KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Automatic Joke Generation: Learning Humour from Examples

Thomas Winters

Thesis submitted for the degree of
Master of Science in Engineering:
Computer Science

Thesis supervisor:
Prof. dr. Danny De Schreye

Assessor:
Prof. dr. Luc De Raedt,
Dr. Stefano Teso

Mentor:
Vincent Nys

Academic year 2016 – 2017

Preface

I would like to express my sincerest gratitude to prof. dr. Danny De Schreye and Vincent Nys for their enthusiasm and encouragements, for their trust, for allowing extensive initial experimentation, for ensuring that the topic of this thesis was steered towards scientifically interesting aspects, for their patience and for the many inspiring discussions, interesting arguments and enlightening suggestions throughout the year.

Thanks are also due to all the users of JokeJudger.com, the platform we created for this thesis, where hundreds of volunteers created and rated jokes for the training dataset and the evaluation of the system.

I would also like to thank the jury for reading the text.

Finally, I would also like to thank my family and my friends for their support.

Thomas Winters

Contents

Abstract

This thesis explores how a program is able to generate humour by learning insights from a set of rated example jokes. We construct a framework called Goofer to fulfil this task, and use humour theory and generalise computational humour concepts to guide the construction of its components. These components allow the framework to learn the structures of the given jokes and estimate how funny people might find specific instantiations of joke structures. This knowledge is then employed to generate good jokes.

Following the theoretical foundation and specification for the humour generating framework, we implement and evaluate a subset of the framework. This system, called Gag, learns how to generate analogy jokes using the *"I like my X like I like my Y, Z"* template based on given, rated analogy jokes. Since it uses generalised components and learns its own schemas, this program successfully generalises the best known analogy generator in the computational humour field.

List of Figures

List of Abbreviations and Symbols

Abbreviations

AI	Artificial Intelligence
CH	Computational Humour
GTVH	General Theory of Verbal Humour
KR	Knowledge Resource (of GTVH)
NLG	Natural Language Generation
NLP	Natural Language Processing
RegEx	Regular expression
SSTH	Semantic Script Theory of Humour

Abbreviations for Our Systems

Goofer	Generator Of One-liners From Examples with Ratings
Gag	Generalised Analogy Generator

Symbols

X	The first variable used in the *"I like my X like I like my Y, Z"* template
Y	The second variable used in the *"I like my X like I like my Y, Z"* template
Z	The third variable used in the *"I like my X like I like my Y, Z"* template

Chapter 1

Introduction

1.1 Problem Statement

This thesis explores how a computer program is able to learn humour from human-rated examples. We achieve this by using humour theory and concepts from related research as a guidance to design a generalised computational humour framework. We then implement several components of this framework to construct a generalised analogy generator. This research is situated in computational humour (CH), a branch of natural language processing (NLP) and artificial intelligence (AI). In this field, researchers perform several kinds of tasks on humour using programs. There are three main categories of tasks within computational humour [6]. The first type of task is the generation of humorous artefacts using programs. The second type of task is the automatic detection of certain types of humour. The third type is the use of computational concepts to understand and evaluate humour theory [48]. This thesis focuses on the first and the last categories: generating and understanding humour. In this thesis, we create a framework that learns humour theory from input jokes and uses this knowledge to generate new jokes. We call this framework GOOFER, which stands for *Generator of One-liners From Examples with Ratings*. It builds upon multiple existing computational humour systems and theories and can be extended in the interest of generalising other computational humour systems. In order to show an implementation of such a generalisation using this framework, we created a system called GAG (*Generalised Analogy Generator*). This program focuses on a restricted, specific humorous domain, being the understanding and generation of analogy jokes using the *"I like my X like I like my Y, Z"* template.

The ideal computational humour system would be able to perform all three mentioned task categories on all types of humour. However, just like the problem of understanding natural language text [33], computational humour has been referred to as an AI-complete problem [6][56]. A computational problem being AI-complete (also called AI-hard) means that the difficulty of solving the problem is equivalent to solving the central AI problem, which is the problem of making computers as intelligent as a human. As a consequence, most computational humour systems tend to focus on a single task, performed on a specific type of humour [47]. These programs

are for example capable of creating punning riddles [7][34], generating punning witticisms [23], generating analogies [41], generating funny acronyms [57], generating taboo misspellings [62], generating punny captions [9], detecting innuendos [26], recognising correct *knock knock* jokes [58], recognising one-liners [36] and recognising good cartoon captions [52]. We discuss some of these programs in more detail in section 2.4.

Computational humour researchers tend to create new systems that do not really extend on previous systems (although there are exceptions such as Standup [34] building upon the Jape system [7]). They focus on doing new types of tasks on new types of humour. It is thus more of a broad field than a deep field. It is however important to create systems that generalise previous research in order to create systems that are capable of doing more types of tasks on more types of humour. Creating a framework for such type of systems is one of the goals of this research.

The effectiveness of a system capable of generating humorous artefacts depends on several factors. For textual jokes, examples of factors that influence the joke quality include the word choice [57], word order (narrative) [43], amount of conflict [48], amount of incongruity [24] and cultural understanding [41]. These have been widely researched in computational humour. Lesser studied dependencies of humour are the individual and temporal dependencies. The only humorous systems, to the best of our knowledge, that adapt the jokes they present to their user, do not generate said jokes themselves. The use of Eigentaste in the Jester system for example merely functions as a humour recommendation system [17]. No system is thus known to be capable of generating humour based on current humour trends or user preferences without external human support. The reason for this absence is that most of the existing humour generators utilise some form of rule set or assumptions about their type of joke. This implies that they are neither capable of automatically learning new types of humour nor updating the content of their jokes without humans having to manually update these rules. This has some serious implications for both the individual and temporal dependency of the quality of the generator's humour.

The perceived quality of a joke depends on the time when it is read. When reading jokes from your grandparents' childhood, they are perceived as less funny now than at the time they were published. There could be several reasons for this, such as the fact that society evolved beyond using certain objects, practises or words used in the joke. Another possibility is that due to the internet, sharing jokes has become easier, causing jokes to be more prevalent. The result of people seeing a larger quantity of jokes, is that they might set a higher standard for what they perceive as truly funny. A similar trend is noticeable on short term, when certain types of internet jokes go viral for a small amount of time, after which people perceive these kind of jokes as dull [5]. Joke generators that are unable to adopt to how humour evolves thus tend to lose value over time. Another issue for the temporal dependency with having a fixed rule set for humour generators, is their limited possibility space. The possibility space of a generator denotes the mathematical set of all possible generations it can produce. Having a rule set or focussing on a particular type of humour might bore users after they have seen sufficient examples of the possibility space to not be surprised any more by its output. Being able to adopt and learn

new rule sets would allow generators to keep presenting fresh jokes and surprising their user.

The neglect of the individual appreciation dimension of humour generators has consequences for the user interfaces employing such generators. It has often been suggested that computational humour could play a significant role in making human-machine interfaces feel more human-like [56], especially for chat bots [42]. Seeing the disagreement between users in the evaluations of jokes from previous research [17][41][57] as well as in ours (see section 4.3.6), it might be sensible to account for individual user preferences when generating jokes. An integrated humour generator that is capable of generating jokes based on the learned user's preferences might thus outperform a better generator that does not account for these preferences. This is comparable to how friends might cause people to laugh harder than any comedy on TV ever would, since the latter is independent of their viewers' preferences and personal knowledge.

Being able to learn humour from examples along with their perceived quality to the user(s), enables a program to account for the temporal and individual factors of humour appreciation. It can perform such a feat by feeding its own generated jokes with ratings to its training data. For human-machine interfaces, the program can execute sentiment analysis algorithms to learn how well the user perceived this joke. This way, it can evolve to accommodate for the users humour preferences. In order to solve the temporal dependency, the program can learn from new jokes posted on popular humour sites. It could also post its generated jokes to popular platforms with a rating system in order to get insight in the perceived quality.

In this thesis, we present our framework for generating humour from examples. Our goal is to create a stepping stone towards computational humour systems that are capable of solving the temporal and individual issues mentioned earlier. We analyse and reuse concepts and results (such metrics, parameters, theories, methods ...) from other computational humour research and use it to create an extendible framework. We also show how this framework can be used to generalise previous research by focusing its domain on analogy jokes and implementing an generalised analogy generator. We end the thesis with a comparison of our implementation to an existing analogy generator.

1.2 Research Questions

- Can we make a computer program learn relations between words that establish humorous interpretation when used in a particular template?

- Is it possible to create a program capable of humour generation that can learn from rated examples, and if so:

  - does this make it possible for the program to improve the quality of its generations for a single user? In other words, can we create a computer program that is capable of learning humour from corpora of human-rated

jokes, as well as from feedback on its own generated content and generate jokes based on these rated jokes?

- can the program explain (elements of) the learned knowledge in such a way that humans can understand and use this theory in practise?
- are there aspects where this program outperforms a generator which uses hand-crafted rules or assumptions made for a specific type of joke?

- Is it possible to create a framework that builds upon theories, metrics and other concepts discovered by other computational humour research in order to build a more generic, extensible program that learns humour from examples, and if so, are there aspects where this program outperforms a generator which uses hand-crafted rules or assumptions made for a specific type of joke?

1.3 Relevancy of Computational Humour

Computational humour is a relevant research domain in a world that becomes increasingly reliant on natural language human-machine interactions. Humour has always been an important aspect in human communication [49]. Hence, it only makes sense that humour also plays an important role for systems aiming to realistically communicate with humans using natural language. Other uses for computational humour can also be found in advertisement [56], studying humour theory, education and edutainment[64], massive production of jokes and in comedy writers' software.

1.3.1 Computational Humour in Human-Machine Interaction

Leading companies like Google, Facebook, Amazon, Apple, Slack and Skype have recently been massively investing in virtual assistants and chat bots [40][19][12]. Virtual assistants are being portrayed as the new way for users to interact with computers, while chat bots are depicted as a modern way for businesses to interact with (potential) customers [40]. Being able to react to users in a funny way could increase users like using this interface [6]. However, it has been shown that these assistants are notably bad at humour [50]. Both Google and Apple recognise the need for humour in their virtual assistants and are at the moment of writing actively looking for comedy writers and comedians to humanise these assistants [18][65]. Computational humour could thus play an important role in making a bot or virtual assistant feel more human-like to their users [42]. The use of computational humour has the capability to outperform pre-written humour in adopting to the user, as it would enable a system to refer to personal topics (such as locations, friends and hobbies) in a humorous way.

A less obvious use of computational humour in human interaction might be in advertisements. Not only are advertisers capable of using the comedy writing tools mentioned in section 1.3.2, computational humour can also be deployed to create advertisements for a single user [56][57].

1.3.2 Theory and Tools for Creating Comedy

Using computational concepts to talk about humour theory is one of the three categories studied in computational humour [48]. Although there has been some debate about what a theory on computational humour should look like [44], it is still a powerful medium to test out humorous hypotheses. A lot of different theories about humour exist, but they are often rather vague or contradict either themselves or other theories. Raskin, a famous linguistic humour theorist, believes that a good humour theory has five components. These components are a *body* of the theory with a set of predictive statements, the *purview* stating what the theory is about, the *premises* stating the axioms this theory builds upon, the *goals* of the theory and *methods of falsification* as well as *methods of justification* [44]. This kind of rigid structure is meant for more serious theories, such as those we discuss in section 2.2.

We are not convinced that a theory of humour should necessarily be as verbose as formal theorists like Raskin want it to be. One simpler way of verifying a theory is to translate the theory into an algorithm. The difficulty of translation strongly depends on the theory. Once we have a program that generates jokes using this theory, it is possible to evaluate the generated jokes. The theory can then be evaluated using the ratings of these generated jokes. There are several ways to rate a humorous system, such as calculating the average score or calculating the frequency of ratings above a certain threshold [61]. Evaluating a humour theory using an algorithm generating possible jokes stated by the theory exposes bad jokes that might not have been considered otherwise due to confirmation bias. This type of bias occurs when only good jokes that conform to the theory are considered, while bad jokes conforming to this theory are ignored.

Computers are not only capable of verifying humour theory, but also of finding their own theories using machine learning on sets of jokes [36][53]. This type of research uses algorithms to find features good jokes comply to, and that distinguish them from bad jokes. Being able to computationally distinguish jokes and non-jokes is a useful tool for human-machine interaction systems. The findings may or may not be translatable to human-understandable humour theory. We also explore this category of computational humour in this thesis.

One might ask why identifying and verifying humour theory is useful. It has been argued that these types of theories are necessary to advance the study of computational humour and to prevent the field from evolving further blindly [44]. The theory of humour is also useful to practitioners, teachers and students of the art of comedy. We believe that there is also a market for programs to assist these comedy writers, using (computational) humour theory. These type of programs could assist comedians by enhancing their comedy writing process. Computers have been able to outperform humans in certain search problems due to their exhaustive power. If a program is capable of distinguishing more humorous artefacts from less humorous artefacts, it could provide comedy writers with humorous suggestions to improve their scripts and jokes. Such a program could for example suggest synonyms with stronger, funnier connotations or suggest funny associations of the used words. Another feature could be similar to Photoshop's content-aware fill, which is capable of filling in holes

in pixel layers using neural networks [4]. This feature of our hypothetical program would fill in a joke, fitting the topic of the text surrounding it, which the user can then polish to make it fit in with the rest of the comedy set.

In this thesis, we look into generating analogies using the *"I like my X like I like my Y, Z"* template. This template is also used in an improvisational comedy game, where X and Y are assigned through audience suggestion and the improvisers on stage have to fill in Z, the punch line. The theory about the relation between the template values found by our system might thus be useful as a teaching device for improvisational comedy workshops.

1.3.3 Other Uses

Computational humour has also been used in several other domains. In the education and edutainment field, the Standup joke generator has been developed in order to improve the literacy of children with impaired speech by enabling them to interact with generated punning riddles [34][64]. Another obvious use of a humour generator would be for people requiring a large number of jokes. It has been argued that British Christmas crackers firms, which include simple riddles in their crackers, could require a large number of jokes, and thus use a joke generator for this [47].

A use for humour detection would be to enable computer programs to filter out humorous statements. This could help governmental instances' machine learning algorithms filter out noise data when looking for messages related to for example terrorism. Such a technique has to be used cautiously though, since there also has been some research in automatically hiding messages into textual jokes [11]. Another use for humour detection is being able to detect potential humour in serious documents, which enables the user to eliminate unintentional wordplays [49].

1.4 Thesis Structure

In chapter 2, we discuss related computational humour research and terminology in more detail. First, we discuss humour theory in more depth and summarise its history. We then use this theory to argue why certain relevant natural language tools and techniques have been powerful tools in computational humour. These concepts and tools are important for the framework we create in later chapters. Finally, we discuss some related computational humour systems in more detail.

In chapter 3, we present our Goofer framework and its components. This framework is used to generate humour from examples. We start by generalising and extending on the NLP and CH techniques discussed in chapter 2. We then present framework's components based on these generalisations, and discuss the flow of the framework.

In chapter 4, we show an implementation of the Goofer framework. This implementation, which we called Gag, is capable of generating analogies using the *"I like my X like I like my Y, Z"* template. The Gag system implements most of the components of the Goofer framework and allows us to compare the performance of our generalised analogy generator to an existing analogy generator. Since this

implementation requires a large dataset of rated jokes to learn from, we created a platform called JokeJudger. Users of this platform are asked to rate and create jokes, which are used to train our system. We discuss the implementation, philosophy and the learned lessons of this platform in more detail. We discuss the classifier and regression algorithms used in the Gag system, and argue why certain algorithms are powerful for systems using the Goofer framework in general. We then evaluate our system in several ways. We compare the ratings received for the two different versions of our system to each other, as well as to human generated jokes and to an existing analogy generator. In the evaluation, we conclude that compared to jokes submitted by JokeJudger users, the jokes created by Gag are half as many times perceived as funny.

Chapter 2

Related Research and Terminology

2.1 Introduction

This chapter describes related computational humour research. We first discuss several relevant humour theories. These theories are used to verify techniques for humour generation as well as to derive relevant metrics and the structure for the Goofer framework, which we create later in this thesis. We also discuss several concepts and techniques that are often used in computational humour. We extend and use these concepts for the implementation of the Goofer framework in the next chapter. We end the chapter with a discussion about several existing humour generators and detectors.

2.2 Humour Theory

2.2.1 History

Ever since Ancient Greek times, humans have been wondering about the principles that cause laughter and humour [2]. As such, a multitude of theories about humour have emerged over the years. Three main categories for these theories can be identified based on their source of humour, being superiority, the release of tension and incongruity [29]. The superiority theory states that humour stems from feeling superior to a certain political, ethnic or gendered group of people [29][39]. This theory has been viewed as quite an old fashioned theory due to several large issues. The most significant issue is that this theory can not explain why texts that are as aggressive as certain aggressive jokes are not as funny [47]. The relief theory sees humour as the release of tension and psychic energy [55][15]. This energy is due to not having to put this energy into mental censors [38][29][6]. This theory is however not capable of explaining several types of humour [29].

The incongruity theory is the most widely accepted theory, and is also most relevant to computational humour research. It states that humour originates from

incompatible views being resolved. This category contains a large number of theories listing necessary conditions for humour. Many philosophers and linguists helped shaping this theory. Immanuel Kant was the first to formally link the concept of incongruity with humour. Other philosophers such as Schopenhauer and Bergson extended on his work in the 19th century [51][6]. They argued that humour is not in the incongruity itself, but that humour arises when suddenly noticing the incongruence. The humour thus stems from the congruity's appropriate resolution. This explains why the theory is often called the incongruity-resolution theory.

Another important evolution in this theory occurred when Koestler introduced the term "bisociation" in his book The Act of Creation [28]. A bisociation describes the two different viewpoints an act of creativity tends to have [47][29]. He argued that humour stems from a sudden mental bisociation. Such a bisociation manifests itself as a jump between two self-consistent but incompatible frames of reference. In order for a bisociation to create a humorous effect, the narrative is required to have the right emotional tension [28]. Koestler saw laughter as a reaction to a wide variety of impulses, and concluded that no single theory would be able to capture it. [29] Most humour theorists seem to agree on the idea of humour being the combination of two frames, since most jokes can be distilled to a set-up and a punchline [47].

2.2.2 Semantic Script Theory of Humour

From the 1970s onwards, linguistics joined the central humour researchers [2]. Raskin was the first to formally describe the earlier incongruity theory notions as a formal linguistic theory in his Semantic Script Theory of Humour (SSTH) [43][29]. This theory is also compatible with the other humour theory categories, and has been argued to be one of the biggest contribution to incongruity-resolution theory [29]. The focus of the theory is on verbal humour, more specifically on jokes that contain a punchline. Raskin utilises the same notion of frames of reference as the other incongruity-resolution theorists, but calls them "scripts". In his theory, he states that these scripts had to oppose each other. An argument for this opposition is that just like jokes, poetic images have two self-consistent, incompatible frames. These poetic images are not considered humorous, which he argues is due to the lack of opposition [43].

The main hypothesis of SSTH is that there are two linguistic conditions which are necessary and sufficient for a text to be characterised as a single-joke-carrying text[1]

- *"The text is compatible, fully or in part, with two different scripts.*
- *The two scripts with which the text is compatible are opposite in a special sense [...]. The two scripts with which the text is compatible are said to overlap fully or in part on this text."* [43]

[1] This hypothesis is literally quoted from Raskin's book *"Semantic Mechanisms of Humor"* on p.99 [43].

In SSTH, Raskin also provides a set of the possible special senses mentioned in the second condition. Examples of these different types of oppositions are "bad ↔ good", "life ↔ death", "sex ↔ non-sex" and "real ↔ unreal". If a second script does not exist or is not opposed to the first script, a text is not considered as a joke. He also establishes rules for disambiguation between scripts. As such, he states that a joke is unambiguous until the punchline has been reached. Raskin demonstrates SSTH by analysing every word of joke 1, which has become one of the classic jokes in humour theory ever since.

JOKE 1:
"Is the doctor at home?" the patient asked in his bronchial whisper.
"No," the doctor's young and pretty wife whispered in reply.
"Come right in." [43]

He identifies the two scripts in this joke as (visiting the) DOCTOR and LOVER. He shows exhaustively how every word of the joke can be linked to either or both scripts using their dictionary definition and related topics. He summarizes the idea created by the first sentence as *"somebody who was recently treated for an illness who wants to know if the owner of the house is physically present"* [43]. He creates a similar summary of the exact intentions of the other two sentences. He argues the meaning of the last sentence makes the listener wonder why the woman wants him to come in, since this does not allow him to reach his intent. The solution to this question comes from what Raskin calls "world information", which gives the listener the answer. Since DOCTOR and LOVER are regarded as opposites according to the theory, this resolution causes the text to be considered humorous.

2.2.3 General Theory of Verbal Humour

In 1991, Attardo and Raskin incorporated SSTH into their General Theory of Verbal Humour (GTVH) [3]. The goal of this theory is to explain verbal humour. Verbal humour is one of the two types of humour, the other being situational humour. One of the issues of SSTH according to Attardo found in SSTH is that it could not tell how similar jokes were. He thus decided to revisit this theory with Raskin [3], and incorporated it in into GTVH. This theory states that a joke can be distilled into six parameters, which they call knowledge resources (KR): [3]

1. LANGUAGE: The language KR specifies the linguistic components of the joke. It specifies the exact wording and placement of the punchline.
2. NARRATIVE STRATEGY: This KR specifies (micro)genre of the joke, the way it is told. The theory states that the text should be non-redundant enough not to spoil the punchline.
3. TARGET: The target KR specifies the butt of the joke. It is the only optional parameter, as not all jokes are aimed towards someone or something.
4. SITUATION: The situation KR talks about the props of the joke, such as the participants, surroundings, activities.

5. Logical mechanism: This KR explains how the logic is undermined, played with and justified in the joke. This mechanism allows the two scripts to be combined into one joke.

6. Script opposition: This KR specifies the two scripts that are being opposed in the joke. This is the part of GTVH that incorporates SSTH, and thus the same oppositions as specified in section 2.2.2 apply here.

The order of these KR are important, as their order is from weakest to strongest. If two jokes are only different on a certain KR, and two other jokes are only different on a stronger KR, then the first set of jokes is more similar to each other than the second set. To illustrate these parameters and to show that a difference in a stronger KR causes a joke to be less similar, we create several variations of a joke. We utilise joke 2 as the root joke and create several variations using this theory.

Joke 2:
A cowboy parks his horse and walks into a bar.
"What would you like to drink?" the bartender asks.
"I'll have a bucket of beer for my horse, and a glass of water for myself. I still have to drive!"

By changing the Language KR, and thus changing the effectiveness of the punchline (for the worse, in this case), we get joke 3.

Joke 3:
A cowboy parks his horse and walks into a bar.
"What would you like to drink?" the bartender asks.
"I'll have a bucket of beer for my horse, and since I still have to drive, a glass of water for myself."

In joke 4, we change the Narrative Strategy of 2 from a story to a riddle.

Joke 4:
Why did the cowboy only order water for himself after ordering a big bucket of beer for his horse?
Because he still has to drive!

When the Target of the root joke 2 is changed, it mocks another group instead of cowboys. A possible variation on this parameter is joke 5.

Joke 5:
An Irish man parks his horse and walks into a bar.
"What would you like to drink?" the bartender asks.
"I'll have a bucket of beer for my horse, and a glass of water for myself. I still have to drive!"

Changing the Situation KR of joke 2 changes the situation it occurs in. In joke 6, we change the wild western setting with a snow setting, the bar with his house, his horse with sled dogs, beer by wine and the bartender by his wife.

Joke 6:
A cowboy parks his sled dogs next to his igloo.
"What would you like to drink?" his wife asks.
"I'll have a some wine for the dogs, and a glass of water for myself. I still have to drive!"

Changing only the Logical Mechanism changes the joke even more. This is a logical implication of GTVH, as it states that this parameter is stronger than all the previously discussed KR. It still has to keep the structure of the punchline in the end (language), the structure being a *"man walks into a bar"* joke (narrative strategy), keep cowboys as the butt of the joke (target) and remain in the wild west setting with a bartender, ordering beer (situation). We created a possible variation in joke 7, where we changed the faulty logic of driving a damaged vehicle to a pun (*water/what her*), keeping as much of the original joke as possible.

