MEMORIAL UNIVERSITY OF NEWFOUNDLAND
FACULTY OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

Text Summarization

Computer Science 4750

Jacob House Noah Gallant

Friday, November 30th, 2018

Contents

1 Topic and Motivation

Text summarization is the process of reducing an input text to a smaller, more concise version of itself. Motivation to study computerized text summarization — referred to by the term *automatic abstracting* by those in the field — stemmed from curiosity about the mechanisms used in an automatic abstraction PowerShell script found years ago on the Internet [4]. Further research has shown automatic abstraction's usefulness in many interesting fields, including but not limited to the legal and medical professions, scholarly research, and search engine result sorting and summarization.

2 Background Research

Automatic abstracting can be done using various techniques in both strong and weak NLP. Three techniques, nested tree, evolutionary algorithms, and graph-based are discussed in this report, all using strong NLP. These techniques rely on text pre-processing, which prepares the text by converting it to an appropriate format to perform the summarization. Pre-processing can be done in fine detail, or can simply involve splitting the text into sentences. The higher the level of detail applied to the pre-processing, the more likely the algorithm will provide valuable results.

2.1 Nested Tree

The nested tree automated abstracting technique focuses on finding and classifying inter-sentence and inter-word dependencies. The first step to this algorithm is to divide the text into sentences, adding a node to the tree for every sentence. The edge connecting the sentence nodes in the tree represents the dependency between the two sentence nodes. This is shown in Figure 1. Nodes connected by edges are adjacent sentences and the type of dependency between the nodes is what is held in the edge. Words are stored similarly.

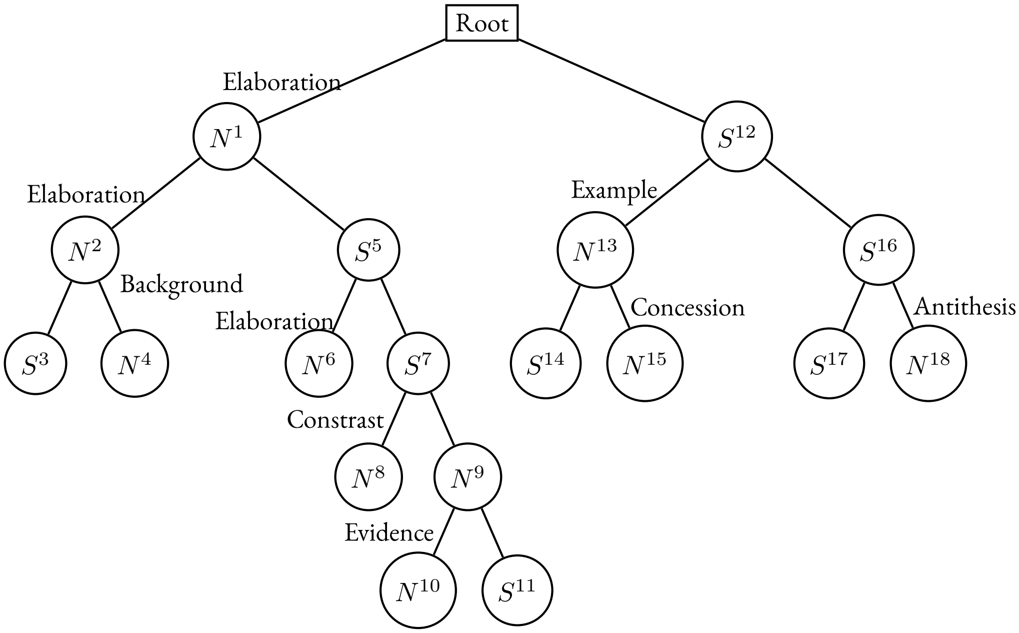

Figure 1: Conceptual Diagram of the Nested Tree Approach

Each sentence is split into words which are each stored in a word node. These nodes are each stored in a tree within each sentence node. In each word tree, the dependency between the words is held in the edge between the two dependent word nodes. Therefore when conceptualizing the tree, the dependency between word nodes forms the meaning of the sentence and the dependency between sentence nodes determines the meaning of the entire text. When forming edges, there are many different types of dependencies, each of which can be classified as either a core attribute or an additive attribute which adds additional meaning or value to a core sentence. To form the summary, we start from the lowest branches in the tree and begin trimming. When performing the tree trimming to obtain our summary, we first trim the sentences whose dependency trees indicate that they add little value to the text as a whole. Core dependencies are the sentence nodes to be trimmed last as they are the core meaning of the sentence and are therefore an important sentence to the summary. This method will form a concise summary by continuously trimming the tree until we reach the desired summary length. It's important to note that this summary method allows us to trim the summary to any length we choose. The disadvantage of this, is that very short summaries may be concise but are likely to also lack information that is important to the meaning of the sentence, resulting in an incomplete and therefore inaccurate summary of the text.

