check [Las], *Structured Queries on Large Scale* chapter). This complicates the debugging process.

5.3. Preprocessing

In this section, we explain which techniques and tools we used for preparing our data for the classifier.

Punctuation and Stop Words

In similar works ([SRS12], [Koz15]), titles were used as features for item classification. In general, punctuation was removed, and the text was tokenized. We use additional features besides titles. However, we also removed punctuation and tokenize our features. We didn't use any stop word list to omit words. Titles in general already contain a small amount of words (on average 10). The other fields on average contain even less, except for descriptions that contain an average of 50 words. However, we think that a deeper study has to be done to determine which words are not contributing to solve our problem. With more time this could be done, and could be interesting to compare results. But for now, this is not in our scope.

German Grammar

The language used in the whole data set is German. However, some products are in German from Germany and some others from Austria. The products file is of *json* type. Here, it is specified which German is the employed one for each item (exclusively one). We used both Germans for our study. No distinction is made.

Even though German uses the Latin alphabet, some singularities have to be taken into account (as we mentioned in Section 4.3). In particular, German uses diaeresis and sharp S (Eszett: β) as part of its ortography. Both have an *equivalent* writing form

that doesn't use these particular characters[2]. That is ä, ö, ü is changed to ae, oe, ue respectively, and ß to ss.

However, as we are handling only text features, we should be sure that these differences do not affect our calculations. In order to check this, we used a Word2Vec model trained with the titles of the products. Then, related words for *größe, groesse* and *grösse* were queried. The top synonyms for a word $w \in V$ (where V is the vocabulary) are selected calculating the distance to the other words in V, and taking the closer ones (in our case the top 5). The distance measure in the Spark implementation of Word2Vector is the cosine distance.

Back to our example. It is important to note that the three words (*größe, groesse* and *grösse*, all meaning "size") are part of the vocabulary. They are considered different words (although they should be the same), as they have associated different synonyms.

- Größe: the synonyms are closer to wearable items (such as values for sizes and colors). Which makes sense since *Größe* means size (see Table 5.2). Notice that the similarities are all closer to 0.6 than to 0.5.
- Groesse: the first two synonyms and the last one, are colors (see Table 5.3). Koelner and staedte aren't. In any case, non of them have diaeresis (just like *Groesse*).
- Grösse: the synonyms here appear to be less meaningful than with the other two variations (i.e. größe and groesse). Only the first one (gewünschte, i.e. desired) seems to be in context with the word *size*, as is something we would like.

Table 5.1.: Synonyms for variations of Größe

word	**similarity**
schwarz	0.622
s	0.599
rot	0.590
gr	0.565
3xl	0.564

Table 5.2.: Größe

word	**similarity**
maigruen	0.534
skyblue	0.529
koelner	0.526
staedte	0.518
graumeliert	0.516

Table 5.3.: Groesse

word	**similarity**
gewünschte	0.665
oben	0.546
oslo	0.484
suchfenster	0.473
windsor	0.427

Table 5.4.: Grösse

We changed all diaeresis for its non-diaeresis form, and the sharp S to double s. Of course, no synonyms for *größe* or *groeße* were found (as they don't belong to the vocabulary anymore). But the synonyms for *groesse* changed: see Table 5.5.

In particular, they are closer to the prior list of *größe* synonyms. However, similarity values are higher.

[2] Usually, one is preferred over the other one depending on the case. E.g. *Straße* is preferred over *Strasse*.

word	**similarity**
schwarz	0.648
m	0.615
l	0.614
weiss	0.608
royalblau	0.586

Table 5.5.: Final Table of Synonyms for größe

Neural Embedding: Word2Vector

The research made in [Koz15]) compares different feature modeling techniques that can be employed to perform multiclassification in a similar setting as ours, using Yahoo! shopping data. They showed how the word embeddings method proposed by Mikolov et al. ([MSC+13]) outperformed some others, akin to mutual information dictionaries. In particular this algorithm, has gained popularity in very recent years in the area of Natural Language Processing, and specifically sentences and text classification tasks ([Kim14, ZWZ16]). We have only text fields as attributes, that's why this method seems promising for this investigation.

The skip-gram model is used in the implementation of Word2Vec. "The training objective of the Skip-gram model is to find word representations that are useful for predicting the surrounding words in a sentence or a document" ([MSC+13]). Given a sentence with words: $w_1, w_2, w_3, ..., w_T$, the objective to maximize the average log-likelihood:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{j=-n}^{j=n} log(p(w_t + j|w_t)),$$

where n is is the window size. That is, we are trying to predict w_t's surrounding words, where w_t is in the center. With larger values of n, more words around w_t are considered, but the training time will be also longer (see Figure 5.5).

Every word w is associated with two vectors u_{w_i} and v_{w_i} which are vector representations of w as word and context respectively ([Spaa]). It uses the soft-max function to predict the embedding:

$$p(w_i|w_j) = \frac{\exp(u_{w_i}^{\top} v_{w_j})}{\sum_{l=1}^{V} \exp(u_l^{\top} v_{w_j})},$$

where V is the vocabulary size. The skip-gram model weighs nearby context words more heavily than more distant context words. Therefore, words that occur in similar contexts have similar embeddings. Spark has implemented this using hierarchical softmax. This way, the complexity of calculating the embedding for one word is not proportional to V, but $O(\log(V))$ ([Spaa]).

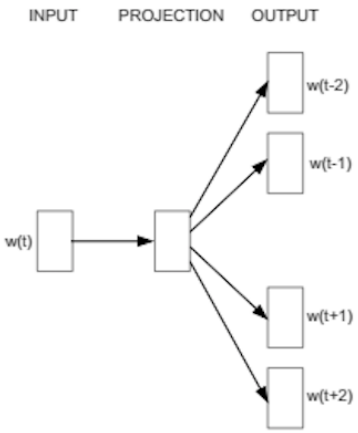


Fig. 5.5. Skip-gram model

For every word in a title, we calculated its vector representation. Afterwards, we calculate the pool average among them, to have a final vector representation for the whole title. That is, having a title with M words: $(w_1, w_2, ...w_M)$, and their respective embedding: $(e_1, e_2, ..., e_M)$, where $e \in \mathbb{R}^E$ (E is the size of the vector, this can be set arbitrarily). Vector $v \in \mathbb{R}^E$ would be the final vector representation for the title:

$$v = \frac{1}{M} \sum_{m=1}^{M} \sum_{i=0}^{E-1} e_m(i).$$

Spark has two implementation for Word2Vec. One belongs to the MLLIB library, and the second to the ML library. In the first one, the pool average calculation is not contemplated, but in the latter it is. Because this was desirable, we use the second one.

The default relevant variables' values in this implementation are:

- Min. Count: Determines how many times a word has been seen to be considered to be a part of the vocabulary, the default is 5.
- Vector size: Size of the vector (E in our equations). The default is 100.
- Window size: The default is 5 (i.e., n in our equations).

PCA for Outlier Detection

Principal Component Analysis (PCA) is an unsupervised algorithm that seeks the projection that best represents the given data, in order to lower the dimensionality of it ([DHS+01]). See Appendix A.1 for more information about the algorithm, and how to calculate a PCA model.

At this point PCA was used in order to visualize our data (in 2D). We projected our data's feature vectors (calculated with word2vec), into the two first principal component's space.

In Figure 5.6 we can see the projection of a random subset of the data, grouped by their first level category. The category *Freizeit und Outdoor* is the one with the largest variance in the first principal component, i.e. the x axis. It also overlaps with all the others categories at some point. Then, it would be expected that in the results, this category is confused with any other. On the other hand, we can see the dominant categories are *Mode und Accessoires, Auto und Motorrad* and *Freizeit und Outdoor*. As this is a subset, the proportion of products per categories is maintained. However, if we go back to Table 4.4 where we see the distribution of products over first level categories, *Haushalt und Wohnen* is missing in our PCA visualization, even though it is the second most populated category.

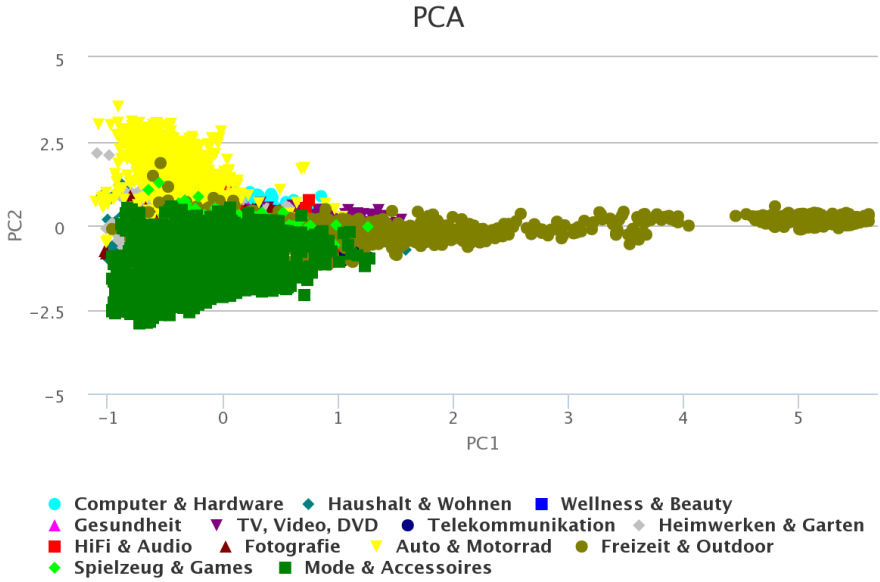


Fig. 5.6. PCA of a subset of the data

If we filter out all categories but *Haushalt und Wohnen* and *Freizeit und Outdoor*, for example, we get an idea of what is happening (see Figure 5.7). We also showed the projection of *Haushalt und Wohnen* and *Heimwerken und Garten* together (see Figure 5.8). The second doesn't overlap as much as the first one the *Haushalt und Wohnen* category.

In the experiments section, we will confirm whether our thoughts about this section concurring this relationship between categories is true.

With this method we also tried to detect outliers by inspecting the data points that were seen out of the distribution presented by every category. Some of them were indeed

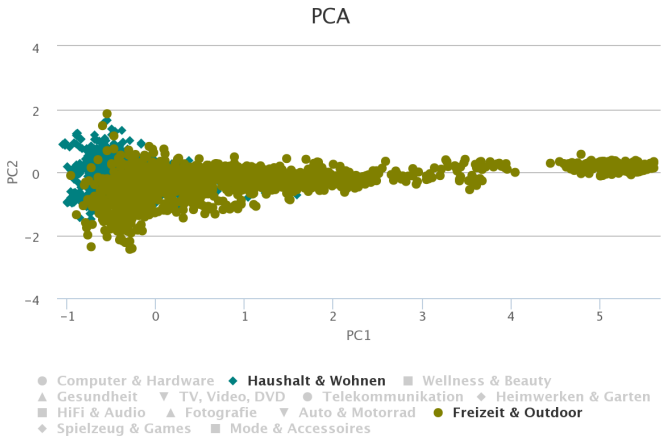


Fig. 5.7. PCA: *Haushalt und Wohnen* with *Freizeit und Outdoor*

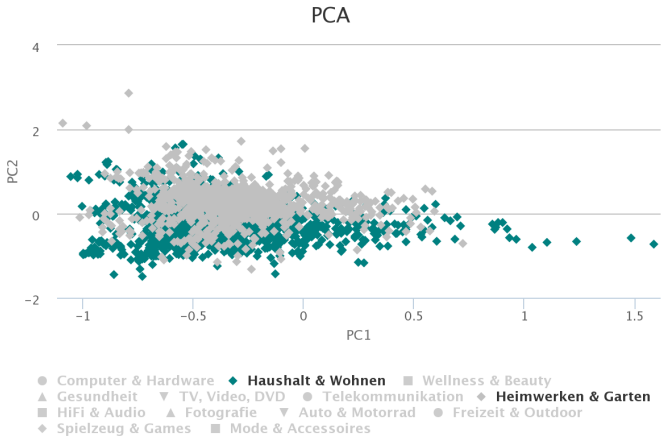


Fig. 5.8. PCA: *Haushalt und Wohnen* with *Heimwerken und Garten*

misclassified points in the training data, but some others were simply outliers for that category.

For example, a lot for *cushions* were found. These products had their information in English, instead of German. And they were wrongly preclassified.

We ran multiple times the projection, looking for wrongly classified products. If we have too much of this products our trainer is going to learn wrongly the products' labels. However, because of lack of time (we spent around 10 hours to *clean* the data with this method), we could only detect around 200 misclassification errors by means of PCA projections. In any case, we empirically proved that it's a good method to clean the data set a bit. This helped us to detect outliers in a visual way.

This cleaning step didn't severely impact our results, as 200 products out of 100K represents only 0.02% of the total.

More projections of the categories individually and by pairs can be seen in the Appendix C.1.

5.4. Classification: kNN

K Nearest Neighbor is a "lazy-learning" algorithm ([YZK03]). This means that we will defer any calculation till the testing phase. So basically, a very small amount of computation is required in the training phase (e.g., the creation of the features).

In particular, there is a well known issue when using kNN especially with many attributes. kNN results rely on the calculation of the distance between two data samples, which are located in feature space. T. Mitchell in his Machine Learning book ([Mit97]) explains an example when this occurs: Imagine one instance is described by 20 attributes, but only 2 of these are relevant for our classification. In this case, if two instances that are similar (w.r.t. what we are trying to classify), they would have equal (or very similar) values for this two relevant features. However, because of the other 18 dimensions that also count, the two points might end up very distant from each other. "The distance between neighbors will be dominated by the large number of irrelevant attributes", mention Mitchell, "Nearest-neighbor approaches are especially sensitive to this problem" talking about curse of dimensionality.

This algorithm is in particular useful when the classes have many prototypes and the decision boundary is very irregular ([HTF01]). Therefore, it makes sense to try it for our problem, despite the curse of dimensionality. We have a wide variety of products inside every subtree that is formed from the first level categories, i.e. each of the 13 classes have many prototypes. How many different accessories and types of clothes can you think of? How many different furniture, decorations and items can you find in a home? What if we add to them the diversity of edibles? It is difficult to find only one prototype that represents all items in a branch. Many studies ([YZK03], [SRS12]) have used kNN successfully for hierarchical classification similar to ours.

Implementation of KNN

Spark doesn't include an implementation of kNN in any of its machine learning libraries. This is why it was implemented.

We implemented three versions of kNN for determining the neighbors. The first one uses a fixed k, i.e., k is set in the beginning. This is the typical kNN algorithm, where we take the k nearest neighbors of the test sample.

The second one, selects k dynamically (as proposed in [SRS12]). This is, the chosen neighbors aren't determined by a determined number of them but by their distance to the sample. We select only those neighbors whose similarity distance is lower than a threshold. The threshold is set similarly as suggested in Shen et al. work [SRS12]: double of the distance of the first neighbor.

Third, we implemented a version of weighted kNN, where weights are assigned to training samples, depending on their distance to the test sample. This has the advantage that no sorting is required.

For all three we used cosine similarity. This is because Spark's word2vec (i.e. where our feature vector comes from) uses this as similarity measure too.

Independently of the implementation version, the calculation of the distances between the test set samples and the training set ones is required. First we calculate the Cartesian product between the train and the test samples. Let's say we have a train set $T = \{t_1, t_2, ...\}$ and a test set $E = \{e_1, e_2, ...\}$, where $e_i, t_i \in \mathbb{R}^F$ (F is the size of the feature vector, or the space dimensionality). Then,

$$TxE = \{(t, e) | t \in T, e \in E\}.$$

In order to do this in Spark, we assume that the test set is smaller than the training set. The test set is broadcasted among all the nodes, and then joined with the train set.

Then, a simple *map* function with the preferred distance measure (i.e. cosine similarity[3].) is executed for calculating all distances between tests and training samples. See Figure 5.9.

Now to proceed we need to differentiate between the three implementations.

We based the first one in the implementation specified in [Par15]. In every partition and test sample the distances are sorted. We take the first k in every partition for every test sample. Then we group by test sample ID. Next, we *reduce* them, having all partitions together (notice that we are only then sending k times the number of partitions of records over the network). We sort them again and take the final k neighbors for every sample. The selection of k was done in an empirical way. However, it is known that smaller k produces higher variance, and therefore makes the algorithm less stable. On the other hand, large values of k produces higher bias, which translates into less precision in the prediction. More about this in the evaluation's Section 6.4.

For the second implementation, we need to find the closest neighbors in a determined distance radio around the test sample. And then a *filter* function would do the rest. After

[3] Cosine similarity between $t \in T$ and $e \in E = \frac{\sum_{i=1}^{F} t_i \cdot e_i}{\sqrt{\sum_{i=1}^{F} t_i^2} \cdot \sqrt{\sum_{i=1}^{F} e_i^2}}$

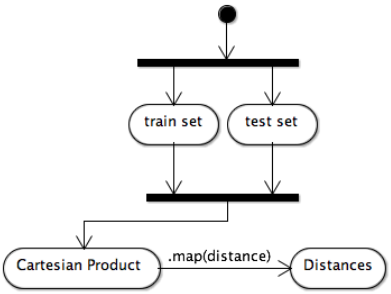


Fig. 5.9. kNN distance calculation between train and test set samples.

empirical experiments, we noticed that there was no significant change in the accuracy when compared to the first implementation. However, it was more time consuming. Therefore, we stuck to the other two implementations, and disregard this one. Maybe it would be better to explore it more deeply.