Joke 7:
A cowboy parks his mare and walks into a bar.
"What would you like to drink?" the bartender asks.
"I'll have a bucket of beer for my mare, and a glass of water for myself. It's water heart desires!"

GTVH states that jokes being different on just Script Opposition are the least similar compared to any set of jokes differing on any other single parameter. In joke 8, we try to keep as many of the other KR of the original, root joke intact, while changing the two opposed scripts of "life ↔ death" (of the vehicle) to "sex ↔ non-sex".

Joke 8:
A cowboy parks his horse and walks into a bar.
"What would you like to drink?" the bartender asks.
"I'll have a bucket of beer for my horse, and a glass of lube for myself. I still have to ride him!"

Other humour researchers criticise several aspects of SSTH and GTVH. One big issue in the theories is that they do not set jokes in a social or interpersonal context [47]. Several different computational humour researchers argue that these theories are just analytical constructs, and that they do not provide any sort of guidance for joke production. Venour argues that these theories are far from implementable, and creates a more computationally tractable theory based on their theory in his thesis [63]. With her joke production engine Jape, Binsted argues that Raskin is wrong in saying that the scripts should oppose each other. She states that the only requirement for a joke is that they should be different [7]. In her evaluations of variations of jokes created with GTVH, Taylor also shows that these theories can neither predict nor explain why certain variants of the same joke have higher ratings [59].

2.2.4 Ritchie's Incongruence-Resolution Theory

As mentioned at the end of section 2.2.3, computational humour researchers criticised GTVH for its lack of giving insight in how to produce humour. Most notably, Ritchie argued that this formal theory was far from being implementable enough for computational humour, and created his own formal theory, extending the incongruity-resolution theory. He argues that the problem with most incongruity-resolution theories is that ambiguity can either create a joke or a misunderstanding [46]. Like Raskin, he described previously discovered informal notions by others more formally, such as the surprise disambiguation and Suls' two-stage model [46]. His theory also only covers verbally expressed humour, the humour conveyed in language, the opposite of situational humour [46]. Surprise disambiguation states that two different interpretations for the set-up of a joke must exist. The first interpretation is the most obvious one, whereas the second is the hidden meaning. The audience should only become aware of the second meaning through the punchline, which forces the second, hidden meaning as the only remaining possible interpretation. Suls' two-stage model on the other hand states that the punchline creates incongruity for which a cognitive rule has to be found to make it follow naturally from the set-up [46]. The model works almost like an algorithm that processes text until it finds a mismatch in its prediction. If the text is then ending and a cognitive rule is found to explain the mismatch, laughter ensues. If it ends before finding a mismatch, no laughter occurs. If the cognitive rule is not found, puzzlement occurs [46]. In contrast to the surprise disambiguation model, Suls' model does not require the initial set-up to contain two interpretations.

Ritchie proposed several properties to identify the relationships created by these less formal theories. Each property reveals a subproblem, making the properties more concrete for computational humour generation and detection than GTVH [48][46].

- Obviousness: This property states that the first interpretation should be more likely to be noticed than the second interpretation. This quantifies how obvious the initial interpretation of the set-up is. The subproblem here is to find the parameters that make one interpretation more obvious than another one.

- Conflict: This property entails that the punchline does not make sense with the initial interpretation of the set-up. The subproblem of conflict is finding the threshold for which a re-evaluation of the set-up has to occur.

- Compatibility: This property expresses that the meaning of the punchline is compatible with the second, hidden interpretation of the set-up. The subproblem is the difference between two interpretations being different and being amusingly different.

- Comparison: This property requires that a contrasting relationship between the two possible interpretations of the set-up should exist. The subproblem of Comparison is the same as Compatibility.

- Inappropriateness: This property is the critical factor. It states that the second interpretation should be inherently odd, inappropriate or taboo. This property entails that a text is a joke if it leads to absurdity or taboo. The subproblem is finding the factors that make an interpretation more amusing.

2.3 Related Concepts & Methods

In this section, we discuss several methods that have shown their use in previous systems and/or argue why they are useful for computational humour using humour theory. We leave out methods meant for humour generators and detectors for visual humour [10][52]. We focus on natural language processing and generation since our Goofer framework, discussed in the chapter 3, is intended for textual jokes.

2.3.1 Templates & Schemas

There are several approaches when it comes to text generation, such as using templates, grammars, Markov chains using n-grams (as discussed in section 2.3.2) and more recently recurrent neural network trained on characters [25]. Templates are probably one of the most simplistic and naive methods, but they are a powerful tool mostly employed in macros, user interfaces and chat bots [42]. They are also extensively used in computational humour projects [7][34][63][30][45].

A template, in the meaning we intend, can be defined as a text with variables, also called slots. These slots are filled in later by another data source. In this work, we call the values to be filled into a particular template "template values". The slots in a template for these template values tend to have their own variable name each, such that the data source can easily fill these in. It also allows data sources to work with different templates.

Schemas are often used as the data source for templates in computational humour [7][34][63]. They are used in the first computational joke production engine, JAPE. In this system, schemas are defined as the structure defining the relationships between key words in a joke [7]. They are responsible for generating the template values, often using a lexicon. A schematic view of a schema for the template *"What's <CharacteristicNP> and <Characteristic1>? A <Word1> <Word2>."* can be seen in figure 2.1. Along with its template, the instantiated schema generates joke 9.

Joke 9:
What's green and bounces? A spring cabbage. [7]

We can argue why templates and schemas are such an effective approach using GTVH. Templates often do not contain humour intrinsically. They are linkable to GTVH's weakest parameters, being Language and Narrative Strategy, as templates fix these parameters for the jokes they generate. We see schemas as responsible for GTVH's other parameters, as they provide the content of the joke. The relations used in a schema decide which Logical Mechanism is applied. The values generated by a schema decide on the Situation and possibly on a Target.

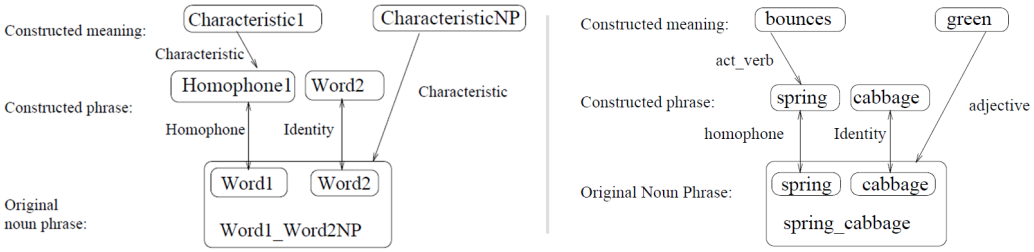


Figure 2.1: An uninstantiated JAPE schema (left) and an instantiated JAPE schema (right), taken from the JAPE paper [7]

The relations and the generated values together are responsible for finding the scripts that oppose each other for Script Opposition.

2.3.2 N-grams

N-grams are sequences of N words and the frequency they occur in a certain text [22]. Comparing the counts of different N-grams trained on different corpora is useful as a way to determine whether a text is more related to a certain corpus than another. N-grams in general are mostly used in natural language processing as the set of co-occurring words. A 1-gram, also called unigram, is equivalent to the frequency of a specific word in a corpora. 2-grams, also called bigrams, are used to find relations such as the frequency certain adjectives are used with certain nouns [41]. Creating an N-gram from a given corpus is a trivial task. A program splits the text into a list of words and counts every set of N words that occur in the list of words. This process results in an N-gram for sequences of size N. For example, the 2-gram for the sentence *"The quick fox jumps over the quick dog"* is:

- *the quick* (x2)
- *quick fox*
- *fox jumps*
- *jumps over*
- *over the*
- *quick dog*

An N-gram can also be used as a text generator using a Markov process. If you start with $N-1$ words, a Markov process fed with an N-gram model picks a word based on the chances provided by the frequencies of sequences of words that start with these $N-1$ words. The Markov process can then repeat this process with the most recent $N-1$ words of the text. This process results in a Markov chain of words, which can be turned into text.

Even though creating N-grams is an easy process, creating N-grams with little bias and high quality is a very rigorous task. N-grams are strongly dependent on their corpora, and balancing them is a difficult task. Luckily, companies like Google

provide pre-trained N-grams in Google Ngrams[2]. This set of N-grams contains N-gram models for $N \in [1..5]$ trained on 12 different large corpora containing n-grams from one million books with one frequency per n-gram per year.[3]

We have created and published an N-gram and Markov chain Java library[4], capable of executing complex N-gram and Markov operations. We used this library as part of the initial experimentation of this thesis.

Although N-grams are a very powerful tool for computational humour, they are not powerful enough to function as reliable humorous text generators using Markov processes and without further modifications. We can use Ritchie's incongruence-resolution theory [46] (see section 2.2.4) as an argument to prove this. N-grams are useful for detecting if a text is possible[5], and generating locally plausible texts. This implies that it is suitable for the Obviousness and Compatibility properties of this theory. Although n-grams can be used to detect Conflict when applied to different parts of a joke [41], the N-gram Markov process only knows about the last $N-1$ words. The model is not capable of ensuring the Conflict or Comparison properties other than by random chance. Since Markov chains often sound rather absurd, the Inappropriateness property could get fulfilled by chance. Due to the lack of Conflict and Comparison however, this absurdity does not lead to humour but to confusion [46].

2.3.3 WordNet

WordNet[6] is a lexical database of English grouping cognitive synonyms, also called synsets, of nouns, verbs, adjectives of verbs [13]. The database also contains meaningful relations between synsets, such as:

- Hypernymy/Hyponymy, the "is a" relation, e.g. *colour* is a hypernym of *blue* and *blue* is a hyponym of *colour* because *blue* is a *colour*.
- Meronymy/Holonymy, the part-whole relation, e.g. *processor* is a meronym of *computer* and *computer* is a holonym of *processor* because a *computer* has a *processor*.
- Antonymy, the opposite adjective relation, e.g. *good* $\leftrightarrow$ *bad*.
- Troponymy, the presence of manner relation, e.g. *jogging* is a troponym of *walking* because *jogging* is a way of *running*.

These relations are used to improve a computer program's understanding of natural language. Several interesting measures can be calculated using these relationships. Using shortest-path algorithms on the graph of relations, semantic similarities can be calculated [54]. Before WordNet, computational humour projects had to

[2] https://storage.googleapis.com/books/ngrams/books/datasetsv2.html
[3] http://commondatastorage.googleapis.com/books/syntactic-ngrams/index.html
[4] https://github.com/TWinters/Markov
[5] and has been used for this purpose in computational humour [58]
[6] https://wordnet.princeton.edu/

create their own lexicons, severely limiting their natural language knowledge and possibility space [7][63][30]. Extending previous systems with WordNet strongly improved their performance [34]. Researchers have also extended WordNet in various ways. WordNet Domains is a database linking WordNet synsets to WordNet topics [32]. It also contains other extensions such as WordNet-Affect [16], which correlates synsets with their affective words.

We can argue using humour theory why WordNet has been used in several successful computational humour projects [34][57][41]. First of all, WordNet adds simple natural language understanding, which helps the Compatibility property to find ambiguous words. Using similarity measures, contrasting relations for Comparison can be found. Lastly, the domains of WordNet Domains can help ensure the dual opposing scripts necessary for SSTH and GTVH's Script Opposition.

2.3.4 Part of Speech Tagging

Part of Speech (POS) taggers are capable of marking words with their POS based on their definition and context. WordNet can be used as a simple POS for tagging single words as either an adjective, noun, verb or adverb, and is only capable of using the word definition, not the context. A more popular POS tagger, often used in computational humour programs [6], is the Stanford Log-linear POS tagger[7]. This tagger is capable of tagging sentences with their appropriate POSes based on their context and uses more specific POS[8] than WordNet.

Just like WordNet, POS taggers bring more natural language understanding to computer programs. They are for example capable of filtering unsupervised data such that values are of the correct part of speech for certain jokes [26][41].

2.3.5 Levenshtein Distance

The Levenshtein distance, also called the edit-distance, is a similarity measure to determine the minimum number of additions, deletions or substitutions a sequence has to receive in order to be transformed to a second sequence [31]. For example, for the words (which are sequences of characters) "tree" and "tired", the Levenshtein distance is 2, because the shortest way to transform tree to tired requires two steps, being tree → tiree → tired.

This distance measure is used in computational humour to find similar sounding words in order to create puns [50][63]. Often, the words are first transformed into sequences of phonemes, as they represent the way a words is verbalised. We can recognise this technique as beneficial by linking it to the Compatibility property of Ritchie's IR theory. Similar sounding words can correlate the punchline to the two interpretations in their verbal or written form. The second interpretation can then later emerge from the semantic meaning of the whole punchline.

[7] https://nlp.stanford.edu/software/tagger.shtml

[8] See http://www.comp.leeds.ac.uk/amalgam/tagsets/upenn.html for a full list of the used POS tags

2.4 Related Humour Generators

2.4.1 JAPE and STANDUP

JAPE (*Joke Analysis and Production Engine*) [7] and STANDUP (*System To Augment Non-speakers' Dialogue Using Puns*) [34] are humour generators for punning riddle in a question-answer format. JAPE is seen as one of the first significant computational humour systems [6]. It introduces the templates and schemas technique, as discussed in section 2.3.1 and seen in figure 2.1, to generate riddles based on simple phonetic puns. Since JAPE's schemas required certain linguistic relationships, a custom, humour independent lexicon was built [7]. STANDUP extends this generator by modifying and adding schemas in such a way that WORDNET can be used instead of the hand-crafted lexicon, increasing the possibility space drastically. STANDUP also uses the Unisyn dictionary[9] to convert words from WORDNET to their phonemes, which are used with a Levenshtein algorithm with custom weights for each modification to identify homonyms and similar sounding word relations.

STANDUP redefines schemas as consisting of five parts: the header (which is the name with the variables of the template), the lexical precondition (= the syntactic, phonetic, structural or semantic constraints on the variables), the question and answer specifications (the templates to match the assigned variables to, along with some lexical constraints for their template values) and the keywords (used to define equivalence between jokes) [34]. An example of the *newelon2* schema of STANDUP can be found in figure 2.2. This schema uses the template *"What do you call a* `SynHomB` *with a* `MerA`*? A* `A HomB`*"*. Applying `NP`=*computer screen*, `A`=*computer*, `B`=*screen* and `HomB`=*scream* results in joke 11 [34]. Note that `shareproperty(NP, HomB)` defines a subschema for the question specification. This subschema fills in `SynHomB` as a synonym of `HomB` and `MerA` as a meronym of `NP`.

Header: `newelon2(NP, A, B, HomB)`
Lexical preconditions: `nouncompound(NP,A,B), homophone(B,HomB), noun(HomB)`
Question specification: `shareproperties(NP, HomB)`
Answer specification: `phrase(A,HomB)`
Keywords: `[NP, HomB]`

FIGURE 2.2: The newelon2 schema, from the STANDUP system [34]

Examples of JAPE generations are joke 9 mentioned earlier and joke 10. Examples of STANDUP generations are joke 11, 12 and 13.

JOKE 10:
What do you get when you cross a sheep and a kangaroo?
A woolly jumper. [7]

[9] http://www.cstr.ed.ac.uk/projects/unisyn

Joke 11:
What do you call a shout with a window?
A computer scream. [34]

Joke 12:
Why is a sick character different from a static letter?
One is an ill star, the other is a still r. [10]

Joke 13:
What kind of a debt is solitary?
A lone loan. [11]

2.4.2 Petrovic & Matthews Analogy Generator

Petrovic & Matthews have created a model for generating analogy jokes using the *"I like my X like I like my Y, Z"* template [41]. They argue their program is the first fully unsupervised humour generation system, as they did not rely on a hard-coded schema approach, but on relations used in a minimisation model. Their model encodes five relations about the X, Y and Z *"I like my X like I like my Y, Z"* jokes. It fixes the template values such that every template value is a single word, more specifically that X and Y are both nouns and that Z is an adjective. The system requires X to be defined by the user. The system generates further values using Google Ngrams by choosing Y and Zin a way such that Z is an adjective usable for both X and Y. The relational assumptions used in the model are that the joke is funnier the more often the attribute is used to describe both nouns, the joke is funnier the less common the attribute is, the joke is funnier the more ambiguous the attribute is and the joke is funnier the more dissimilar the two nouns are [41]. These assumptions are all shown to be implementable to be a metric resulting in a number. For most of these metrics, they use Google Ngrams to find the appropriate values. The research also uses WordNet[13][37] to look up the number of senses of a word. We discuss these metrics in more detail in section 4.5, where we extend this model. In order to rank how funny a joke is, the program minimizes the product of these five relations. Unlike our approach, this research thus does not use machine learning techniques on training data to generate jokes.

This system is quite successful, as its generations were considered funny 16% of the time. Human-produced jokes using the same template were considered to be funny in 33% of the time. [41]. Examples of jokes generated by this system are jokes 14 and 15.

Joke 14:
I like my relationships like I like my source, open [41]

[10] https://twitter.com/jokingcomputer/status/887258183147950080
[11] https://twitter.com/jokingcomputer/status/876024138481233922

Joke 15:
I like my coffee like I like my war, cold [41]

2.5 Related Humour Detectors

2.5.1 DEviaNT: Innuendo Classification

DEviaNT (*Double Entendre via Noun Transfer*) is a system that detects innuendos. More specifically, it detects if it is appropriate to say *"That's what she said"* after a certain, given sentence [26] An innuendo is a piece of text that has a double meaning, where the second meaning is often sexual. An example of a *That's what she said* joke is shown in joke 16.

Joke 16:
Michael: "They taste so good in my mouth."
Stanley: "That's what she said!" [12]

The system executes the detection of innuendos by checking if the nouns that are present could be euphemisms for sexually explicit nouns and if the structure of the sentence is similar to one in the erotic domain [26]. It uses 2-grams of an erotic corpus with the 2-grams trained on the Brown corpus [14] in order to calculate adjective vectors. These adjectives vectors are then compared to create metrics like noun sexiness, adjective sexiness and verb sexiness, which calculate how likely this word is to be used as a euphemism.

Other research on innuendos also exists, like systems for replacing words in sentences as if they were corrected by auto-correct to a profane word [62] or in common sayings [61].

2.5.2 One-liner Recognition

Mihalcea and Strapparava created a system capable of distinguishing non-humorous texts from humorous texts, more specifically one-liners [36]. The latter researcher is well known in computational humour for having co-founded the first European funded computational humour project, HAHAcronym [57]. The only other research done in humour recognition before this research is a knock knock joke recognition system [58]. The system analyses short sentences (less than 15 words) for structural features common in jokes, such as alliteration, antonyms, adult slang (using WordNet Domains, rhymes and simple syntaxes. Training data is collected by starting with a small set of one-liners. They created a scraper that scraped the web for pages containing one-liners from the already collected set of one-liners, resulting in a bootstrapping process for joke collection. The end-result of this process is a dataset of 16 000 one-liners. They use several other types of short sentences as the negative training data, such as Reuters news lines and lines from the British National Corpus. They then apply Naïve Bayes and SVM classifiers on this training set, and achieve

[12] Lines from TV series *The Office*, season 3 episode 5

79% accuracy when testing one-liners against sentences from the British National Corpus, and 97% accuracy against Reuters news lines. One of the conclusions of the research is that future research should focus on getting the right metrics, rather than a big dataset, as the system's learning curves flattened after about 10% training data.

2.5.3 T-PEG: Template Extraction

T-PEG, for *Template-Based Pun Extractor and Generator* is a system created for the extraction of templates, aimed at pun templates[20]. Their generator works similar to Standup, and is thus less relevant for this research. This system is meant for punning riddles such as those created by Japeand Standup. T-PEG uses very similar relationships as those used in the latter two systems, such as similar pronunciation, hyponymy and meronymy. Just like Standup, they rely on WordNet and UniSyn as their lexical resources. One difference is that they also rely on ConceptNet[13] for some of the semantic relationships [20]. In order to find a template, the system is just given a single punning riddle, for which it replaces some words with variables. The template extraction algorithm is capable of detecting three types of variables for the slots in a template. The first type is the regular variable, a noun, adjective or verb. The second type of variable is a similar-sound variable, such as *bear* and *bare*. The last type they identify is the compound-word variable, which combines two similar-sound variables. T-PEG is also capable of using hidden variables related to variables that are actually used in the template. In the author's evaluation evaluation, 69.2% of the found templates were actually usable for joke generation [20]. They noted however that their system was heavily reliant on the existence of linguistic relationships between the words of the joke.

Other researchers tested T-PEG by clustering several similar Standup-generated jokes based on structural similarity [1]. Their system extracts templates using T-PEG, and employs agglomerative clustering on these templates using a single majority rule. They use semantic similarity evaluation function used in [35] and algorithms used for aligning of complex expressions, which a template is. They tested this system by automatically verifying whether the templates and schemas used in Standup generated jokes were correctly found. It has an overall precision of 61% [1].

2.6 Conclusion

Computational humour is a challenging field with several ways of achieving different types of humour. There are several different humour theories, some more formal than others. We used these theories to argue for the relevancy of certain technologies and concepts used in computational humour. Few theories have however actually been practical to guide the actual creation of existing computational humour. As such, existing computational humour systems are mostly focused on single tasks on a single type of humour, as we have shown by discussing existing systems.

[13] http://conceptnet.io/

Chapter 3

Generalised Joke Generation

3.1 Introduction

In this chapter, we specify a theoretical computational humour framework that is capable of generating humour based on given rated jokes. To achieve this, we extend and generalise several concepts, methods and theories employed in previous research, in order to make it applicable to a broader range of humour. More specifically, we generalise the notion of schemas and create a theoretically founded set of metrics for computational humour purposes. We then describe the flow of the framework, as well as a detailed description of every data structure and component. We call this framework GOOFER, which stands for *"Generator of One-Liners From Examples with Ratings"*. We use the framework to implement a generalised system for generating *"I like my X like I like my Y, Z"* jokes in the next chapter.

The GOOFER framework first extracts the templates from the given jokes and transforms the dataset to a dataset containing the template values with their ratings for each template. This transformed dataset is used to learn classification schemas for every template. A generator then generates a large quantity of template values. The classification schema picks the template values that it considers best, based on its learned humour knowledge. These template values are then inserted into their template to create a set of output jokes.

3.2 Generalising Schemas

As discussed in section 2.3.1, a schema defines the relationships between the variables of templates of the jokes. In this section, we generalise the normal, constraint-based schema to a probabilistic schema, and use this to create a new type of schema capable of using classification algorithms.

3.2.1 Constraint-based Schemas

JAPE, STANDUP and several other humour generators use schemas that enforce strict relations between the variables of a template using a constraint based ap-

proach [7][34][63][45]. They often use lexical relations such as synonyms, meronyms, hyponyms, (quasi-)homonyms and their combinations. This technique makes it straightforward to generate jokes based on a seed: once a schema word is filled in, the possibilities for the other words are limited to those that are in these strictly defined relations with the already decided word(s). Not all of these words have to eventually occur when filled in into the template, some words might serve as hidden links between other words. The limited search space of constraint based approaches has two effects: it has the benefit of being efficient, but their possibility space might appear small compared to generators using probabilistic approaches. In LiBJoG, a light bulb joke generator [45] for example, the schema specification is almost as big as the generation space. The generator uses a schema that defines stereotypes for types of people, and links stereotypes to punchlines of jokes that follow the set-up line *"How many [type of people] do you need to change a light-bulb?"*. This mapping is close to a 1-to-1 mapping of types of people with the punchlines. This means that their possibility space is almost as big as their schema specification, which could be seen as a strong argument for saying that this generator is less effective than for example the Standup generator, since most of its generated jokes have essentially been pre-written.

These type of schemas can be written in a Prolog-like notation, as can be seen in figure 2.2). This notation reveals that this specification of a schema is both a generator and a checker (although the real implementation of Standup translates these schemas to SQL database lookups instead of using real Prolog [34]). This notation only works for constraint-based schemas though, and we have to come up with a different notation for schemas if we want to incorporate probabilistic metrics.