2.2 Evolutionary Algorithm

The evolutionary algorithm approach to automated abstraction is a more complicated approach in comparison to the nested tree method. This method also requires more time and computational power.

Initially, a population of candidate summaries are created from the original text. This population is essentially just a multiset containing not necessarily distinct subsets of set of sentences from the original text. Then, each candidate summary is scored.

Some possible scoring parameters are the sentence's position in a paragraph (the first sentence in a paragraph being awarded one point, the second $\frac{4}{5}$, the third $\frac{3}{5}$, and so on), font style such as capitalized or bold-face type, or the presence of numerical or temporal information in the sentence. Each of these scoring parameters which identify favourable text features would add to a sentence's overall score. Hence sentences with higher scores are regarded as being more significant to the text's overall meaning.

Following this scoring — known as *fitness evaluation* in the genetic algorithm field — candidate summaries are chosen to become parents and produce offspring. Like reproduction in nature, reproduction in an evolutionary algorithm relies on crossover and mutation operators. An example of crossover to maximize child fitness is the case where two candidate summaries S_1 and S_2 each contain a very fit sentence (the fit sentence in S_1 not equal to the sentence in S_2). Then S_1 and S_2 may produce an offspring summary S_3 which contains both fit sentences, and may omit another less fit sentence instead. This summary, which combines the best parts of both parents, should have a higher fitness than either parent.

Then, after offspring have been created, some survivor selection operation is performed to decide which individuals will move on to the next generation. The above steps are repeated to form subsequent generations, the goal being that the maximum fitness per generation increases with each iteration. The algorithm stops when maximum fitness plateaus or a maximum number of generations is met.

This approach seems to have potential to provide much more accurate results than the nested tree method, however this is reliant on effective recombination and mutation operators, as well as parent and survivor selection, and on developing effective fitness scoring metrics. Together all of these parameters make implementing this solution extremely complex.

2.3 Graphs

Graph-based automated abstraction techniques are similar to the tree method. First the text to summarize is broken down into individual sentences. These sentences are then placed into nodes which are initially connected by edges representing the adjacency of sentences in the text. Once all the nodes have been added to the tree, the edges representing other criteria within the sentence are added. These criteria can range from the type of sentence, to shared or similar words. When all the edges have been added to the graph, the graph is now considered full. To get the summary of the text from the graph, we must simply determine the sentences with the most dependencies (*i.e.,* largest number of edges connected to the node). The number of sentence to be chosen to create the summary will dictate the length of the summary. While the graph-based method can give accurate detailed summaries, it is extractive. This meaning that it will not add anything additional to the text to form the summary, but will instead simply extract the important elements in the text and add them to the summary. Applying such an algorithm to summarize text may result in an extremely fragmented summary as we are merely cherrypicking sentences from the text. This does however lend itself to effective creation of summaries in bullet point form rather than paragraph-based summaries that read smoothly and are grammatically correct.

3 Implementation and Program Structure

The implementation chosen consisted primarily of the graph method. Three Python classes were written to implement this approach: the `Node`, the `Edge`, and the `Text` class. These are called by a Python program `Summarizer.py` which has the role of opening the input text file and ensuring the correct format is used before performing the summarization.

3.1 The `Text` Class

The role of the `Text` class is to perform all operations on the text. This includes any pre-processing required, the initial object creation, determination of inter-sentence relationships, as well as using the determined relationship information to form a reasonably accurate and concise summary.

When first initializing the text, the text to be summarized must be passed into the constructor of the `Text` class. This text is then saved as in instance variable for future use, and undergoes pre-processing. Pre-processing is done by the `preProcessing()` method, and is tasked with splitting the text into sentences.