For the third implementation, we calculated the weight of distance d as $w(d) = \frac{1}{d^2}$. This way the closer points are weighted more than far away points. After this, we sum together all weighted distances that belonged to the same label. For more efficiency (i.e. reduce the amount of traffic between nodes), we first sum in every partition (using *mapPartitions*) the weighted distances of the data points that belonged to the same class. This way we would have at the end in every partition a maximum of $T \cdot C$ sums to shuffle with a *groupBy* category (where T is the test set size and C is the number of classes). After that, similarly to the first implementation, we just need to reduce the set to have all the $T \cdot C$ summed weighted distances. From this we just need to take the highest value per test (see the code in Listing 5.1)

For the well known Iris data set, we tried the first and the third implementations having approximately 96% accuracy. If we compare this accuracy to the one gotten by running the same data set with the scikit-learn implementation (with the same k and distance measure), we obtain the same result. This was done to confirm that our implementation was correct.

Choosing k is a hard task. Using cross-validation for this matter works, but it is unstable, as we need to recalculate k every time we have new data. The classification is very sensitive to k.

Listing 5.1: kNN Code

```
{
//cartesian Product
val cross = train
  .join(broadcastedTest.value)

//distance between every two vectors
```

```
val distances = cross
  .withColumn("distance",
    cosineDistance(col("trainFeatures"), col("testFeatures")))

// weight for every distance.
val weighted = distances
   .withColumn("weight", weight(col("distance")))

//partial sum of weighted distances: (testID, category, sum)
val sumWeightsPart = weighted
   .mapPartitions(it => sumWeights(it, categoryColumn))

//final sum of weights
val finalSum = sumWeightsPart
   .groupByKey(_._1, _._2) // (testID, category)
   .reduce((a,b) => (a._1, a._2, a._3+b._3))
}
```

The parallelism level chosen by Spark to use with its RDDs or DataFrames, depend on the underlying storage system. If the data is stored in HDFS (as in our case), by default the number of partitions created will be the number of blocks in the HDFS file where it is stored. The number of partitions is also changed when shuffling. If a DataFrame or RDD is a consequence of a shuffle, the number of partitions will be inherited from the size of their parent data structure ([KKWZ15]).

In the case of our kNN implementation, our Cartesian product combinine a Dataframe with 200 partitions with a second Dataframe with 200 partitions, this implies that at the end we end up with 40.000 partitions in the result. It is a fact that at some point, some unknown partitions contain a lot of records in comparison to the others: We have skewed data. This causes some tasks to go very slow, which is of course a problem.

Yang et al. in [YZK03] analyze the complexities involved in classification algorithms. They divide the algorithms into two groups: Binary classification algorithms, such as SVM, and m-way classification algorithms, such as kNN. In the first set of algorithms the complexity grows linearly in the number of categories. This is because, having C categories, means that we need to train C different binary models (i.e. when using *one versus the rest* approach[4]), in order to have our classification results.

In [YZK03] it's considered that the hiearchical classification is done level by level, i.e. we classify into every single level of the taxonomy. However, our approach corresponds to a two level classification, where the second level categories are all categories that belong to the categories in the branch (i.e. all levels from 2 to 6). And the first one is, obviously the first level of our taxonomy.

[4] Even if we use the *one versus one approach* the number of models to train grow proportionally to the number of categories we have.

Having this in mind, let us analyze the complexity formulas given by Yang et al. ([YZK03]). For kNN, they proved that the hierarchical complexity is:

$$Q_{knn} \leq h \cdot O(N),$$

where h is the depth of the hierarchy, N is the number of data points in the training set, and $O(N)$ is the single-classifier complexity on the training set. As said, our setting is equivalent of having two levels.

Our distributed implementation (with Apache Spark) has a slight difference in the calculation of the complexity for classifying one level. Our training set (of size N) is distributed among the partitions. Let us call n_i the size of the training sample part that is in partition i. Notice that $\sum_{i=0}^{P-1} n_i = N$, where P is the number of partitions. We then select the shortest k distances from every n_i. This is linear with respect to the number of samples (i.e. we are still in $O(N)$). After we shuffle all the kP samples, we need to find the shortest k distances from them. This is also linear with respect to the subset of samples $k \cdot P$. Then our final complexity for one level is $O(N \cdot k \cdot P)$. Notice that $k << N$, therefore $O(N) \leq O(N \cdot k \cdot P) \leq O(N^2)$.

We make a bit more of calculations for the time complexity of our solution. Let us say that the time for calculating the k closest neighbors in partition i (i.e. among p_i samples) is K_{p_i}. Then, the time we need for calculate the final k neighbors from all partitions is K_P. Then, in total we will have a worst case scenario that is:

$$time_{knn} = max(K_{p_i}) + K_P + n(k),$$

where $n(k)$ is the network time required for sending the kP samples. If we add more parallelism, the time of calculating the k distances in a partition will decrease, i.e. K_{p_i}. However, K_P will increase, as well as the amount of data that is being sent through the network. This doesn't consider sorting the k neighbors, which should be $O(k \cdot log(k))$

If no parallelism is used, this is all done in sequentially: The time complexity is $O(N)$.

The space complexity is high (i.e. our training set size), as the model is basically our training sample.

Notice that all of these calculations are independent from the number of categories we have. Therefore, scaling in terms of categories won't affect time, space or computational complexity. On the other hand, an increase in the number of training samples is linear w.r.t. all complexities.

5.5. Classification: SoftMax Function (Multinomial Logistic Regression)

Binary logistic regression can be generalized to output more than two different classes. This generalization is explained in Appendix B.1.

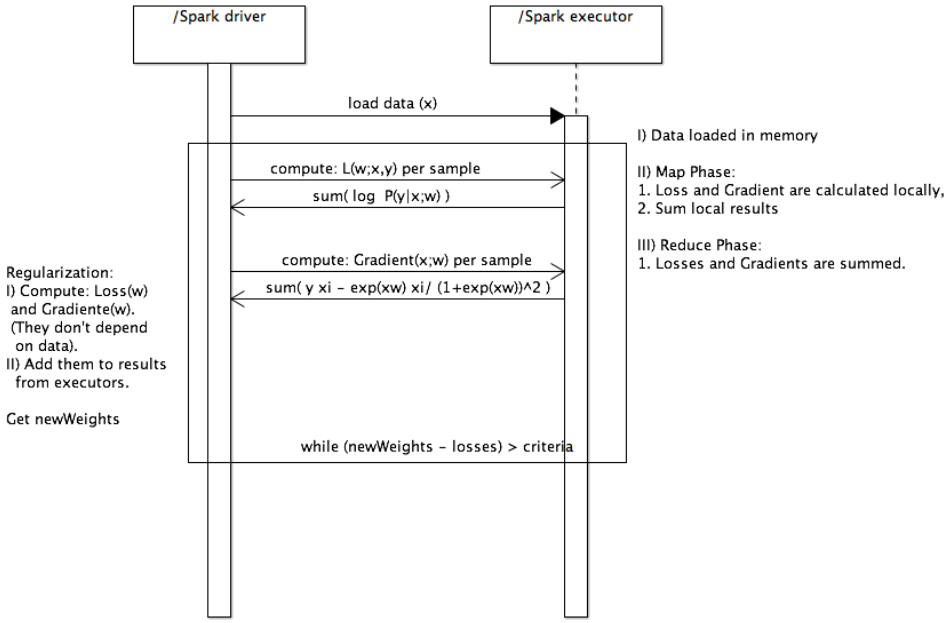


Fig. 5.10. Implementation of Logistic Regression in Spark

Spark's implementation of Multinomial Logistic Regression is parallelized. A general sequence diagram of its training's implementation can be seen in Figure 5.10.

There, we are basically calculating iteratively the loss function $l(w, x)$:

$$l(w, x) = l_{model}(w, x) + l_{reg}(w). \tag{5.2}$$

Spark's original prediction function for this model was slightly changed, in order to return the probabilities for all the classes instead of just the predicted class. When predicting a label for a data point x_i, we calculate the probability for all possible classes Y_i:

$$P(Y_i = K + 1|x_i; \beta) = \frac{e^{\beta_k \cdot x_i}}{1 + \sum_{k=1}^{k-1} e^{\beta_k \cdot x_i}}. \tag{5.3}$$

With the first class being the pivot, with probability of:

$$P(Y_i = 1|x_i; \beta) = \frac{1}{1 + \sum_{k=1}^{k-1} e^{\beta_k \cdot x_i}}. \tag{5.4}$$

This is returned as part of the *predict* function.

Two algorithms for logistic regression were already implemented in Spark: mini-batch gradient descent and Limited Memory-BFGS (L-BFGS). We used L-BFGS because it converges faster than the first one [Spad].

The Sparks default values for the optimizer (L-BFGS) are:

- Criteria or convergence tolerance: 1E-6.
- Maximal number of iterations: 100.
- Regularization Parameter: 0.0
- Updater: type of regularization. We used L2, i.e. $L2 = \lambda \sum_{i=1}^{N} w_i^2$. This updater penalizes high values of w.

In the next chapter, we will experiment changing some of this values, in order to tune our models. Scaling in terms of feature dimensions or the number of categories affects the time, space and computation complexities. As we need to estimate a variable per class and per feature's dimension, that need to be then stored as the model.

5.6. Classification: Random Forest

A random forest classifier is an ensemble of decision trees ([Bre01]). The risk of overfitting with this algorithm is reduced using many decision trees: It is like asking a lot of experts for their opinion, given them different parts of the information they might need. That is, it injects randomness into the training process, i.e. avoiding overfitting. At the end, the predictions of all trees are combined ([Bre01], [Spac]).

Training every tree can be done in parallel: Every tree is independent from each other. The randomness in the training process comes in basically two forms: bootstrapping, i.e. subsampling of the original data set. This is what we mentioned about *telling every expert* (i.e. tree) only a part of the information. And second, in every node of every decision tree that is built, a different random subset of features to split is selected.

Deeper trees reduce the bias, whereas more trees reduce the variance in the predictions. But of course, having many deep trees will slow down the whole task significantly. "Training time increases roughly linearly in the number of trees" ([Spac]). This is also true for the number of training instances and features.

The final label is calculated by aggregating by majority of vote, the predictions of every tree. The predicted label is the one which receives the most votes ([Spac]).

The relevant default values in Spark's implementation of Random Forest are:

- 20, the number of trees in the forest.
- 5, the maximum. depth of each tree. This value *starts counting* on 0 (i.e. depth 0 means 1 leaf node).
- 100, the maximum number of bins used for splitting features.

5.7. Classification: Support Vector Machines (SVM)

SVM is a supervised learning algorithm for binary classification. Its objective is to find the hyperplane that maximally divides the data into two classes (see Appendix B.2). With an infinite number of hyperplanes that could separate our two classes, we will select the one that does it by maximizing the margin like in Figure 5.11 where we would choose the margin from the right ([MRT12]).

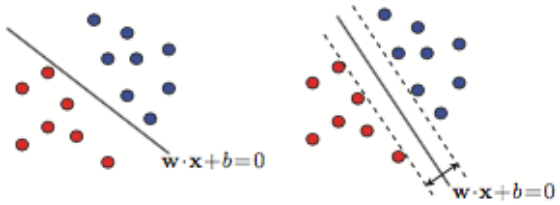


Fig. 5.11. Both hyperplanes separates the classes. However, the one from the right would be chosen by an SVM model, because it is the one with larger margin. From [MRT12]

Spark's implementation is by default trained with L2 Regularization (i.e. $\frac{1}{2}\lambda \cdot \|w\|_2^2$). But we can also choose L1 Regularization. Also, the parameter lambda (λ) is adjusTable as usual.

One versus the Rest (OVR)

One versus the rest, also called one-versus-all, is a method used for extending binary classification models to allow for multiclassifcation. In this case, we need to train C different binary models, where C is the number of categories we have. Each will discriminate between one class and the rest of the classes (i.e. distinguish if a data point belongs or not to a class). At the end, a decision is taken by majority vote or other techniques.

Despite Spark has already implemented a version of OVR, we implemented our own. This is due to the incompatibility of it with their own SVM version (for what we need it for).

The implementation was done in a sequential way: Each classifier for every label (against the rest) was trained one by one. This was done like this, because the whole data is needed for each of the trained models.

However, the online classification phase is implemented differently. The C models were broadcasted. Then, every point is predicted in parallel with every model. After that, the most probable class is selected as the predicted one.

5.8. Grained Level

For the grained level the same algorithms could be considered. We need to assure that only the products that are associated, i.e. labeled with categories on that branch, are taking part in the training of it. One problem that we could face is that having a lack of products for training a certain category. This is a problem that might occur also in the first level classification phase, however in the grained level is more likely to happen due to the quantity of rare categories (i.e. those with less than 10 products).

This can be solved by splitting our dataset assuring that we have at least one item in the training set for every category. This was solved by taking the same train-test ratio for every different category that was going to be part of our classes set. That is, if our train-test ratio was 80:20, for every category we randomly[5] take 80% of the items we had and place them in the training set. Then take the rest and consider it as test.

Now we are ready to start experimenting with all this.

[5] The randomSplit and sampleByKey functions in Spark were used. They use "a sampler based on Bernoulli trials for partitioning a data sequence" (taken from Spark's API 2.0.0, i.e., https://goo.gl/Sp3z8W). This is an implementation of Java's RandomSampler, which uses a pseudorandom number generator (i.e., it is deterministic), and involves using the system clock to get a different seed every time.

6. Time to Evaluate it All

> "So the universe is not quite as you thought it was. You'd better rearrange your beliefs, then. Because you certainly can't rearrange the universe".
>
> — Isaac Asimov, *The Gods Themselves*

Once we have everything implemented, it is time to experiment and evaluate our work. First we will talk about which measurements we are going to use in order to evaluate everything, including the baselines. After this, we are going to describe the test environments. Next, we will explain the executed experiments, with their respective results, analysis and conclusions. Finally, we will discuss the proposed solution, its problems and remarks.

6.1. Performance Measures

Results can be quantitatively or qualitatively measured. The first one is easy to calculate, and will be explained in the next subsection. The second one is hard, as we can recall from Section 4.3, where we talked about misclassification problems.

All the results are based on the given labels of the data, which aren't absolutely correct. Moreover, some errors are worse than others (e.g. a mountain bike classified as an avocado plant is a worse error than it been classified as a city bike). But the degree of error is hard to measure.

A solution for this, would be to manually check for errors similar to what we did in Section 4.3. However, this can be a very expensive task, because it can't be automatized. Therefore, we leave these types measures out of the scope. We will rely only on quantitatively methods.

From the types of misclassification errors listed in Section 4.3, we are going to treat all three equally. However, we will calculate how many of them are subtle errors (i.e. the real category is in the path of the assigned category to the root of the hierarchy) in our experiments.

The Accuracy Paradox

Accuracy is a measure that calculates the ratio of correct predictions to the total number of cases evaluated (basically the percentage of correct predictions). The accuracy paradox for predictive analytics affirms that we can be in a situation where we would prefer lower accuracy levels than higher [VAPM14]. This could happen for example, when the predictive model we are evaluating has greater predictive power, than others

that have higher accuracy. A good example for this kind of situation is when we seek to detect outliers. Imagine this occurs 1 out of 100 times. The accuracy of one possible model for this problem could be 99%. Nevertheless, the model might report "This is not an outlier" in every case. Its predictive power is very low, i.e. it's not predicting anything at all. In this case, we would prefer to have less accuracy, but more (real) predictive power. This problem shows up specially when we face unbalanced data (such as ours).

The accuracy metric may be better to avoid, in favor of other metrics, such as precision and recall. We will use the F_1 Score, which is a weighted average of precision[1] and recall[2].

With a delta function δ :

$$\delta(x) = \begin{cases} 1, & \text{if } x = 0. \\ 0, & \text{otherwise.} \end{cases} \tag{6.1}$$

Let us name the classes $L = \{\ell_0, \ell_1, \ldots, \ell_{M-1}\}$, and the true output vector y with N elements (N in our case is the number of products in the test set): $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{N-1} \in L$. The overall or weighted F_1-Score calculation, depends on the F_1-Score for label l. This can be calculated with the use of two formulas [Spae]. Equation 6.2 calculates the F_1-Score for label l. And formula 6.3 corresponds to the overall weighted score:

$$F_1(\beta, l) = (1 + \beta^2) \frac{precision_l \cdot recall_l}{\beta^2 \cdot precision_l + recall_l} \tag{6.2}$$

$$F_1(\beta) = \frac{1}{N} \sum_{l \in L} F_1(\beta, l) \cdot \sum_{i=0}^{N-1} \delta(y_i - l) \tag{6.3}$$

Remember that $l_i \in \mathbf{R}$, and this implies that y_i does too. We use $\beta = 1$. The Weighted F_1-Score will be used always as measure of accuracy. From this point on, we will refer to it as F_1-*Score* for simplification.

Confusion matrices[3] are particularly useful to identify, which classes are false positives or false negatives.

6.2. Test Environment

Experiments were done locally, and in a small cluster provided by the company. In both, Spark 2.0.0 with its standalone cluster manager was used.

Locally, the host operating system is Linux Mint 17.3 'Rosa'. The machine has four cores, each with an Intel Xeon 3.2GHz CPUs, and 16GB of memory.