3.2.2 Probabilistic Schemas

Recognising components used for joke generation as templates or schemas is a useful technique. The approach of using templates and schemas can be applied to CH systems that are not using schemas and templates explicitly. Venour showed how a Tom Swifty joke generator[30] was implicitly using templates and schemas [63]. We believe that we can extend his approach even further, and even map other systems that do not use constraint based approaches onto a more general type of schema. In order to achieve this, we first need to introduce the notion of a probabilistic schema.

Probabilistic Schema

We define a probabilistic schema as having the following components, inspired by the definition of a schema of the Standup generator [34], as discussed in section 2.4.1.

- **Header:** the name of the schema, as well as the variables used in this schema.
- **Metrics:** the metrics used and which variables they are used on. These metrics map the input variables to a number.

- **Aggregation:** The way to aggregate the metrics and to choose the output jokes, e.g. all metrics must be above a certain value or the sum of certain metrics is higher than the sum of other metrics.
- **Generator:** a more primitive generator to generate proposals for jokes.
- **Template:** the template with slots for the defined variables.
- **Keywords:** the keywords, being the most relevant variables, used for calculating equivalence between schema outcomes.

Generalising Constraint-based Schemas to Probabilistic Schema

In order to show that a probabilistic schema is a generalisation of the constraint-based schema, we have to show that a theoretical mapping from the latter to the former exists. Firstly, the header is transferable, just like the keywords and template. Secondly, the constraints can be mapped to functions that outputs 0 if the given constraint is violated and 1 if it is satisfied. These functions form the metrics of a probabilistic schema. Since the schema only allows the assignment of variables that make all the functions map to 1, the aggregation function is defined as a function that only returns *true* if all metrics return 1. The only attribute left to define now is the generator, since this came for free using the Prolog-like constraints. As this is a theoretical mapping and efficiency can thus be ignored, and we can define the generator such that it generates all possible combinations of assignments to the variables using all words present in the lexicon of the constraint-based generator. This concludes our mapping from a constraint-based schema to a probabilistic schema.

Mapping Existing System to Probabilistic Schema

With this new notion of a probabilistic schema, we can map the analogy generator discussed in section 2.4.2 to the template and schema approach. As discussed earlier, this system uses metrics to calculate five metric values from a joke using the *"I like my X like I like my Y, Z"* template. These five values are numbers stating the dissimilarity between X and Y, the relatedness of X and Z, the relatedness of X and Y, the ambiguity of Z and the uncommonness of Z. It then applies minimisation of the product of these values over the space of possible jokes. In order to find these possible jokes, it uses Google Ngrams to find suitable sets of noun-noun-adjective [41]. Since the system is only generating *"I like my X like I like my Y, Z"* jokes, we say it has one template. Their model represents how to fill this template. We can thus conclude that this system uses one template with one schema. It is quite straightforward to map the used model to our probabilistic schema. This mapping can be found in figure 3.1.

3.2.3 Classification Schemas

One big hurdle in creating computational humour systems using the templates and schemas approach is that these schemas tend to be handcrafted. This requires

Header: `petrovic_matthews_analogy_model(X,Y,Z)`
Metrics: `relatedness(X,Z)`, `relatedness(Y,Z)`, `dissimilarity(X,Z)`, `ambiguity(Z)`, `uncommonness(Z)`
Aggregation: Product of the metrics is below threshold t.
Generator:

1. Take X from input.
2. Generate Z from X as a possible adjective used with X with Google Ngrams.
3. Generate Y from Z as a possible noun used after Z with Google Ngrams.

Template: `I like my <X> like I like my <Y>, <Z>.`

Keywords: `[X,Y,Z]`

Figure 3.1: Our mapping of the analogy generating model created by Petrovic & Matthews [41] to a probabilistic schema

knowledge about the jokes beforehand, which is found manually. An advantage of these handcrafted systems is that they can demonstrate the validity of simple humour theories about a certain joke type by translating the theories to schemas and judging the jokes that result from such a system.

In this thesis, we focus on finding ways of crafting these schemas automatically using the machine learning tools available today. For this, we use our newly defined notion of a probabilistic schema. A classification algorithm can be used to learn which metric values are correlated with which discrete rating (e.g. from the mode score of a collection of ratings, or the score of one specific person). A regression algorithm can be used in a similar fashion to estimate a non-discrete rating (e.g. from the average rating). These algorithms are able to fulfil the aggregation functionality by training them to distinguish good jokes from bad jokes, and only allow jokes with an estimated score above a certain threshold. They also need to be accompanied by a content selector (a naive value generator) as the generator used in the probabilistic scheme. The classification algorithms thus judge whether any of the possibilities generated by the content selector exceeds a certain threshold in order to be considered "good". We call this type of probabilistic schema a *classification schema*.

3.3 Metric Set Identification

Now that we have defined schemas such that they are capable of using classification and regression algorithms, we need to define the metrics usable for calculating features from template values. These metrics should be metrics that make sense for a joke judging algorithm. Ritchie's incongruity-resolution theory, as discussed in section 2.2.4, identifies five properties necessary for verbal humour [46]. We use these properties, along with other existing research, to identify and argue a solid set of potential metrics to identify humour.

3.3.1 Obviousness Metrics

The Obviousness property states that the first interpretation should be more obvious than the other, hidden interpretation. The more the set-up of the joke is linked to the first interpretation than to the second, the bigger the impact of the punchline. Possible metrics for approximating this property are:

- **Association / Semantic distance:** This metric uses a lexicon, such as WordNet or ConceptNet, in order to find the distance between the nodes in a lexical graph. Several (combinations of) different types of lexical relationships such as meronymy and hyponymy could be used to create variants on this metric. Smaller distances between nearby words in the set-up are correlated to higher obviousness.
- **Word frequency:** This metric uses 1-grams to calculate how often a certain word occurs. If the word frequency is high, chances are that this word is a common word, where people associate this word with one specific, obvious meaning more easily. This metric can be implemented using for example Google Ngrams' 1-grams. This metric often used in research [41].

3.3.2 Conflict Metrics

The Conflict property states that the punchline does not make sense with the first interpretation of the set-up. Possible metrics to approximate this property are:

- **Association / Semantic distance:** Just like the metric used for the Obvious property. The larger distances are however correlated to higher conflict.
- **Frequency of word combinations (n-grams):** N-grams are capable of revealing how often words are used together. If particular words of the punchline are used more with certain other words of the setup, the meaning of this combination of words suddenly becomes more important than all the other used words linked to the first interpretation, increasing the conflict with these words. This metric can be calculated using Google Ngrams. Other computational humour research also uses this metric [41].

3.3.3 Compatibility Metrics

The Compatibility property states that the punchline should make sense with second, hidden interpretation of the set-up. Possible metrics to approximate this are:

- **Frequency of word combinations (n-grams):** The same metric as discussed in the section 3.3.2. The higher the frequency particular words are used together, the higher the compatibility.
- **Number of meanings metrics:** The number of meanings a word has is related to its ambiguity [41]. This metric can be calculated using the number

of different definitions for a certain word in WordNet. In Ritchie's IR-theory, he states that the surprise disambiguation IR-theory requires the set-up to be ambiguous, with one obvious meaning. Words being more ambiguous thus help the set-up. This metric is used in Petrovic's model for generating *"I like my X like I like my Y, Z"* jokes [41].

- **Homonyms Metrics:** Similar sounding words and homonyms (words that sound exactly alike) can link the first and the second interpretation, but ensure that only the second interpretation is appropriate in a context. This can be implemented using a pronunciation library like Unisyn. The Datamuse API[1] can also be used to query similar sounding words. Several computational humour researchers have used this metric [34][7][62][63].

3.3.4 Comparison Metrics

The Comparison property states that there should be a contrasting relationship between the two possible interpretations of a set-up. This property can be linked to the opposing scripts of the SSTH theory. Possible metrics for property are:

- **Opposing Domains:** Mapping two different words to their appropriate domain(s) could reveal opposite domains. HAHAcronym, a generator for funny acronyms, uses WordNet Domains to find the appropriate domains and uses predefined domain oppositions (based on SSTH) to check if the words could be opposing [57]. This method can also be extended by mapping domain oppositions to different values.

- **Word Similarity between Related Interpretations:** By using similarity measures to find the word similarity between words that are related to certain words used, contrasting relationships between interpretations of used words are discoverable. The definitions and distances between the words can be found using WordNet.

- **Adjective Vector Difference:** An adjective vector can be formed by calculating the adjectives and their frequencies used with a particular noun. Comparing the adjective vectors of a noun results in a value describing how different the contexts are that certain words occur in. Other computational humour research also uses similar values [26][41].

3.3.5 Inappropriateness Metrics

The Inappropriateness property states that the second interpretation should be odd, inappropriate, taboo and/or absurd. Possible metrics to measure this property are:

[1] http://www.datamuse.com/api/

- **Word sexiness:** Adjective sexiness and noun sexiness are used in DEViaNT innuendo detection system (see section 2.5.1) to calculate how likely it is that a word is an innuendo-related word [26]. For an adjective, it compares the frequency of the word in a sexual corpus with a non-sexual corpus. For a noun, it calculates this value by comparing the frequency of the noun's adjective vector to adjective vectors of body parts in a sexual corpus.

- **Word frequency:** The frequency of a word is related to its unpredictability, and is thus capable of identifying odd words [41]. This implementation metric has already been discussed in the section 3.3.1.

The identified metrics form a foundation for the knowledge base of our generic framework and are used by the machine learning algorithms to extract features from given jokes. This metric set is not exhaustive, and some metrics are complement each other. However, it shows how to cover as much as possible with a small amount of metrics from a humour theory point of view. The knowledge base used in the framework we present is extendible, such that it can be used for a particular type of joke, to improve its performance. This also allows the system to be used to test new theories and hypotheses.

Note that the identified metrics are used to map single words or sets of single words to a feature value. As mentioned earlier, being able to deal with multiple word template values is seen as an extension, discussed further in section 5.1.2. We offer some possible solutions however in our implementation in section 4.4.1 .

3.4 GOOFER Framework

In the previous sections of this chapter, we have generalised core computational humour concepts. We use these generalised concepts to build the pipeline of a framework that is capable of learning how to generate jokes based on a corpus of human-rated jokes. Figure 3.2 shows an overview of this framework.

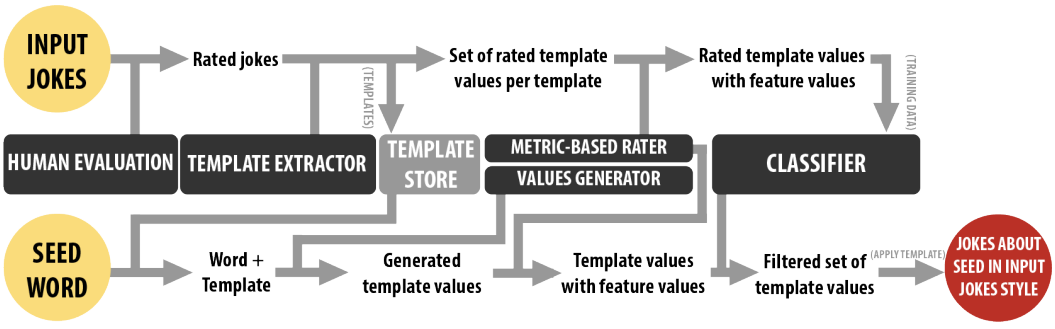


FIGURE 3.2: A schematic overview of the generic GOOFER framework for joke generation from examples.

3.4.1 Flow of the framework

1. A user provides a set of textual jokes to the system. These input jokes could be obtained from any source, being from the human evaluation component, other/previous computational humour systems and/or other external sources.

   Example:

   a) I like my relations like I like my source, open. [41]

   b) I like my coffee like I like my war, cold. [41]

   c) The sailor bears a stress. Pier pressure. [63]

   d) I like my girlfriend like I like my estate: real.

   e) The lord cultivates a region. A baron land. [63]

2. The given jokes are evaluated by the `Human Evaluation` component. This is the only step that requires human interaction. The component converts the set of jokes to a set of jokes with scores.

   Example: (using the jokes from step 1)

	rater 1	rater 2	rater 3
joke a	5	4	5
joke b	4	4	3
joke c	5	3	4
joke d	2	1	3
joke e	3	2	2

3. The `Template Extractor` component then detects the templates present in the given jokes using a template extraction algorithm, such as the one discussed in section 2.3.1. The component stores the templates in the `Template Store`. It also extracts the template values of the input jokes and groups them per template, along with their ratings, and continues executing the following steps until step 5 separately for every template.

   Example:

   a) **Template 1:**
   "I like my X like I like my Y, Z."
   Values:
   - relations, source, open | [5,4,5]
   - coffee, war, cold | [4,4,3]
   - girlfriend, estate, real | [2,1,3]

   b) **Template 2:**
   "The A B a C. D."
   Values:
   - sailor, bears, stress, Pier pressure | [5,3,4]

- lord, cultivates, region, A baron land | [3,2,2]

4. The `Metric-based Rater` uses its metrics in order to generate feature values for every set of template values. It does this by applying every unary metric to every template value, every binary metric to every couple of template values etc.

   Example:

X	Y	Z	score	$freq_X$	$freq_Y$	$freq_Z$	$freq_{XY}$	...
relations	source	open	[5,4,5]	3 831 210	4 050 904	4884757	0	...
coffee	war	cold	[4,4,3]	784 065	9 211 826	2 010 173	0	...
girlfriend	estate	real	[2,1,3]	75 392	998 669	5 284 057	0	...

5. The `Classifier` receives the calculated feature values and rating of the rated template values. It trains a model using these values that is capable of assigning scores to other template values. This enables the system to distinguish good jokes from mediocre jokes from non-jokes. The trained model functions as the classification schema for the template.

6. The second phase, the generation phase, starts with the user entering possible seed words for the generated jokes to be about. They can be left blank if the generator is free to generate jokes about any subject.

   Example: coffee, women

7. A learned template from the template store is combined with this seed. The following steps are done separately for every template.

   Example: coffee, women | Template 1

8. The system uses the `Values generator` to generate candidate template values based on the seed word and possibly instructions of a schema associated to the template.

   Example: The example generation system uses 2-grams: it finds words used both as adjectives for coffee as for women. This assumes the classifier step knows that X and Y tend to be nouns and Z tend to be adjectives, and has informed the generator.

   - coffee, women, final
   - coffee, women, black
   - coffee, women, popular
   - coffee, women, rich
   - coffee, women, Irish

9. The generated template values are then evaluated by the same metrics from the knowledge base as the training data in step 4.

X	Y	Z	score	$freq_X$	$freq_Y$	$freq_Z$	$freq_{XY}$	...
coffee	women	final	?	784 065	9 929 965	3 070 183	0	...
coffee	women	black	?	784 065	9 929 965	5 707 260	0	...
coffee	women	popular	?	784 065	9 929 965	2 311 546	0	...
...	...	...	...	...	...	...	...	...

10. This set of template values with their feature values are then scored by the `Classifier` component by detecting similar patterns as the high scoring jokes from the training data, using the learned classification schema. It only passes template values with a score above a given threshold to the next step. It might also filter jokes that are too similar to higher rated jokes using the keywords of the classification schema.

    Example The output of the classification algorithm might be the following set of template values, rated by the `Classifier`.

    - coffee, women, final | 3
    - coffee, women, black | 5
    - coffee, women, popular | 2
    - coffee, women, rich | 2
    - coffee, women, Irish | 4

    If the threshold is 4, the template values "coffee, women, black", "coffee, women, Irish" are passed through.

11. The filtered set of values is then applied to the template that was chosen in step 7, which creates a textual joke. The algorithm finishes by outputting these jokes.

    Example The output of the program is the following set of jokes.

    - I like my coffee like I like my women: black.
    - I like my coffee like I like my women: Irish.

3.4.2 Components

Human evaluation The jokes have to be rated for the classifier to distinguish good jokes from bad jokes. The human evaluation component is responsible for this task. It is the only place, apart from the user of the system (who delivers seed word(s), and potentially the input jokes), where humans are involved in the algorithm. Alternatively, rated input jokes can be delivered straight to the Template Extractor.

Template extractor The template extractor component is responsible for detecting templates in jokes, and extracting the values that are filled into these templates from the jokes. This component thus detects and extracts the Narrative and Language KR of GTVH. It stores the detected templates in a template store. Each of these templates is also associated with their own trained classifier. We discussed in section 2.5.3 how template extraction works.

Template store The template store is responsible for storing all the templates found by the template extraction component. It also stores the schemas trained from the jokes using this template.

Metric based rater The metric based rater is responsible for assigning values from the metrics to the values, which are to be inserted in a template. The metrics are part of the the knowledge base, as specified in section 3.3 and are each responsible for mapping template values to a number or a label. The component thus is responsible for the LOGICAL MECHANISM and SCRIPT OPPOSITION KR of GTVH.

This component should be extendible such that users can extend the knowledge base to give a priori information about metrics they deem important for a particular set of jokes. It can also be extended in order to test certain hypotheses about a set of jokes.

Values generator The values generator component is responsible for selecting candidate content to be inserted in a template to create a joke. It might receive a seed word to generate jokes with. We can see that this component is responsible for taking care of the SITUATIONAL and optionally the TARGET KR of the GTVH theory. It receives instructions from the related template about how to generate these values. The instructions from the template could include constraints like the type of word to be generated. E.g. the classifier component could have detected that certain template values are (almost) exclusively nouns. This information decreases the size of the possibility space, but should increase the score the classifier assigns to the generated values on average.

The generator could deploy any kind of text generation techniques. Since template values used in this thesis are constrained to be a single word, an interesting way is using n-grams to generate words that are related to each other. We discuss a possible extension for (sub-)sentence template value generation in section 5.1.2.

The values generator could also be another computational humour system that employs a template and schema approach, or can be shown to use an equivalent system, as we did in section 3.2.2. Using such a generator in the GOOFER framework would extend an existing system by allowing it to learn from previous generations, be it to solve the temporal or individual dependency of humour or both, as discussed in the introduction of this thesis.

Classifier The classifier component is used to assign scores to values to be inserted into templates using a classification schema. The classification schema is different from template to template. This classification schema could either use a classification or regression algorithm as the metric aggregator of the schema.

Using classification algorithms is useful as there are a lot of interesting algorithms to pick from, going from decision trees like RandomForest and J48 (Java version of C4.5 in WEKA) to neural network approaches. If a simple decision tree approach is used and the knowledge base is composed of understandable humour theory metrics, the resulting decision trees might be understandable for humans. Decision trees can

also help by stating which attributes cause the largest impurity decreases. These statistics can help humans learn about what makes a specific type of joke funny.

Regression algorithms are useful because they are capable of dealing with real numbers. This means that they can use the average score a joke gets as the score. This is in contrast with classification algorithm that only support a certain number of classes, for which we use the mode or the rating of a single user. This might also work in the algorithm's disadvantage, as the perceived funniness can be entirely different between people, meaning that the average rating might often be close to the middle of the rating range.

One important factor to account for when choosing a classifier or regression algorithm, is the ability to deal with noisy metrics used in Goofer. The framework is deploying a large number of metrics on all possible combinations of the template values. This means that some of the features might be noisy. The chosen algorithm should thus be quite resistant to possible noise.

3.5 Notes about the Framework

Note that this framework could be extended to domains other than humour. For example, it could be used to understand and generate texts in the romantic domain. By adopting the metric set to a metric set appropriate for romantic texts and using ratings based on how romantic the words are, this framework could be modified to generated romantic analogies, given that the texts in the new domain also tend to work with template-like texts.

A secondary goal of our framework is for it to be able to comprehensively explain what distinguishes these good jokes from bad jokes. This can be done by using for example a decision tree as the classifier component in the framework and using human understandable metrics. This knowledge could then be used to build models for more efficient, smaller systems. It could even be used by humans to study humour theory. In other words, this framework is able to be used to answer the following question: "What relations between the template values are the most relevant in order to be funny?"

3.6 Conclusion

In this section, we described the specifications of a generic framework that is capable of generating jokes in a similar style to the given jokes, and can distinguish between (and thus filter) good and bad jokes. We did this by generalising the existing computational humour notions of templates and schemas in such a way that it is possible for a machine to learn features of jokes that follow a certain pattern. Using humour theory, we also identified a knowledge base of generally applicable potential metrics to use in the framework. This framework is capable of finding its own proper schemas and thus could free computational humour researchers of the task of manually crafting those. We also showed how this framework could be used for setting up humour theory experiments.

Chapter 4

Generalised Analogy Generator

4.1 Introduction

In the previous chapter, we presented the theoretical specification of our Goofer framework, which is capable of learning jokes from examples. In order to prove that such a system could work and to evaluate its results, we implemented a subset of its components. The system is capable of generating analogies. We call it the *Generalised Analogy Generator*, or Gag for short.

The implementation covers all of the Goofer framework steps (and required components) except for the template extraction step (see step 3 of section 3.4.1). There is no need showing that the template extraction step of the pipeline of the Goofer framework is implementable, since considerable research has been done on template extraction, and previous research on this topic for jokes has already shown its success [20][1][27]. We still cover several subtasks of the template extraction component to make the system more robust. The steps following the template extraction step in the Goofer are executed for every single template separately. Focusing on these steps allows us to examine on the automatic learning of classification schemas. These schemas are used to generate jokes following a single template from example jokes using this template.

We choose to focus on jokes of the template *"I like my X like I like my Y, Z"* for the implementation of the subset of components of the Goofer framework. The first known published research about *"I like my X like I like my Y, Z"* jokes was done by Petrovic and Matthews [41], as we discussed in section 2.4.2. Their goal was proving that a system using unsupervised learning techniques (namely Google Ngrams) was able to generate funny analogies using the model they created. Their approach uses a human-crafted probabilistic schema, whose mapping from their model we created in section 3.2.2. Our approach presented here, on the other hand, employs a generalised computational humour framework that learns schemas from examples. Since the same template is chosen for Gag, it becomes possible to use similar metrics and to compare these approaches.

We require a considerable quantity of training data in order to make the Gag system function. Since a large dataset of *"I like my X like I like my Y, Z"* jokes

with a large number of ratings does not exist, we spent a significant amount of time developing a platform for the collection of jokes and ratings for these jokes. This platform is called JokeJudger. We discuss its implementation, features, philosophy and learned lessons in this chapter.

4.2 Simplified Pipeline

The pipeline for our system to create *"I like my X like I like my Y, Z"* jokes, called Gag, is depicted on Figure 4.1. Compared to the Goofer framework (see figure 3.2), some components are removed or simplified. We discuss the implementation of the components in the rest of this chapter.

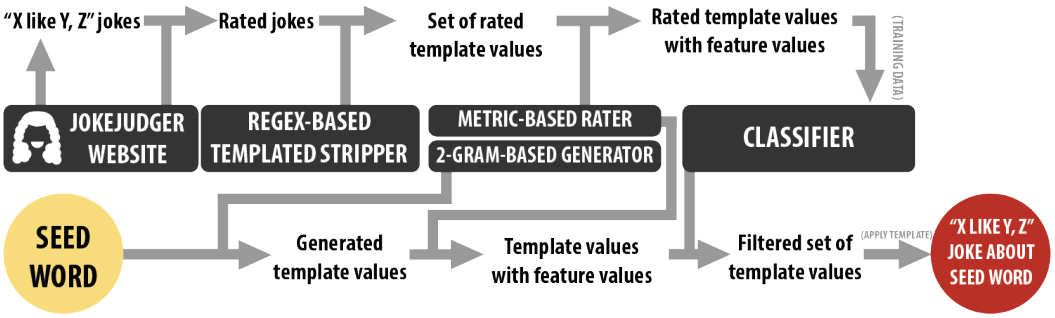


Figure 4.1: A schematic overview of how *"I like my X like I like my Y, Z"* jokes could be generated following the design of our Goofer framework.

