A Gentle Introduction to Deep Learning for Natural Language Processing

Mihai Surdeanu
Department of Computer Science
University of Arizona

April 10, 2020

Contents

Preface

An obvious question that should immediately pop up when seeing this material is: why another deep learning and natural language processing book? Several excellent ones have been published recently covering both theoretical and practical aspects of deep learning and its application to language processing. However, from my experience in teaching courses on natural language processing, I argue that, despite their excellent quality, most of these books do not target their most likely reader: an expert in a domain other than machine learning and natural language processing whose work relies, at least partially, on the automated analysis of large amounts of data, in particular texts. Such experts include, among others, social scientists, political scientists, biomedical scientists, or computer scientists and computational linguists with limited exposure to machine learning.

Existing deep learning and natural language processing books generally fall in two camps. The first camp focuses on the theoretical foundations of deep learning. This is certainly useful to the readers above (one should understand a tool before using it!). However, these books tend to assume the typical background of a machine learning researcher, which our reader usually does not have. As a consequence, I have repeatedly seen students that match this profile rapidly getting lost in such material, and struggling to keep up. To mitigate this issue, the second type of books that exist today focus on the machine learning practitioner, that is, on how to use deep learning software, with minimal attention payed to the theoretical aspects. I argue that focusing on practical aspects is similarly necessary but not sufficient. Considering that deep learning frameworks and libraries have gotten fairly complex, the chance of misusing them due to theoretical misunderstandings, which I have also commonly seen in my courses, is high.

This book aims to bridge the theoretical and practical aspects of deep learning for natural language processing. I cover the necessary theoretical background, but assuming minimal machine learning background from the reader. The aim is that anybody who took linear algebra and calculus courses be able to follow the theoretical material. To address practical aspects, the book includes pseudo code for the simpler algorithms discussed, and actual Python code for the more complicated architectures discussed. The code should be understandable to anybody who took a Python programing course. After reading this book, I hope that the reader has the necessary foundation to immediately dive into building real-world, practical natural language processing systems, as well as expanding their knowledge by reading research publications on these topics.

Acknowledgments

Your acknowledgments: TODO: Thank you!

Mihai Surdeanu

1 Introduction

Machine learning (ML) has become a pervasive part of our lives. Pedro Domingos, a machine learning faculty at University of Washington, discusses a typical day in the life of a 21st century person in his recent book, showing how she is accompanied by machine learning applications throughout the day, from early in the morning (e.g., waking up to music that the machine matched to her preferences) to late at night (e.g., taking a drug designed by a biomedical researcher with the help of a robot scientist) [Domingos 2015].

Natural language processing (NLP) is an important subfield of ML. As an example of its usefulness, consider that PubMed, a repository of biomedical publications built by the National Institute of Health,[1] has indexed more than one million research publications *per year* since 2010 [Vardakas et al. 2015]. Clearly, no human reader (or team of readers) can process so much material. We need machines to help us manage this vast amount of knowledge. As one example out of many, in our previous work we showed that machine reading discovers an order of magnitude more protein signaling pathways[2] in biomedical literature than exist today in humanly-curated knowledge bases [Valenzuela-Escárcega et al. 2018]. Only 60 to 80% of these automatically-discovered biomedical interactions are correct (a good motivation for *not* letting the machines work alone!). But, without NLP, all of this would remain "undiscovered public knowledge" [Swanson 1986], and would limit our ability to understand important diseases such as cancer. Other important and more common applications of NLP include web search, machine translation, and speech recognition, all of which have had a major impact in almost everybody's life.

Since approximately 2014, the "deep learning tsunami" has hit the field of NLP [Manning 2015], to the point that today a majority of NLP publications use deep learning. For example, the percentage of NLP publications at four top NLP conferences has increased from under 40% in 2012 to 70% in 2017 [Young et al. 2018]. There is good reason for this domination. Deep learning systems are relatively easy to build due to their modularity, and they perform better than many other ML methods.[3] For example, the site nlpprogress.com, which keeps track of state-of-the-art results in many NLP tasks, is dominated by results of deep learning approaches.