3.1.1 The `preProcessing()` Method

Pre-processing is a crucial step in achieving a high quality summary. Since pre-processing is the first operation performed on the text, the method in which pre-processing is conducted can have a significant bearing on the quality of the summary. The goal of the pre-processing in this application is primarily to split the text into sentences. This is done by considering sentence terminating symbols including '.', '!', and '?'. The algorithm also considers if a sentence contains a quotation, and other situations in which a terminating symbol should be ignored.

The secondary operation of the pre-processing in our application is to replace special Unicode characters with their plain text equivalent, or, in extreme edge cases, to simply remove the character. Many common text editors use Unicode characters as opposed to the plain text characters for many text symbols. Some of these symbols include typographer's quotation marks, accents (*e.g.,* ç, ä, ñ, etc.), en dashes and em dashes, as well as non-Latin characters (*e.g.,* ø, æ, ß, etc.).

Operating on Split Sentences

Once split, the text is processed by creating a `Node` object for each sentence. This node contains the original sentence, a list of all the words in the sentence which have been passed through a lemmatizer [1], a list of edges which are connected to the node, as well as the sentence number to keep track of the location of the sentence in the original text. During the `Text()` constructor which parses the sentence, `Edge` objects are created to connect sentences that are adjacent to each other in the original text. These adjacency edges are the first edges to be created in the graph.

3.1.2 Creating the Dictionary

After all the sentence nodes have been created, and a list of words in each sentence node has been processed, the instance of `Text` will then proceed to loop through each sentence node to create a complete dictionary of all the words found in the text. This dictionary will not only store the available words, but will also store all the sentence nodes in which the word is contained. The goal of creating this dictionary is to decrease the computational complexity in determining the relationships between sentences.

3.1.3 Creating Edges

By creating a dictionary that contains words and the nodes they are contained within, it then suffices to loop through all the words (*i.e.,* dictionary keys) and, when a word is contained within more than one node, to link these nodes with an edge. Each `Node` object will thus contain both proximity `Edges` and `Edges` associated with common words.

This step also offers the opportunity for further improvement to the summary. The more relation edges that are made, using different criteria, the better the summary. Therefore by only counting the number of word relations, we limit the quality of our summary. Some additional criteria to be added to improve the summary include quotation detection, statistics, names, negations, and modifiers. Of course, this means creating a hybrid graph-and-text-element approach in which sentences accumulate weight (*i.e.,* their importance score) from inter-sentence content relationships as well as in-sentence text elements that do not associate one sentence with another.

3.1.4 Creating a Summary

The next step is to create a summary using the nodes and edges created in the previous steps. After this processing is complete, the software prompts the user to input how many sentences the summary should contain and supplies a recommended number of sentences to choose in case the user is not completely aware of the length of the text supplied. This recommendation R is computed using Equation (1), where N is the total number of sentences in the text.

$$R := \begin{cases} \frac{1}{2} \cdot N, & \text{if } N < 10 \\ \\ \frac{1}{3} \cdot N, & \text{if } N \geqslant 10. \end{cases} \tag{1}$$

Then, the R highest-ranking Node objects are sorted according to their associated sentence's position in the original text and the stored sentences are printed in bullet-point format.

3.2 Node Class

The Node class is the application's representation of a sentence. The Node() constructor takes in a sentence as a string, and will immediately split the string into words which are saved individually in a list.

Despite no longer being in the pre-processing stage, the final step in pre-processing is conducted in the constructor of the Node object. Each Node object first accepts a sentence as a string, which is split into individual words. Before storing the individual words in a list, they are lemmatized to improve the quality of the summarization and ensure that the same words with different endings still result in creating a dependency between two sentences. This is performed during the access of each word to reduce the time complexity of the preProcessing() function, and of the overall application.

3.2.1 The parse(sentence) Method

This method takes a string argument called sentence which is the sentence the Node object is representing. parse() then takes the sentence and splits it into individual words. These words are then passed through a lemmatizer using the NLTK library to remove the endings of words, *e.g.,* -ing, -er, -ed, etc. Once passed through the lemmatizer, the lemmatized words are added to a list which is returned by the function.

3.2.2 The `findWords()` Method

The `findWords()` method finds and returns all the words in the sentence as a list. As the words are already parsed and stored in lists by the constructor, this function simply returns that list, and doesn't perform any additional modifications to the `Node` object.