The first simplification we executed is simplifying the template extraction component. As mentioned in section 4.1, there is no need to have a template extraction component since a lot of research on template extraction already exists [20][1][27]. The role of the template store has also become redundant, since all jokes use the *"I like my X like I like my Y, Z"* template.

The second simplification is the content selection generator used. As we discussed in section 3.4.2, the generator could use N-grams to generate combinations of single words as template values. We have chosen N-grams to be the only content selection source of the system. Since we know that jokes using the *"I like my X like I like my Y, Z"* template mostly use nouns for X and Y and adjectives as Z, we built our template values generator accordingly[1]. Note that the information of the POS of the words of a template are found for the template extraction algorithms, as discussed in section 2.5.3. No extra information is thus added to the system. The template values generator of this system applies n-grams in order to create single-word template values. This is similar to the way the Petrovic's *"I like my X like I like my Y, Z"* system generates their template values [41]. We discuss this generator in more detail in section 4.6.

[1]This is the most obvious kind of content to be selected for this template. In reality however, we noted that a significant amount of people diverge from these word types when creating this type of joke themselves, for example by naming a relation to another noun as Z.

The third simplification/extension is that the human evaluation component is also responsible for generating the input jokes used in the training data. For this human evaluation component, we have built a platform, called JokeJudger, for the creation and the evaluation of jokes. This ensures that the format of the joke is following the supported template (*"I like my X like I like my Y, Z"*). This does not violate any of the assumptions made in section 3.4.2, since the input jokes were defined as coming from any source.

4.3 Data Collection: JokeJudger

We have implemented a data collection web application called JokeJudger[2] in order to comply to the specification of the `Human Evaluation` component of the GOOFER framework (see section 3.4.2). The web application allows users to create *"I like my X like I like my Y, Z"* jokes and rate jokes created by other users. We have invested significant effort into this platform. We employ several tactics in order to make the site as user-friendly as possible as well as making the users revisit the site, as we discuss in sections 4.3.2 and 4.3.4.

Another possibility to handle the data collection and rating that is often used in research [41][36] is scraping popular sites equipped with like-based systems containing jokes, such as Twitter[3] and Reddit[4]. The problem with these like-based systems are that they do not necessarily always reflect how well the joke is received. This is due to sites not exposing all jokes equally, as well as having a significant bias towards a specific audience. Visitors of a specific website as well as followers of a comedian on Twitter both have a correlated taste of humour. The amount of exposure for a tweet on Twitter is correlated to several other factors, such as the number of followers of the posting user as well as to the number of followers of any user retweeting or liking the tweet (both these actions make the tweet show up in other users' timelines) and even the hashtags used. Higher exposure means that the tweet has a larger amount of people that can like the tweet. The audience size of a tweet is however not freely available except for the user posting this tweet and is thus hard to factor out. Reddit has similar problems, since top-scoring comments end up higher on the page, and thus receive more exposure as well. This larger audience gives the opportunity to the comment to get even more likes, as other equally good jokes might remain at the bottom.

4.3.1 Platform

We acknowledged the need for a platform that presents jokes independently of the quality perceived by other users as a requirement to create a generic computational humour framework as specified in the previous chapter. We built JokeJudger in generalisable way (as specified in Appendix A.3) such that it can fully comply to the

[2] `http://jokejudger.com` and `https://jokejudger.herokuapp.com`
[3] `https://twitter.com`
[4] `https://reddit.com`

needs of the general computational humour framework if required. The most similar platform to our platform is The New Yorker's platform for voting captions for their image of the week[5]. The captions collected by this platform have also been studied for computational humour purposes [53][52]. There are however large differences between their platform and our provided platform. One such difference is that our platform is built to work for any textual joke and does not force a specific challenge if the steps of section A.3 are followed. Another difference is that users are able to get help creating their joke if necessary, and that they are rewarded with several insightful analyses about their created jokes, such as shown on figures 4.6 and 4.5. JokeJudger also uses a Likert scale of size five, instead of three. It is also the first time to our knowledge of that an open platform was created to acquire jokes of a specific format and of which the source code is released for reuse.

We implemented the full stack of JokeJudger ourselves, allowing us to tweak every aspect of the site to our liking. We used several frameworks in order to make this platform as modular as possible, both on the front-end and the back-end, such that components of this platform could be reused by others if they prefer to use a slightly different stack. For the back-end, we employed a MySQL[6] database and a NodeJS[7] server using the Express[8] framework. The front-end was designed using Bootstrap[9], extended with our custom CSS, and AngularJS[10] to communicate with our back-end and to dynamically load in all content separately on every page, such as the jokes and profile. JokeJudger is also built in a mobile-friendly way, as it employs a different interface when opened on mobile. One of the optimisations is that texts that are shown by hovering over them on desktop, are written as text on the page instead, such as the the textual meaning of every number of stars. JokeJudger is also optimised to be run as a proper web app on iOS, Android and Windows Phone.

4.3.2 Interface Decisions

When designing the platform, we have aimed to create a minimalistic feeling to the input pages (depicted in figures 4.2 and 4.3) in order to eliminate the need for explanations. We also made use of several icon fonts to make the site more visually clear.

Rate page The rate page (see figure 4.2) presents the jokes one at a time, with the variable template values in semi-bold to increase the speed of reading the important part of the jokes. The punchline is always on a new line in order to make the distinction between Y and Z very clear. The jokes are loaded in a least-ratings-first order, as to balance the number of ratings per joke over the whole site in general. The rate page remembers which joke has already been rated by a user, such that

[5] http://www.newyorker.com/cartoons/vote
[6] https://www.mysql.com/
[7] https://nodejs.org/en/
[8] https://expressjs.com/
[9] https://getbootstrap.com/
[10] https://angularjs.org/

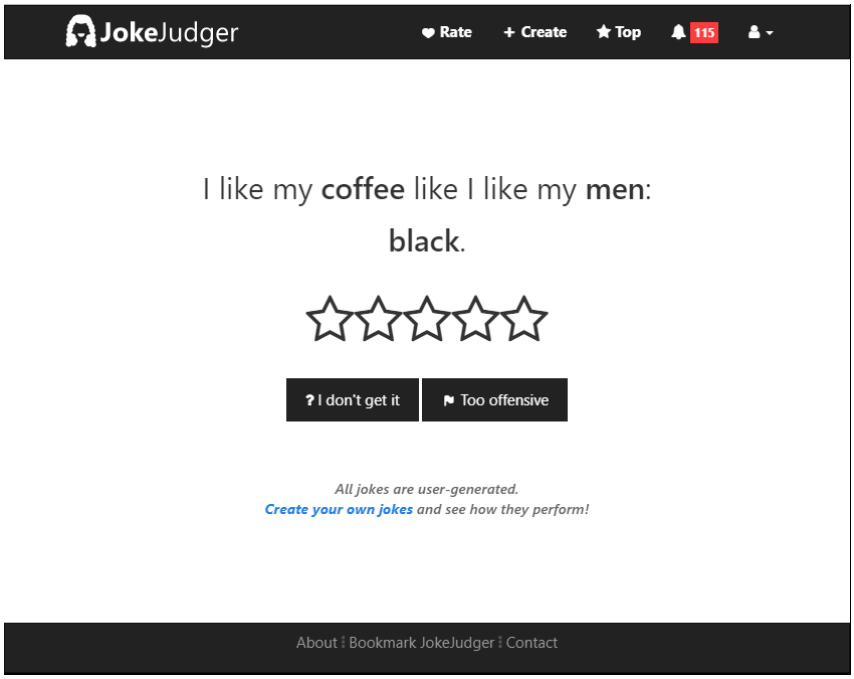


FIGURE 4.2: The "rate" page, where users rate other users' jokes.

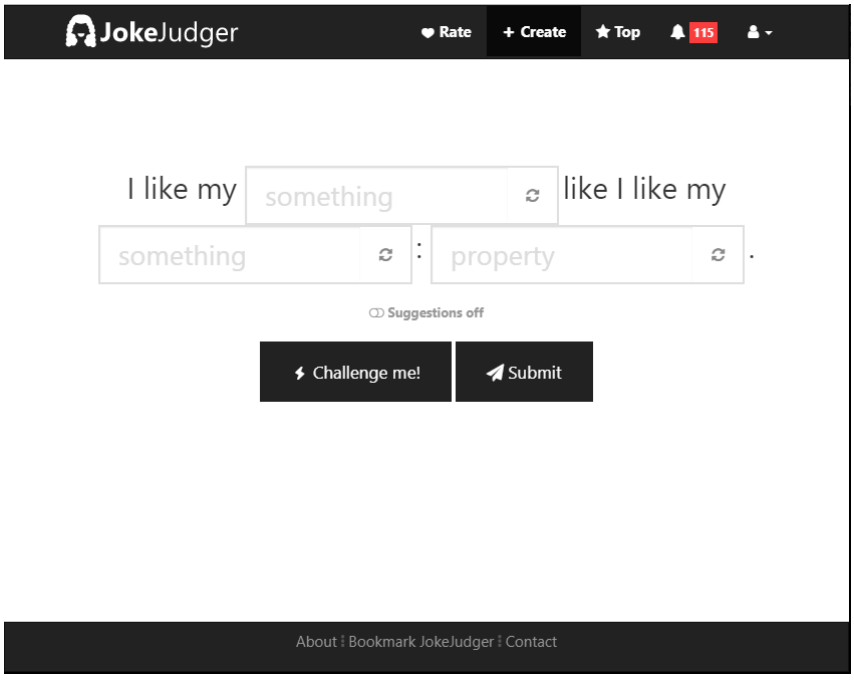


FIGURE 4.3: The "create" page allows users to create jokes with helpers like challenges, suggestions and a randomiser.

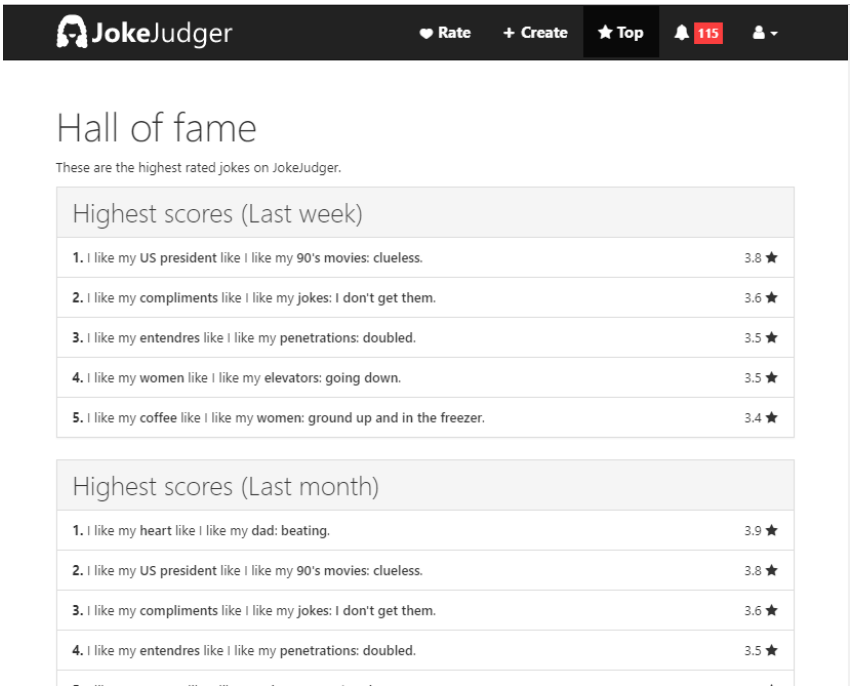


Figure 4.4: The "hall of fame" lists the best jokes of the week, month and of all time.

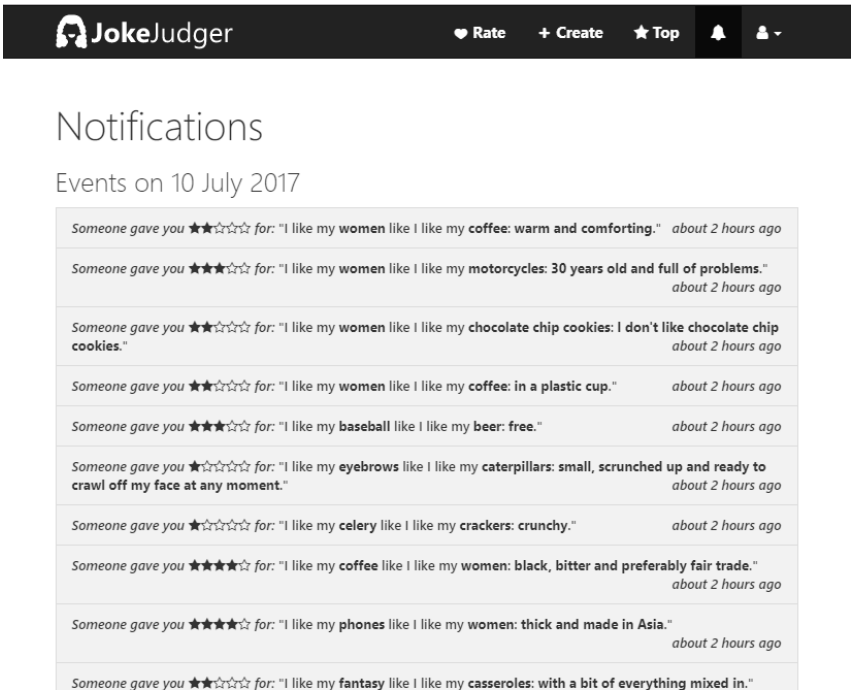


Figure 4.5: The "notifications" page shows the user the most recently received ratings on their jokes.

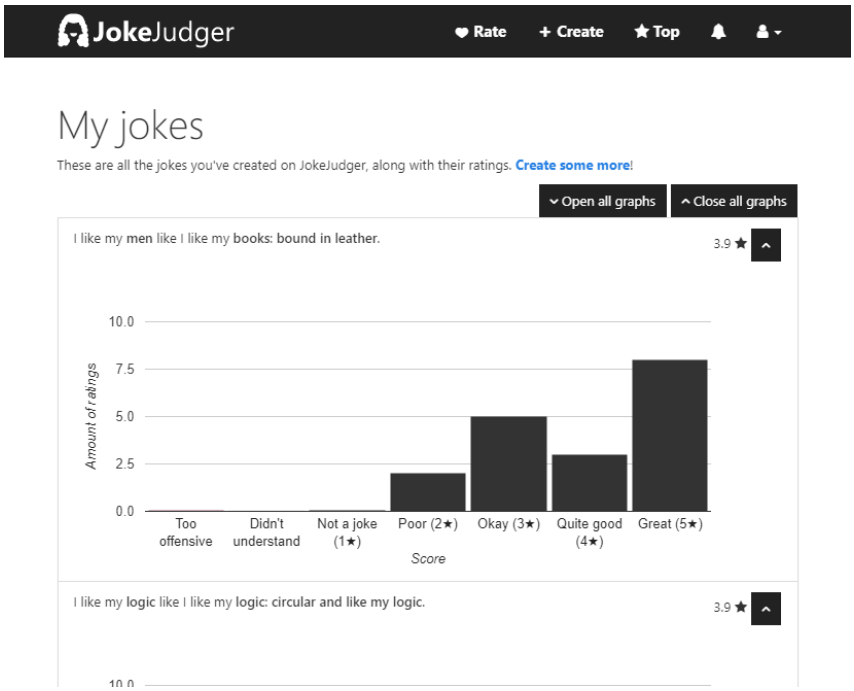


FIGURE 4.6: The "created jokes" page shows a histogram of the received ratings for every created joke.

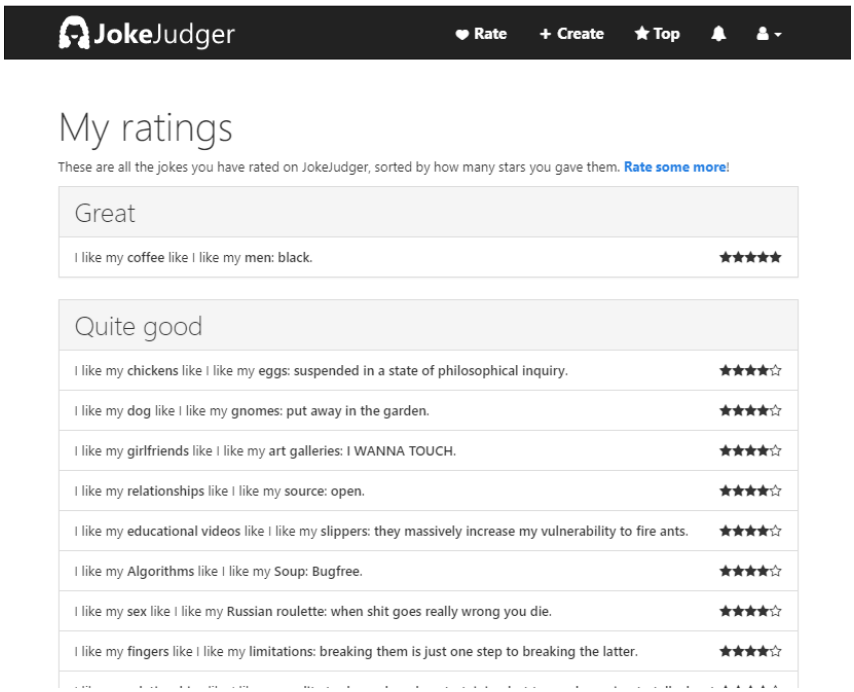


FIGURE 4.7: The "ratings overview" page allows users to easily rediscover their favourite jokes on the site.

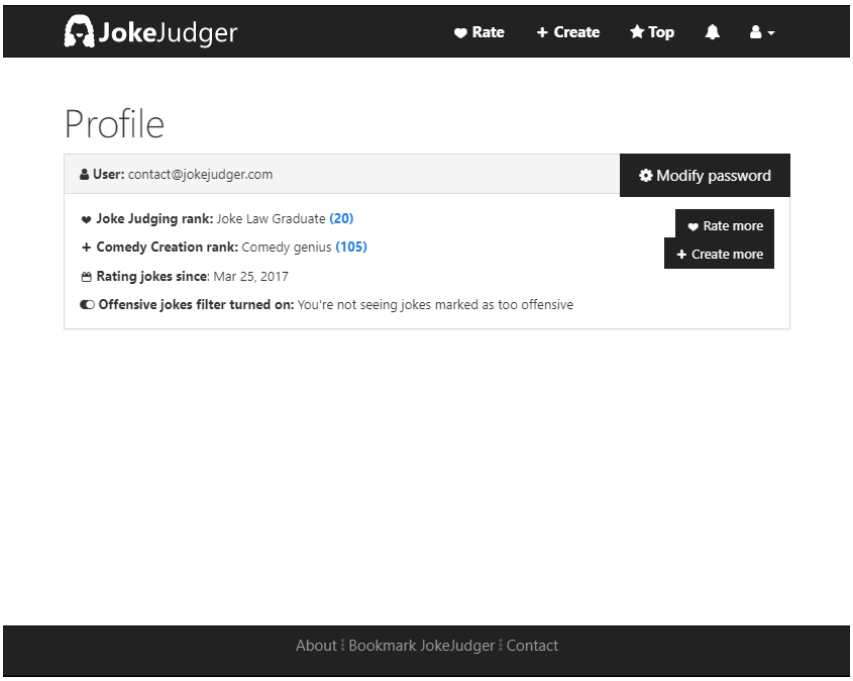


Figure 4.8: The "profile" page allows to change several settings and displays achievement ranks.

a joke is never rated twice by the same user. It also never displays a joke that is created by the user himself. The page initially only contained the joke with five empty stars that are being filled along with the stars to the left whenever the user hovers over one. When hovering over a star, all stars left to it also get filled. On hovering over, each star shows the meaning of this rating. On mobile, a paragraph containing the explanation for each rating is shown permanently underneath the rating buttons. We use a Likert scale of size five and use meanings similar to other computational humour research using Likert scales [64], being:

1. Not a joke
2. Poor
3. Okay
4. Quite good
5. Great

After doing some user tests, we noticed users did not know what to do when they felt that they did not get a reference, but did not want to give a joke only one star for it either. They had a similar response when they felt that the joke was too sexual or too offensive for their normal liking. In order to improve the quality of the dataset, we added two types of markings underneath the star ratings, being "I don't get it" and "Too offensive". The rate page presents the jokes with the smallest

number of ratings first, in order to stabilize the number of ratings received on every joke.

Create page The create page (depicted in figure 4.3) allows users to input jokes into the system. There are three input fields, since *"I like my X like I like my Y, Z"* jokes have three places for template values. The page offers several ways to inspire a user and alleviate their possible blank page syndrome. The user can toggle on the suggestions option. Each input field has its own randomisation button that fills in the field with a random element from the suggestion. There is also a challenge-button on the page, which generates a random challenge. We discuss the exact implementation of these suggestion options, randomisation system and generator in section 4.3.3.

Data analysis pages We also aim to reward JokeJudger's raters and creators for their work by offering them detailed data insights on their created jokes (see figures 4.4, 4.5, 4.6, 4.7 and 4.8). For every joke they submit, they can see a histogram displaying the ratings received on these jokes (see figure 4.3). They can also see which ratings they recently received on the notifications page (see figure 4.5) with a notion of how long ago this rating was given. If there are any notifications, the number of notifications are displayed as a number in a red box in the navigation bar, like many other social networks (e.g. Facebook, Twitter, Pinterest etc) tend to do.

4.3.3 Challenge Generator

In order to alleviate users' blank page syndrome, we implemented several tools to inspire users to create jokes.

Suggestion system The user can toggle the suggestion button underneath the input fields on the Create page (see figure 4.3). This suggestion option queries the Datamuse API[11] for the suggestions. If Z is filled in with an adjective, it is able to suggest nouns for X and Y that can be used with this adjective. This also works the other way around: if X or Y is filled in with a noun, the suggestions for Z are the adjectives that can be used for either of them. If only X or Y is filled in, for the other one (X or Y), it generates a list of nouns that can be used for an adjective that can be used for the noun that is already filled in.

Field randomiser Each input field (X, Y and Z) has its own randomiser button. Pressing this button fills the field with a random value picked from the suggestions list, as specified in the paragraph above.

Challenge creator The *"Challenge me"* button allows the user to generate a random challenge. There are two modes between which the generator picks: a list mode and Datamuse mode. The first mode is based on a principle used in improvisational comedy theory that suggestions should be clear and capable of being

[11] https://www.datamuse.com/api/

put in a multitude of contexts. We made lists of random objects, jobs, people and objects that are close to people, events, genres and locations. The generator has a high chance of picking a noun from the list of *close subjects*, in order to suggest making an analogy about a meaningful subject, for X. It picks a random element from the remaining categories for Y. This leaves room for a strong punchline. These lists can be found in the source code of JokeJudger[12]. The lists were initially made for a Twitterbot called ImprovChallenge[13], which generates suggestions similar to those of the improvisational comedy show *"Whose line is it anyway"*. People could then reply with their punchline to this bot, and like other people's replies. We created this bot when we were still experimenting with how to collect data for computational humour research.

The other mode for the challenge generator is the Datamuse mode. This starts by generating a noun that can be preceded by "my", in order to again create a meaningful analogy. It finds adjectives that could be used for this word, and then nouns that could be used with an adjective from this list. This step is repeated several times as a way to be able to generate a somewhat random suggestion by stepping away from the initial word in random directions. The first word, the last found adjective and the last found noun are filled in into respectively X, Z, Y after which one or two of the words are deleted. This leaves the user with respectively two or one words filled in already.

The existence of these generators added some of our assumptions about analogy jokes to the dataset. This would be worrisome if our research would use these jokes without their ratings to distinguish them from completely non-humorous texts, like the research done on one-liner recognition [36]. For our system however, this does not matter, as even if only a subset of the possible *"I like my X like I like my Y, Z"* jokes were given, the task of differentiating good and bad jokes is still dependent on the ratings they receive. If all jokes were generated by these generators, our system might only be able to classify jokes as good if they were following a similar pattern, thus limiting the possibility space. However, we added a lot of *"I like my X like I like my Y, Z"* jokes found online. We also noticed that a larger number of people were not using the challenge systems when submitting jokes. This means that the possibility space of the generator is wider than the helping generators of the create page, since more diverse data is also present in the dataset. In fact, having jokes generated by these generators is useful to our system, as we designed these to suggest single words, which is what Gag and its metrics are created for.