[1] Measures the classifier's exactness. $= TruePositives/(TruePositives + FalsePositives)$

[2] Measures the classifier's correctness. $= TruePositives/Positives$

[3] Summary of predictions in a table showing correct predictions (in the diagonal) and the types of incorrect predictions (i.e. what classes incorrect predictions were assigned) in the other cells.

The cluster on the other hand is comprised of three nodes. Each node have a virtual machine where Spark 2.0.0 is running. In every node, a total of 6 CPUs and 12GB is assigned for the Spark's virtual machines. They also have a simulation of the hardware with *QEMU Virtual CPU* version 2.1.2 with 1.8GHz. The three hosts, where the virtualizations are have 48 CPUs and 128GB each. All have Intel(R) Xeon(R) CPU E5-2650L v3 with 1.80GHz as processor.

It is worth mentioning that Spark virtual machines are not the only ones hosted in the real boxes. While CPU and RAM are provsioned exclusively for the Spark virtual machines, there might be contention for network and disk I/O from other guest machines on the same host, making measurements less reliable.

Even though the network load is not constant in time, we will assume it is, in order to compare our results in the cluster without any further calculations. However, we should keep this in mind.

6.3. Data for Experiments

For a few experiments, additional data was used. Due to misclassification problems encountered using this data set (see Section 4.3), results (that should be corrected) were expected.

To avoid this problem, in some works that use eBay products as data ([SRS12, SRSS11]), articles that were successfully purchased were used for training. Due to the belief that those had more chances to be correctly labeled.

As we know, idealo doesn't sell any products (just compare prices and redirects to those who sell it), thus this option was not possible. As a solution for this, a new (larger) data set was provided containing products that were manually worked. This new set of products contain a (manually prepared) product sheet, and therefore were supposed to have fewer misclassification problems. Figure 6.1 contains the distribution of the features in this data set.

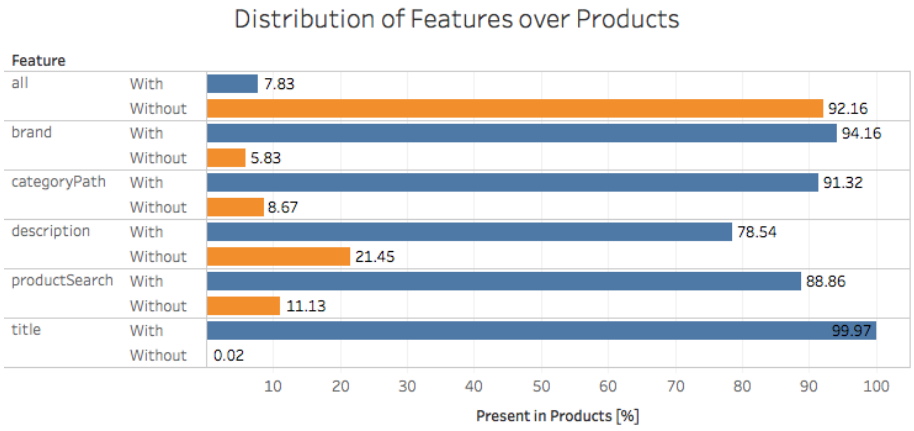


Fig. 6.1. Existence of Features in Large Dataset

For experiments, the data was always split (e.g. in train and test set) in a way that ensured that every category was represented in the training set.

Finally, a third data set was provided. This was very similar to the first one (introduced in Chapter 4) and representative of products that the classifier in production will receive (i.e. it is more similar to the data that will be classified in the end). This was used only for testing purposes. Table 6.1 summarizes the dataset sizes.

Data Set	**Size [MB]**	**#Products**
A: The one from Chapter 4	257	93112
B: New Large one	1500	589464
C: New Small one	282	84162

Table 6.1.: Sizes of Datasets

In general, sets A and B will serve as training and validation sets. Dataset C will be used for testing the classification steps.

6.4. Experiments

For comparison, the next baselines were taken (in ascending order of importance):

1. Better than randomly chosen. If we are classifying into *total* number of categories, this would be $\frac{1}{total}$.
2. Majority Class. Because we are dealing with unbalanced data, we need to check that our classifier has predictive powers (see Section 6.1). This means, that our accuracy needs to be better than the percentage of representation of the dominating class.
3. We will compare our methods between them.

The F_1-Score for the current classifier at idealo has not been determined, since the final categorization is determined as a result of several interacting processes (see Figure 3.1)

In the following subsections experiments in the subsequent phases (explained in Chapter 5) will be discussed.

Cleaning

First of all, we need to clean the data following this steps:

1. Select only the useful information from the products, such as the features to be used, category of the product in order to evaluate our classifier at the end, and its identification.

2. We clean the text we have by removing all punctuations. Also, we lower case every word.

3. In order to apply some algorithms to our data, we need the text in our features to be tokenized. For this we used the Spark's tokenizer algorithm (see [Spaf]).

Feature Selection

Figure 6.2 shows all the preprocessing steps we need to tune, in order to perform our classification. We highlight there the current step of experiments for achieving that.

As indicated in the earlier chapters, only five features were considered: *title, description, category path, product search and brand.* All consisting of text fields, as explained in Section 4.2. The objective of this experiment is to confirm which features we should use for classifying. We tried different combinations of features, and evaluated their performance.

We use dataset A with a split 80:20, respectively for train and validation set. The model was build for classifying among the first level categories, using multinomial logistic regression with its default values for the parameters (see Section 5.5). It was run in the local environment.

The *title* is the only feature that appears in almost every product (see Figure 4.6), therefore was used in all eight experiment of this section.

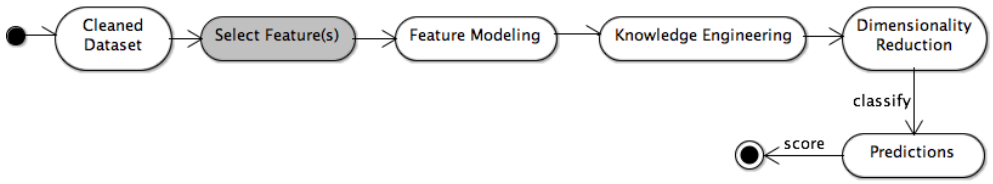


Fig. 6.2. Preprocessing steps: The gray box indicates where we are.

#Exper.	Feature(s)	F_1-Score	Time [min]
1	Title	0.74	2.50
2	Title + Description	0.78	3.38
3	Title + Category Path	0.87	2.59
4	Title + Product Search	0.74	3.08
5	Title + Brand	0.77	3.00
6	Title + Description + Category Path	0.87	3.52
7	Title + Description + Category Path + Brand	0.87	3.63
8	All Five Features	**0.88**	3.76

Table 6.2.: Feature Selection Results

In Table 6.2, we see that the best results were obtained when using all of the five features, as expected. It is important to note that the greatest difference with respect to

using the title, happened when *Category Path* was added (improvement of 13 percentage points). Followed with a big difference by *description* which "only" improved the results 4 percentage points.

All five features will be selected (i.e. title, description, brand, category path and product text search). The last one is used despite of its lack of input for classification. The reason is that *product search* doesn't contribute much to the output of the classification. This might be because it is not that present frequently in the products (i.e. only around 30% of the products have this feature. See Figure 4.6). In any case, it doesn't contribute to the final results, hence it doesn't harm them either.

The most common misclassification errors selecting only the *title* as a feature, happened when classifying products into the following categories: Audio *Cds*, *DVDs* and *Sportbekleidung* (sportswear). In the end, (in experiment 8) one of the misclassification errors remained: classifying *Sportbekleidung* mistakenly into the branch of *Mode & Accessories* (i.e. Fashion) instead of *Freizeit und Outdoor*.

Word2Vec

In the last chapter, Section 5.3, we discussed how Mikolov's word2vec algorithm works, and how we can use it for calculating the pooled average for sentences. Also, we described the Spark implementation, as well as its default values.

For all our experiments using word2vec, we employed Spark's machine learning ML library implementation. The word2vec models were trained using the same domain data: we used all the titles and the descriptions from the dataset A.

Let us say we produce a vector of size v. After processing all of these five features and concatenating the outcomes, we will obtain a final vector of length $5 \cdot v$ (see Figure 6.3).

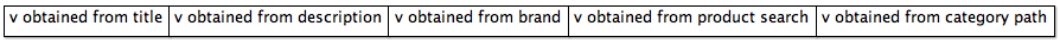


Fig. 6.3. Word2Vecs Vectors Concatenated

In this section, we describe varying experiments involving the use of different vector sizes for Word2Vec, i.e. v. The goal is to determine which size of vector v performs the best in order to model our features (see Figure 6.4).

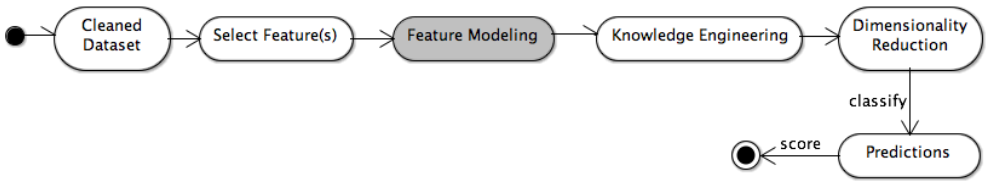


Fig. 6.4. Preprocessing steps: The gray box indicates where we are.

Dataset A was used in the local environment. For this experiments, we didn't select all five features as established in last section, but only the titles and the descriptions.

Table 6.3 shows that the best size for our vectors is 200. The vocabulary contains 141967 words.

Vector Size (v)	$\boldsymbol{F_1}$**-Score [%]**	**Training Time [min]**
50	0.750	3.43
100	0.782	4.21
150	0.820	7.56
160	0.818	7.62
200	**0.829**	10.11
220	0.795	10.32

Table 6.3.: Word2Vec experiments with Vector Size.

The rest of the Word2vec parameters, such as window size and iterations were left with their default values.

A feature vector for a product will then be of size 1000 (from $5 \cdot 200$). The number of features normally is a value that directly affects the number of variables to be estimated when creating a model. During the first level classification phase, we have enough training samples to discriminate between the 13 categories. However, when we are attempt to classify a product into a (branch's) category we might not have enough instances for training (as shown in Chapter 4). Hence, we used a word embedding model with vectors of size 200 for the first level classification. However, when classifying at a grained level we preferred to use vectors of size 150.

Dataset A was used for both: training the word2vec model, and the classification methods. Overfitting problems were at this point discarded with empirical tests, although this algorithm is unsupervised.

For discarding overfitting at this point, our data set was divided in a 80:20 proportion. The first one, was used to train the word2vec model. After that, the obtained model was used to create the features for both: the same 80% of the data, and the *new* 20% section. Then, we trained a classifier with the 80% part, and test it on the 20% (see Figure 6.5). In the second part of the experiment, we created the word2vec model with the whole data. As before, we transform our data with the built word2vec model. Then, we proceeded as before: training a classifier with only the 80% of the data and testing it with the remaining 20% (see the first one in contrast with Figure 6.6).

The difference between the two results is negligible (i.e. $F_1 - Score_{firstExperiment} - F_1 - Score_{secondExperiment} \approx 0.00001$).

The final characteristics of the models used in the first and grained level classification, that were discussed previously, are summarized in Table 6.4.

Domain Specific Changes

Machine learning algorithms are always dependable on the information and type of data that is used for learning. Everywhere you look (papers, books, articles...), they

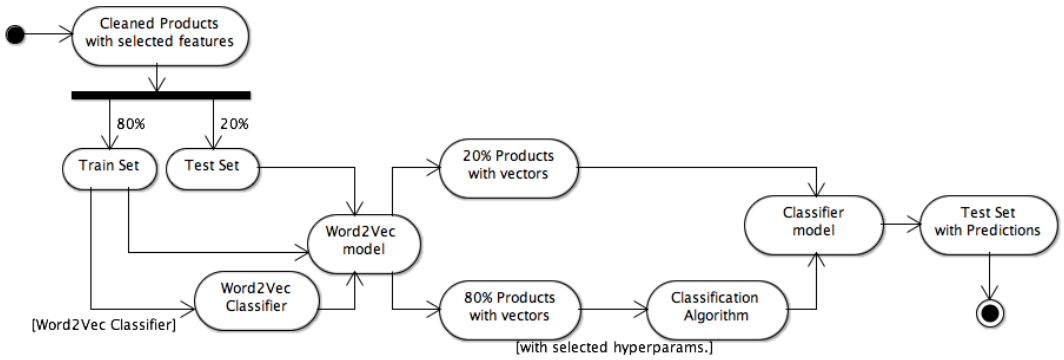


Fig. 6.5. Word2Vec being trained with Classifier

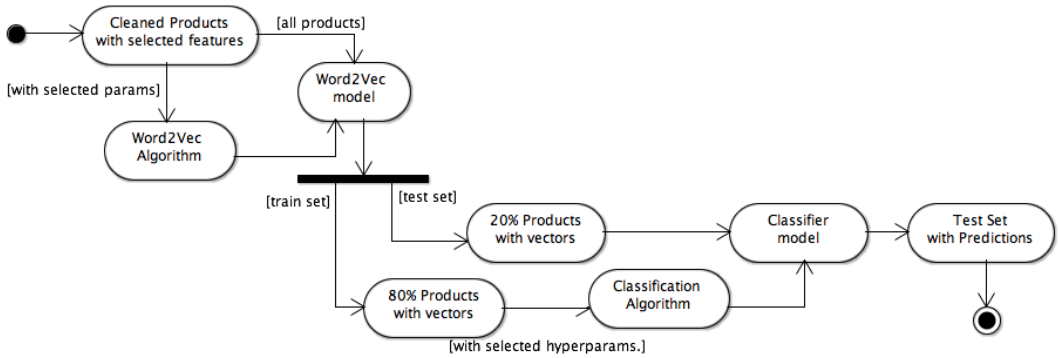


Fig. 6.6. Word2Vec independent of Classifier

Classification	**Vector Size**	**Model Size [MB]**
First Level	200	115.2
Grained Level	150	89.3

Table 6.4.: Word2Vec Vector Sizes depending on Classification Level.

always tend to include a phrase with something similar to "...but it will always depend on the data you have", referring to the sensitivity to features that this algorithms have. There is no master algorithm that works for every case. And that is why it is so important to make a proper exploration of the data under investigation.

Even when a perfect algorithm is selected, its results depends strongly on the features one provide for the task. Most of the time of this work was spend in finding this features that will suit our algorithms, i.e. feature engineering. Shen et al. ([SRSS11]) manage to improve their classifier's prediction for eBay items by 1 or 2 percentage points in one of their subcategories, only by making some changes in the used features. This is considered significant in their experiments, "since the base number of items to be categorized is in the order of hundreds of millions and one percent error rate may impact millions of users' experiences" [SRSS11]. Of course, this requires previous knowledge of the domain, errors and the reason they are happening. Fortunately, we already have this covered (see Chapters 4 and 5). In this section, we explore additional input parameters we can tweak based on our domain knowledge, in order to improve our classification (see Figure 6.7).

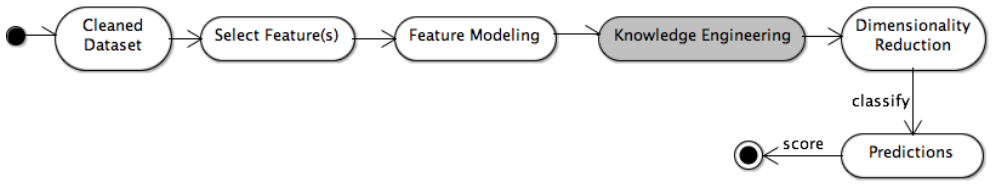


Fig. 6.7. Preprocessing steps: The gray box indicates where we are.

Basically two different approaches were taken. The first one adds information to the feature vector (i.e., the one obtained using a word2vector model), hoping that this would help the classifier to decide for the best category. The second approach is related to the hierarchical structure we have. As we are classifying first into the first level categories and then into the selected branch's categories, we assume that all items inside a branch are similar between them. This is not always the case as we explained in Section 4.3.

Adding Information to the Vector

When our word2vector model converts sentences into numeric vectors, it doesn't consider one word more important than others, just because of its semantic or syntactical significance. In "Samsung Galaxy 7 Cover", the word *cover* may be the one that determines if the products will go to the smartphones category or to a cover's category. Every probability per word is treated the same.

However, depending on the context some words are more important than others. Let us say we have a title like: *"Blindness. ISBN: 9707311150"*. Due to the ISBN code[4], we expect this item to be a book. If the word *blindness* occurs more frequently in movie

[4] ISBN stands for International Standard Book Number. It is a unique numeric commercial book identifier. Though, audio books in particular can have ISBN as well.

titles, perhaps this phrase will be closer to other movies in the feature vector space. We would like to emphasise the *ISBN* word.

In our dataset we found that 2% of the products are books and less than 1% are audiobooks. Prediction errors are prevalent and very frequent in the two subcategories, i.e., audio books and books ($\approx 5\%$). This happens due to the diversity and ambiguity in books titles. If a book has "Tchaikovsky's Suites" as its title, there is chance that it will be classified as an audible product (i.e. CD, DVD, or similar), instead of as Sheet Music (book). Cases like these occur, throughout the catalog of categories.

In Chapter 4, Section 4.2, it was shown that a number of available/used features in Figure 4.2 can be retrieved from products. After running some empirical tests, we noticed that some of these features, *Product Search* and *Description*, contained the information about the ISBN when it was a book. Therefore, an extra binary element was concatenated to the created (word-to-) vector (see Section 5.3).