3.2.3 The `addEdge(edge)` Method

Each `Node` object contains a list of both the words in the sentence, and a list of all the edges with which the `Node` is associated. This function adds the passed in `Edge` object to the list of edges held in the `Node`.

3.2.4 The `returnEdgeNum()` Method

The `returnEdge()` method returns the number of edges associated with the current `Node` object.

3.2.5 The `returnEdges()` Method

The `returnEdges()` method returns the list of `Edge`s held in the `Node` object. This includes both the adjacency edges added in the `Text` constructor, as well as edges which represent common characteristics such as sharing the same word.

3.3 Edge Class

The `Edge` class is very simplistic and only contains a constructor and a single object variable. The `Edge` class connects two nodes which share a dependency. For this reason, the single object variable called `node`, simply holds a tuple of the two nodes the `Edge` is connecting. Instances of the `Edge` class are stored in both the `Node` and `Text` classes and are crucial to forming the summary.

References

[1] Steven Bird. *NLTK: The natural language toolkit*. COLING-ACL '06. Sydney, Australia: Association for Computational Linguistics, 2006, pp. 69–72. DOI: `10.3115/1225403.1225421`. URL: `http://dx.doi.org/10.3115/1225403.1225421`.

[2] Noah Albert Gallant. *A Review of Heterogeneous System Architecture Design Techniques*. 2018.

[3] J. K. Rowling. *Harry Potter and the Order of the Phoenix*. 2010.

[4] Prateek Singh. *PSSummary*. 2018. URL: `https://github.com/PrateekKumarSingh/PSSummary`.

Appendix

A Demonstration

A.1 Heterogeneous Computing

One test case was to give the application an except (included below) of a research paper on heterogeneous computing [2].

Heterogeneous Computing Test File

> The usage of the CPU as a generalized process, and the GPU as an accelerator to the CPU has been a common practice just before and continuing after 2010. This configuration is what's considered as general-purpose computing on a GPU. This configuration offers significant performance benefits in a system, particularly in data-parallel, and computationally heavy process. Heterogeneous system architecture is used to leverage the value of each unique device, The CPU being well designed for tasks were latency is critical, while the GPU is the choice processor in throughput-oriented tasks. This creates two different classifications; a latency compute unit (LUT) being a general CPU, and a throughput compute unit being a general GPU. Both processing units are capable of performing the same calculations or tasks, however the different architectures have different strengths and weaknesses. The strengths and weaknesses of each processor boils down to their internal structure. CPUs are composed of a few large, flexible, and fast clocked cores, while GPUs are built using thousands of cores that are both smaller and slower, but are highly parallelized. In addition to performance, energy efficiency is also an important factor while GPUs have considerably greater computational power than CPU's, the have approximately the same energy cost. For example, when comparing the Intel Xeon E7 CPU, 150 watts delivers around 100GFlops/s, while a small increase of energy to 250 Watts in the NVIDIA GK110 provides 1.3 TFlops/s. At the time of writing of this article in 2015, these were both state of the art processors.

For this file we requested that the application return five sentences. The sentences returned were:

1. The usage of the CPU as a generalized process, and the GPU as an accelerator to the CPU has been a common practice just before and continuing after 2010.
2. This configuration offers significant performance benefits in a system, particularly in data-parallel, and computationally heavy process.
3. Heterogeneous system architecture is used to leverage the value of each unique device, The CPU being well designed for tasks were latency is critical, while the GPU is the choice processor in throughput-oriented tasks.
4. This creates two different classifications; a latency compute unit (LUT) being a general CPU, and a throughput compute unit being a general GPU.
5. CPUs are composed of a few large, flexible, and fast clocked cores, while GPUs are built using thousands of cores that are both smaller and slower, but are highly parallelized.

[Reflect on this summary here.]

A.2 Harry Potter

The next test case being demonstrated is the use of the software with an excerpt from the first chapter from J.K. Rowling's *Harry Potter and the Order of the Phoenix* [3].

Excerpt from Chapter 1 from *Harry Potter and the Order of the Phoenix*

The hottest day of the summer so far was drawing to a close and a drowsy silence lay over the large, square houses of Privet Drive. Cars that were usually gleaming stood dusty in their drives and lawns that were once emerald green lay parched and yellowing; the use of hosepipes had been banned due to drought. Deprived of their usual car-washing and lawn-mowing pursuits, the inhabitants of Privet Drive had retreated into the shade of their cool houses, windows thrown wide in the hope of tempting in a nonexistent breeze. The only person left outdoors was a teenage boy who was lying flat on his back in a flower bed outside number four.