4.3.4 Deployed Aesthetics of Play

We used several gamification techniques in the JokeJudger platform to make it more fun to use, with the goal of increasing the amount of data collected. Gamification is the application of game elements in order to encourage engagement of users with a service. To implement this, we studied and used the MDA framework (Mechanics, Dynamics, Aesthetics). This research uses the term "aesthetic" to denote the desirable

[12] https://github.com/TWinters/JokeJudger
[13] https://twitter.com/ImprovChallenge

emotional response evoked in a player [21]. An aesthetic can thus be used to identify what type of fun a particular person likes. They identify eight different aesthetics, some of which we discuss in this section. They argue that successful games have many of these aesthetics in order to increase the reach of their audience. To gamify JokeJudger, we aim it to have several of these aesthetics.

The first of their aesthetics we aim for is the *discovery* aesthetic, which sees a game as uncharted territory. There is no complete list of all jokes on JokeJudger available to the user , other than the ones they rated. By only showing one joke at a time when rating, there was always the random chance of discovering a great joke as the next one.

The second aesthetic we aim for is the *challenge* aesthetic, which sees a game as an obstacle course. We achieve this by giving the player insight into the scores their jokes get (with the notification page, figure 4.5, and the joke ratings page, figure 4.6), and by showing the top jokes of a certain period. We also have the *"Challenge me"* button that fills in some of the words, as a challenge for the user to fill in. JokeJudger also has an achievement system with titles that increase dependent on the number of jokes the user has rated and the number they have created. We hope that these features encourage challenge-seeking users to submit more jokes.

The third aesthetic we aim for is their *expression* aesthetic, which sees a game as self-discovery. We achieve this by having a create page where they can totally express themselves using jokes, and which immediately shows their joke to all other users of JokeJudger. We also aim to help people requiring assistance with this type of self-expression using the joke suggestions on the creation page, and thus helping them to express themselves.

The last aesthetic JokeJudger aims to get is the *submission* aesthetic, which sees a game as a pastime. We do this by having a minimalistic interface that immediately opens a joke on the rating page. By adding the notification system, we also hope that people open JokeJudger more often in order to check their notifications, just like most social networks do to make their users return.

4.3.5 Reinforcement Learning

Learning to generate jokes from rated examples has the advantage of allowing rated, generated jokes to return as training data for the generator. This principle thus allows the system to learn from its own jokes. As discussed in the introduction of this thesis, this is an interesting attribute, as it implies that this algorithm is capable of accounting for the temporal dependency of humour, as well as being able to adopt to the preferences of a single user.

JokeJudger can be used as an interface in order to achieve reinforcement learning with the Gag system. The Gag system starts with its initial dataset, collected by the online version of JokeJudger. A local version of JokeJudger with an empty database has to run on the same computer the Gag system is running on. The Gag system automatically creates an account on JokeJudger and submits jokes. The user surfs to his local JokeJudger websites and rates these jokes. The Gag system detects when all jokes have been rated by the user, and uses both the original JokeJudger

dataset with the local, generated JokeJudger dataset to re-train its classifier and repeats the whole process.

4.3.6 Collected Data

During the training data collection phase, the platform was marketed on Facebook, such as in groups for improvisational theatre actors and comedians. It was also marketed on sub-reddits about improvisational theatre, comedy and datasets. During this phase, the platform collected 336 jokes and 4828 ratings by 106 users. This dataset can be found in section A.2. The percentage of ratings per number of stars can be found on figure 4.9. We can see that the most given rating type is two stars, being the rating that stands for "Poor". We can calculate the agreement for a joke as the percentage of people who submitted a rating equal to the mode of all ratings, or the average of these percentages if there are multiple modes. The mode of a dataset is the most occurring class. The average agreement for all the human-created jokes is 41.36%. Alternatively, if we define agreement as picking one of the modes (instead of averaging the agreement percentages of the modes), the average agreement about the ratings in the dataset is 48.61%. The dataset thus exposes the large disagreement between raters about the quality of a joke. This is one of the typical characteristics of humour, which makes it even harder for programs to identify humour.

At the end of the evalution phase, we collected 524 jokes (100 of which are generated by Gag), 9034 ratings, 418 markings (*"Too offensive"* or *"I don't understand"*) coming from 203 users. The agreement on the jokes was of a largely similar magnitude, being 41.87%. If the definition of agreement is "any mode", the result also stays the very similar with 48.52%.

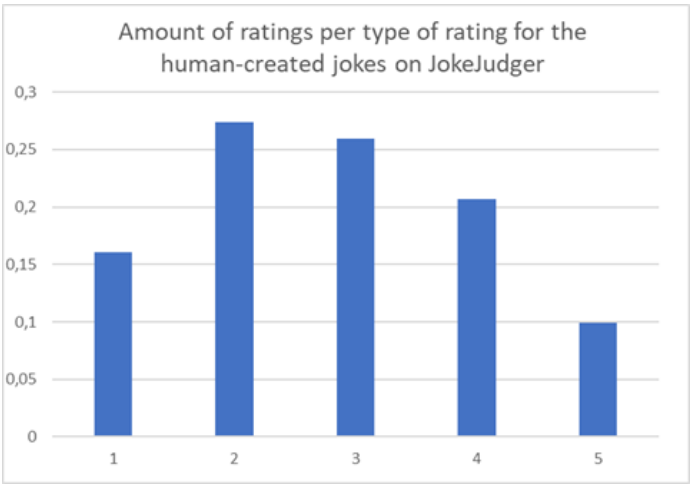


Figure 4.9: The percentages of the ratings for human-created jokes on JokeJudger at the end of the training data collection phase.

4.3.7 Reflection

The platform received a relatively large number of jokes and ratings for a project with a non-existent marketing budget. It seemed that a lot of people felt incentivised to rate jokes, probably because reading jokes is quite an enjoyable activity.

We received some comments from people that did not want to create an account in order to rate jokes. We decided on having this obligatory log-in system for several reasons. For the research, it was important to know which ratings were given by the same person. This was important because it allowed us to notice trends in their ratings if this deemed necessary, as humour is a subject everyone has completely different views on and "there's no right answer". It also made sure that we could filter out ratings if this deemed necessary, as someone could possible have flooded the website with nonsense ratings or jokes. We did not want to have to erase most ratings from unregistered users after such an attack. For the platform itself to function, it was also important because the site sorts the jokes to be rated on their number of ratings received, so that new jokes get way more ratings than older jokes. If the users did not have to be registered, they would tend to keep on seeing these newer, already jokes on fast consequent visits. This would skew the dataset as the same person can express his opinion on a joke multiple times and count as different people. Another important reason for this decision is that the site is built to give (anonymous) notifications when people rate their jokes, give insights with histograms on the total ratings received for every joke, so that users can learn how well their jokes scored. This would be impossible to do if people were not required to log in, as they could lose access to the statistics about their created jokes otherwise.

4.4 RegEx Based Template Stripper

We used a regular expression (RegEx) to be able to extract the template values from jokes in using the *"I like my X like I like my Y, Z"* template. The RegEx can be found in figure 4.10. The RegEx also allows for extracting slight variations on the *"I like my X like I like my Y, Z"* template in order to make the system more robust. We obviously did not have to use this complicated RegEx when dealing with JokeJudger data, as they were neatly formatted, but we had to use it when initially dealing with the jokes scraped from the web.

This component also functions as the template store to provide the *"I like my X like I like my Y, Z"* template so it can be applied in the last step of the pipeline, in order to satisfy the generalisation of the system specification specified in section 3.4.

```
I (like|love) (my )?([\w\s-]+) (the way |how |like )(I|he)
(like |love )?(my )?([\w\s\-]+)[,.;:\-]+ ?(.+).?!?
```

FIGURE 4.10: The regular expression to extract template values from the *"I like my X like I like my Y, Z"* template.

4.4.1 Multi-word metric aggregation

In the Goofer framework, we limit each template value to only be a single word. This is an issue for the collected training data, as multiple words are often used in user created jokes. In order to use single-word metrics to evaluate template values that exists of multiple words, we propose several solutions. In order to convert this problem to a smaller problem, we could use the knowledge of the template extractor to find what part of speech is used for a certain template slot. The given multi-word template value is then searched for words of this part of speech. If there is only one, the problem is solved. If there are multiple, a first solution is that the metric is used on every possibility, after which the maximum, minimum, average or other aggregation function is used on these feature values. Another solution when dealing with multiple values, is to create all permutations of all candidate words for the template values and replace the original template values with this set. Jokes such as *"I like my coffee like I like my war, gruesome and cold"* are transformed to the two jokes *"I like my coffee like I like my war, gruesome"* and *"I like my coffee like I like my war, cold"*. This is the solution we picked for our implementation, as it is the most general solution, since it does not require any knowledge about the metrics. This operation transforms our collected data from JokeJudger to 528 single-word template values.

4.5 Metrics used

We implemented several metrics of the Goofer framework proposed in section 3.3, and made sure to cover all humour theory properties by implementing at least one metric per property. As the Goofer framework specifies, these metrics are applied to every template value that fits the (POS) criteria. These POS tags are found during template extraction in the Goofer framework. Since this system only has one template, we know that the POS tags are *noun*, *noun* and *adjective* for respectively X, Y and Z. The metrics we used are quite similar to the metrics of Petrovic's analogy system [41], as their metrics have been shown to be successful. Having similar metrics also allows us to validate their importance using the classification algorithm. However, we added some variations, since we did not need to use the values in a minimisation problem, and extended it to include several more metrics to satisfy all categories of the theory we use. The metrics used in Gag are:

- **Word Frequency:** We used the 1-gram model of the English One Million dataset from Google Ngrams[14] to calculate the frequency of any word. These datasets are provided as zipped comma separated value files, making them hard to search through. In order to query these datasets more easily, we created a Java program to extract these files to a MySQL database and query them later, for this research. We made this program public for anyone to use[15].

[14] https://storage.googleapis.com/books/ngrams/books/datasetsv2.html
[15] https://github.com/TWinters/google-ngrams-to-mysql

- **Frequency of word combinations (2-grams)** The frequency of word combinations can be found using 2-grams. We use Google Ngrams and our created program to load these n-grams and calculate the frequency of adjective-noun combinations on all compatible template values. Since we only require adjective-noun combinations, we combine POS evidence of WordNet and Stanford Log-Linear POS tagger[16] to only store possible adjective-noun pairs.
- **Number of meanings metrics** The number of meanings metric is calculated using the number of definitions provided by WordNet for a particular word.
- **Adjective Vector Similarity** The similarity in the list of adjectives used with a noun can be found by using 2-grams of Google Ngrams, which can be loaded into database using our program. In order to calculate the adjective vector, the metric finds all adjectives and their frequencies that they are used before these nouns in sentences by dividing their number of appearances by the sum of these counts over all the adjectives used with the noun. The metric then sums all products of these frequencies of the adjectives used in both vectors.
- **Word sexiness** The word sexiness is calculated by comparing the frequency of a word in a corpus of a sexual domain to a normal domain. We already discussed how to calculate the word frequency in a normal corpus above. In order to calculate this value, we use a similar, but simplified, approach as used in the DEViaNT system [26]. This research used the texts from *textfiles.com* under the erotica category[17] as their sexual corpus. In order to scrape all these files, we created a Java tool that downloads all documents of this site from any given category[18]. We added both the frequency in sexual corpora as this frequency divided by the Google N-gram word frequency. This metric is used for the Inappropriateness property, which is a property that is not explicitly accounted for by any metric used in Petrovic's analogy system.

Applying these metrics to their possible template slots results in a total of 15 feature values for each joke. Each feature value name has a suffix identifying the index of the template value(s) they are used on. The features are thus:

- `frequency_0`
- `frequency_1`
- `frequency_2`
- `sexual_freq_0`
- `sexual_freq_1`
- `sexual_freq_2`
- `relative_sexual_0`
- `relative_sexual_1`
- `relative_sexual_2`
- `adjective_vector_similarity_0_1`
- `relative_frequency_2_0`
- `relative_frequency_2_1`
- `word_senses_0`
- `word_senses_1`
- `word_senses_2`

[16] https://nlp.stanford.edu/software/tagger.shtml
[17] http://textfiles.com/sex/EROTICA
[18] https://github.com/TWinters/TextFilesComScraper

4.6 Template Values Generator

The generation of candidate template values is done using 2-grams. The 2-grams used in our system are Google Ngrams[19]. It finds nouns for X and Y and an adjective that could be used to the left of both words. If seed words are given, these can be assigned to X if one is given, or to X and Y if two seed words are given. A similar generator is also deployed in Petrovic's system [41], making it easier to compare the differences in the evaluation of our approaches.

In order to make the specification of the Goofer framework (see section 3.4) fit, we assume the framework has detected that X, Y and Z are respectively mostly nouns, nouns and adjectives in the template extraction phase, as discussed before. Another way the system could have achieved this using a statistical analysis based on part-of-speech (POS) tagging, discussed in section 2.3.4. The generation of sentence candidates instead of these word candidates for template values is discussed in section 5.1.2.

4.7 Classifier

As discussed in section 3.2.3, our schema generalisation allows us to use either classification or regression algorithms as the aggregation parameter of a classification schema. The difference between these two types algorithm is that regression involves predicting a continuous value and classification identifies group membership to a label. A regression algorithm would thus allow us to predict the average rating a joke might get. Using a classification algorithm on the data set, we could predict the most popular Likert-scale rating for a joke, since this defines five possible classes. Using a classifier thus gives the system the same five star option a human has. We use WEKA in our implementation for both options.

4.7.1 Classifying the Mode Rating

When using classification algorithms, we use the mode rating received for each joke. Ties are broken by picking the lowest mode. The distribution of the modes of the jokes can be seen in figure 4.11. The most frequent mode is the two star rating, with 174 instances. The ZeroR algorithms is a classifier that is often used to establish a baseline to compare other classifiers to. This algorithm classifies every given instance to the most frequent class, in this case the two star rating. This means that the ZeroR classifier correctly classifies $\frac{174}{528} = 32.95\%$ of the training data.

We compared several classification algorithms on their number of correctly classified instances using 10-fold cross-validation. Random Forest was the best scoring classifier for this task, classifying $\frac{325}{528} = 61.55\%$ of the training instances correctly. The Random Forest algorithm creates several random decision trees using subsets of the features and subset of the training data. These decision trees then all vote for a specific class when classifying an instance [8]. Systems using the Goofer

[19] https://storage.googleapis.com/books/ngrams/books/datasetsv2.html

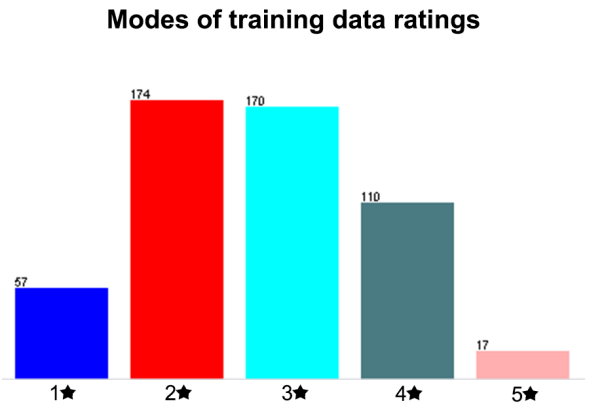


FIGURE 4.11: The number of jokes in the training dataset per mode.

framework are prone to using too many, possibly noisy, features since a multitude of humour metrics is applied to every compatible template value. It thus makes sense that this type of algorithm, which uses subsets of the features, works relatively well for the GAG system.

A weakness of using classification on the mode rating is that the classification algorithm does not explicitly know about the ordering of the labels. For example, it does not know that score 4 and 5 are closer to each other than score 2 and 5.

Classified as →	1 star	2 stars	3 stars	4 stars	5 stars
1 star	30	16	10	1	0
2 stars	1	114	46	13	0
3 stars	1	40	119	10	0
4 stars	1	26	24	57	2
5 stars	2	3	6	1	5

FIGURE 4.12: The confusion matrix of the Random Tree classifier on mode ratings of the training data.

Comparing the performance of the Random Tree algorithm (61.55%) to the agreement between users (41.36%, explained in section 4.3.6) shows that the algorithm picks the most frequent rating more often than the human evaluators do in the training data.

The recall of a classification algorithm determines the chance that an instance receives a correct classification. We can see that this value is rather low for the five star rating class, as it is very often misclassified ($\frac{2+3+6+1}{2+3+6+1+5} = 70.5\%$ of the time). This might be due to a lack of jokes with a five star rating or five star jokes being intrinsically very unpredictable. These misclassifications cause the GAG system to miss a lot of good jokes. This might still be okay, since the system generates thousands of candidate jokes. More worrisome is the fact that two star ratings are often misclassified as four stars.

The precision of a classification algorithm determines the chance that a classification to a certain class is correct. It is a useful value for evaluating a classification algorithm in the Goofer framework, since it will have to pick high quality jokes from a randomly generated set of jokes. This value is especially important for high rating labels such as 4 and 5 stars, since it is more important that jokes rated as such are great jokes than that we might miss good jokes because they got a lower label. The confusion matrix (fig 4.12) shows us that no joke classified as having a mode of five stars by the algorithm has a score lower than four. However, the subset of jokes with a mode rating of five is rather small and thus does not allow for large variety. We thus use both the four and five star rating to filter the set of generated jokes. According to the confusion matrix, filtering out four and five star classifications gives us a precision of $\frac{57+2+1+5}{1+13+10+57+2+1+5} = \frac{65}{89} = 73\%$ jokes that actually have four or five star mode rating in the training data.

4.7.2 Regression on the Average Rating

Regression algorithms can be used on the average rating of each joke in the dataset in order to predict the quality of the joke. The average joke ratings can be seen in figure 4.13. Regression algorithms can be evaluated by comparing their predictions to the predictions of ZeroR for regression. This ZeroR algorithm just predicts the average rating over the whole dataset as the value for every joke. The root mean square error (RMSE) is a measure for the average difference between the predictions and the actual value. The root relative squared error (RRSE) is computed using the RMSE of the algorithm divided by the RMSE of ZeroR. A RRSE of over 100% thus implies that it is worse than just predicting the average rating of the whole dataset. We used the RRSE value to compare several regression algorithms. The regression version of the Random Forest algorithm scored the highest, with a RRSE of 84.04% and a RMSE of 43.25%. The reason we assume the Random Forest algorithm performs best is the same as the reason it performs well for classification, being that it creates trees that use subsets of the features and data.

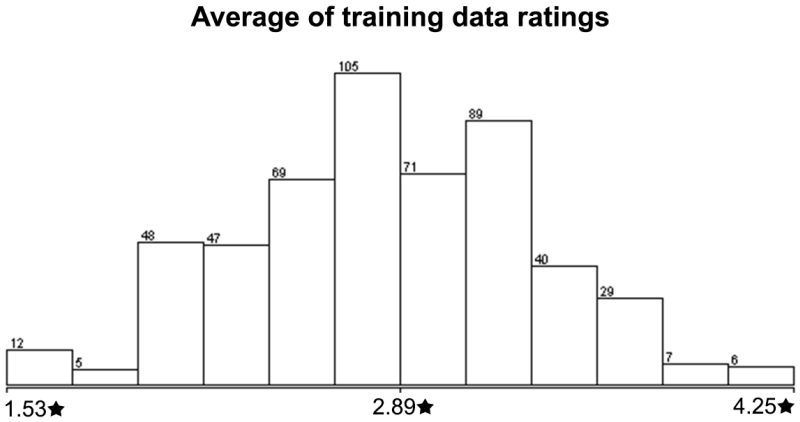


Figure 4.13: The average rating for the jokes in the training dataset.

Another important advantage the Random Forest algorithm brings, is that it is capable of finding the importance of each attribute by checking its average impurity decrease in the decision trees. This means that the algorithm is capable of stating the importance of each metric for each position. A humour theorist can thus use the GOOFER framework with a Random Forest algorithm to validate and discover assumptions about a certain collection of rated jokes. The importance of each attribute for the training data in the GAG system is given in figure 4.14. One thing to notice is that the attribute importance seems to be conforming to the theory of Petrovic's model [41]. His five assumptions and metrics roughly map to our `relative_frequency_1_0` , `relative_frequency_2_0` , `adjective_vector_similarity_2_1` , `word_senses_2` and `frequency_2` metrics, most of which have a relatively high importance.

Importance	**Applied metric**
0.67	`relative_sexual_freq_2`
0.67	`relative_frequency_2_1`
0.62	`adjective_vector_similarity_0_1`
0.59	`relative_sexual_freq_0`
0.56	`sexual freq_0`
0.53	`word_senses_2`
0.52	`relative_frequency_2_0`
0.52	`sexual freq_2`
0.50	`word_senses_0`
0.50	`relative_sexual_freq_1`
0.50	`frequency_0`
0.45	`frequency_2`
0.43	`sexual freq_1`
0.36	`frequency_1`
0.26	`word_senses_1`

FIGURE 4.14: The attribute importance according to the regression version of the Random Forest algorithm on the average score of the training data .

4.8 Evaluation

We generated 100 jokes using the GAG system. Half of them were generated using the classification algorithm, the other half using the regression algorithm. We generated these jokes based on values for X present in the training data, similar to how Petrovic created his evaluation data [41]. Since classifying all possible jokes with a fixed X generated by 2-grams, which is roughly $\sim$10.000.000 jokes, requires a lot of processing power, we decided to add a random sampler in the generation process. The generator randomly samples adjectives used with X in a uniformly distributed way as candidates for Z and then randomly samples candidates for Y, also uniformly

distributed. Since we do not want the output jokes to be similar, we added the option to Gag to eliminate jokes that are too similar to higher scoring jokes from its output. We do this by only allowing one word to equal between jokes. For example, if we're generating jokes where X is fixed to be "coffee", and the generator generates *"I like my coffee like I like my arguments, hot"*, then any joke using either *"arguments"* for Y or *"hot"* for Z will be filtered out. This is similar to how Standup filters its jokes, using the "keywords" attribute in its schemas to calculate its similarity.

The amount of jokes having a mode of 5 is significantly smaller than the others, which limits the variety of the output of jokes with five stars when using the classification algorithm. To solve this, we added the option to Gag of only outputting generated jokes with a score above a certain value. We use this option to generate jokes with a score of either 4 of 5, and then randomly sample a quantity of jokes proportional to the frequency of X in the training data.

We uploaded all these jokes on JokeJudger, in between the human created jokes that were already there. Since JokeJudger sorts jokes on their amount of ratings, we also added 67 human created jokes to make sure that the first raters did not only see the jokes generated by Gag. We contacted several top JokeJudger users and asked them to submit several new jokes before the start of the evaluation phase. They added 30 of the new 67 human generated jokes. We also added 37 jokes that were recently posted to Twitter, Reddit and to our ImprovChallenge bot[20]. This ensured that the jokes presented during the evaluation phase to the first raters were also mixed with human created jokes. This decreased the chance that the evaluating users knew that they were rating generated jokes as well as giving us a more reliable control group of human-created jokes.

During the evaluation phase, 4424 ratings and markings were given to jokes on JokeJudger. The 50 jokes from the classification algorithm variant received 745 ratings, whereas the 50 regression algorithms jokes received 721 ratings. The 424 human-created jokes on JokeJudger received the remaining 2958 jokes. The platform was marketed in a public post on personal Facebook accounts, where it was shared several times. We also received (slightly smaller) traffic by promoting JokeJudger on Twitter, reaching out to the original JokeJudger users through a newsletter and from getting our old post on the /r/datasets sub-reddit[21] promoted by one of the moderators to the top of the sub-reddit.

4.8.1 Comparing Classification with Regression

We can see on figure 4.15 that the jokes generated by the classification algorithm are more positively received than jokes generated using the regression version of Gag. The regression jokes received 5% more 1 star ratings, as well as 1.5% more "I don't get it" markings. The classification jokes receive more ratings than the regression jokes for every rating category larger than or equal to two stars. Since three stars mean "Okay", 24.3% of the classifier jokes were seen as average jokes or better, compared to 21.36% of the regression jokes.