Let $f_j \in F$, be any of the before mentioned features, i.e., $F = \{productSearch, description\}$, and p_i a product (i.e. $p_i \in P$), then p_{ij} refers to the feature f_j that belongs to product p_i. The function $isBook(p_i)$ can then be generalized as:

$$isBook(p_i) = \cup_{j=1}^{|F|} \mathbb{1}_{contains(p_{ij},''ISBN'')}{}^{5}.$$

Both functions $isBook(p_i)$ and $contains(p_{ij}, x)$ are a Boolean functions, with $p_i \in P$, and x a sequence of characters.

The function $contains(p_{ij}, x)$ is defined as:

$$contains(p_{ij}, x) = \begin{cases} true, & \text{if } x \text{ can be found in } p_{ij} \\ false, & \text{otherwise,} \end{cases} \tag{6.4}$$

where p_{ij} is the value of the feature j in product p_i.

Analogously, a binary element was added to the vector when *"hörbuch"* (audiobook) was found.

With these two changes, an improvement of 1% (together) was observed (Table 6.5). After this change, our vectors sizes changed from 1000 and 750, to 1002 and 752, for the first and grained level classification features, respectively. It is important to note that these kind of tweak can be implemented independently at any level of the classification, i.e., we could only apply this to the feature vectors of an arbitrary branch.

Moving Subcategories

In Section 4.3, we introduced the hierarchical problem, concerning the presence of very similar categories, that are located in different tree branches (e.g., the category *Smartwatches* is under the *Telekommunikation* branch, and category *Sportuhren* (sport

[5] $\mathbb{1}$ is an indicative function. It returns 1.0 if '$contains(p_i,'' ISBN'')$' is true, and 0.0 otherwise.

watches) can be found under the *Mode und Accessoires* category). When we perform the second classification phrase, i.e., the grained classification step, we assume that categories that are close to each other in the hierarchy have a lot more in common with each other than those that are far apart. However, this is not necessarily true.

For this reason, we would like to move the whole subtree formed by the *problematic* category to a more intuitive branch.

For example, 1.8% of the data belongs to the branch formed from *Sportbekleidung* (Sport Clothes). This is a subcategory of *Freizeit & Outdoors* (Leisure and Outdoors), and not from *Mode & Accessories* (Fashion and Accessories)[6].

In many cases, determining if a piece of cloth is meant to be used for practicing sports or for something else, is a difficult choice. For example, in Figure 6.8 we show some t-shirts with a skiing motive: Does this mean that the person who wears it will want to wear it while practicing her/his favorite sport?, or is it worn for everyday purposes? Should it be simply classified as any fashion item?

Fig. 6.8. Ski T-Shirt: For skiing or for Everyday Use?

This is a very subjective classification decision. Therefore, this subcategory (*Sportbekleidung*) was moved since it would naturally belong to Fashion and Accessories, instead of Leisure and Outdoors, as shown in Figure 6.9. There, we show the initial position of the whole subtree of *sportbekleidung* in *Freizeit und Outdoor*'s branch (first tree). And then, the result after moving *sportbekleidung* to some more intuitive place, such as *Mode und Accessoires*'s branch this is shown in the second tree.

Unfortunately, this doesn't completely solve our problem: There are other subcategories that don't originally belong to *Fashion and Accessories* and still contain articles of clothes. For example, the *Schwimmsport* (swimming) category contains some shorts for swimming and belongs to the *Freizeit und Outdoor* category. Very similar shorts are found in the subcategory *Bademode* (swimwear), which belongs to the first level category *Fashion and Accessories*. We also moved this category, similarly as we did with the *Sportbekleidung* subtree. The results are shown in Table 6.5 in column "Schwimm.".

Similarly, all of these subcategories were moved to *Haushalt und Wohnen*:

- *Fotozubehör*. This category contains portraits, photo albums and similar items and was originally placed under *Fotografie*. However, categories that *Wanddekoration*

[6] Funny thing: there is a category called *Freizeitbekleidung* (leisurewear), that is a subcategory of *Mode und Accesoires*, and not *Freizeit und Outdoor*.

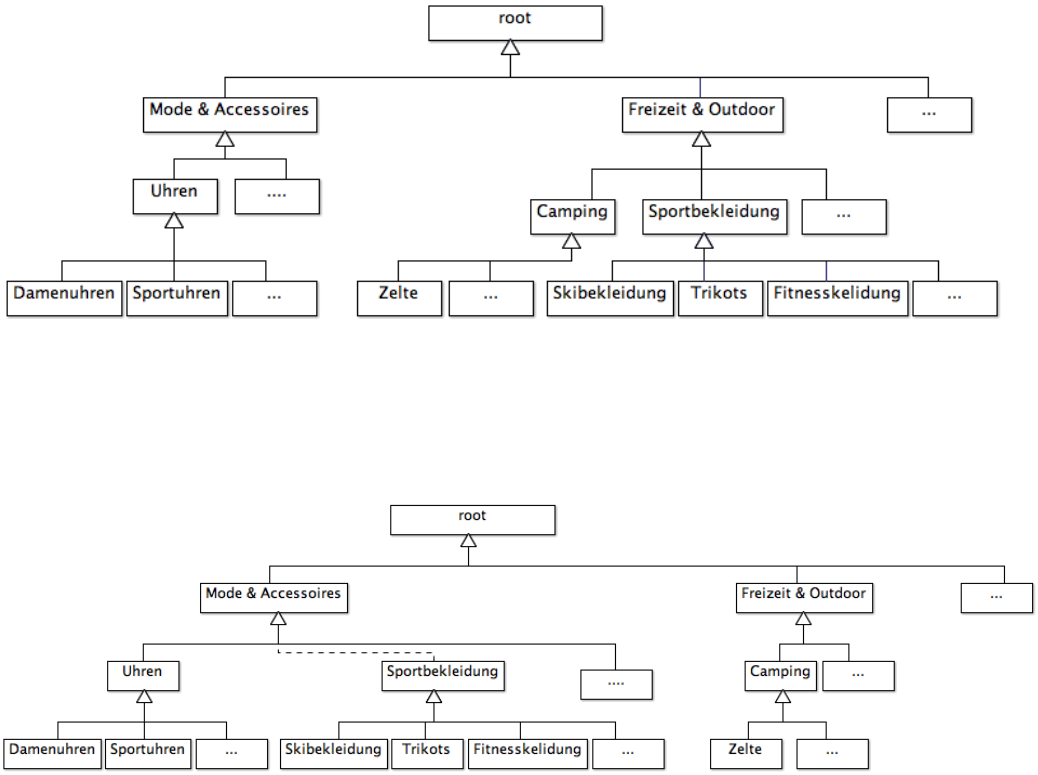


Fig. 6.9. Moving of Subcategories in the Tree

Exp.	**Books**	**Bekleid.**	**Schwimm.**	**Wohnen**	**Gartenmöb.**	**A. books**	F_1**-S. [%]**
1							0.873
2	x						0.873
3	x	x					0.884
4	x	x	x				0.885
5	x	x	x	x			0.890
6	x	x	x	x	x		0.892
7	x	x	x	x		x	0.899
8	x	x	x	x	x	x	**0.900**

Table 6.5.: Domain Changes Results. Note that an "x" on cell C_{ij}, where i are rows and j are columns), means that the category (or group of categories being referred in column j, was/were included in experiment i.

(posters) and *Wohnideen* (home decor), which are more related to photo albums, for example, belong to *Haushalt und Wohnen*.

- *Kinderzimmer* is located in the *Freizeit und Outdoor* branch. It includes all types of furniture for children's rooms. However, furniture in general is found in *Haushalt und Wohnen*, and therefore moved there.
- *Gartenmöbel*, *Badewannen*, *Waschbecken*, *Duschkabinen*, *Küchenarmaturen* and *Spülen*, that belong originally to *Heimweken und Garten* were moved.
- *Aufkleber und Sticker* is placed under *Freizeit und Outdoor*. This type of product was often mistakenly classified as *Wanddekoration*, which belong to *Haushalt und Wohnen*.

The results of these experiments are depicted in Table 6.5. In the "Wohnen" column of this table, we include all these changes, except the ones related to *Gartenmöbel*. The latter is in column "Gartenmöb." It improves our results by 1 percentage point.

Table 6.5 summarizes the results of the experiments made using domain knowledge. Books and Audio (A.) books columns refer to the experiments adding one binary element to our feature vector (i.e. Section 6.4).

The rest are experiments on moving subcategories. In the table, the column "Bekleid." (for Bekleidung) doesn't include the *Sportbekleidung* category (i.e. Figure 6.9), but all of the categories that didn't originally belong to *Mode und Accessoires*, but referred anyway to clothes (e.g. *Berufsbekleidung*).

These experiments were run on the dataset A, as before we use Spark Multinomial Logistic Regression implementation with its default values. All five features were selected. This results correspond to classify the products into the first level categories.

The best results were obtained when adding a binary element for both: books and audio books, and simultaneously changing all the subcategories to places where they (arguably) more naturally belong to.

Many of these problems can't be solved easily. There are categories, like *Kletterzubehör* (climbing accessories) that is host for products like climbing gloves, helms and chalk bags. These items are more related to *Mode und Accessoires* than to *Freizeit und Outdoor* - where the category belongs-. However, this category also have products, such as chalk, brushes and rockhammers, which definitely don't belong to *Mode und Accessoires* (see Figure 6.10). A solution like the moving subcategories trees, explained earlier wouldn't work in this case, since not all the products in the category belong completely to the first level category. This is reflected in the results: No matter where we place this category, mistakes labeling its items are made. On the left part of Figure 6.10 we see a chalk package which doesn't fit into the *Mode und Accessoires* category. But on the right hand, we have some gloves that should be placed somewhere in that branch. Both belong to *Kletterzubehör*, which is in the *Freizeit und Outdoor* branch.

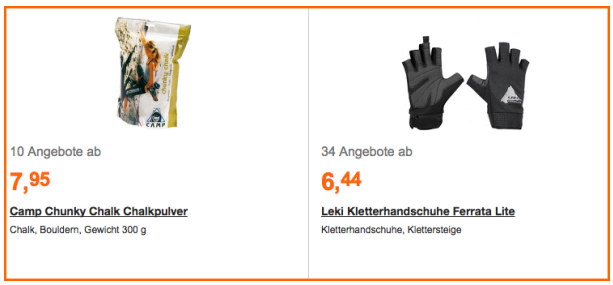


Fig. 6.10. Climbing Products Example

Dimensionality Reduction

We explore the possibility of reducing the dimensionality (from the 1002 and 752 we have, depending on the classification level). PCA is a common algorithm used extensively for these types of tasks. The goal is to find orthogonal dimensions where the variance is maximized, and greater discrimination is possible. In our preprocessing dataflow, we see our location in Figure 6.11.

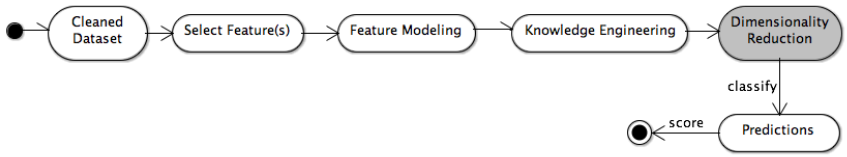


Fig. 6.11. Preprocessing steps: The gray box indicates where we are.

The principal difficulty for this algorithm is to know how many dimensions (i.e. principal components) we want to use. For this, it is useful to plot the eigenvalue spectrum. Higher eigenvalues are related to the firsts eigenvectors (i.e. principal components). The eigenvalues indicate how much variance can be explained by its associated eigenvector. Plotting the eigenvalue spectrum is just a simple visual way to detect how many eigenvalues have a significant variance. We can then select the number of principal components based on that, see Figure 6.12.

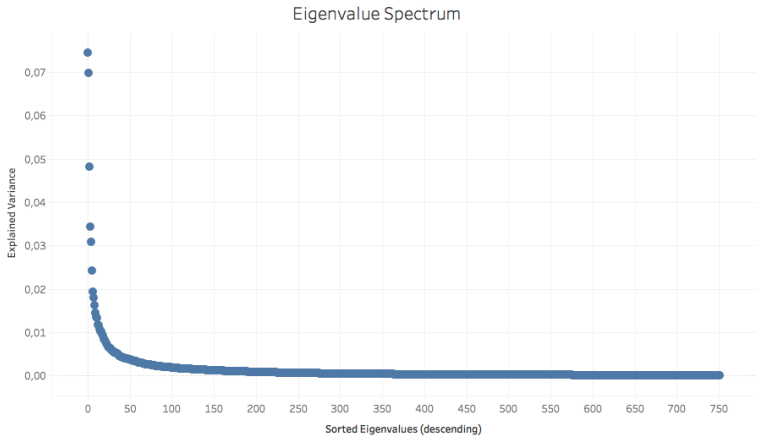


Fig. 6.12. It can be seen that two eigenvalues are have a notable difference in their values, with respect to the others.

Based on what is seen in Figure 6.12, we selected 2, 3, 6 and 110 principal components, and made experiments. Table 6.6 summarizes the results.

These experiments were run in the local environment, and in the first level classification phase, however using vectors of size 750 (this time). We use the dataset A, locally. All five features were selected. As usual, we use multinomial logistic regression with its default values.

PC	F_1**-Score**	**Time[min]**
2	0.46	4.58
3	0.46	6.95
6	0.45	6.94
110	0.52	7.10
without	**0.90**	8.28

Table 6.6.: Experiments with PCA

The difference between the eigenvalues is not that big. Also, the explained variance is not very high. This could explain why we don't have a very good performance by projecting the data into the chosen principal components (Table 6.6), as the one we have by using all the dimensions. However, this behaviour was expected from the PCA

projections we got in Figure 5.6, where all categories appear to overlap. We didn't reduce our data set dimensionality.

First Level Classification

Classifying products into the first level categories is very important for accuracy. If a mistake is made at this level, the error will propagate down to the more specialised classification tasks, where no matter which classification procedure is made or how good is the classifier, the result will be wrong. For this reason, a lot of experiments were made over the first level classification task, including the exploration of different algorithms with different parameters. Because having a good performance is particularly important at this level, we focus especially in solving this part of the problem well.

For knowing how much data we need for training, we made an experiment with this goal. This was done for data sets A and B. Both datasets were split with different train and test set proportions. Performance on the first level classification using multinomial logistic regression was evaluated afterwards.

These experiments were executed in the cluster (see Section 6.2 for information about the cluster). The times refer to training and testing time together.

For dataset A, we saw more or less similar performance having a training set represented by 70 and 80% of the data set. But an increase of 1 percentage point with 90%. Table 6.7 summarizes these results.

Proportion	**Train Samples**	**Test Samples**	F_1**-Score [%]**	**Time [min]**
70:30	65163	27949	0.89	4.27
80:20	74470	18642	0.89	4.16
90:10	83767	9345	0.90	4.32

Table 6.7.: Experiments with the number of samples in the small (first) data set.

With the large data set B, we obtained similar results using different training sizes. This is probably because there are already enough samples that have only 70% for creating a *good* model. It already converged with 70% of the samples. Table 6.8 summarizes this results.

Proportion	**Train Samples**	**Test Samples**	F_1**-Score [%]**	**Time [min]**
70:30	412571	176863	0.863	9.158
80:20	471549	117939	0.864	9.65
90:10	530383	59105	0.868	9.3

Table 6.8.: Experiments with the number of samples in the large data set.

Here we can already notice that the results using the dataset A seems to be better. This was unexpected as it is believed that the dataset B was better constructed (in terms of correctness of the labels). From this we can conclude that the misclassification problems described in Section 4.3 are not that severe.

Earlier in this chapter we introduced the dataset B. In Figure 6.1 we showed how features are not as present in this data set, as they are in the small one (see Figure 4.6 in Chapter 4)

The dataset B doesn't have as many descriptions as dataset A. In general, the first data set contained all features in approx. 30% of its products, while the large one only in around 10%. This led us believe that this could be one of the reasons dataset A serves better for our classification purposes.

Having misclassification problems (like the ones present in the small data set) affects less the final output from the classifier (i.e. the F_1-Score) than lacking of some useful features, such as descriptions. Of course, probably the best would be to have both: A good labeled data set and all features present.

Afterwards, for a final experiment we combined both of the datasets. First we used the small one for training, and the large one for testing the model. With this, we got an F_1-score of 0.78. Then, we did the opposite. This gave us worse results. Finally, we combined both and divide them the train and test set (so every set have samples from both data sets). This outperformed the two experiments from above: the F_1-score was 0.80. However, this was not better than 0.86 or 0.90 (results from above, where data sets are used independently).

This is still a good scenario for us, as dataset A better represents the type of products that need to be classified, see Section 3.2. The products in the dataset B are already treated by company's experts (and therefore there is no need to pass them through the classifier).

Comparison between Algorithms (in first level)

The chosen algorithms for the first level classification include multinomial logistic regression (see Section 5.5), random forest (see Section 5.6), k-nearest neighbors (see Section 5.4) and SVM (see Section 5.7). As explained in Chapter 5 all are part of the Spark machine learning Library except for kNN, which was implemented for this work.

Each of these four algorithms have different hyperparameters that need to be tune with cross validation.