[1] https://www.ncbi.nlm.nih.gov/pubmed/

[2] Protein signaling pathways "govern basic activities of cells and coordinate multiple-cell actions". Errors in these pathways "may cause diseases such as cancer". See: https://en.wikipedia.org/wiki/Cell_signaling

[3] However, they are not perfect. See Section 1.3 for a discussion.

This book explains deep learning methods for NLP, aiming to cover both theoretical aspects (e.g., how do neural networks learn?) and practical ones (e.g., how do I build one for language applications?). The goal of the book is to do this while assuming minimal technical background from the reader. The theoretical material in the book should be completely accessible to the reader who took linear algebra and calculus courses, or who is willing to do some independent work to catch up. From linear algebra, the most complicated notion used is matrix multiplication. From calculus, we use differentiation. The code examples should be understandable to the reader who took a Python programming course.

Starting nearly from scratch aims to address the background of what we think will be the typical reader of this book: an expert in a discipline other than ML and NLP, but who needs ML and NLP for her job. There are many examples of such disciplines: the social scientist who needs to mine social media data, the political scientist who needs to process transcripts of political discourse, the business analyst who has to parse company financial reports at scale, the biomedical researcher who needs to extract cell signaling mechanisms from publications, etc. Further, we hope this book will also be useful to computer scientists and computational linguists who need to catch up with the deep learning wave. In general, this book aims to mitigate the impostor syndrome [Dickerson 2019] that affects many of us in this era of rapid change in the field of machine learning and artificial intelligence (this author certainly has suffered and still suffers from it![4]).

1.1 What this Book Covers

Chapter 2 starts hoping to convince the reader that machine learning is easy: we will go over a children's book where key ML concepts and one learning algorithm are introduced. From this example, we will start building several basic neural networks. In the same chapter, we will formalize the Perceptron algorithm, the simplest neural network. In Chapter 3, we will transform the Perceptron into logistic regression, another simple neural network that is surprisingly effective for NLP. In Chapter 4, we will generalize these algorithms into feed forward neural networks, which operate over arbitrary combinations of artificial neurons.

The astute historian of deep learning will have observed that deep learning had an earlier impact in image processing than in NLP. For example, in 2012, researchers at University of Toronto reported a massive improvement in image classification when using deep learning [Krizhevsky et al. 2012]. However, it took more than two years to observe similar improvements in performance in NLP. One explanation for this delay is that image processing starts from a very low-level unit of information, i.e., the pixels in the image, which are then hierarchically assembled into blocks that are more and more semantically meaningful, e.g., lines and circles, then eyes and ears in the case of faces. In contrast, NLP starts from words, which are packed with a lot more semantic information than pixels and, because of that, are

[4] Even the best of us suffer from it. Please see Kevin Knight's description of his personal experience involving tears (not of joy) in the introduction of this tutorial [Knight 2009].

harder to learn from. For example, the word *house* packs a lot of commonsense knowledge, e.g., houses generally have windows and doors and they provide shelter. Although this information is shared with other words, e.g., *building*, a learning algorithm that has seen *house* in its training data will not know how to handle the word *building* in a new text it is exposed to after training. Chapter 5 addresses this limitation. We will discuss methods to transform words into a numerical representation that captures (some) semantic knowledge, using Firth's old observation that "you shall know a word by the company it keeps" [Firth 1957], i.e., learning from the context in which words appear in large collection of texts. Under this representation, similar words such as *house* and *building* will have similar representations, which will improve the learning capability of our neural networks.

We will then take a break from the theoretical discussion. We will discuss best practices when training neural networks in Chapter 6, and how to implement them in a popular deep learning framework that is tailored for NLP, DyNet,[5] in Chapter 7.

Chapter 8 introduces sequence models for processing text. For example, while the word *book* is syntactically ambiguous, i.e., it can be either a noun or a verb, knowing that it is preceded by the determiner *the* in a text gives strong hints that this instance of it is a noun. In this chapter, we will cover neural network architectures designed to model such sequences including recurrent neural networks, convolutional neural networks, long short-term memory networks, and long short-term memory networks combined with conditional random fields. In Chapter 9 we will show how some of these architectures can be implemented in DyNet.