[20] https://twitter.com/ImprovChallenge
[21] https://www.reddit.com/r/datasets/

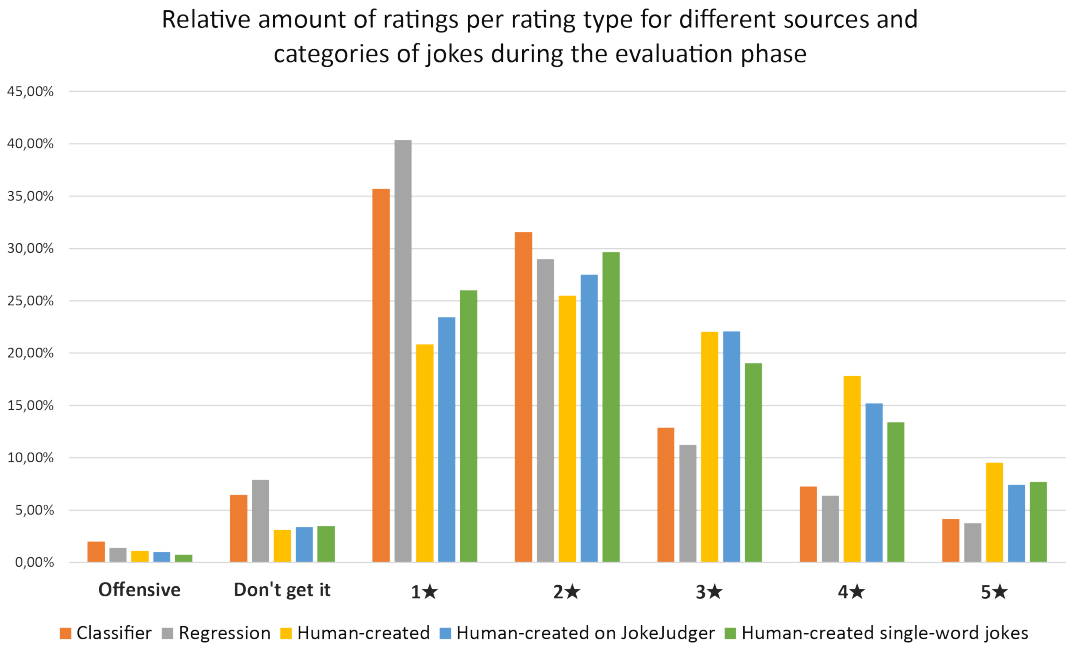


FIGURE 4.15: The percentages of the rating categories for the classification GAG system, the regression GAG system, all human-created jokes, all human-created jokes created on JokeJudger (= excluding jokes scraped from other sites such as Twitter and Reddit for the initial dataset), and all the human-created jokes that only use a single word in for every template value.

One possible explanation for the higher quality jokes created by the classification algorithm is that we took a random sample from the generated four and five star rated jokes. We chose to sample from the four star rated jokes, because the class of five star rated jokes in the training data was substantially smaller than the other classes. It also had a low precision when using the Random Forest algorithm, probably due to the unpredictability of truly good quality jokes. For regression however, these high-scoring, difficult-to-predict jokes might affect the given scores in a way that is more difficult to account for.

4.8.2 Comparing Generated Jokes with Human Created Jokes

We can see that human-generated jokes in general receive higher scores than the jokes generated by GAG. Human-created jokes are considered to be *"quite good"* and *"great"* jokes 27.38% of the time, whereas our classification algorithm only receives these types of ratings 11.41% of time. One possible reason for the lower quality of generated jokes is that they all have a similar shape, since they only use single-word template values. These types of jokes might be annoying to raters, making them press lower scores as soon as they saw a shorter joke. It might also be that jokes using only single words as template values in *"I like my X like I like my Y, Z"* jokes are generally perceived as worse jokes. We can see evidence for this claim by looking

Source	*4+ ratings*
Gag (Classifier)	11.41%
Gag (Regression)	10.12%
Human (All)	27.38%
Human (JokeJudger)	22.61%
Human (Single words)	21.08%

Figure 4.16: The percentage of ratings higher than or equal to four stars out of five for each category of figure 4.15.

at the "Human-created single-word jokes" statistics. These jokes are only perceived as better than "quite good" 21.08% of the time. This means that the classification algorithm, with 11.41% jokes with four or more stars, performs more than half as well as human generated jokes using the same single-word constraint.

One thing to note is that computer generated jokes receive more "too offensive" markings. This might be because jokes during the training data phase were flagged manually to be hidden unless the user disabled the offensiveness filter on his profile. During the evaluation phase, this page was left untouched for evaluation integrity, meaning that the generated jokes, unlike the old human-created jokes, were unfiltered.

Another possibility is that by executing our multi-word aggregation (see section 4.4.1), we transformed the dataset too much and made the generator learn from low quality jokes and still perceive them as high quality. For example, these types of jokes sometimes have several adjectives describing both nouns as Z, where the last one is often the funniest, as can be seen in joke 17. Although both receive the same ratings in our transformed dataset, the perceived funniness of the "silent" punchline might not be similar to the the "pretty" punchline.

Joke 17:
I like my women like I like my art: pretty and silent.

4.8.3 Comparison with Existing Analogy Generator

In the evaluation of his analogy generator, Petrovic used a Likert scale of size 3: "funny", "somewhat funny" and "not funny" [41]. The human generated jokes (collected from Twitter) in his data got 33.1% funny ratings, his model 16.3% and his baseline, which generated random jokes with his 2-gram generator and maximises only adjective vector difference between X and Y, 3.7% funny ratings.

We can map our Likert scale of size 5 to his scale by saying that 1-2 maps to "not funny", 3 to "somewhat funny" and 4-5 to "funny". Using this new classification, human generated jokes are 27.38% of the time perceived as funny, while our classification version of the generator is perceived as funny 11.41% of the time. Both the percentage of the funniness of the human generated jokes as well as the percentage of our generator is lower. There are several possible reasons to explain this. A first reason might be that Petrovic's research only used five different raters. These raters might have been more inclined to rate jokes as funny than the large,

diverse population on JokeJudger. A second reason might be that the quality of jokes submitted to JokeJudger is lower due to the barrier being lower. The barrier might be lower because users are able to submit jokes in an anonymous way. It might also be that jokes found on platforms like Twitter, are generally more funny due to the higher threshold for posting, as well as better jokes receiving more exposure, which we discussed in 4.3.1. We can see proof for this claim in the fact that human-created jokes on JokeJudger were perceived as funny 27.38% of the time, whereas when we filter out our initial dataset collected on such platforms, the average funniness of human-created jokes on JokeJudger drops to 22.61%. This means that, using this constraint, the Gag system with its 11.41% funniness is again more than half the time as funny as human-created jokes.

4.8.4 Example jokes per type

In order to illustrate the sources of jokes used by the evaluation section, we present some of the higher rated jokes per category. For Gag using the classifier algorithm, well-scoring jokes are joke 18 and joke 19. An honourable mention goes to joke 20 (not used during evaluation).

Joke 18:
I like my sex like I like my activities: illicit.

Joke 19:
I like my sex like I like my emotions: basic.

Joke 20:
I like my thesis like I like my suffering: long.

Well scoring jokes of Gag using the regression algorithm are jokes 21 and 22.

Joke 21:
I like my sex like I like my friend: lesbian.

Joke 22:
I like my women like I like my laughter: silly.

Well scoring jokes scraped from Reddit and Twitter and submitted to JokeJudger are jokes 23 and 24.

Joke 23:
I like my logic like I like my logic: circular and like my logic.

Joke 24:
I like my women like I like my Mac: inferior, but pretty.

The best scoring jokes made by JokeJudger users are jokes 25 and 26.

Joke 25:
I like my chess climaxes like I like my Australian dinners: check, mate.

Joke 26:
I like my US president like I like my 90's movies: clueless.

Some of the highest rated jokes created by humans that only have a single word for each template value are jokes 27 and 28.

Joke 27:
I like my heart like I like my dad: beating.

Joke 28:
I like my women like I like my men: androgynous.

4.9 Conclusion

In this chapter we have created a system that implements several components of Goofer framework we proposed in chapter 3. This system, combined with existing template extraction research [20][1][27], shows the Goofer framework is implementable.

We have also created a platform for collecting jokes and ratings as the training data for the Gag system. The platform suffers less under the typical exposure bias towards good content that sites usually have. The platform is made available to help future computational humour researchers gather data. We have also contributed to other future humour research by making the data set used available for other researchers.

We have compared the advantages and disadvantages of the usage of classification compared to regression algorithms as the aggregation parameter of classification schemas in the Gag system.

The Gag system effectively generalises the system created by Petrovic, as it eliminates the need to explicitly model the minimisation function and it is capable of detecting even more possible schemas. This shows that our Goofer framework (specified in section 3.4) can effectively make a more generic version of existing research.

When comparing the performance of our classification and regression versions of Gag, we noted that the version using classification algorithms outperforms the regression version. The frequency of funny jokes generated by Gag is over 50% of the frequency of funny *"I like my X like I like my Y, Z"* jokes with single words template values created by human users. This result made it harder to compare to Petrovic's analogy generator, since its jokes are also perceived as funny roughly half as many times as the human-created jokes.

We see our implementation as a stepping stone towards the Goofer framework. This implementation shows that the pipeline and theory suggested in the previous chapter is capable of learning jokes from examples. The most prominent unimplemented component of the Goofer framework is the template extraction, which already has been proven to work [20].

Chapter 5

Conclusion

5.1 Future work

In this final section, we discuss several possible extensions of the GOOFER framework for further research.

5.1.1 Generalised Templates

In this thesis, we chose to work with templates that are a list of fixed strings. One issue with this is that they represent fixed sentences that do not allow small (grammatical) variations. One possible way of solving this is by defining templates using grammars with variables instead of strings and template slots alternating each other. The template extraction methods would then need to be updated, e.g. require new distance measures for grammar trees in order to find these grammar tree templates. This would increase the amount of training data per template if done correctly, as the training data that would be found for each of the similar templates can now be merged.

As we discussed, SSTH defines "narrative" to be one of the six parameters that define a joke [43]. The narrative of a joke could change, but still convey the same joke, being of a different quality. For example, jokes 29 and 30 are the same joke, but using a different narrative. Since there might be both grammatical variations and narrative variations, templates can be generalised even further, namely as a set of lower-level templates that can be filled with similar content generated by the same schema(s). In order for this to work, a new component needs to be introduced that is capable of detecting these similar templates and schemas, and merge them together.

JOKE 29:
How many psychiatrists does it take to change a light-bulb?
Only one, but the bulb has got to really want to change. [63]

JOKE 30:
It takes only one psychiatrist to change a light-bulb, but the bulb has got to really want to change.

5.1.2 Sentence Generation

The Goofer framework as discussed in this thesis assumes for simplicity that template values are single words. Both the discussed metrics and the generated template values have been focused for single word template values. In section 4.4.1, we already proposed several methods for dealing with sentences using single-words metrics. In order to successfully generate jokes using sentences as template values, the template value generator has to be updated to be capable of proposing such sentences. There are several ways we would like to suggest to generate sentences related to other template values.

Firstly, WordNet relations could be used to create small sentences. Meronyms of a word could be used to form the punchline *"having a <meronym>"*, while hyperonyms could be used to form the the small sentence *"being a <hyperonym>"*.

Previous research found that jokes often tend to be about humans and human actions [36]. One way of generating related human activities is through using WikiHow[1], a website that explains how to execute certain tasks by giving a step-by-step guide of the actions required. Since this is often quite nicely structured, the tasks and actions have been suggested as a useful source for human actions [60]. This could generate template values describing an action that another template value is linked to.

We used adjectives that describe nouns based on 2-grams as a relationship in this thesis. We can generalise this to sentences by using descriptions of (parts of) this noun as a sentence. One possible way to retrieve such descriptive sentences is by scraping product descriptions from web stores selling these products or from online reviews.

Another useful source of sentences and useful word combinations is using news headlines. One of the highest rated human-created jokes on JokeJudger is joke 31, which is about a tower that recently burned down. Using the news in generated texts helps with the temporal dependency of humour and proves to the users that the jokes are recently generated.

Joke 31:
I like my coffee order like I like my Grenfell Tower jokes: too soon.

5.2 Applications in a Real World Context

The bane of most computational humour generators is that the perceived quality of their generated jokes might reach a peek followed by a sharp decline as the jokes become predictable. Since the perception of a certain type of humour might hype (e.g. internet memes), some generators that are built for these types of humour can rapidly get out of style. The generic computational humour framework proposed by this research is capable of learning new types of jokes from examples without requiring additional human effort. This means that if the system were automatically fed with new jokes from sites where people post humorous artefacts, it could constantly learn

[1] http://www.wikihow.com

these new types of humour. It would thus be possible to create an interface or bot that learned new memes or hypes through analysing Twitter feeds, sub-reddits or other sources of humour. The bot could thus learn how to generate jokes that are currently trending on its own. The possibility space of this generator thus has the potential of growing larger over time. This solves the dullness other generators often experience.

This system could also help chatbots and virtual assistants personalise their humour for their users on an individual basis. Chat bots often already use templates when responding [42]. This implies that our framework could help them discover hidden schemas that their user prefers. One of the issues is getting feedback on how well the user perceived the jokes. This could possibly be done using sentiment analysis on the reply.

5.3 Conclusion

In this research, we created a computer program that is capable of learning to generate humour based on human-rated examples. We achieved this by extending and generalising computational humour concepts such as schemas. We also used humour theory and other computational humour research in order to argue, identify and evaluate a knowledge base of humorous metrics. These findings are used in the Goofer framework, which is capable of learning humour from rated examples. This framework shows how machine learning algorithms, more specifically classification and regression algorithms, can alleviate humans from the elaborate task of crafting schemas for humour generation by hand. These schemas find correlations between metrics applied onto template values used in templates that make a joke humorous. We also showed how the framework can be used to help explain in a human-comprehensible way what the most important metrics of the classification schema for a particular set of jokes are.

We created and published a reusable platform called JokeJudger to collect and rate jokes. We showed why such a platform was necessary for our proposed framework to function. With this platform, we created a dataset of human-rated *"I like my X like I like my Y, Z"* jokes. Such a dataset is useful for computational humour researchers since it provides ratings that have significantly less exposure bias on their scores than jokes from other sites.

During this research, we implemented a large set of tools, which we made available for other computational humour and computational linguistic researchers to use, namely JokeJudger[2], the data created with JokeJudger[3], Goofer and Gag[4], textfiles.com scraper[5], Google N-gram MySQL loader[6], an extensive Markov chain

[2] https://github.com/TWinters/JokeJudger
[3] https://github.com/TWinters/JokeJudger-Data
[4] https://github.com/TWinters/Goofer
[5] https://github.com/TWinters/TextFilesComScraper
[6] https://github.com/TWinters/google-ngrams-to-mysql

library[7] and a Datamuse Java API[8].

Finally, we created an implementation for a system using the Goofer framework for generating *"I like my X like I like my Y, Z"* jokes from examples. This implementation shows that components of the Goofer system are functional but also what their limitations are. We evaluated the system by comparing jokes generated using classification and using regression with each other. We concluded that our version of Gag using a classification algorithm outperforms our version using a regression algorithm. We also compared the performance of the jokes generated by Gag with human-created jokes. When we constrain the human-created to either not being filtered by another online platform or only using single-word template values, the frequency of funny jokes generated by Gag is over 50% of the frequency of funny human-created jokes.

[7] https://github.com/TWinters/Markov
[8] https://github.com/TWinters/Datamuse-Java

Appendices

Appendix A

Program manuals

This section explains how to set up the programs created for this thesis.

A.1 Deploying JokeJudger

1. **Clone the git repository:** The repository can be found on https://github.com/TWinters/JokeJudger

2. **Set up a MySQL server:** This can be done in several ways, depending on the platform. For Windows, WampServer[1] can be used.

3. **Load MySQL database schema:** The JokeJudger schema can be found in `JokeJudger > design > database_diagram.mwb`. This schema needs to be imported into the MySQL database.

4. **Install NodeJS:** In order to run the JokeJudger server, NodeJS[2] has to be installed.

5. **Set up the environment and run NodeJS:** In order for JokeJudger to run, it has to know where the database is. The environment has to contain the following variables with the correct values for the created database: `MYSQL_HOST`, `MYSQL_USERNAME`, `MYSQL_PASSWORD` and `MYSQL_DATABASE`. Optionally, to allow JokeJudger to send mails when creating an account and allowing users to reset their password, an SMTP server can be linked with the variable `SMTP_HOST`, `SMTP_USER` and `SMTP_PASSWORD`. After the environment is set up, the server can be started using the command "`node server`".

A.2 JokeJudger Data

We made the data collected by JokeJudger during both the training data collection phase as well as the evaluation phase available for download as an anonymised

[1] http://www.wampserver.com/en/
[2] https://nodejs.org/en/

MySQL database and as a comma-separated value file with all ratings per joke on https://github.com/TWinters/JokeJudger-Data.

A.3 Extending JokeJudger to Other Types of Jokes

The JokeJudger is made for collecting jokes using the *"I like my X like I like my Y, Z"*-template. As such, jokes are saved as X, Y, Z. In order to collect jokes using any template, several small steps have to be undertaken in order to make the platform look right.

1. **Removing the template application:** The jokes service component, which can be found under *JokeJudger* > *public* > *js* > *services* > *jokes.js* has to be changed to just return `"joke.x"`, as this is where we will save the full joke.

2. **Updating the create page:** The create page, found in `JokeJudger > public > views > create.html` should be modified such that the input fields for Y and Z, the generator and the suggestion elements are removed, as they would not make sense any more. The input field of X should also be changed to a text field to a text area, as to allow long jokes.

3. **Updating the database:** In the MySQL database header of the *jokes* table, the type of x needs to be changed to `"TEXT"` in order to allow for more jokes longer than 255 characters.

A.4 Deploying Generalised Analogy Generator

This section explains how to set up the Gag joke generator.

1. **Setting up Java environment:** In order for Gag to work, Java 8 SE[3] and JDK [4] needs to be installed.

2. **Setting up required Google N-gram databases:** In order for Gag to work, the Google Ngrams[5] needs to be present in a MySQL database. More specifically, both English One Million 1-gram and 2-gram needs to be loaded in using our Java tool[6] in a database following database design specified in the repository. Loading this database will take several hours.

3. **Cloning GOOFER repository:** The Gag system can be found in the same repository as the Goofer framework repository[7].

[3] http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html
[4] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
[5] https://storage.googleapis.com/books/ngrams/books/datasetsv2.html
[6] https://github.com/TWinters/google-ngrams-to-mysql
[7] https://github.com/TWinters/Goofer

4. **Running GAG system:** The GAG system can be executed by using Java to run the main method of the `GeneralisedAnalogyGenerator.java` class. It supports following arguments:

Argument	**Description**
-outputModel	Path where the program should output the training model file
-output	Path where the program should output the training model file
-maxSimilarity, -maxSim	If given, GAG will only output generations if it differs enough (no more words similar than this value) from previous generations
-outputWords	Allow the template values in the training model file: classifiers have diffulty dealing with strings though!
-inputJokes	Path to the input jokes file
-sortRating, -sort	Wether or not the output should be sorted by their rating
-minScore	Minimal score threshold to be considered a good joke
-sqlHost	Host of the SQL database of the n-grams database
-sqlPost	Port of the SQL database of the n-grams database
-sqlUser	Username of the SQL database of the n-grams database
-sqlPassword	Password of the SQL database of the n-grams database
-sqlDB	Database name of the SQL database of the n-grams database
-dictionary	Path to the WordNet dictionary
-posFile	Path to the Stanford POS tagger
-classifier	The classifier to use to learn from the input jokes
-aggregator	The rating aggregator to combine the ratings with
-x	First template value of an analogy joke
-y	Second template value of an analogy joke
-z	Third template value of an analogy joke
-generator, -g	Type of template values generator: *sql*, *datamuse* or *twogram*

Bibliography

[1] T. Agustini and R. Manurung. Automatic evaluation of punning riddle template extraction. In *ICCC*, pages 134–139, 2012.

[2] S. Attardo. The semantic foundations of cognitive theories of humor. 10:395–420, 01 1997.

[3] S. Attardo and V. Raskin. Script theory revis(it)ed: joke similarity and joke representation model. *Humor*, 4(3):293–347, 1991.

[4] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman. Patchmatch: a randomized correspondence algorithm for structural image editing. *ACM Trans. Graph.*, 28(3), 2009.

[5] C. Bauckhage. Insights into internet memes. In L. A. Adamic, R. A. Baeza-Yates, and S. Counts, editors, *ICWSM*. The AAAI Press, 2011.

[6] K. Binsted, B. Bergen, S. Coulson, A. Nijholt, O. Stock, C. Strapparava, G. Ritchie, R. Manurung, H. Pain, A. Waller, and D. O'Mara. Computational humor. *IEEE Intelligent Systems*, 21(2):59–69, 2006.

[7] K. Binsted and G. Ritchie. An implemented model of punning riddles. *CoRR*, abs/cmp-lg/9406022, 1994.

[8] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, Oct. 2001.

[9] A. Chandrasekaran, D. Parikh, and M. Bansal. Punny captions: Witty wordplay in image descriptions. *CoRR*, abs/1704.08224, 2017.

[10] A. Chandrasekaran, A. K. Vijayakumar, S. Antol, M. Bansal, D. Batra, C. L. Zitnick, and D. Parikh. We are humor beings: Understanding and predicting visual humor. In *CVPR*, pages 4603–4612. IEEE Computer Society, 2016.

[11] A. Desoky. Jokestega: Automatic joke generation-based steganography methodology. *Int. J. Secur. Netw.*, 7(3):148–160, Mar. 2012.

[12] J. Dunn. We put siri, alexa, google assistant, and cortana through a marathon of tests to see who's winning the virtual assistant race - here's what we found. http://uk.businessinsider.com/siri-vs-google-assistant-cortana-alexa-2016-11, 2016.

[13] C. Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[14] W. N. Francis and H. Kucera. Brown corpus manual. Technical report, Department of Linguistics, Brown University, Providence, Rhode Island, US, 1979.

[15] S. Freud and J. Strachey. *Jokes and Their Relation to the Unconscious*. Complete Psychological Works of Sigmund Freud. Norton, 1960.

[16] S. Gella, C. Strapparava, and V. Nastase. Mapping wordnet domains, wordnet topics and wikipedia categories to generate multilingual domain specific resources. In N. C. C. Chair), K. Choukri, T. Declerck, H. Loftsson, B. Maegaard, J. Mariani, A. Moreno, J. Odijk, and S. Piperidis, editors, *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland, may 2014. European Language Resources Association (ELRA).

[17] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval*, 4(2):133–151, July 2001.

[18] B. Heate. Google is looking to creative writers and comedians to help humanize assistant. https://techcrunch.com/2016/10/10/google-laughsistant/, 2016.

[19] R. Hirst. Siri, echo and google home: are digital assistants the future of the office? https://www.theguardian.com/media-network/2016/nov/21/siri-echo-google-home-digital-assistants-future-office, 2016.

[20] B. A. Hong and E. Ong. Automatically extracting word relationships as templates for pun generation. In *Proceedings of the Workshop on Computational Approaches to Linguistic Creativity*, CALC '09, pages 24–31, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[21] R. Hunicke, M. LeBlanc, and R. Zubek. Mda: A formal approach to game design and game research. 2004.

[22] D. Jurafsky and J. H. Martin. N-grams. 2014.

[23] B.-M. Justin McKay. Generation of idiom-based witticism to aid second language learning. *Proceedings of April Fools' Day Workshop on Computational Humour (TWLT 20)*, pages 77–87, April 2002.

[24] J. T. Kao, R. Levy, and N. D. Goodman. A computational model of linguistic humor in puns. *Cognitive Science*, 40(5):1270–1285, 2016.

[25] A. Karpathy, J. Johnson, and F. Li. Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078, 2015.