The process for evaluating any of the first level models (i.e. any of the before mentioned four algorithms) is:

1. Select hyperparameters to try (depending on the algorithm).
2. Run 3-fold cross validation on 10% of the data set A with the selected hyperparameters. At the end, we should have, the set of hyperparameters that performs well (i.e. maximize the F_1-Score).
3. Train 90% of the data set A with the chosen hyperparameters.
4. Test with dataset C.

We will shortly explore the hyperparameters that were chosen for algorithms we employed.

kNN

For kNN when choosing the proper k a good rule of thumb, according to Duda et al. [DHS+01], is $k_n = \sqrt(n)$, where n would be the number of products. Since $n \approx 90k$, k should be around 300. After some empirical tests, we found 250 to be a better estimate. We also experimented with weighted kNN, however, the results were around 10 percentage points lower than the traditional kNN. Therefore, it was no longer considered. On average, the F_1-Score for kNN (with k=250) was 0.79% when performing first level classification (see Table 6.9).

However, considering only those predictions whose probability was higher than 80%, the F_1-Score with the same set up was around 0.90%. That is, its posterior probability is:

$$P(y|x) \geq 0.80,$$

where x is the feature vector of a product, and y is the category we want to predict.

SVM

Use of an SVM classifier in a multinomial context requires the use of a generalization technique to "convert" the binary problem into a multinomial one.

As explain in the last chapter, we implemented a One Versus the Rest (OVR) approach. This approach is sometimes also called, One versus All. This was selected over the similar, All Versus All (AVA) approach, because the latter requires the construction of more models. This approach is also called One Versus One. With OVR we only need to train as many models as we have classes in our case it would be 13. The OVO approach requires us to build a binary model for every pair of classes, for us it would be $\frac{13\,(13-1)}{2} = 78$. ([HPK11]).

For all of the 13 models we used the same regularization parameter (for the L2 optimizer), i.e. λ, and the maximum number of iterations. This could have been optimized (e.g., choosing a regularization parameter) that is better suited. The values were $\lambda = 0.002$ and a maximum of 100 iterations. This was explored using cross validation. For lambda, we tried values between 0.0001 and 1.0. As for the number of iterations, we considered values between 50 and 130.

Random Forest

The hyperparameters to create a random forest model, we need to set hyperparameters such as the number of trees in our forest and their depth. Other parameters such as maximum number of bins, the subsampling rate and the feature subset strategy were left with their default value.

For the number of trees we tried values that oscillate between 10 and 250. And for the depth, values between 5 and 15. The best results were 70 trees with a maximum depth of 15.

We didn't try higher values for the max. depth parameter because it would take much longer to train and we wanted to avoid overfitting the model.

Multinomial Logistic Regression

For the multinomial logistic regression algorithm, different hyperparameters were considered.

- Regularization Parameter: Values between 0.0001 and 1.0, where the best one for the first level classification problem was 0.001.
- Maximum Iterations: Values between 20 and 200, where the best value was 70.

An L2 updater was used as an optimizer. The rest of the hyperparameters were set to be their default values.

Larger weights in the Logistic model indicate increased complexity and fragility. It is desirable to keep weights small. Figure 6.13 shows the distribution of the values of the weights of the model used for the first level classification. They present a more or less symmetric distribution around zero.

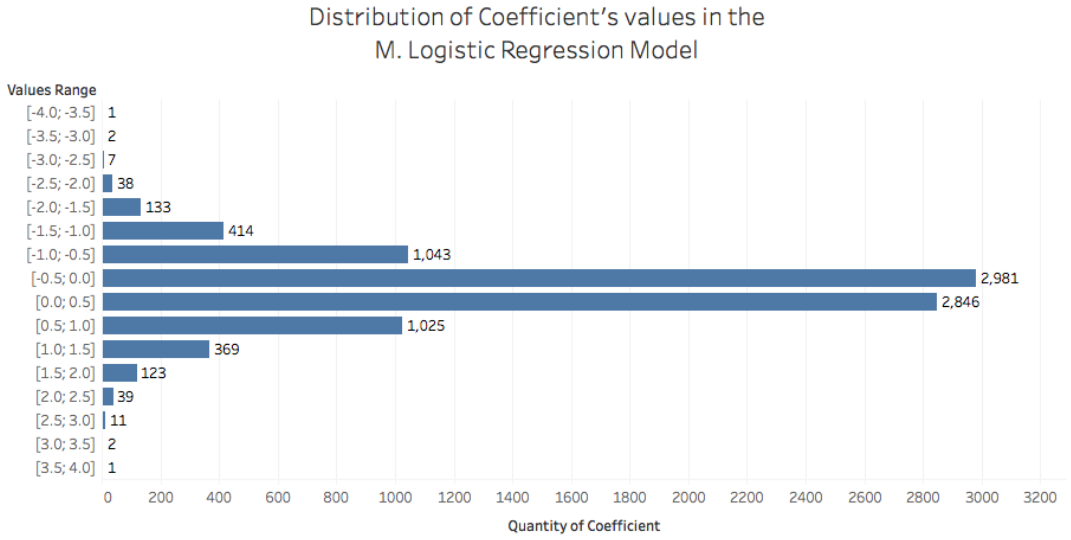


Fig. 6.13. Distribution of Coefficient Values in the Multinomial Logistic Regression Model

The Spark Multilayer Perceptron algorithm was also employed. It behaved as well as the Logistic Regression, with the input and output layers set to 1002 and 13, respectively, without any hidden layer. This should come as no surprise, since the nodes in the intermediate layers use sigmoid (logistic), and in the output the softmax function, which is the way we calculate the probability for the possible outcomes. No extensive experiments were conducted with a varied number of hidden layers. However for the

Algorithm	**F_1-Score [%]**	**Training Time [min.]**
kNN	0.79	83.20
wkNN	0.66	100.21
MLR	**0.93**	**18.54**
RF	0.69	44.19
OVR-SVM	0.75	33.67

Table 6.9.: First Level Classification Results. Multinomial Logistic Regression appears to outperforms the other ones.

ones we executed, they did not surpass the logistic regression results, thus they were not reported.

MLR outperformed all of the algorithms at this stage of the classification. Hence, the rest of the experiments involving the first level utilized MLR.

At this point, it is interesting to investigate, similar to what was done with kNN above, the proportion of samples that were classified with MLR with a certain amount of probability.

Taking a proportion 80:20 for training and testing from dataset A, respectively, we classified the test set samples into the first level categories. Then, we calculated the probabilities of each of the predictions. We grouped all those that were between 0 and 10%, 10% and 20%, 20% and 30% and so on. Then, we calculated the errors in each of the probability buckets. Of course, this must be seen jointly with the quantity of products which the predicted label probability in the respective bucket. This is shown in Figure 6.14. There we can see that 90% of the products (i.e. $\approx 20k$) were classified with 95% probability.

Let x be a feature vector, such that $x \in \mathbb{R}^f$, f is the features space dimensionality, and X a set with all the feature vectors in the test ($\approx 20k$ of products). The label assigned by our classifier is y, and the real label is l. Then, for calculating the last error bucket (i.e. where $p(y|x;\beta) \geq 0.9$ we use:

$$\frac{\sum_{x \in X} \mathbb{1}_{P(y|x) \geq 0.90 \wedge y \neq l}}{\sum_{x \in X} \mathbb{1}_{P(y|x) \geq 0.90}} = 0.049$$

The calculation of the other buckets is analogous.

If we calculate this for probabilities between 80% and 100%, we find 5.90% of the errors. This last calculation is important for idealo's decision, whether to consider (or not) the classifier prediction. If a prediction was done with a probability higher than 80%, it will be considered correctly classified. Then, at least in the first level classification step, there will be an error less than 6% of the times.

We have seen the overall F_1-score for the first level classification with MLR. However, something that also may be worth to explore is the F_1-score for every first level category. In Table 6.10 the F_1-Score for every first level category is shown. This is the result of calculating the multinomial logistic regression algorithm with the best hyperparameters

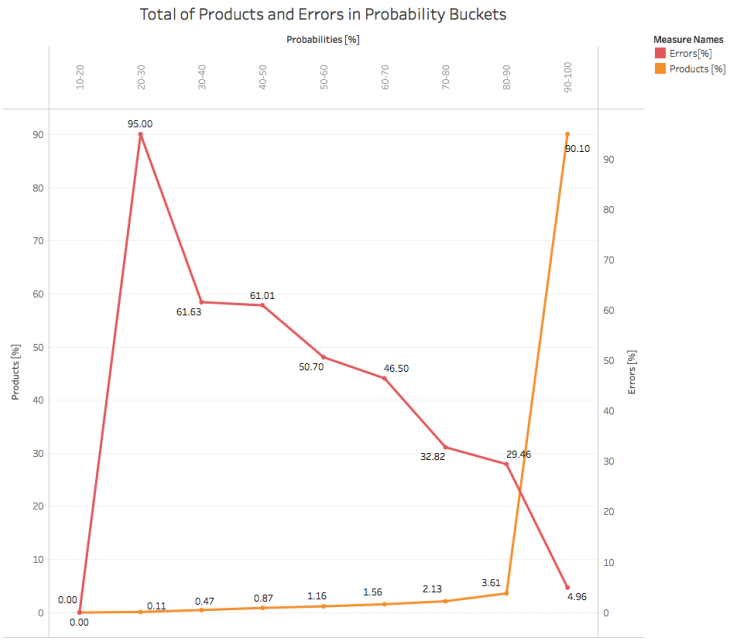


Fig. 6.14. The majority of the products are labeled with a high probability of being right.

on the first level. We also included in the table the percentage of products that were classified into each category. Categories where more products were assigned, have in general higher F_1-score which is of course good for us.

Finally, in order to investigate the correlation between categories Table 6.11 shows the confusion matrix of the results in the first level. The diagonal of a confusion matrix represents the correct predictions. In the rows we have the true categories with its false negatives, and in the columns the predictions made by the classifier with its false positives, e.g., 94 products labeled as *Mode und Accessoires* (true class), were said to belong to *Freizeit und Outdoor* by the classifier (prediction). Here we can see high values in both relations between *Mode und Accessoires* and *Freizeit und Outdoor*. In general, *Freizeit und Outdoor* seems like a complicated category. This category is the cause of false positives with almost every other category. Going back to the PCA projection we did in Section 5.3, we can confirm what we saw there about *Freizeit und Outdoor* overlaping the other categories. As well as our hypothesis about the category *Haushalt und Wohnen*.

In general, it is important to notice that all first level scores are greater than $\frac{1}{13} = 0.07$, which corresponds to our first baseline, i.e. just guessing the category. Further, our more populated first level category is *Mode und Accessoires* with $\approx 27\%$ of products (see Figure 4.4). It seems that we don't have any problem related to classifying everything to this category.

Category	**Products [%]**	**F_1-Score [%]**
Mode & Accessoires	24.50	0.95
Freizeit & Outdoor	18.00	0.87
Haushalt & Wohnen	15.18	0.91
Auto & Motorrad	14.00	**0.96**
Heimwerken & Garten	9.00	0.85
Computer & Hardware	6.60	0.89
TV, Video, DVD	3.60	0.93
Gesundheit	2.20	0.88
Spielzeug & Games	2.17	0.82
Fotographie	1.60	0.75
Wellness & Beauty	1.31	0.71
Telekommunikation	1.15	0.82
HiFi & Audio	0.70	0.66

Table 6.10.: First Level error distribution over categories. The category with highest score is *Auto und Motorrad*. The one with the lowest score is *HiFi und Audio*. This might be because it is one of the categories with less products.

True Class \ Prediction	Fotografie	Telekom.	Computer and HW	TV,Video,DVD	HiFi and Audio	Haushalt und Wohnen	Auto and Motorrad	Spielzeug and Games	Freizeit und Outdoor	Heimwerke and Garten	Gesundheit	Wellness and Beauty	Mode and Accs.
Fotografie	72	2	6	1	2	3	2	0	8	10	0	0	1
Telekom.	0	155	20	1	1	0	5	0	4	4	0	0	19
Comp. und HW	3	2	1086	9	7	23	2	2	42	19	0	0	14
TV,Video,DVD	1	2	9	621	3	0	0	0	26	8	0	0	0
HiFi und Audio	1	2	7	9	81	8	4	1	9	4	0	0	12
Haus. und Woh.	1	0	12	2	4	2878	22	8	96	97	3	5	26
Auto und Motorr.	0	0	4	2	1	17	2469	1	17	33	1	0	9
Spielzeug	0	0	7	0	1	12	4	277	41	4	0	0	7
Freizeit	0	0	31	11	0	62	9	12	2484	35	12	7	111
Heim. und Garten	3	2	13	0	2	93	20	6	45	1306	4	3	12
Gesundheit	1	0	0	0	0	3	1	0	12	7	316	13	6
Welln. und Beau.	0	0	1	0	0	8	1	0	16	11	15	133	12
Mode und Accs.	3	4	9	2	2	26	8	8	94	9	8	14	4905

Table 6.11.: Confusion Matrix in First Level

The results from this section were higher than expected. In other works in similar domain (e.g., eBay items [SRS12]), and different domain (e.g., web content [DC00]) the results for the first level classification were lower. However, as we didn't use our method with their data this is inconclusive.

Grained Level Classification

At this stage of the classification task, we need to create 13 different models: One for each of the 13 categories that we have in the first level.

In order to train each branch, the data set A was filtered to have only the products that belonged to the respective branch. The quantity of products per branch can be seen in Figure 4.4. Afterwards, the data set was split in a 90-10 train-test proportion. This way, every branch was tuned independently.

Table 6.12 summarizes the results of logistic regression in every branch, with its respective hyperparameters. For all of the branches, the F_1 Score of the models using the same training sample was higher than 0.90. This, in order to assess underfitting. Since the score was high, we can discard this problem. In particular categories *HiFi und Audio* and *Wellness und Beauty* have a very low score. This might be due to small amount of products we have for training them.

Category	**Params**	**‖Train‖**	**‖Test‖**	F_1**-S. [%]**	**Train[min]**
Mode und Accessoires	(1.0E-4, 70)	23021	2588	0.73	3.69
Freizeit und Outdoor	(1.0E-4, 120)	13063	1516	0.83	4.79
Haushalt und Wohnen	(5.0E-6, 100)	14410	1504	0.80	4.48
Auto und Motorrad	(2.0E-4, 55)	11698	1272	0.78	1.76
Heimwerken und Garten	(1.0E-3, 100)	6941	733	0.56	3.40
Computer und Hardware	(5.0E-6, 100)	5681	579	0.69	2.39
TV, Video, DVD	(5.0E-4, 80)	3077	333	**0.87**	1.34
Gesundheit	(1.0E-4,150)	1621	170	0.64	1.95
Spielzeug und Games	(1.0E-3, 100)	1629	174	0.55	1.63
Fotografie	(7.0E-5, 61)	445	55	0.69	1.24
Wellness und Beauty	(5.0E-5, 80)	859	101	0.44	1.35
Telekommunikation	(3.0E-7, 80)	908	105	0.78	1.28
HiFi und Audio	(5.0E-6, 100)	611	68	0.44	1.27

Table 6.12.: Logistic Regression in Branches

We noticed that a lot of the errors are subtle, such as classifying an item as a *Book*, when it belongs to a more general category like *Reading and Literature*. Later, we will calculate the overall error, taking this into account.

In this level of classification, categories are more similar between them. Table 6.13 summarizes the most common errors found in each branch.

Category Branch	**Real Category**	**Predicted Category**
Mode & Accessoires	Herren-Shirts	Sportbekleidung
	Herren-Shirts	Funshirts und Fanshirts
	Handytaschen und Covers	Tablet-Taschen
Freizeit & Outdoor	Bücher	Lesen und Literatur
	Schulbücher und Lernhilfen	Bücher
Haushalt & Wohnen	Küchengeräte-Zubehör	Dekoschalen
Auto & Motorrad	Autoteile	Auto-Karosserien und Schließanlagen
Heimwerken & Garten	Fenster	Rollläden
	Bohrer & Bits	Fräsmaschinen
Computer & Hardware	Druckerpatronen	3D-Drucker
TV, Video, DVD	Audio-CDs	Schallplatten
Gesundheit	Medizintechnik	Medizinische Instrumente
	Ärzte- & Krankenhausbedarf	Injektion
Spielzeug & Games	Babyspielzeuge	Spieluhren
	Gesellschaftsspiele	Modelleisenbahn-Zubehör
Fotographie	Spiegelreflexkameras	Systemkameras
Wellness & Beauty	Herrendüfte	Damendüfte
Telekommunikation	Handy-Ersatzteile	Smartphones
HiFi & Audio	Audiokabel	Vorverstärker
	Audiokabel	MIDI-Controller

Table 6.13.: Common Errors

It is interesting that the categories that behaved better during the first level classification (see Table 6.10), aren't necessary the ones that performed the best in its branch, (e.g., *Auto und Motorrad*).

For the grained level classification, we only considered the Multinomial Linear regression because it was the one with better results. However, the beauty of this method is that one can play with the components as one was building a Lego structure: There is no need to use the same color for every part. As we aren't tied to use one specific method or algorithm for every branch.

From the performance point of view, training the 13 branches can be done in parallel. This is an advantage over the flat classification approach. Further, the grained classifiers will be smaller and faster to train than building a model that considers all the $\approx$ 1700 categories at once.

Overall Experiments

We ran different experiments on the whole classification task (i.e. classifying products through both stages). In Listing 6.1 a simplified pseudocode for this is presented. Some intermediate steps are ignored, such as converting our labels, i.e. category ids to the real positive numbers Spark needs. The *grainedLevel* function is also presented in a very simplified way in Listing 6.2. Just to show how, despite of having several classifiers (depending on the branch we are classifying into), we compute the final prediction in a parallel way.