He was a skinny, black-haired, bespectacled boy who had the pinched, slightly unhealthy look of someone who has grown a lot in a short space of time. His jeans were torn and dirty, his T-shirt baggy and faded, and the soles of his trainers were peeling away from the uppers. Harry Potter's appearance did not endear him to the neighbors, who were the sort of people who thought scruffiness ought to be punishable by law, but as he had hidden himself behind a large hydrangea bush this evening he was quite invisible to passersby. In fact, the only way he would be spotted was if his Uncle Vernon or Aunt Petunia stuck their heads out of the living room window and looked straight down into the flower bed below.

On the whole, Harry thought he was to be congratulated on his idea of hiding here. He was not, perhaps, very comfortable lying on the hot, hard earth, but on the other hand, nobody was glaring at him, grinding their teeth so loudly that he could not hear the news, or shooting nasty questions at him, as had happened every time he had tried sitting down in the living room and watching television with his aunt and uncle.

Almost as though this thought had fluttered through the open window, Vernon Dursley, Harry's uncle, suddenly spoke. "Glad to see the boy's stopped trying to butt in. Where is he anyway?" "I don't know," said Aunt Petunia unconcernedly. "Not in the house."

Uncle Vernon grunted.

"Watching the news..." he said scathingly. "I'd like to know what he's really up to. As if a normal boy cares what's on the news — Dudley hasn't got a clue what's going on, doubt he knows who the Prime Minister is! Anyway, it's not as if there'd be anything about his lot on our news—"

"Vernon, shh!" said Aunt Petunia. "The window's open!"

"Oh — yes — sorry, dear..."

The Dursleys fell silent. Harry listened to a jingle about Fruit 'N Bran breakfast cereal while he watched Mrs. Figg, a batty, cat-loving old lady from nearby Wisteria Walk, amble slowly past. She was frowning and muttering to herself. Harry was very pleased that he was concealed behind the bush; Mrs. Figg had recently taken to asking him around for tea whenever she met him in the street. She had rounded the corner and vanished from view before Uncle Vernon's voice floated out of the window again.

"Dudders out for tea?"

"At the Polkisses'," said Aunt Petunia fondly. "He's got so many little friends, he's so popular..."

Harry repressed a snort with difficulty. The Dursleys really were astonishingly stupid about their son, Dudley; they had swallowed all his dim-witted lies about having tea with a different member of his gang every night of the summer holidays. Harry knew perfectly well that Dudley had not been to tea anywhere; he and his gang spent every evening vandalizing the play park, smoking on street corners, and throwing stones at passing cars and children. Harry had seen them at it during his evening walks around Little Whinging; he had spent most of the holidays wandering the streets, scavenging newspapers from bins along the way.

The opening notes of the music that heralded the seven o'clock news reached Harry's ears and his stomach turned over. Perhaps tonight — after a month of waiting — would be the night.

"Record numbers of stranded holidaymakers fill airports as the Spanish baggage-handlers' strike reaches its second week —"

"Give 'em a lifelong siesta, I would," snarled Uncle Vernon over the end of the newsreader's sentence, but no matter: Outside in the flower bed, Harry's stomach seemed to unclench. If anything had happened, it would surely have been the first item on the news; death and destruction were more important than stranded holidaymakers.

He let out a long, slow breath and stared up at the brilliant blue sky. Every day this summer had been the same: the tension, the expectation, the temporary relief, and then mounting tension again...and always, growing more insistent all the time, the question of why nothing had happened yet...

He kept listening, just in case there was some small clue, not recognized for what it really was by the Muggles — an unexplained disappearance, perhaps, or some strange accident...but the baggage-handlers' strike was followed by news on the drought in the Southeast ("I hope he's listening next door!"