[26] C. Kiddon and Y. Brun. That's what she said: Double entendre identification. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 89–94, Portland, OR, USA, June 2011. http://dl.acm.org/citation.cfm?id=2002756ACM ID: 2002756.

[27] C. Kim and K. Shim. Text: Automatic template extraction from heterogeneous web pages. *IEEE Transactions on Knowledge and Data Engineering*, 23(4):612–626, April 2011.

[28] A. Koestler. *The Act of Creation*. An Arkana book : psychology/psychiatry. Arkana, 1964.

[29] A. Krikmann. Contemporary linguistic theories of humour. *Folklore: Electronic Journal of Folklore*, (33):27–58.

[30] G. Lessard and M. Levison. Computational modelling of linguistic humour, tom swifties. *ALLCIACIi92 Conference Abstracts*, pages 175–178, 1992.

[31] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[32] B. Magnini and G. Cavaglià. Integrating subject field codes into wordnet. In *Proceedings of the Second International Conference on Language Resources and Evaluation (LREC-2000)*, Athens, Greece, May 2000. European Language Resources Association (ELRA). ACL Anthology Identifier: L00-1167.

[33] J. C. Mallery. Thinking about foreign policy: Finding an appropriate role for artificially intelligent computers. In *Master's thesis, M.I.T. Political Science Department*, 1988.

[34] R. Manurung, G. Ritchie, H. Pain, A. Waller, D. Mara, and R. Black. The construction of a pun generator for language skills development. *Applied Artificial Intelligence*, 22(9):841–869, 2008.

[35] R. Manurung, G. Ritchie, and H. Thompson. Using genetic algorithms to create meaningful poetic text. *Journal of Experimental & Theoretical Artificial Intelligence*, 24(1):43–64, 2012.

[36] R. Mihalcea and C. Strapparava. Making computers laugh: Investigations in automatic humor recognition. In *HLT/EMNLP 2005, Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference, 6-8 October 2005, Vancouver, British Columbia, Canada*, pages 531–538. The Association for Computational Linguistics, 2005.

[37] G. A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, Nov. 1995.

[38] M. Minsky. *Jokes and the Logic of the Cognitive Unconscious*, pages 175–200. Springer Netherlands, Dordrecht, 1984.

[39] M. Mulder and A. Nijholt. Humour research: State of art. 02(34):–, 9 2002. Imported from CTIT.

[40] MyOxygen. Rise of the bots. http://www.myoxygen.co.uk/blog/rise-of-the-bots/, 2016.

[41] S. Petrović and D. Matthews. Unsupervised joke generation from big data. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 228–232, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

[42] G. Pilato, A. Augello, G. Vassallo, and S. Gaglio. Ehebby: An evocative humorist chat-bot. *Mobile Information Systems*, 4(3):165–181, 2008.

[43] V. Raskin. *Semantic Mechanisms of Humor*. Studies in Linguistics and Philosophy. D. Reidel, 1 edition, 1985.

[44] V. Raskin. A little metatheory: Thought on what a theory of computational humor should look like, 2012.

[45] V. Raskin and S. Attardo. Non-literalness and non-bona-fide in language: An approach to formal and computational treatments of humor. *Marcelo Dascal, Pragmatics & Cognition 2:1*, pages 31–69, 1994.

[46] G. Ritchie. Developing the incongruity-resolution theory. 1999.

[47] G. Ritchie. Current directions in computational humour. *Artificial Intelligence Review*, 16(2):119–135, 2001.

[48] G. Ritchie. The structure of forced reinterpretation jokes. *Proceedings of April Fools' Day Workshop on Computational Humour (TWLT 20)*, pages 47–56, April 2002.

[49] G. Ritchie. Computational mechanisms for pun generation. In *Proceedings of the 10th European Natural Language Generation Workshop.*, pages 125–132, Department of Computer Science, University of Aberdeen, Aberdeen AB243UE, Scotland, 2005. ACL Anthology.

[50] B. Rose. Stand-up comedy using only siri, alexa, cortana and google home. https://www.youtube.com/watch?v=r0-89oBeBbQ.

[51] A. Schopenhauer. *The World as Will and Representation*. Number v. 1 in Dover books on philosophy. Dover Publications, 1958.

[52] D. Shahaf, E. Horvitz, and R. Mankoff. Inside jokes: Identifying humorous cartoon captions. In *KDD*, pages 1065–1074. ACM, 2015.

[53] D. Shahaf, E. Horvitz, and R. Mankoff. Inside jokes: Identifying humorous cartoon captions. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1065–1074, New York, NY, USA, 2015. ACM.

[54] A. Sharan and M. L. Joshi. An algorithm for finding document concepts using semantic similarities from wordnet ontology. *IJCVR*, 1(2):147–157, 2010.

[55] H. Spencer. *The Physiology of Laughter*. Macmillan, 1860.

[56] O. Stock and C. Strapparava. Getting serious about the development of computational humor. In G. Gottlob and T. Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 59–64. Morgan Kaufmann, 2003.

[57] O. Stock and C. Strapparava. Hahacronym: Humorous agents for humorous acronyms. *Humor-International Journal of Humor Research*, 16(3):297–314, 2003.

[58] J. Taylor. *Computational Recognition of Humor in a Focused Domain*. University of Cincinnati, 2004.

[59] J. Taylor. Do jokes have to be funny: Analysis of 50 "theoretically jokes". In *Artificial Intelligence of Humor, Papers from the 2012 AAAI Fall Symposium, Arlington, Virginia, USA, November 2-4, 2012*, volume FS-12-02 of *AAAI Technical Report*. AAAI, 2012.

[60] Thricedotted. Found lexicons, generated grammars. https://www.youtube.com/watch?v=e7lxHlp2gGU.

[61] A. Valitutti. How many jokes are really funny? towards a new approach to the evaluation of computational humour generators. In *Proceedings of International Workshop on Natural Language Processing and Cognitive Science (NLPCS 2011)*, 2011.

[62] A. Valitutti, H. Toivonen, A. Doucet, and J. M. Toivanen. "let everything turn well in your wife": Generation of adult humor using lexical constraints. In *ACL (2)*, pages 243–248. The Association for Computer Linguistics, 2013.

[63] C. Venour. The computational generation of a class of puns. Master's thesis, Queen's University, Kingston, Ontario, 1999.

[64] A. Waller, R. Black, D. A. Mara, H. Pain, G. Ritchie, and R. Manurung. Evaluating the standup pun generating software with children with cerebral palsy. *ACM Transactions on Accessible Computing (TACCESS)*, 1(3):1–27.

[65] G. V. D. Zwaag. Apple zoekt een grappenmaker voor siri: heb jij genoeg humor om te helpen? https://www.iculture.nl/nieuws/siri-vacature-grappen-bedenken/, 2016.

Automatisch humor genereren vanuit voorbeelden

Thomas Winters
Departement Computerwetenschappen
KU Leuven
thomas.winters@student.kuleuven.be

Prof. dr. Danny De Schreye
Departement Computerwetenschappen
KU Leuven
danny.deschreye@kuleuven.be

Vincent Nys
Departement Computerwetenschappen
KU Leuven
vincent.nys@kuleuven.be

***Samenvatting*—Dit artikel bespreekt hoe een computerprogramma kan leren om humoristische teksten te genereren op basis van voorbeelden. We construeren een generiek framework om deze taak te vervullen, en gebruiken humortheorie en generaliseren computationele humor concepten om de constructie hiervan te sturen. Dit framework gebruiken we om een systeem te implementeren dat in staat is om analogiehumor te genereren.**

1. Introductie

In dit artikel onderzoeken we een methode voor een computerprogramma om humor te leren genereren uit humoristische voorbeelden. Het domein van dit artikel is computationele humor, een tak binnen natuurlijke taalverwerking en artificiële intelligentie. In dit domein zijn er drie soorten taken die onderzocht worden: het genereren, het detecteren en het begrijpen van humor [1] [2]. Computationele humor is een AI-compleet probleem, wat inhoudt dat AI eerst zo intelligent als mensen moet zijn vooraleer men computationele humor volledig kan oplossen [2]. Bijgevolg behandelt het merendeel van het bestaande onderzoek in computationele humor slechts één taak op één type humor [1]. Zulk onderzoek produceert programma's die bijvoorbeeld in staat zijn tot het genereren van raadsels met woordspelingen [3] [4], analogieën [5], grappige acroniemen [6], cartoon bijschriften [7] en het detecteren van toespelingen [8], klop-klop moppen [9], en korte grappen [10].

Bestaande humorgeneratoren beschikken meestal over voorgedefinieerde regels. Dit beperkt zowel de tijdsafhankelijke, als de individuele dimensie van de gegenereerde humor. Met de tijdsafhankelijke dimensie bedoelen we dat de kwaliteit van de perceptie van humor tijdsafhankelijk is. Humor van vroeger is vaak minder grappig doordat o.a. de herkenbare situaties en gebruiksvoorwerpen veranderd zijn door de evolutie van de maatschappij. De waardering van bepaalde soorten humor verandert ook door de hogere blootstelling aan humor door de geëvolueerde media. Ook op korte termijn zijn fluctuaties in humorperceptie zichtbaar, bv. in internet memes [11]. Huidige computationele humorgeneratoren zijn niet in staat om zich automatisch aan te passen aan deze nieuwe vormen van humor, en verliezen hierdoor waarde doorheen de tijd.

Een ander aspect waar weinig rekening mee wordt gehouden in computationele humorgeneratoren is de individuele dimensie van humor. De perceptie van humor verschilt van persoon tot persoon. Verscheidene onderzoeken, waaronder het onze, tonen grote onenigheid tussen menselijke beoordelaars over de kwaliteit van bepaalde moppen [12] [5] [6]. Het is dus logisch om te trachten gegenereerde humor aan te passen aan de voorkeuren en de kennis van een individu.

2. Probleemstelling

In dit artikel bieden we oplossingen aan om de tijdsafhankelijke en individuele dimensies van humor bij humorgeneratoren te verbeteren. We doen dit door aan te tonen hoe classificatie- en regressiealgoritmes benut kunnen worden voor humor. Een generator die humor kan leren uit andere humoristische artefacten kan zich aanpassen aan de hedendaagse humor. Een generator die humor kan leren uit de beoordelingen van zijn vorige generaties is in staat zich aan te passen aan het individu waarmee het converseert. Beide problemen kunnen opgelost worden door een generator te creëren die in staat is zinvolle kennis te leren uit humoristische voorbeelden.

Om dit framework te creëren, generaliseren we het concept van schema's zodanig dat het classificatie- en regressiealgoritmes kan hanteren. Daarna passen we humortheorie toe om goede metrieken voor humor te identificeren en te beargumenteren. Deze componenten verwerken we in ons framework, genaamd GOOFER. Om aan te tonen dat dit framework werkt, implementeren we de belangrijkste componenten en creëren we een gegeneraliseerde analogie generator, afgekort tot GAG. Dit stelt ons in staat om de kwaliteit van humor gegenereerd met classificatie en regressiealgoritmes onderling te vergelijken, alsook met door mensen gecreëerde grappen en met bestaande systemen met vaste regelsets.

3. Achtergrond

3.1. Humor theorie

Er zijn verschillende categorieën van humortheorie. De drie belangrijkste theorieën beschouwen ofwel opluchting,

superioriteit of incongruentie als bron van humor [13]. De laatste categorie is de meest geaccepteerde theorie alsook de meest relevante voor computationele humor. Een formele humortheorie binnen de ongerijmdheidscategorie is de *General Theory of Verbal Humour* die zes parameters identificeert waarin grappen verschillen [14]. Deze parameters zijn, van zwakste naar sterkste: de taal, het verhaalstijl, het doelwit, de situatie, het logische mechanisme en de tegenstrijdige scripts [14]. Hoewel deze theorie linguïstisch nuttig kan zijn, krijgt ze vaak de commentaar moeilijk implementeerbaar te zijn en niet de grappigheid te kunnen voorspellen [15] [16]. Een meer implementeerbare theorie is Ritchie's incongruentie resolutie theorie [17]. Deze theorie ziet humor als een set-up gevolgd een pointe. De set-up heeft twee mogelijke interpretaties, waarvan de eerste meer voor de hand ligt dan de tweede. De pointe veroorzaakt eerst ongerijmdheid met de set-up, gevolgd door het naar boven halen van de tweede interpretatie van de set-up. Deze theorie stelt vijf eigenschappen van humor voor, zijnde de DUIDELIJKHEID van de eerste interpretatie van de set-up, het CONFLICT tussen de pointe en de eerste betekenis van de set-up, de VERENIGBAARHEID van de pointe met de verborgen interpretatie van de set-up, het CONTRAST tussen de twee mogelijke interpretaties en de ONGEPASTHEID van de tweede betekenis [17].

3.2. Gerelateerd onderzoek

Een veelgebruikte techniek binnen computationele humor is het gebruik van sjablonen en schema's [3] [4] [15] [18]. Sjablonen zijn vaste stukken tekst met variabelen die ingevuld worden door een andere databron. Een schema fungeert als zulke databron. Een schema is een structuur die de lexicale woordrelaties tussen de invullingen in een sjabloon definiëren [3]. Zo kan het bijvoorbeeld in een woordspeling de relatie *"synoniem van een homoniem"* tussen sleutelwoorden van de grap modelleren [3] [4].

T-PEG (*Template-Based Pun Extractor and Generator*) is een programma dat humor ontleedt en genereert. Het detecteert een sjabloon op basis van de aanwezige, gedetecteerde linguïstische relaties in een gegeven woordspeling. Het gebruikt deze woordrelaties om een schema te maken gelijkaardig aan de schema's van JAPE en STANDUP [3] [4]. Onderzoek over hoe goed T-PEG de sjablonen en schema's van dit laatste systeem kan achterhalen stelde een precisie van 61% vast voor de gegenereerde dataset [19]. Dit onderzoek clustert de gevonden templates samen met een semantische gelijkaardigheidsfunctie [20]. We gebruiken dit sjabloon extractie algoritme als bewijs voor de sjabloon extractie component in het GOOFER framework.

Petrovic & Matthews hebben een computationeel model geconstrueerd voor het genereren van analogiegrappen met het *"I like my X like I like my Y, Z"* sjabloon [5]. Dit onderzoek heeft het doel aan te tonen dat grappige analogieën gegenereerd kunnen worden met behulp van GOOGLE NGRAMS. Het model minimaliseert het product van vijf metrieken, waarvan er vier berekend worden aan de hand van N-grammen. Deze metrieken minimaliseren de gemeenschappelijkheid tussen X en Y en de frequentie van Z en maximaliseren het aantal definities van Z en de frequentie van zowel X als Y na Z. De kandidaatgrappen worden gegenereerd op basis van adjectief- en substantiefcombinaties aanwezig in de N-grammen. Volgens de evaluatie van het onderzoek is 16,3% van de gegenereerde grappen grappig.

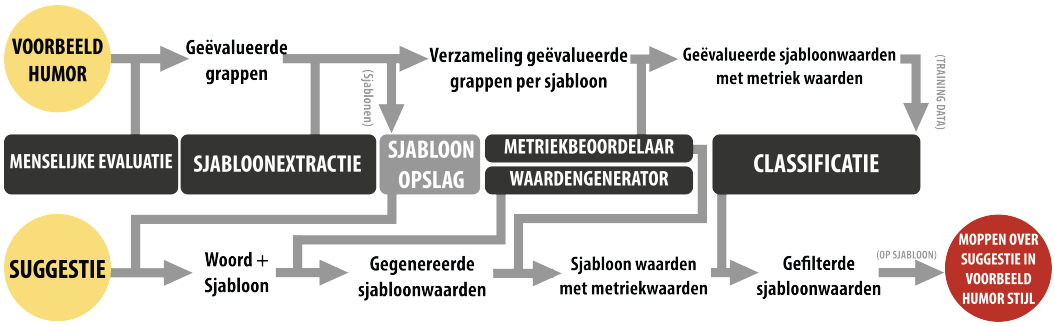
4. GOOFER framework

We construeren in dit artikel een framework dat verder bouwt op de besproken humortheorie, -concepten en -systemen zodat het in staat is om humor uit voorbeelden te leren genereren. We noemen dit het GOOFER framework, wat staat voor *"Generator Of One-liners From Examples with Ratings"*. Het framework bereikt dit doel door de sjablonen te detecteren en de variërende sjabloonwaarden uit de gegeven grappen te halen. Het leert de verbanden tussen deze waarden met behulp van humortheorie bekrachtigde metrieken en classificatie- of regressiealgoritmes. Een meer primitieve generator kan dan sjabloonwaarden kandidaten genereren, waarvan het classificatiealgoritme dan de beste mogelijkheden kiest. De gekozen waarden worden in het sjabloon ingevuld om een grap te construeren.

4.1. Schema generalisatie

Om dit systeem te laten functioneren, moeten we eerst laten zien hoe de notie van een schema gegeneraliseerd kan worden zodanig dat het classificatie- en regressiealgoritmes kan gebruiken. Schema's gebruiken typisch lexicale relaties tussen woorden als restricties, en kunnen hierdoor ook gebruikt worden als generator [3] [4] [15]. De mogelijkheidsruimte van de generator is dus beperkt door deze relaties. Ieder element in deze ruimte wordt door de generator typisch als gelijkwaardig gezien. Een schema in het STANDUP systeem wordt gedefinieerd met vijf attributen, zijnde de hoofding (= de naam van het schema en de variabelen), de lexicale voorwaarden, twee sjabloonspecificaties (voor de vraag en het antwoord, met mogelijk verdere lexicale restricties) en de sleutelwoorden om equivalentie te bepalen.

Deze definitie van een schema kunnen we generaliseren naar een probabilistisch schema, zodanig dat het metrieken met een reële waarde als uitkomst kan gebruiken in plaats van binaire lexicale restricties. We definiëren een probabilistisch schema als een schema met zes attributen. We nemen de hoofding, de sjabloonspecificatie en de sleutelwoorden over vanuit de klassieke definitie, en voegen drie attributen toe. De nieuwe attributen zijn de metrieken met bijhorende sjabloonvariabelen waarop ze gebruikt worden, de aggregator van de metriekwaarden en de primitieve generator van sjabloonwaarden. Het besproken model van Petrovic's analogie generator (zie sectie 3.2) past binnen dit nieuwe schema. De metrieken zijn de vijf veronderstellingen die gemaakt worden, de aggregator is de minimalisatie van het product van deze metriekwaarden en de generator is de N-gram generator.



Figuur 1. Een schematisch overzicht van de componenten en doorstroom van het GOOFER framework.

De notie van een probabilistisch schema generaliseert de restrictieschema's. We kunnen namelijk de lexicale restricties vertalen naar metrieken die 0 of 1 geven als de relatie respectievelijk onvervuld of vervuld is. De aggregator is een functie die checkt of alle waarden 1 teruggeven. De generator wordt ingevuld door alle mogelijke combinaties van lexiconwoorden te gebruiken als sjabloonwaarden. Gezien de andere drie parameters uit de definitie van een restrictieschema komen, toont dit aan dat een probabilistisch schema de normale definitie van een schema generaliseert.

We definiëren een classificatieschema als een extensie van een probabilistisch schema. Een classificatie schema is een probabilistisch schema dat als aggregatorattribuut een classificatie- of een regressiealgoritme gebruikt. Indien dit algoritme een waarde boven een bepaalde drempel heeft, zal het algoritme de kandidaat sjabloonwaarden accepteren.

4.2. Metrieken identificatie

De eerder besproken incongruentie-resolutie theorie van Ritchie bevat vijf parameters (zie sectie 3.2) waarop we een goede standaard set van metrieken voor het framework kunnen definiëren [17]. Voor de DUIDELIJKHEID parameter kan er gebruik gemaakt worden van de semantische afstand tussen twee woorden in een lexicon (bv. WordNet) alsook de frequentie van een woord in een 1-gram. Voor de CONFLICT parameter kan dezelfde semantische afstand gebruikt worden, alsook de frequentie van het samen voorkomen van twee woorden met behulp van 2-grammen. De VERENIGBAARHEID parameter kan benaderd worden met de frequentie dat twee woorden samen voorkomen in 2-grammen, alsook door het meten van ambiguïteit met metrieken zoals het aantal betekenissen van een woord en de homoniem relatie. De CONTRAST parameter dient voor het contrast tussen twee interpretaties. Dit contrast kan gemeten worden met behulp van tegenovergestelde domeinen met WORDNET DOMAINS, de gelijkaardigheid tussen gerelateerde woorden met behulp van een lexicon en het verschil in de frequenties van adjectieven gebruikt bij bepaalde substantieven. Als laatste kan de ONGEPASTHEID parameter benaderd worden door de woordfrequentie in een erotische corpus, gezien dit benadert hoe seksueel getint een woord is [8]. Ook kan het benaderd worden door de N-gram frequentie in een neutrale corpus, gezien dit lage voorspelbaarheid benadert.

4.3. Doorstroom van het systeem

Nu we schema's gegeneraliseerd hebben en de metrieken geïdentificeerd, kunnen we uitleggen hoe we het GOOFER framework geconstrueerd hebben. Dit framework is in staat om humor uit voorbeelden te leren genereren. Een model met de componenten en doorstroom van data van het GOOFER framework is te zien op figuur 1.

Initieel worden er in GOOFER voorbeeldgrappen ingevoerd. Deze dienen voorzien te worden van beoordelingen in de *Menselijke Evaluatie* component. Dit is de enige component die menselijke tussenkomst nodig heeft. De *Sjabloon Extractie* component haalt de sjablonen uit de grappen, onthoudt ze in de *Sjabloonopslagplek* en geeft de sjabloonwaarden met hun beoordelingen door aan de *Metriek Beoordelaar* component. Deze component voegt functiewaarden toe met behulp van de geïdentificeerde metrieken, zodanig dat de *Classificatie* component hieruit kan leren.

Na geïnitialiseerd te zijn kan een gebruiker aan het systeem woorden suggereren waarover de gegenereerde grappen moeten gaan. Aan deze suggestie wordt een sjabloon uit de *Sjabloonopslagplek* verbonden, en worden er waarden gegenereerd met behulp van de *Sjabloon Waarden Generator*. Deze generator kan beperkingen opgelegd krijgen door de *Sjabloon Extractie* component, die bijvoorbeeld de woordsoort van de sjabloonwaarden gevonden heeft. Deze sjabloonwaarden krijgen dan functiewaarden van de *Metriek Beoordelaar* component, zodat de *Classificatie* component de juiste waarden eruit kan filteren. Op het einde worden de gefilterde waarden op het sjabloon toegepast, zodat het proces gegenereerde grappen terug geeft.

Dit framework kan ook gebruikt worden om een systeem dat humor genereert uit te breiden zodat het kan leren uit eerder gegenereerde humor. Dit kan bereikt worden door de *Sjabloonwaardengenerator* in te vullen met dat generatie systeem. Ook humortheorieonderzoekers kunnen dit framework gebruiken om hypotheses over welke attributen belangrijk zijn in een set van humor. Dit gebeurt door het classificatiealgoritme zo te kiezen dat de de belangrijkste attributen gevonden worden. In een algoritme met beslissingsbomen worden deze attributen gevonden door te kijken welke attributen zorgen voor de grootste vermindering van onzuiverheid in de knopen.

5. GAG systeem

Om aan te tonen dat ons GOOFER framework werkt en te kunnen evalueren, implementeerden we een deelverzameling van de componenten in onze *Gegeneraliseerde AnalogieGenerator*, oftewel GAG. Een schematisch overzicht van dit systeem is te zien op figuur 2. Dit systeem is in staat om uit grappen met het *"I like my X like I like my Y, Z"* sjabloon te leren. Dit is dezelfde soort grap als Petrovic zijn analogie generatie systeem. We kunnen gelijkaardige beslissingen maken zodat we onze gegeneraliseerde aanpak kunnen vergelijken met zijn handmatige aanpak.