Chapter 10 discusses sequence-to-sequence methods, i.e., methods tailored for NLP tasks where the input is a sequence and the output is another sequence. The most common example of such a task is machine translation, where the input is a sequence of words in one language, and the output is a sequence that captures the translation of the original text in a new language. Chapter 11 introduces transformer networks, a more recent take on sequence-to-sequence methods, which replaces the sequence modeling traditionally used in these approaches with "attention", i.e., a mechanism that computes the representation of a word using a weighted average of the representations of the words in its context. These weights are learned and indicate how much attention each word should put on each of its neighbors (hence the name).

Chapter 12 discusses methods that (begin to) address the "brittleness" of deep learning when transferring a model from one domain to another. For example, the performance of a part-of-speech tagging system, i.e., identifying which words are nouns, verbs, etc., that is trained on well-formed texts such as newspaper articles drops precipitously when used on social media texts (see Section 1.3 for a longer discussion).

Lastly, Chapter 13 discuses approaches for training neural networks with minimal supervision. For example, training a neural network to detect spam emails normally requires many examples of emails that are/are not spam. In this chapter, we introduce a few recent directions

[5] http://dynet.io

in deep learning that allow the training of a network from a few examples that are annotated with the desired outcome (e.g., spam or not spam) and many that are not.

1.2 What this Book Does Not Cover

It is important to note that deep learning is only one of the many subfields of machine learning. In his book, Domingos provides an intuitive organization of these subfields into five "tribes" [Domingos 2015]:

Connectionists This tribe focuses on machine learning methods that (shallowly) mimic the structure of the brain. The methods described in this book fall into this tribe.

Evolutionaries The learning algorithms adopted by this group of approaches, also known as genetic algorithms, focus on the "survival of the fittest". That is, these algorithms "mutate" the "DNA" (or parameters) of the models to be learned, and preserve the generations that perform the best.

Symbolists The symbolists rely on inducing logic rules that explain the data in the task at hand. For example, a part-of-speech tagging system in this camp may learn a rule such as `if` previous word is *the*, `then` the part of the speech of the next word is noun.

Bayesians The Bayesians use probabilistic models such as Bayesian networks. All these methods are driven by Bayes' rule, which describes the probability of an event.

Analogizers The analogizers' methods are motivated by the observation that "you are what you resemble". For example, a new email is classified as spam because it uses content similar to other emails previously classified as such.

It is beyond the goal of this book to explain these other tribes in detail. Even from the connectionist tribe, we will focus mainly on methods that are relevant for language processing.[6] For a more general description of machine learning, the interested reader should look to other sources such as Domingos' book, or Hal Daumé III's excellent Course in Machine Learning.[7]

1.3 Deep Learning Is Not Perfect

While deep learning has pushed the performance of many machine learning applications beyond what we thought possible just ten years ago, it is certainly not perfect. Gary Marcus and Ernest Davis provide a thoughtful criticism of deep learning in their book, Rebooting AI [Marcus and Davis 2019]. Their key arguments are:

Deep learning is opaque While deep learning methods often learn well, it is unclear *what* is learned, i.e., what the connections between the network neurons encode. This is dangerous, as biases and bugs may exist in the models learned, and they may be

[6] Most of methods discussed in this book are certainly useful and commonly used outside of NLP as well.

[7] http://ciml.info

discovered only too late, when these systems are deployed in important real-world applications such as diagnosing medical patients, or self-driving cars.

Deep learning is brittle It has been repeatedly shown both in the machine learning literature and in actual applications that deep learning systems (and for that matter most other machine learning approaches) have difficulty adapting to new scenarios they have not seen during training. For example, self-driving cars that were trained in regular traffic on US highways or large streets do not know how to react to unexpected scenarios such as a firetruck stopped on a highway.[8]

Deep learning has no commonsense An illustrative example for this limitation is that object recognition classifiers based on deep learning tend to confuse objects when they are rotated in three dimensional space, e.g., an over-turned bus in the snow is confused with a snow plow. This happens because deep learning systems lack the commonsense knowledge that some object features are inherent properties of the category itself regardless of the object position, e.g., a school bus in the US has a yellow roof, while some features are just contingent associations, e.g., snow tends to be present around snow plows. (Most) humans naturally use commonsense, which means that we do generalize better to novel instances, especially when they are outliers.