Listing 6.1: Pseudocode for whole classification process

```
//load test data
val testData = sqlContext.read.json(...)

//load classifiers
val coarseClassifier = loadClassifier(...)
val firstLevelCategories = ...

//array of pairs with catId and grained classifiers
val branchClassifiers = firstLevelCategories
    .map( catId => (catId, loadClassifier(catId, ...)) )

//coarse level classification
val coarsePrediction = coarseClassifier.predict(testData)

//grained level classification
val broadClassifs = sc.Broadcast(branchClassifiers)
grainedLevel(coarsePrediction, broadClassifs, "firstLevelPred")
```

Listing 6.2: Pseudocode for grained level Classification

```
grainedLevel(test:DataFrame,
    models: Broadcast[Map[Long, ClassifierModel]],
    branchCol:String){
  test.map{ x =>
     val thisModel = models.value.get(x.getAs[Long](branchCol))
     thisModel.predict(x)
  }
 }
```

In the experiments from this section, we are not tuning models anymore. We already have our models built. The goal here is to evaluate the performance of the whole classification process together, and explore different interpretations of the predictions to improve the overall $F_1 - score$.

We cleaned and selected all the five features discussed before for every item in the dataset. Afterwards, we only considered word embeddings of size 200 for the first level classification. And for all the branches we used word embeddings of size 150. This size is proportional to the number of variables we need to estimate for our models. As we don't have too many samples for training, we thought it would proper to reduce its complexity by reducing the word embedding size of the features that are used for creating the model of this classifiers. We made all domain specific changes explained previously. And then we classified.

The test set i.e., the complete dataset C, is classified with the first level classifier (i.e. Multinomial Logistic Regression), into a category in the first level. Then, depending on the first level prediction, every test sample is "sent" to its respective grained classifier. For example, if a *Pictionary* was labeled by the first classifier as *Spielzeug & Games*, it will continue its classification process with this branch classifier. As mentioned several times, one downside of this method is that we are propagating the errors made in the first level to the others. We ran experiments with the dataset C, which is very similar to the one introduced in Chapter 4. It has 84162 products (i.e. 282MB raw). This way, we can assure that our results are reliable for the company's data.

As seen before, the first level classification F_1-Score was 0.93 (see Table 6.9). This means that we can expect around 7% of the items (i.e. $\approx 7k$) to be classified into the wrong branch for the second classification step. This was expected from the first level experiments results in Section 6.4.

Next, a few overall experiment designs will be explained. Afterwards, the results will be shown and explained.

Overall Experiments: First Level Filtered

Previously (see Figure 6.14), we saw that most of the products were classified correctly in the first level, when they were associated with more than 90% of probability. In any case, only products with an overall probability (i.e. after having passed the whole classification process) greater than 80% will be considered, see Figure 3.1. That's why

we filtered out those products that during the first level classification step didn't have a probability greater than 90% (arbitrarly chosen). This means that we will send directly between 4%-5% of the products to be classified by alternate means, and don't count them in the final results.

Overall Experiments: In Path

Some products are originally classified in a category that is more general than the one where our classifier predicts. When this happens, we say that the real category (or label) is in the path from the predicted one to the root of the category tree (see subtle misclassification in Section 4.3). In these experiments the results were assumed correct when this happened. One percentage point increase in the total score, can be seen, see Table 6.14.

Overall Experiments: Siblings

In many cases, the real category and the predicted category were very close in the category tree. For example, we saw in Table 6.13 that *Handytaschen Cover* and *Tablet-Taschen* were often confused with the other. Both categories belong to the category *Taschen* (which is in *Mode und Accessoires*). In this section of experiments we then accepted as correct classified those products that were classified into a certain category instead of one of its siblings category. This improved the overall classification by 10 percentage points, see Table 6.14. This tell us that we are very often in a category that is close to ours. Even if we don't classify the product into the correct category, we could be in the case where the item could be classified in more than one category, or is just a subjective case. Or course, it could be too that it's wrongly classified, since many sibling categories are not very related.

In order to measure this, Sun et al. ([SL01]) proposed a metric for evaluating the prediction of the classifier, that includes a cosine distance calculation between the *real category* and the one predicted. Sun et al. proposal sounds promising, and could be worth trying in the future.

Overall Experiments: Top Experiments

After running the whole experiment through the two stages of classification (i.e. first level and branches), the top 2, 3 and 5 categories for every product was calculated. That is, not only one prediction was made, but the top X (e.g. 2, 3 or 5) predictions of a category for a product. In previous works ([SRS12], [WHS16], [SRMS12]) experiments like this were calculated. Two (or more) categories can be very similar. Since the products can only be assigned to one category (i.e. it is not a multilabeling problem) "errors", such as predicting a category instead of other that is very similar, are common. With this experiment we attempt to inspect how far the model was from knowing the "truth".

Flat Classification

The flat classification experiment from Table 6.14, was done with multinomial logistic regression. The regularization was set to 0.0001, and a maximum of to 100 iterations. The model as usual, was trained with 90% of the data set presented in Chapter 4. However, the shown results like the others, come from the data set C.

Finally, The Results

Table 6.14 summarizes the three overall experiments. The F_1-Score column refers to the final prediction without considering the path, siblings or top. Different subsets of the data were used. The *Overall* row represents the whole data, i.e. no subset. Next, we filtered the data as explained earlier. The third experiment was explained also above. The last two (*Without Rares* and *Filtered without rares*) are experiments that exclude rare categories (i.e. those with less than 10 products in the dataset used for training, i.e., dataset A). The first one just exclude rare categories. In the latter, we made the *Filtered Overall* experiment, but this time without using the rare categories.

Experiment	**Final F_1-S.**	**In Path**	**Siblings**	**Top 2**	**Top 3**	**Top 5**
Overall	0.67	0.68	0.78	0.73	0.76	0.80
Filtered Overall	0.71	0.72	0.82	0.77	0.81	0.84
Flat	0.58	0.58	0.72	0.66	0.69	0.73
Without Rares	0.70	0.71	0.81	0.76	0.80	0.83
Filt. and without rares	0.73	0.74	0.84	0.80	0.83	0.86

Table 6.14.: Overall Experiments Results

In order to compare, we also ran the kNN algorithm as a flat classifier. That is, we didn't consider the hierarchy, but we tried to classify directly into the target categories. We got an F_1-Score of 0.49 (with k=250). With our data, Multinomial Logistic Regression performs better than kNN in a flat setup too.

We can see how considering the TOP 2 improves a minimum of 6 percentage points in every experiment, compared to the TOP 1 ("Final F_1-score" column in the table). TOP 3 and 5 improve a minimum of 9 and 13 percentage points, respectively. If we consider our classification to be right, when the true label is in the "predicted path", we obtain results 1% better. Considering siblings normally pushes the score about 10 percentage points higher, however as explained before, this isn't completely fair as there are some siblings categories that are not as similar as others may be.

The experiments regarding *rares categories* presented in the summary Table 6.14 shows the results of running our whole classification task excluding rare categories (and products that were supposed to be classified there), i.e. we had 89090 products in the data set C. The F_1-score was between 2 and 3 percentage points better. For confirming that rare categories are not being classified that well, we executed the same experiment, but this time only with the products that belong to the rare categories (i.e. the complement

set with 3755 products, which is 4.4% of the items in dataset C). The F_1-score was: 0.14. With rare categories (there are 789 rare categories) the classifiers did not perform well.

But, where did all the errors come from? Specifically from which branches? In Table 6.15 we tried to answer these questions. We calculated how many errors were in every branch, that is:

$$\%Error(b_i) = \frac{(\text{Errors In Branch } b_i) \cdot 100}{\text{Test Datapoints}}.$$

Branch \ $Error(b_i)$	**Total** [%]	**In Path**[%]	**Siblings**[%]
Mode & Accessoires	**6.96**	**6.86**	**3.66**
Freizeit & Outdoor	2.22	2.17	1.06
Haushalt & Wohnen	5.28	5.26	3.12
Auto & Motorrad	4.29	3.79	2.94
Heimwerken & Garten	3.84	3.81	2.93
Computer & Hardware	2.01	1.97	1.42
TV, Video, DVD	0.56	0.53	0.23
Gesundheit	0.61	0.61	0.45
Spielzeug & Games	0.69	0.63	0.38
Fotographie	0.17	0.16	0.13
Wellness & Beauty	0.52	0.52	0.38
Telekommunikation	0.22	0.22	0.16
HiFi & Audio	0.27	0.27	0.25

Table 6.15.: Where do the overall **errors** come from?

In Table 6.15 we can see the impact of the siblings experiment. In the majority of the branches, the error decrease by half. Only in categories like *HiFi und Audio* or *Fotographie*, this measure doesn't help significantly. Notice that the same amount of products that is classified to a determinate branch in the first place, for all the three cases (overall experiment, in path experiment or siblings experiment) will be the same.

The classification performance taking the hierarchical approach is better than the flat one (see Table 6.14. We can conclude from this that tuning parameters for every phase, despite of having the error propagation problem (i.e. we can't recover later from an error done in the first level), can be more useful. This was expected, from the conclusions in many other works ([LYW+05], [SRS12], [DC00] [KS97])

Low classification performance in some branches might be due to the data sparseness problem, as we saw in Chapter 4, most categories don't have enough evidence for our algorithms to learn accurate models. In order to confirm this, we classified the same data set as in the experiments, but without considering those products that belonged to rare categories. With that, our overall F_1-score increased from 0.67 to 0.70.

Finally, we checked where were the most common errors after the overall classification experiments. In Table 6.16 we show some. There we point out if this misclassification could be solved by any of the methods described in our experiments (in path or siblings).

Real Category	Predicted Category	Can be solved?
Staubsaugerbeutel	Staubsauger-Zubehör	No
Bücher und Schulbücher	Lernhilfen	With Siblings
Sommerkompletträder	Kompletträder	No
Bücher	Reiseführer	With Siblings
Bücher	Biographien	With Siblings
Foto-Zubehör	Deko- und Porzellanfiguren	No
Wanddekoration	Deko- und Porzellanfiguren	With Siblings

Table 6.16.: Common Errors Overall

Many common errors can be solved then when we assume as correct categories predicted instead of their siblings (i.e. two categories are siblings, if they are sharing the same parent in the category tree). Table 6.17 shows the common errors after doing the experiment explain in Section 6.4:

Real Category	Predicted Category
Sommerkompletträder	Kompletträder
Herren-Shirts	Sportbekleidung
Lackpflege (from Auto & Motorrad)	Farben und Lacke
Autoaufbereitungsmittel	Farben und Lacke
Foto-Zubehör	Deko- und Porzellanfiguren
Motorradzubehör	Aufkleber und Sticker
Herren-Shirts	Damen-Shirts

Table 6.17.: Common Errors in Siblings Experiment

Efficiency

Before, we have always shown training times for our algorithms. In this section, we want to show how much time elapsed in order to classify products. These experiments were done using the cluster. As before, we need to prepare the data: clean it, select the features, tokenize them and transform them with a word2Vec model. Then we classified it into the first level. And finally, classified into the respective branch.

The lifecycle of a Spark job was explained in Section 5.2. However, as we want to run some experiments related to this, we will repeat the information that concerns us at this point.

First, we have the driver process, i.e., where the main method runs. It schedules the tasks, that it needs to convert from the user code first on the executors. Executors are processes that run in the worker nodes. They are responsible for running tasks, and sending the results back to the driver. They have in-memory storage, (e.g., for cached objects). Their lifetime is typically the duration of the application. The executors register themselves with the driver, once they start. They communicate directly with

the driver node. The workers, on the other hand communicate with the cluster manager, i.e. they share the knowledge about the availability of their resources. However, if the Standalone Spark's cluster manager is used (as is true in our case), the number of executors per worker node is usually 1.

As mentioned in Section 5.2, some flags can be used in order to control the number of cores and memory we want Spark to use. "Every Spark executor in an application has the same fixed number of cores and same fixed heap size" ([Ryz]). In order to run spark with differnet configurations, we set some flags when running spark-submit (http://spark.apache.org/docs/latest/configuration.html), such as

- executor-memory MEM: Memory per executor (e.g. 1000M, 2G) (Default: 1G). We experimented with 2, 6 and 10GB.
- total-executor-cores NUM Total cores for all executors. We experimented with values between 1 and 18. This sets also indirectly the number of nodes that Spark will be using, if the number of cores is smaller than the number of nodes (if the total of executor cores is set to 1, Spark can only use one node).

The values related to the driver were set to in their default values for these experiments: driver-cores (i.e. Cores for driver. Default is 1) and driver-memory (i.e. Memory for driver default is 1024M).

These values were selected arbitrarily. A deeper study should be made in order to determine which values are really the most optimum.

Let us make some remarks before we start the experiments. Changing the number of cores controls the concurrency level of our executors (how many simultaneous tasks can -each- run at the same time). Changing the memory heap, impacts on the amount of data that can be cached by Spark. Also is reflected on the maximum sizes of the shuffled data (e.g. when grouping, joining, etc) ([Ryz]).

As the driver node is a non-executor node, we can set the cores and memory that it will use independently (using –driver-memory and –driver-cores respectively).

The times in minutes are measured after spark context is created. And before analysing the results (i.e. calculating the F_1-score, statistics about the results, etc), because this won't be done during online classification phase. Basically we are measuring then, the time needed for all the steps mentioned in the first paragraph of this subsection to run.

Remember that we have 3 nodes, 6 cores per node: in total 18 cores. And 3 executors (one per node). As well as a total of 32GB of memory (10.8 in each node). Spark automatically distribute the maximum number of cores among the available nodes, e.g. if we say we would like a maximum of 6 cores, every node will use 2 cores (from 6 available). Table 6.18 shows the results of this experiments.

Setting more than 18 cores (i.e. physically not possible in our cluster), makes Spark to take automatically 0, which results in our job waiting forever. Running executors with too much memory often results in excessive garbage collection delays. It is important to remember to avoid allocating 100% of the resources (memory and cores), as the node needs some resources to run the OS and Hadoop daemons.

Nodes	**Cores in Total**	**Memory per node [GB]**	**Time [min]**
1	1	2	8.397
1	1	6	8.300
1	1	10	8.341
3	3	2	4.063
3	3	6	4.109
3	3	10	4.097
3	6	2	3.028
3	6	6	3.052
3	6	10	3.036
3	12	2	2.571
3	12	6	2.566
3	12	10	2.519
3	18	2	2.517
3	18	6	**2.365**
3	18	10	2.371

Table 6.18.: Experiments on the Cluster with number of cores and memory

The best time performance was seen when we used all the 18 cores of the cluster, with 6GB of memory. However, changes were more radical when we increased the number of cores, i.e., increased the level of parallelism.

7. Conclusion

7.1. Summary

Since we are children, we classify things. We start distinguishing between dogs and cats, and then move to more complex stuff: categorizing things is a cognitive ability we acquire and polish through the years. However, this is a challenging task to conceptualize: It is hard to explain how we classify an item as X and not as Y. Moreover, it is a subjective task that might be biased by our experiences or our backgrounds.

If we want to extend this *talent* to machines, we face a very challenging problem. In the last pages, we specifically talked about classifying products originating from an e-commerce domain. Products are heterogeneous and are highly skewed across the existing categories.

Categories on the other side are structured in a hierarchy. Categories in the higher levels correspond to more general concepts, while categories in the lower levels are more specific. This hierarchy is immutable from our point of view: It is a scheme that declares where we should place products.

Prior works ([SRS12, DC00]) have shown that considering the underling hierarchical structure of the categories, derives better results than classifying products in a flat structure. That is, classifying products in the first level categories disregarding the majority of the other subcategories, and then, classifying among the categories that belong to the first level category hierarchy. We adopted this approach and showed it to outperform the flat approach, consistent with previous results.

In our solution, we explored different features from the products, such as title, description, brand, categories from other retailers, etc. This is to the best of our knowledge, a novel characteristic: Prior works usually consider only titles and sometimes also descriptions. However, idealo's data contains other features in their products obtained by the different retailers that can be used. This gave us some advantage already.

Then, we used Mikolov's word embeddings ([MSC+13]) to transform the text features to numeric vectors. We explored different domain based changes in the features, improve our results. The first one was related to adding information to the feature vectors. The second one involved the organization of categories in the hierarchical structure. We also considered using PCA for dimensionality reduction, but this didn't work well. However, we used it for visualization purposes, and outlier detection (although the impact of this is negligible).

Finally we classified the products, experimenting with different algorithms, such as multinomial logistic regression, SVM and kNN. This was done in two stages: First we

selected a category among the first level ones, and then we classified into the branch formed by this one.

We evaluated our performance using F_1-Score. We ran different experiments, where some of them tried to overcome the problem of subjectivity, such as considering the sibling categories as correct predictions, or selecting the top X predictions instead of only one.

All this was done with Apache Spark. An engine for easing parallel and distributed large-scale data processing. Spark was used with the idea of improving times of the training and prediction tasks. Some experiments were run with respect to this, such as changing the number of nodes, and evaluating the performance. This, to the best of our knowledge, was never tried before with a hierarchical setting as the mentioned before. In general, this work can be summarized into the following steps:

1. Exploration of the Data.
2. Cleaning of the Data.
3. Feature Selection.
4. Feature Modeling.
5. Hierarchical Classification: containing first level classification and a flat classification for every branch.
6. Evaluation of the proposed solution.