> bellowed Uncle Vernon, “with his sprinklers on at three in the morning!”); then a helicopter that had almost crashed in a field in Surrey, then a famous actress’s divorce from her famous husband (“as if we’re interested in their sordid affairs,” sniffed Aunt Petunia, who had followed the case obsessively in every magazine she could lay her bony hands on).
>
> Harry closed his eyes against the now blazing evening sky as the newsreader said, “And finally, Bungy the budgie has found a novel way of keeping cool this summer. Bungy, who lives at the Five Feathers in Barnsley, has learned to water-ski! Mary Dorkins went to find out more...”
>
> Harry opened his eyes again. If they had reached water-skiing budgerigars, there was nothing else worth hearing. He rolled cautiously onto his front and raised himself onto his knees and elbows, preparing to crawl out from under the window.
>
> He had moved about two inches when several things happened in very quick succession.
>
> A loud, echoing crack broke the sleepy silence like a gunshot; a cat streaked out from under a parked car and flew out of sight; a shriek, a bellowed oath, and the sound of breaking china came from the Dursleys’ living room, and as though Harry had been waiting for this signal, he jumped to his feet, at the same time pulling from the waistband of his jeans a thin wooden wand as if he were unsheathing a sword. But before he could draw himself up to full height, the top of his head collided with the Dursleys’ open window, and the resultant crash made Aunt Petunia scream even louder.
>
> Harry felt as if his head had been split in two; eyes streaming, he swayed, trying to focus on the street and spot the source of the noise, but he had barely staggered upright again when two large purple hands reached through the open window and closed tightly around his throat.

With this text file as the input, the software recommended a summary of 15 sentences. This was chosen and the output was as follows.

1. Deprived of their usual car-washing and lawn-mowing pursuits, the inhabitants of Privet Drive had retreated into the shade of their cool houses, windows thrown

wide in the hope of tempting in a nonexistent breeze.

2. Harry Potter's appearance did not endear him to the neighbors, who were the sort of people who thought scruffiness ought to be punishable by law, but as he had hidden himself behind a large hydrangea bush this evening he was quite invisible to passersby.

3. In fact, the only way he would be spotted was if his Uncle Vernon or Aunt Petunia stuck their heads out of the living room window and looked straight down into the flower bed below.

4. He was not, perhaps, very comfortable lying on the hot, hard earth, but on the other hand, nobody was glaring at him, grinding their teeth so loudly that he could not hear the news, or shooting nasty questions at him, as had happened every time he had tried sitting down in the living room and watching television with his aunt and uncle.

5. She had rounded the corner and vanished from view before Uncle Vernon's voice floated out of the window again.

6. The Dursleys really were astonishingly stupid about their son, Dudley; they had swallowed all his dim-witted lies about having tea with a different member of his gang every night of the summer holidays.

7. Harry knew perfectly well that Dudley had not been to tea anywhere; he and his gang spent every evening vandalizing the play park, smoking on street corners, and throwing stones at passing cars and children.

8. Harry had seen them at it during his evening walks around Little Whinging; he had spent most of the holidays wandering the streets, scavenging newspapers from bins along the way.

9. The opening notes of the music that heralded the seven o'clock news reached Harry's ears and his stomach turned over.

10. If anything had happened, it would surely have been the first item on the news; death and destruction were more important than stranded holidaymakers.

11. He kept listening, just in case there was some small clue, not recognized for what it really was by the Muggles — an unexplained disappearance, perhaps, or some strange accident...but the baggage-handlers' strike was followed by news on the drought in the Southeast ("I hope he's listening next door!" bellowed Uncle Vernon, "with his sprinklers on at three in the morning!"); then a helicopter that had almost crashed in a field in Surrey, then a famous actress's divorce from her famous husband ("as if we're interested in their sordid affairs," sniffed Aunt Petunia, who had followed the case obsessively in every magazine she could lay her bony hands on).

12. Harry closed his eyes against the now blazing evening sky as the newsreader said, “And finally, Bungy the budgie has found a novel way of keeping cool this summer.

13. He rolled cautiously onto his front and raised himself onto his knees and elbows, preparing to crawl out from under the window.

14. A loud, echoing crack broke the sleepy silence like a gunshot; a cat streaked out from under a parked car and flew out of sight; a shriek, a bellowed oath, and the sound of breaking china came from the Dursleys’ living room, and as though Harry had been waiting for this signal, he jumped to his feet, at the same time pulling from the waistband of his jeans a thin wooden wand as if he were unsheathing a sword.

15. But before he could draw himself up to full height, the top of his head collided with the Dursleys’ open window, and the resultant crash made Aunt Petunia scream even louder.