All the issues raised by Marcus and Davis are unsolved today. However, we will discuss some directions that begin to address them in this book. For example, in Chapter 10 we will discuss algorithms that (shallowly) learn commonsense knowledge from large collections of texts. In Chapter 12 we will introduce strategies to mitigate the pain in transferring deep learning models from one domain to another.

1.4 Mathematical Notations

While we try to rely on plain language as much as possible in this book, mathematical formalisms cannot (and should not) be avoided. Where mathematical notations are necessary, we rely on the following conventions:

- We use lower case characters such as x to represent scalar values, which will generally have integer or real values.
- We use bold lower case characters such $\mathbf{x}$ to represent arrays (or vectors) of scalar values, and x_i to indicate the scalar element at position i in this vector. We use $[\mathbf{x};\mathbf{y}]$ to indicate vector concatenation. For example, if $\mathbf{x} = (1,2)$ and $\mathbf{y} = (3,4)$, then $[\mathbf{x};\mathbf{y}] = (1,2,3,4)$.
- We use bold upper case characters such as $\mathbf{X}$ to indicate matrices of scalar values. Similarly, x_{ij} points to the scalar element in the matrix at row i and column j. $\mathbf{x_i}$ indicates the vector corresponding to the entire row i in matrix $\mathbf{X}$.

[8] https://www.teslarati.com/tesla-model-s-firetruck-crash-details/

2 The Perceptron

This chapter covers the Perceptron, the simplest neural network architecture, which will form the building block for the more complicated architectures discusses later in the book. However, rather than starting directly with the discussion of this algorithm, we will start with something simpler: a children's book and some fundamental observations about machine learning. From these, we will formalize our first machine learning algorithm, the Perceptron. In the following chapters, we will improve upon the Perceptron with logistic regression (Chapter 3), and deeper feed forward neural networks (Chapter 4).

2.1 Machine Learning Is Easy

Machine learning is easy. To convince you of this, let us read a children's story [Donaldson and Scheffler 2008]. The story starts with a little monkey that lost her mom in the jungle (Figure 2.1). Luckily, the butterfly offers to help, and collects some information about the mother from the little monkey (Figure 2.2). As a result, the butterfly leads the monkey to an elephant. The monkey explains that her mom is not gray nor big, and does not have a trunk. Instead, her mom has a "tail that coils around trees". Their journey through the jungle continues until, after many mistakes (e.g., snake, spider), the pair end up eventually finding the monkey's mom, and the family is happily reunited.

In addition of the exciting story that kept at least a toddler and this parent glued to its pages, this book introduces several fundamental observations about (machine) learning.

First, *objects are described by their properties*, also known in machine learning terminology as *features*. For example, we know that several features apply to the monkey mom: `isBig`, `hasTail`, `hasColor`, `numberOfLimbs`, etc. These features have values, which may be Boolean (true or false), a discrete value from a fixed set, or a number. For example, the values for the above features are: `false`, `true`, `brown` (out of multiple possible colors), and 4. As we will see soon, it is preferable to convert these values into numbers. For this reason, Boolean features are converted to 0 for false, and 1 for true. Features that take discrete values are converted to Boolean features by enumerating over the possible values in the set. For example, the color feature is converted into a set of Boolean features such as `hasColorBrown` with the value `true` (or 1), `hasColorRed` with the value `false` (or 0), etc.

Second, *objects are assigned a discrete label*, which the learning algorithm or *classifier* (the butterfly has this role in our story) will learn how to assign to new objects. For example, in our story we have two labels: `isMyMom` and `isNotMyMom`. When there are two labels to

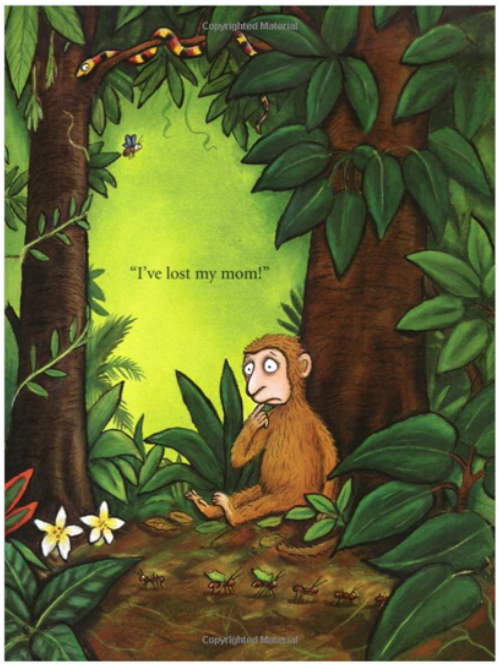

Figure 2.1 An exciting children's book that introduces the fundamentals of machine learning: Where's My Mom, by Julia Donaldson and Axel Scheffler [Donaldson and Scheffler 2008].