Back to the figure we had in the introduction, see Figure 7.1. The first state corresponds to item 2 in the enumeration from above. The second one, i.e. feature extractions and transformations is included in items 3 and 4. The rest of the correspondence between the rest of the states and items should be obvious. However, we included one prior step: Exploration of the Data. This was necessary in order to perform correctly the whole project.

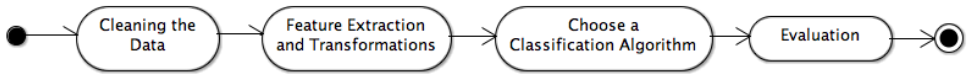


Fig. 7.1. Classification Techniques

7.2. The Challenges

In this section we want to summarize some of the most important problems we encountered in the development of in this thesis.

- Misclassification of the training set: Some of the products were misclassified from the beginning. This not only can be reflected on the final results: When we think something is wrong, and it is right. But also in the training: our classifier learns some information as it was something that is not. We deleted some of this misclassifications manually, with the help of 2D projections with PCA. But cleaning the whole dataset this way would take a lot of time.

- Skew Data: This problem was expected since it was frequently reported in similar works. In the results is evident that is still a problem: Categories that don't have much products (in training set), don't get their F_1-score as high as those that have more products. However, this is easily solvable by adding more training samples to those.

- Vocabulary: German language has some particularities like diaeresis and sharp s. Using word2vec or any other technique which objective is to differentiate between two words, can be losing information when a word is written in more than one way (e.g. *grösse* and *groesse*). We standarized this, by choosing arbitrarily only one form of writing them (e.g. without diaeresis or sharp s), and substituting all by this.

- Hierarchical Structure: There are some categories whose meaning is very close to other category, and somehow they are not close to each other. This was solved by bringing the similar categories together. This was done arbitrarily, and its efficiency was tried with empirical tests. In the recommendation sections, it is suggested to use latent topics, or any clustering techniques in order to overcome this problem.

- Hierarchical Subjectivity: Sometimes it is not clear if a product should belong to one category or to another one. This problem is related to the hierarchical structure from above. In many occasions the product could even belong to two or more categories at the same time. However, as we are not solving a multilabeling task, predicting more than one label is not an option. Different experiments like checking if the real label is on the path, or checking if the real label is a sibling category from the predicted, help to overcome this problem. However, we could also be marking some mistakes with this (specially in the siblings experiments, since not all siblings are necessary related).

- Spark: In general working with Spark is more time consuming than using a standard library for machine learning algorithms (e.g. Weka or scikit-learn). The underlying distribution of the data among nodes can't be forgotten. The use of the User Interface of Spark helps a lot for tracking all problems, detecting slow tasks, and debugging in general.

- Dataframes: The Spark Dataframe structure is an alias for Dataset[Row], and Row is a generic untyped object. Dataframes are then analysed at runtime, which might not be desirable (using Datasets this would be done at compile time). More than that, because we are not explicitly telling the type (or schema) of the Rows in a Dataframe, sometimes is difficult to keep track of all the transformations.

7.3. Recommendations and Future Work

There are some recommendations and further work that can be done for continuing the development of this project.

Recommendations

- We think that a lot of problems were given by the underlying hierarchy. Therefore, maybe changing its structure might be a good improvement. Some past work used Latent Dirichlet Allocation to accomplish this ([CJS11, Koz15]). However, instead of creating this structures *virtually* (i.e. just for prediction), it might work to create them for further uses (i.e. as real hierarchy for the categories). In general, clustering techniques might be applicable to this.
- Create objective rules to differentiate between categories like *Sportuhren* and *Smartwatches*. This way might be easier to identify patterns for classification.
- The classifier could recommend some categories, such as a top X other categories, when a prediction is not good enough. This way, experts would have less categories to select from, and might accelerate their categorization process, see Figure 7.2.

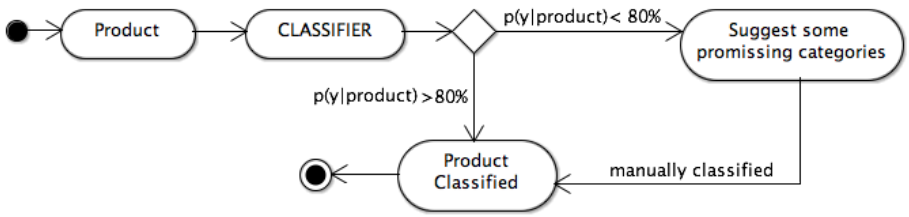


Fig. 7.2. Proposed Change in the Final Classification Part

Further Work

- Explore stop-word lists, in order to determine which words can be removed, without decreasing performance, for alleviating load and processing of words.
- In [WHS16] was shown that hierarchical clustering is a very useful tool for this type of tasks. It could be worth to try, and compare results with other approaches to the presented in this thesis.
- The use of other features, such as images might be useful.

- Select the branches that behave the worst, and take some domain specific actions ([SRSS11]), or different settings (such as other selection of features or algorithms) in order to improve the performance in the respective branch.
- The use of different vector spaces (i.e. different size of vector) depending on the product attribute (e.g. title, description) could give better results.
- When evaluating of the prediction's results, [SL01] weighted the error depending on how far the predicted category was from the real category (using cosine similarity as a distance measure). This way, different degrees of errors can be detected, and evaluated in a more precise way. It might happen, that one category is behaving better than expected: it had too many errors, but perhaps this are due to subjectivity at classification time. This solution should tackle this problem.
- It would be interesting to investigate if the separation of compound words of the German language improves the classification. We are using features generated from word embeddings, which are based on words as units. German uses a lot of compound words, then some meaningful relation between one of the parts of the compound word and some other word in the vocabulary might not be notice.
- Experiment with other corpus as a training set for the word2vec model. Different context produces radical different embeddings, because they induce other word similarities. Then, might be a good experiment to try different ones.

Appendices

A. Unsupervised Learning Algorithms

A.1. Principal Component Analysis (PCA)

"PCA consists of projecting the N-dimensional input data onto the k-dimensional linear subspace that minimizes the sum of the squared L_2-distances between the original data and the projected data"[MRT12].

Let $X \in \mathbb{R}^{dxN}$ be a data matrix with N samples. Every sample corresponds to an $\mathbb{R}^d$ vector. For calculating the PCA of X in a simple way, we can follow this steps:

1. Center the data, such as$\sum_{i=1}^{m} x_i = 0$.
2. Calculate its covariance matrix: $C(X) = \frac{1}{N} X X^T$.
3. Compute the eigenvectors and the eigenvalues of $C(X)$. This can be done using the singular value decomposicion (SVD).
4. Sort (descending) the eigenvectors, using their corresponding eigenvalues. The eigenvector with the largest eigenvalue, correspond to the dimension with the largest variance in the feature space.
5. Take the first k eigenvectors, and form a matrix W, such as its dimension is dxk.

If we want to lower the dimensionality of our data X, we just need to project X into the W space, that is $W^T X$.

B. Supervised Learning Algorithms

B.1. (Multinomial) Logistic Regression

Logistic Regression is an optimization problem that tries to maximize the likelihood function:

$$L(w) = P(y|x; w) = \prod_{i=0}^{K-1} P(y_i|x_i; w)$$

Having K classes and F features. $w \in \mathbb{R}^K$ is a vector with the model's weights, $y \in \mathbb{R}$ is the label we are predicting and $x \in \mathbb{R}^F$ is the features vector. This is read as the likelihood of parameter w is the probability of label y given vector x parametrized by w.

This is equivalent to solving the optimization problem that involves minimizing the respective log-likelihood, that is:

$$l(w) = \sum_{i=0}^{K-1} logP(y_i|x_i; w)$$

Given a feature vector (or data point) x, the (binary) model makes predictions by applying the logistic function:

$$P(y = 0|x; w) = \frac{1}{1 + e^{x^T \cdot w}}$$

For the multinomial logistic regression, we need to choose one of the K classes as a pivot P. Then, we can model K-1 models for every other class, and compare it to the pivot P.

With this, and having $Y = y_1, y_2, ..., y_k$, where $k = 1$ is the pivot, we have out probability functions as:

$$log(\frac{P(y = 2|x; w)}{P(y = 1|x; w)}) = x^T \cdot w_2$$
$$log(\frac{P(y = 3|x; w)}{P(y = 1|x; w)}) = x^T \cdot w_3$$
$$...$$
$$log(\frac{P(y = K|x; w)}{P(y = 1|x; w)}) = x^T \cdot w_K$$

Where we represent the probability with the logistic function like:

$$P(y = 1|x; w) = \frac{1}{1 + \sum_{i=2}^{K} e^{x^T \cdot w_i}}$$
$$P(y = 2|x; w) = \frac{e^{x^T \cdot w_2}}{1 + \sum_{i=2}^{K} e^{x^T \cdot w_i}}$$
$$\ldots$$
$$P(y = K|x; w) = \frac{e^{x^T \cdot w_K}}{1 + \sum_{i=2}^{K} e^{x^T \cdot w_i}}$$

The weights in this generalized form of the problem is a matrix W, where every column corresponds to the weights of a class i.e. $W \in \mathbb{R}^{KxF}$ where F is the number of features. The number of variables we need to estimate is then KxF.

B.2. Support Vector Machines

This is a supervised binary classification algorithm. The goal is to find a decision boundary (i.e. an hyperplane) that maximizes the margin. This is the perpendicular distance between the hyperplane and the closest data points ([Bis06]).

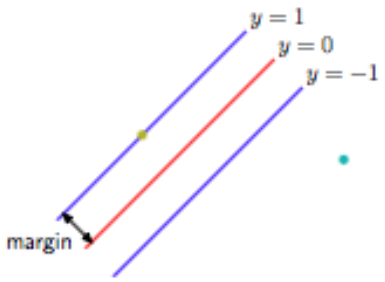


Fig. B.1. The red line is the hyperplane that divides the two classes. The blue lines are *drew* by the support vectors, which are the data points that are closer to the hyperplane. This image is from [Bis06]

Mathematically, we are trying to solve the optimization problem where the weight vector ($w \in \mathbb{R}^d$) is minimized in the objective function $f(w)$:

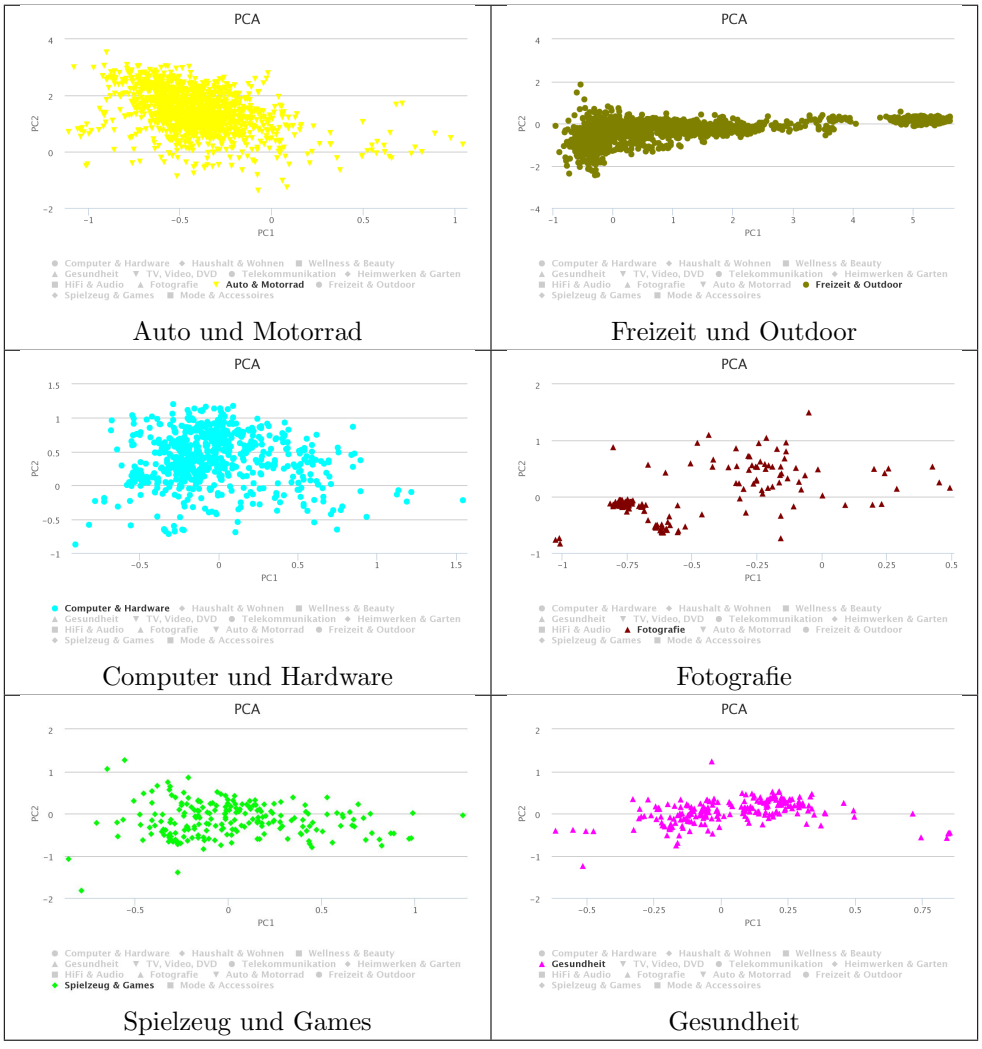
$$f(w) := \lambda \frac{1}{2} \|w\|_2^2 + \frac{1}{n} \sum_{i=1}^{n} \max\{0, 1 - y(w^T x - b)\} \tag{B.1}$$

Here, the vectors $x_i \in \mathbb{R}^d$ are the training data points. and $y_i \in \{-1, 1\}$ are the labels we want to predict ([Spad]). By minimizing this function, we are then penalizing high values for the weights ($\frac{1}{2}\|w\|_2^2$) while minimizing the error ($\sum_{i=1}^{n} \max\{0, 1 - y(w^T x - b)\}$). This trade-off can be controled with λ.

The predictions are calculated by projecting the data point we want to predict into the weight vector. That is $w^T x$. Normally, if $w^T x \geq 0$ then we say $y = 1$, and if not, then $y = -1$.