De implementatie vereenvoudigt en specialiseert enkele componenten van het GOOFER framework. De grootste verandering is dat er geen sjabloon extractie component is. Hier is immers al voldoende onderzoek naar gedaan binnen computationele humor [21] [19]. Ook kozen we ervoor dat de *Sjabloonwaardengenerator* kandidaten voorstelt op basis van Google 2-grammen. Een extensie die we doorvoeren is dat de voorbeeldgrappen gecreëerd worden door *Menselijke Evaluatie* component. We hebben hiervoor het platform JokeJudger gebouwd.

5.1. JokeJudger

Een systeem dat het GOOFER framework gebruikt, heeft nood aan een dataset met grappen met bijbehorende scores. Een mogelijke manier is om populaire platformen die scores bij een post bijhouden, zoals Twitter en Reddit, te gebruiken voor het verzamelen van data. Deze platformen hebben echter meerdere factoren die dit soort data vervormen. Een eerste probleem is dat grappen die al een groot aantal positieve scores kregen meer getoond worden, en zo nog meer positieve reactie krijgen dan een grap van even goede kwaliteit die door andere factoren initieel een minder aantal positieve scores had. Een tweede probleem is dat volgers van een persoon op Twitter of van een sub-reddit meestal een zeer gecorreleerde smaak voor humor hebben. Een derde probleem is dat een "leuk/niet leuk" binair systeem niet precies genoeg is voor onze doeleindes. Om deze problemen op te lossen hebben we JokeJudger[1] gemaakt. Op dit platform kunnen gebruikers anoniem grappen toevoegen, en grappen van andere mensen anoniem beoordelen. De pagina waar de gebruikers hun oordeel kunnen vellen, presenteert steeds één grap, waarbij de gebruiker dan één tot vijf sterren kan geven, of de grap markeren als te beledigend, of markeren dat hij de grap niet snapt. Deze pagina sorteert de gepresenteerde grappen zodanig dat de grappen met het minste aantal beoordelingen die nog niet door de gebruiker beoordeeld zijn vooraan worden gezet. Op deze manier wordt zowel het probleem dat populaire grappen populairder worden vermeden, alsook zorgt dit ervoor dat het aantal beoordelingen per grap dichter bij elkaar zal liggen over de hele dataset. Op de grapcreatiepagina kunnen willekeurige uitdagingen worden gegenereerd ter inspiratie voor de gebruiker. JokeJudger beloont gebruikers die grappen invoeren met data-analysepagina's waar ze overzichtelijk kunnen zien hoe goed hun grappen scoren.

Gedurende de fase waarin we trainingdata verzamelden, hebben we met dit platform 336 grappen en 4828 beoordelingen van 106 gebruikers verzameld. Aan het einde van de evaluatiefase waren dit 524 grappen (waarvan 100 gegenereerd met GAG), 9034 beoordelingen en 418 markeringen komende van 203 gebruikers. Deze data is online ter beschikking gesteld[2].

5.2. Data verwerking

We kozen een deelverzameling van de voorgestelde GOOFER metrieken als metrieken voor GAG. We hebben de verzameling zo gekozen dat iedere parameter van de incongruentie-resolutie theorie van Ritchie minstens één metriek had. De gekozen metrieken zijn de woordfrequentie (Google 1-grams), de relatieve woordfrequentie (Google 2-grams), het aantal betekenissen (met WordNet), het verschil in de adjectief vector (met Google 2-grams) en de 1-gram woordfrequentie van woorden in een erotische corpus. Deze metrieken worden voor alle (combinaties van) sjabloonwaarden berekend waar de woordsoort(en) toepasselijk is (adjectief verschil zal bv. enkel tussen X en Y berekend kunnen worden).

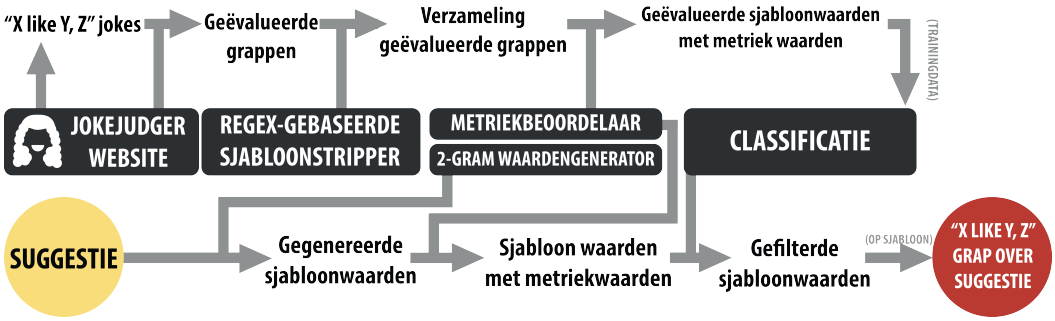
Een probleem voor de dataverwerking is dat GOOFER slechts met één woord per sjabloonwaarde werkt. De verzamelde grappen op JokeJudger bevatten echter vaak meerdere woorden per sjabloonwaarde. Om hiermee om te gaan, hebben we waar mogelijk een permutatie gemaakt van alle woorden met de geschikte woordsoort die in de sjabloonwaarde voorkomen, en deze toegevoegd aan de getransformeerde dataset. Zo wordt bv. de grap *"I like my coffee like I like my war, gruesome and cold"* getransformeerd naar de twee grappen *"I like my coffee like I like my war, gruesome"* en *"I like my coffee like I like my war, cold"*. Dit verandert onze dataset naar 528 sjabloonwaarden met slechts één woord per waarde.

5.3. Classificatie & Regressie

We hebben zowel classificatie op de meest voorkomende score als regressie op de gemiddelde score van de grappen uitgevoerd. Het Random Forest algoritme gaf voor beide versies het beste resultaat. Dit kunnen we verklaren doordat het algoritme willekeurige beslissingsbomen traint op basis van deelverzamelingen van features (en trainingdata), en hierdoor minder gevoelig is voor mogelijke ruis features [22]. Deze ruis features worden veroorzaakt door de vele metrieken die een GOOFER systeem met zich meebrengt, alsook door de complexiteit van het inschatten van humor in het algemeen. We hebben de algoritmes geverifieerd met 10-voudige kruisvalidatie. Voor classificatie was dit algoritme in staat tot het correct inschatten van de meest voorkomende score voor 61,55% van de trainingdata. In de trainingdata zelf stemt slechts 41,36% van de gebruikers op de meest

1. http://jokejudger.com

2. https://github.com/TWinters/JokeJudger-Data



Figuur 2. Een schematisch overzicht van GAG

voorkomende score. Voor de regressie versie was de root relatieve kwadraat fout 84,04%.

Het Random Forest algoritme heeft als voordeel dat het de belangrijkheid van ieder attribuut kan berekenen op basis van vermindering van de onzuiverheid in de knopen van de beslissingsbomen. Met onze mens-begrijpbare metrieken kan een humortheoreticus met het GOOFER framework dus een verzameling grappen evalueren met bepaalde metrieken en hiermee hypotheses verifiëren en genereren.

6. Evaluatie

Om dit systeem te evalueren, hebben we honderd grappen gegenereerd met GAG. Deze grappen werden gegenereerd op basis van vooraf ingevulde waarden voor X vanuit de trainingdata. De helft werd gegenereerd door een versie met het classificatiealgoritme, waarbij willekeurig elementen genomen werden uit de grappen die volgens het algoritme vier of vijf sterren toegewezen kregen. De andere helft werd gegenereerd door het regressiealgoritme. De grappen werden geüpload tussen de bestaande grappen op JokeJudger. Ook hebben we 67 grappen die door mensen werden gecreëerd toegevoegd, zodat de eerste gebruikers die beoordelingen gaven, niet enkel gegenereerde grappen zagen omwille van de "minste beoordelingen eerst" sortering op JokeJudger. Tijdens de evaluatiefase werden 4424 beoordelingen gegeven, waarvan 745 voor grappen van het classificatiealgoritme en 721 voor de grappen van het regressiealgoritme.

Bron	*4+ score*
GAG (Classificatie)	11.41%
GAG (Regressie)	10.12%
Mens (Allemaal)	27.38%
Mens (JokeJudger)	22.61%
Mens (Alleenstaande woorden)	21.08%

Figuur 3. Percentages van het aantal ratings met meer dan vier sterren voor elke bron van grappen.

De grappen van het regressiealgoritme kregen 5% meer één ster beoordelingen en 1,5% meer markeringen dat de gebruiker het niet begreep dan de grappen van het classificatiealgoritme. Voor de andere beoordelingen scoorde het classificatiealgoritme beter. 24,3% van de tijd kregen classificatie grappen een score van drie of meer sterren tegenover 21,36% van de grappen van het regressiealgoritme. Het GAG-systeem produceerde dus betere grappen volgens de JokeJudger gebruikers tijdens de evaluatiefase wanneer het gebruik maakt van classificatie op de meest voorkomende score dan met regressie op het gemiddelde.

In figuur 3 zien we dat grappen gecreëerd door mensen vaker een score van meer dan vier sterren krijgen dan grappen van GAG. De grappen die door mensen gecreëerd zijn, kunnen echter meer woorden bevatten dan de grappen gegenereerd door GAG, gezien ons systeem slechts één woord per sjabloonwaarde invult. Grappen van mensen die slechts één woord voor X, Y en Z gebruiken, krijgen slechts 21,08% van de tijd een beoordeling van vier of meer sterren. De frequentie van grappige GAG grappen is dus meer dan 50% van de frequentie grappige grappen van deze soort.

In de evaluatie van het analogie systeem van Petrovic werd aan vijf beoordelaars gevraagd om grappen te markeren als "niet grappig", "ietwat grappig" en "grappig". Grappen van Twitter waren 33,1% grappig, hun analogie systeem 16,3% en een willekeurige basis systeem 3,7% [5]. Indien we voor ons onderzoek scores van vier en vijf sterren zien als "grappig", zijn in onze evaluatie zowel mensen als ons systeem minder grappig. Hier kunnen verschillende redenen voor zijn. Een eerste reden is dat zijn staal van vijf beoordelaars minder betrouwbaar is dan onze groep van meer dan tweehonderd beoordelaars. Een tweede mogelijke reden is dat grappen van Twitter grappiger zijn dan op JokeJudger omwille van feit dat zulke platformen een menselijke filter vormen waar betere grappen meer worden gedeeld en dus meer laten voorkomen. We kunnen bewijs voor deze claim vinden door de grappen die we van Twitter en Reddit gehaald hebben voor de initiële JokeJudger dataset te elimineren. Na deze operatie zien we dat de overgebleven grappen die door mensen gemaakt zijn slechts 22,61% van de tijd een beoordeling van vier sterren of hoger krijgt. Ook in dit geval is onze generator dus meer dan half zo vaak als mensen grappig.

7. Toekomstig Onderzoek

Voor toekomstig onderzoek zou het interessant zijn om het GOOFER framework uit te breiden zodat het kan omgaan met sjabloonwaarden die langer zijn dan één woord. Ook

zou het interessant zijn om in plaats van vaste zinnen als sjabloon, grammatica's te gebruiken. Dit zou de variatie in de gegenereerde mopjes verhogen. Ook zou het sjabloon extractie algoritme dan zodanig uitgebreid kunnen worden dat het meerdere sjablonen bundelde binnen hetzelfde sjabloon. De trainingdata per sjabloon wordt dan drastisch verhoogd door deze samenvoeging, waardoor betere schema's gevonden kunnen worden.

8. Conclusie

We onderzochten in dit artikel hoe je een programma humor kan leren genereren op basis van voorbeeldgrappen. We hebben hiervoor het GOOFER framework gebouwd dat gebaseerd is op extensies en generalisaties van ander computationele humoronderzoek. We hebben JokeJudger gebouwd om data te verzamelen, en hebben zo een grote dataset van *"I like my X like I like my Y, Z"* grappen gecreëerd. We hebben delen van het GOOFER framework geïmplementeerd in het GAG-systeem om aan te tonen dat de componenten werken, en zodat we classificatie en regressie konden vergelijken. We vonden dat classificatiealgoritmes beter werken. We concludeerden ook dat indien menselijke grappen gelimiteerd werden tot de grappen met slechts één woord per sjabloonwaarde, of tot grappen die niet afkomstig zijn van een ander platform, het GAG-systeem in vergelijking hiermee half zo vaak grappig is.

Dankwoord

Ik zou graag mijn promotor Danny De Schreye en mijn begeleider Vincent Nys bedanken voor hun buitengewone ondersteuning en ideeën doorheen het jaar. Ook zou ik graag de gebruikers van JokeJudger bedanken voor het helpen bij het verzamelen van data. Als laatste zou ik graag mijn familie en vrienden bedanken voor hun steun.

Referenties

[1] G. Ritchie, "Current directions in computational humour," *Artificial Intelligence Review*, vol. 16, no. 2, pp. 119–135, 2001.

[2] K. Binsted, B. Bergen, S. Coulson, A. Nijholt, O. Stock, C. Strapparava, G. Ritchie, R. Manurung, H. Pain, A. Waller, and D. O'Mara, "Computational humor," *IEEE Intelligent Systems*, vol. 21, no. 2, pp. 59–69, 2006.

[3] K. Binsted and G. Ritchie, "An implemented model of punning riddles," *CoRR*, vol. abs/cmp-lg/9406022, 1994.

[4] R. Manurung, G. Ritchie, H. Pain, A. Waller, D. Mara, and R. Black, "The construction of a pun generator for language skills development," *Applied Artificial Intelligence*, vol. 22, no. 9, pp. 841–869, 2008.

[5] S. Petrović and D. Matthews, "Unsupervised joke generation from big data," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Sofia, Bulgaria: Association for Computational Linguistics, August 2013, pp. 228–232. [Online]. Available: http://www.aclweb.org/anthology/P13-2041

[6] O. Stock and C. Strapparava, "Hahacronym: Humorous agents for humorous acronyms," *Humor-International Journal of Humor Research*, vol. 16, no. 3, pp. 297–314, 2003. [Online]. Available: ⟨GotoISI⟩://WOS:000185726800002

[7] A. Chandrasekaran, D. Parikh, and M. Bansal, "Punny captions: Witty wordplay in image descriptions," *CoRR*, vol. abs/1704.08224, 2017. [Online]. Available: http://arxiv.org/abs/1704.08224

[8] C. Kiddon and Y. Brun, "That's what she said: Double entendre identification," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*, Portland, OR, USA, June 2011, pp. 89–94, http://dl.acm.org/citation.cfm?id=2002756ACM ID: 2002756. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002756

[9] J. Taylor, *Computational Recognition of Humor in a Focused Domain*. University of Cincinnati, 2004. [Online]. Available: https://books.google.be/books?id=cwfwjwEACAAJ

[10] R. Mihalcea and C. Strapparava, "Making computers laugh: Investigations in automatic humor recognition," in *HLT/EMNLP 2005, Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference, 6-8 October 2005, Vancouver, British Columbia, Canada*. The Association for Computational Linguistics, 2005, pp. 531–538. [Online]. Available: http://aclweb.org/anthology/H/H05/H05-1067.pdf

[11] C. Bauckhage, "Insights into internet memes." in *ICWSM*, L. A. Adamic, R. A. Baeza-Yates, and S. Counts, Eds. The AAAI Press, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/icwsm/icwsm2011.html#Bauckhage11

[12] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins, "Eigentaste: A constant time collaborative filtering algorithm," *Information Retrieval*, vol. 4, no. 2, pp. 133–151, July 2001.

[13] A. Krikmann, "Contemporary linguistic theories of humour," *Folklore: Electronic Journal of Folklore*, no. 33, pp. 27–58.

[14] S. Attardo and V. Raskin, "Script theory revis(it)ed: joke similarity and joke representation model," *Humor*, vol. 4(3), pp. 293–347, 1991.

[15] C. Venour, "The computational generation of a class of puns," Master's thesis, Queen's University, Kingston, Ontario, 1999. [Online]. Available: http://www.collectionscanada.gc.ca/obj/s4/f2/dsk1/tape3/PQDD_0006/MQ45304.pdf

[16] J. Taylor, "Do jokes have to be funny: Analysis of 50 "theoretically jokes"," in *Artificial Intelligence of Humor, Papers from the 2012 AAAI Fall Symposium, Arlington, Virginia, USA, November 2-4, 2012*, ser. AAAI Technical Report, vol. FS-12-02. AAAI, 2012. [Online]. Available: http://www.aaai.org/ocs/index.php/FSS/FSS12/paper/view/5606

[17] G. Ritchie, "Developing the incongruity-resolution theory," 1999.

[18] V. Raskin and S. Attardo, "Non-literalness and non-bona-fide in language: An approach to formal and computational treatments of humor," *Marcelo Dascal, Pragmatics & Cognition 2:1*, pp. 31–69, 1994.

[19] T. Agustini and R. Manurung, "Automatic evaluation of punning riddle template extraction." in *ICCC*, 2012, pp. 134–139.

[20] R. Manurung, G. Ritchie, and H. Thompson, "Using genetic algorithms to create meaningful poetic text," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 24, no. 1, pp. 43–64, 2012. [Online]. Available: http://dx.doi.org/10.1080/0952813X.2010.539029

[21] B. A. Hong and E. Ong, "Automatically extracting word relationships as templates for pun generation," in *Proceedings of the Workshop on Computational Approaches to Linguistic Creativity*, ser. CALC '09. Stroudsburg, PA, USA: Association for Computational Linguistics, 2009, pp. 24–31. [Online]. Available: http://dl.acm.org/citation.cfm?id=1642011.1642015

[22] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1010933404324

AUTOMATIC JOKE GENERATION: LEARNING HUMOUR FROM EXAMPLES

MOTIVATION

• **Computational humour:** state of the art = systems for **one or several specific types** of jokes. Usually using **templates & schemas**.

• **Holy grail**: system that can **learn humour** from **joke examples**, and then **generate** new ones using this knowledge: improves temporal and individual humour dimensions.

• This research is **first stepping stone** towards a **generalised system** by **generalising previous research**

TEMPLATES

"Sentences with holes"

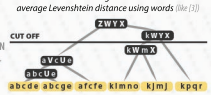
What do you call a **SynHomB** *with a* **MerA**? *A* **A HomB**.

I like my **X** *like I like my* **Y**, **Z**

(from [1] and [2])

GENERALISED FOR EXTRACTION

Hierarchical clustering with minimization of average Levenshtein distance using words (like [3])



HUMOUR METRICS

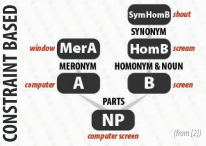
Usually, **carefully chosen relations** based on manually **reverse-engineering** joke based on assumptions

GENERALISED

A **knowledge base** of metrics for features & generation based on **Ritchies Incongruity-Resolution** humour theory parameters

SCHEMAS

CONSTRAINT BASED



(from [2])

GENERALISED

Map **constraints** to **functions** with **0 or 1** as output. Generate possibilities other than strict relations

+

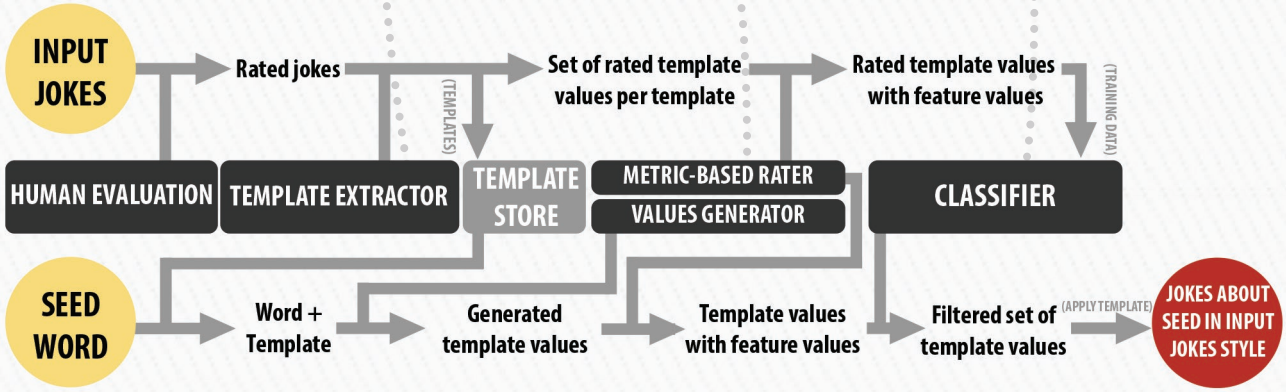
PROBABLISTIC

(from [1])

GENERALISED

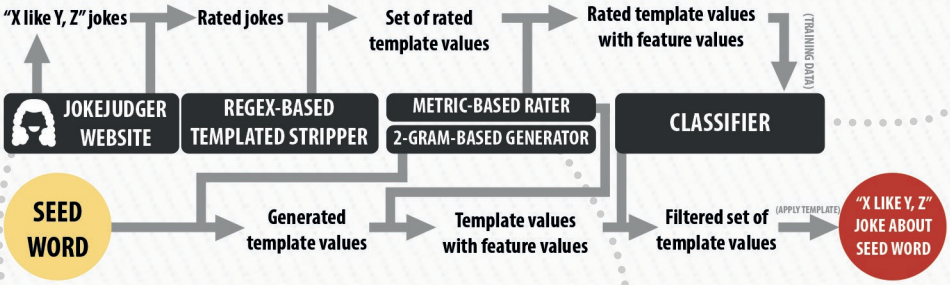
Remove unweighted **minimisation** & introduce **classifier** to learn more than only **hardcoded schemas**

GENERALISED SYSTEM



SIMPLIFIED SYSTEM

System to generate "I like my **X** like I like my **Y**, **Z**" jokes. *(1 template)*



CLASSIFICATION

• **Weka** offers regression & classifier toolkit

• **Regression**: Predict average rating for joke

• **Classification**: Predicts most common rating (1-5)

• **Random Forest** uses subsets of features. Usable for classification and regregession.

• Decision tree based algorithm finds **most important attributes**: useful for humor theory.

JOKEJUDGER

• Website built for this research to collect **human-generated** and **human-rated** jokes, which steers jokes towards form the system can process

• Website allows users to get **insight in their jokes**

• **5-star** Likert scale & no forced ratings

INCONGRUITY METRICS

• Similar to existing analogy generator metrics [1], but included *Inappropriateness* parameter metric from Ritchie's incongruity-resolution theory.

• Base generator & metrics on **N-grams** like [1]

• **X, Y, Z = noun, noun, adjective**

• Adjective is related to these nouns
=> 2-grams as **relatedness feature**
=> 2-grams to generate **possibility space**

(Based on [1])

CONCLUSION

• Possible to **generalise previous research** on the generation of *"I like my X like I like my Y, Z"* jokes.

• **Generalised system** is **promising**, but still requires:
  • **Sentence** generation
  • **Large corpora** of rated jokes
  • Similar **template detection**

References:

[1] S. Petrovic and D. Matthews. Unsupervised joke generation from big data. In Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), pages 228–232, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

[2] R. Manurung, G. Ritchie, H. Pain, A. Waller, D. Mara, and R. Black. The construction of a pun generator for language skills development. Applied Artificial Intelligence, 22(9):841–869, 2008.

[3] B. A. Hong and E. Ong. Automatically extracting word relationships as templates for pun generation. In Proceedings of the Workshop on Computational Approaches to Linguistic Creativity, CALC '09, pages 24–31, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

Master thesis by: Thomas Winters
Promotor: Prof. Danny De Schreye
Daily supervisor: Vincent Nys

DTAI Declarative Languages & Artificial Intelligence

ASSOCIATIE KU LEUVEN

Master thesis filing card

Student: Thomas Winters

Title: Automatic Joke Generation:
Learning Humour from Examples

Dutch title: Automatisch humor genereren vanuit voorbeelden

UDC: 681.3

Abstract:
This thesis explores how a program can teach itself to generate jokes based on a corpus of rated jokes. We create a framework to achieve this and provide an implementation of this framework focusing on jokes using the "I like my X like I like my Y, Z" template.

Thesis submitted for the degree of Master of Science in Engineering: Computer Science

Thesis supervisor: Prof. dr. Danny De Schreye

Assessor: Prof. dr. Luc De Raedt,
Dr. Stefano Teso

Mentor: Vincent Nys