C. Results

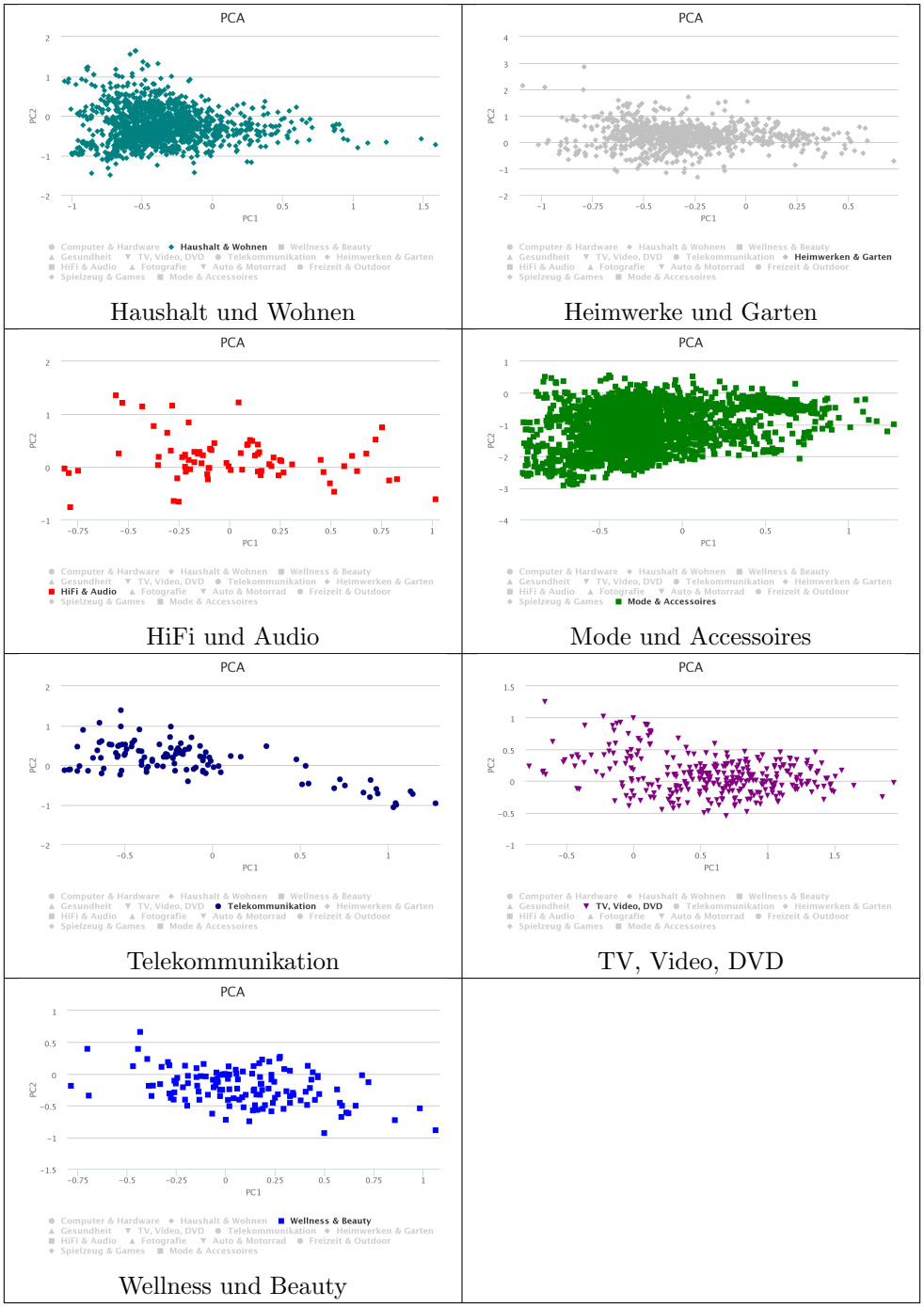
C.1. PCA per First Level Category

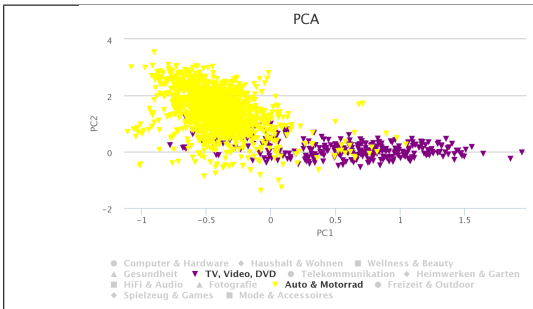


Auto und Motorrad | Freizeit und Outdoor

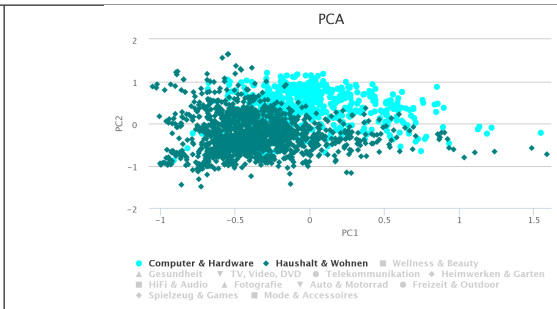
Computer und Hardware | Fotografie

Spielzeug und Games | Gesundheit

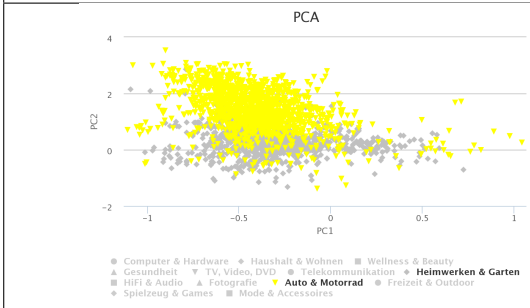

PCA
PC1
PC2
Computer & Hardware
Haushalt & Wohnen
Wellness & Beauty
Gesundheit
TV, Video, DVD
Telekommunikation
Heimwerken & Garten
HiFi & Audio
Fotografie
Auto & Motorrad
Freizeit & Outdoor
Spielzeug & Games
Mode & Accessoires
Haushalt und Wohnen
Heimwerke und Garten
HiFi und Audio
Mode und Accessoires
Telekommunikation
TV, Video, DVD
Wellness und Beauty



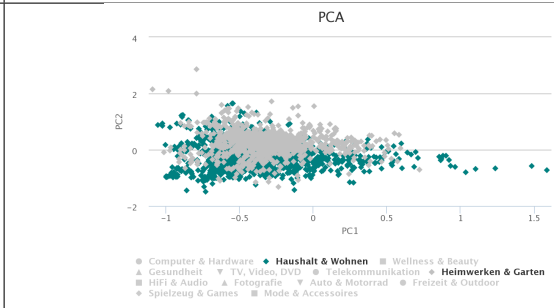
Auto & Motorrad with TV, Video, DVD



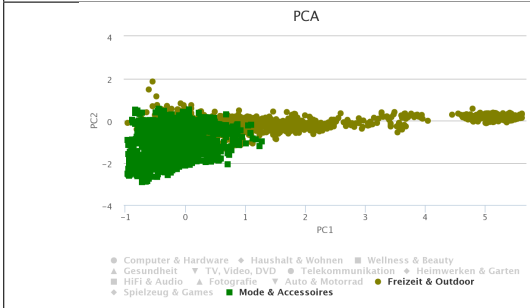
Haushalt und Wohnen with Computer und Hardware



Heimwerke und Garten with Auto und Motorrad



Heimwerke und Garten with Haushalt und Wohnen



Mode und Accessoires with Freizeit und Outdoor

C.2. Spark Experiments with 6GB per Node

This are screenshots from the Spark User Interface, running the experiments reported in 6.18:

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
Active(2)	41	1892.9 MB / 3.4 GB	0.0 B	1	1	0	3839	3840	7,0 m (4,5 s)	15.6 GB	0.0 B	29.8 KB
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
Total(2)	41	1892.9 MB / 3.4 GB	0.0 B	1	1	0	3839	3840	7,0 m (4,5 s)	15.6 GB	0.0 B	29.8 KB

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
0	10.234.144.194:49793	Active	35	1892.8 MB / 3.0 GB	0.0 B	1	1	0	3839	3840	7,0 m (4,5 s)	15.6 GB	0.0 B	29.8 KB	stdout stderr	Thread Dump
driver	172.29.1.251:44948	Active	6	120.7 KB / 366.3 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump

Fig. C.1. One Node. One Core.

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
Active(4)	39	1893.0 MB / 9.4 GB	0.0 B	3	3	0	3847	3850	7,4 m (3,9 s)	15.8 GB	19.0 KB	29.8 KB
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
Total(4)	39	1893.0 MB / 9.4 GB	0.0 B	3	3	0	3847	3850	7,4 m (3,9 s)	15.8 GB	19.0 KB	29.8 KB

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
2	10.234.144.193:56813	Active	12	594.0 MB / 3.0 GB	0.0 B	1	1	0	1257	1258	2,4 m (1,6 s)	4.9 GB	5.8 KB	9.5 KB	stdout stderr	Thread Dump
1	10.234.144.195:33144	Active	13	712.6 MB / 3.0 GB	0.0 B	1	1	0	1235	1236	2,6 m (1,4 s)	5.8 GB	7.7 KB	9.9 KB	stdout stderr	Thread Dump
0	10.234.144.194:44793	Active	10	586.2 MB / 3.0 GB	0.0 B	1	1	0	1355	1356	2,4 m (862 ms)	5.1 GB	5.5 KB	10.4 KB	stdout stderr	Thread Dump
driver	172.29.1.251:48157	Active	4	179.7 KB / 366.3 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump

Fig. C.2. 3 Cores (1 per node)

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
Active(4)	59	1913.5 MB / 9.4 GB	0.0 B	6	1	0	3876	3877	7,8 m (6,9 s)	17.7 GB	18.7 KB	31.5 KB
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
Total(4)	59	1913.5 MB / 9.4 GB	0.0 B	6	1	0	3876	3877	7,8 m (6,9 s)	17.7 GB	18.7 KB	31.5 KB

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
2	10.234.144.193:47118	Active	15	654.1 MB / 3.0 GB	0.0 B	2	1	0	1251	1252	2,6 m (2,4 s)	5.7 GB	5.5 KB	10.3 KB	stdout stderr	Thread Dump
1	10.234.144.195:48492	Active	19	725.8 MB / 3.0 GB	0.0 B	2	0	0	1255	1255	2,7 m (2,7 s)	6.6 GB	7.5 KB	10.5 KB	stdout stderr	Thread Dump
0	10.234.144.194:54405	Active	12	523.3 MB / 3.0 GB	0.0 B	2	0	0	1370	1370	2,5 m (1,8 s)	5.3 GB	5.7 KB	10.7 KB	stdout stderr	Thread Dump
driver	172.29.1.251:49017	Active	13	10.3 MB / 366.3 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump

Fig. C.3. 6 Cores (2 per node)

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
Active(4)	44	1894.1 MB / 9.4 GB	0.0 B	12	1	0	3843	3844	8,7 m (13,3 s)	15.6 GB	18.5 KB	29.8 KB
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
Total(4)	44	1894.1 MB / 9.4 GB	0.0 B	12	1	0	3843	3844	8,7 m (13,3 s)	15.6 GB	18.5 KB	29.8 KB

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
2	10.234.144.193:40977	Active	10	530.3 MB / 3.0 GB	0.0 B	4	0	0	1247	1247	2,8 m (4,3 s)	4.9 GB	5.5 KB	9.4 KB	stdout stderr	Thread Dump
1	10.234.144.195:54261	Active	18	840.1 MB / 3.0 GB	0.0 B	4	1	0	1259	1260	3,2 m (5,3 s)	6.1 GB	8.1 KB	11.0 KB	stdout stderr	Thread Dump
0	10.234.144.194:40905	Active	11	523.6 MB / 3.0 GB	0.0 B	4	0	0	1337	1337	2,7 m (3,8 s)	4.7 GB	4.8 KB	9.4 KB	stdout stderr	Thread Dump
driver	172.29.1.251:44671	Active	5	98.1 KB / 366.3 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump

Fig. C.4. 12 Cores (4 per node)

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
Active(4)	48	1912.9 MB / 9.4 GB	0.0 B	18	1	0	3903	3904	11,8 m (16,7 s)	15.7 GB	19.3 KB	29.8 KB
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
Total(4)	48	1912.9 MB / 9.4 GB	0.0 B	18	1	0	3903	3904	11,8 m (16,7 s)	15.7 GB	19.3 KB	29.8 KB

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
2	10.234.144.193:45836	Active	12	520.9 MB / 3.0 GB	0.0 B	6	0	0	1357	1357	3,9 m (5,2 s)	5.2 GB	6.4 KB	11.7 KB	stdout stderr	Thread Dump
1	10.234.144.195:35076	Active	18	855.7 MB / 3.0 GB	0.0 B	6	1	0	1245	1246	4,1 m (6,6 s)	5.6 GB	6.3 KB	10.8 KB	stdout stderr	Thread Dump
0	10.234.144.194:55338	Active	11	526.3 MB / 3.0 GB	0.0 B	6	0	0	1301	1301	3,8 m (4,9 s)	5.0 GB	6.6 KB	7.4 KB	stdout stderr	Thread Dump
driver	172.29.1.251:47798	Active	7	10.1 MB / 366.3 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump

Fig. C.5. 18 Cores (6 per node)

Bibliography

[Bis06] Bishop, Christopher M.: *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Bre01] Breiman, Leo: *Random Forests*. Mach. Learn., 45(1):5–32, October 2001.

[CJS11] Chen, Mengen, Xiaoming Jin and Dou Shen: *Short Text Classification Improved by Learning Multi-granularity Topics*. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three*, IJCAI'11, pages 1776–1781. AAAI Press, 2011.

[CW13] Chen, Jianfu and David Warren: *Cost-sensitive Learning for Large-scale Hierarchical Classification*. In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, pages 1351–1360, New York, NY, USA, 2013. ACM.

[DC00] Dumais, Susan and Hao Chen: *Hierarchical Classification of Web Content*. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '00, pages 256–263, New York, NY, USA, 2000. ACM.

[DFU15] Dhillon, Paramveer S., Dean P. Foster and Lyle H. Ungar: *Eigenwords: Spectral Word Embeddings*. Journal of Machine Learning Research, 16:3035–3078, 2015.

[DHS+01] Duda, Richard O., Peter E. Hart, David G. Stork, C R. O. Duda, P. E. Hart and D. G. Stork: *Pattern Classification, 2nd Ed.* 2001.

[Far16] Farber, Madeline: *Consumers Are Now Doing Most of Their Shopping Online*. Fortune. 08. Jun 2016.

[FR] Fortmann-Roe, Scott: *Understanding the Bias-Variance Tradeoff*.

[fYhHA+] Yu, Hsiang fu, Chia hua Ho, Prakash Arunachalam, Manas Somaiya and Chih jen Lin: *Product title classification versus text classification*. Technical Report.

[HPK11] Han, J., J. Pei and M. Kamber: *Data Mining: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2011.

[HTF01] Hastie, Trevor, Robert Tibshirani and Jerome Friedman: *The Elements of Statistical Learning.* Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[JM07] Jackson, Peter and Isabelle Moulinier: *Natural language processing for online applications. Text retrieval, extraction and categorization*, volume 5 of *Natural Language Processing.* Benjamins, Amsterdam, Philadelphia, 2007.

[Kim14] Kim, Yoon: *Convolutional Neural Networks for Sentence Classification.* CoRR, abs/1408.5882, 2014.

[KKWZ15] Karau, Holden, Andy Konwinski, Patrick Wendell and Matei Zaharia: *Learning Spark: Lightning-Fast Big Data Analytics.* O'Reilly Media, Inc., 1st edition, 2015.

[Koz15] Kozareva, Zornitsa: *Everyone Likes Shopping! Multi-class Product Categorization for e-Commerce.* In *HLT-NAACL*, 2015.

[KS97] Koller, Daphne and Mehran Sahami: *Hierarchically Classifying Documents Using Very Few Words.* In *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML '97, pages 170–178, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[Las] Laskowski, Jacek: *Mastering Apache Spark 2.*

[LJ99] Lakoff, George and Mark Johnson: *Philosophy in the Flesh: The Embodied Mind and its Challenge to Western Thought.* 1999.

[LYW+05] Liu, Tie-Yan, Yiming Yang, Hao Wan, Hua-Jun Zeng, Zheng Chen and Wei-Ying Ma: *Support Vector Machines Classification with a Very Large-scale Taxonomy.* SIGKDD Explor. Newsl., 7(1):36–43, June 2005.

[MB02] Mayr, Ernst and W. J. Bock: *Classifications and other ordering systems.* Journal of Zoological Systematics and Evolutionary Research, 40(4):169–194, 2002.

[Mit97] Mitchell, Thomas M.: *Machine Learning.* McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[MO] Martin Odersky, Philippe Altherr, Vincent Cremet Gilles Dubochet Burak Emir Philipp Haller Stéphane Micheloud Nikolay Mihaylov Adriaan Moors Lukas Rytz Michel Schinz Erik Stenman Matthias Zenger: *Scala Language Specification.*

[MRT12] Mohri, Mehryar, Afshin Rostamizadeh and Ameet Talwalkar: *Foundations of Machine Learning.* The MIT Press, 2012.

[MSC+13] Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S Corrado and Jeff Dean: *Distributed Representations of Words and Phrases and their Compositionality.* In Burges, C.J.C., L. Bottou, M. Welling, Z. Ghahramani and K.Q. Weinberger (editors): *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[OM07] Ou, Guobin and Yi Lu Murphey: *Multi-class Pattern Classification Using Neural Networks.* Pattern Recogn., 40(1):4–18, January 2007.

[Par15] Parsian, Mahmoud: *Data Algorithms: Recipes for Scaling Up with Hadoop and Spark.* O'Reilly Media, Inc., 1st edition, 2015.

[Ros75] Rosch, Eleanor: *Cognitive Representation of Semantic Categories.* Journal of Experimental Psychology: General, 104:192–233, 1975.

[Ryz] Ryza, Sandy: *How-to: Tune Your Apache Spark Jobs (Part 2).*

[SL01] Sun, Aixin and Ee-Peng Lim: *Hierarchical text classification and evaluation.* In *Proceedings 2001 IEEE International Conference on Data Mining*, pages 521–528, 2001.

[Spaa] Spark, Apache: *Apache Spark.*

[Spab] Spark, Apache: *Apache Spark: Cluster Mode Overview.*

[Spac] Spark, Apache: *Apache Spark: Ensembles Guide.*

[Spad] Spark, Apache: *Apache Spark: Linear Methods.*

[Spae] Spark, Apache: *Evaluation Metrics: Multiclass classification.*

[Spaf] Spark, Apache: *Feature Transformers: Tokenizer.*

[SRMS12] Shen, Dan, Jean-David Ruvini, Rajyashree Mukherjee and Neel Sundaresan: *A Study of Smoothing Algorithms for Item Categorization on e-Commerce Sites.* Neurocomput., 92:54–60, September 2012.

[SRS12] Shen, Dan, Jean-David Ruvini and Badrul Sarwar: *Large-scale Item Categorization for e-Commerce.* In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 595–604, New York, NY, USA, 2012. ACM.

[SRSS11] Shen, Dan, Jean David Ruvini, Manas Somaiya and Neel Sundaresan: *Item Categorization in the e-Commerce Domain.* In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 1921–1924, New York, NY, USA, 2011. ACM.

[VAPM14] Valverde-Albacete, Francisco J. and Carmen Peláez-Moreno: *100% Classification Accuracy Considered Harmful: The Normalized Information Transfer Factor Explains the Accuracy Paradox.* PLOS ONE, 9(1):1–10, 01 2014.

[WHS16] Wu, P., D. He and J. Song: *A mental model approach for category hierarchy maintenance on sellers' self-input items in e-commerce websites.* In *2016 11th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–7, June 2016.

[YZK03] Yang, Yiming, Jian Zhang and Bryan Kisiel: *A Scalability Analysis of Classifiers in Text Categorization.* In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval*, SIGIR '03, pages 96–103, New York, NY, USA, 2003. ACM.

[ZWZ16] Zhu, L., G. Wang and X. Zou: *A Study of Chinese Document Representation and Classification with Word2vec.* In *2016 9th International Symposium on Computational Intelligence and Design (ISCID)*, volume 1, pages 298–302, Dec 2016.