be assigned such as in our story, we call the problem at hand a binary classification problem. When there are more than two labels, the problem becomes a multiclass classification task. Sometimes, the labels are continuous numeric values, in which case we have a regression problem. An example of such a regression problem would be learning to forecast the price of a house on the real estate market from its properties, e.g., number of bedrooms, and year it was built. However, in NLP most tasks are classification problems (we will see some simple ones in this chapter, and more complex ones starting with Chapter 8).

To formalize what we know so far, we can organize the examples the classifier has seen (also called a training dataset) into a matrix of features $\mathbf{X}$ and a vector of labels $\mathbf{y}$. Each example seen by the classifier takes a row in $\mathbf{X}$, with each of the features occupying a different column. Table 2.1 shows an example of a possible matrix $\mathbf{X}$ and label vector $\mathbf{y}$ for three animals in our story.

The third observation is that a good learning algorithm *aggregates its decisions over multiple examples with different features*. In our story the butterfly learns that some features are positively associated with the mom (i.e., she is likely to have them), while some are negatively associated with her. For example, from the animals the butterfly sees in the story,

Little monkey: "I've lost my mom!"

"Hush, little monkey, don't you cry. I'll help you find her," said butterfly. "Let's have a think, How big is she?"

"She's big!" said the monkey. "Bigger than me."

"Bigger than you? Then I've seen your mom. Come, little monkey, come, come, come."

"No, no, no! That's an elephant."

Figure 2.2 The butterfly tries to help the little monkey find her mom, but fails initially [Donaldson and Scheffler 2008]. TODO: check fair use!

Table 2.1 An example of a possible feature matrix **X** (left table) and a label vector **y** (right table) for three animals in our story: elephant, snake, and monkey.

isBig	hasTail	hasTrunk	hasColorBrown	numberOfLimbs
1	1	1	0	4
0	1	0	0	0
0	1	0	1	4

Label
isNotMyMom
isNotMyMom
isMyMom

it learns that the mom is likely to have a tail, fur, and four limbs, and she is not big, does not have a trunk, and her color is not gray. We will see soon that this is exactly the intuition behind the simplest neural network, the Perceptron.

Lastly, learning algorithms produce incorrect classifications when not exposed to sufficient data. This situation is called *overfitting*, and it is more formally defined as the situation when an algorithm performs well in training (e.g., once the butterfly sees the snake, it will reliably classify it as not the mom when it sees in the future), but poorly on unseen data (e.g., knowing that the elephant is not the mom did not help much with the classification of the snake). To detect overfitting early, machine learning problems typically divide their data into three partitions: (a) a training partition from which the classifier learns; (b) a development partition that is used for the *internal* validation of the trained classifier, i.e., if it performs poorly on this dataset, the classifier has likely overfit; and (c) a testing partition that is used *only* for the final, formal evaluation. Machine learning developers typically alternate between training (on the training partition) and validating what is being learned (on the development partition) until acceptable performance is observed. Once this is reached, the resulting classifier is evaluated (ideally once) on the testing partition.

2.2 Use Case: Text Classification

In the remaining of this chapter, we will begin to leave the story of the little monkey behind us, and change to a related NLP problem, text classification, in which a classifier is trained to assign a label to a text. This is an important and common NLP task. For example, email providers use binary text classification to classify emails into spam or not. Data mining companies use multiclass classification to detect how customers feel about a product, e.g., like, dislike, or neutral. Search engines use multiclass classification to detect the language a document is written in before processing it.

Throughout this chapter, we will focus on binary text classification for simplicity. We will generalize the algorithms discussed to multiclass classification in Chapter 6. We will introduce more complex NLP tasks starting with Chapter 8.

For now, we will extract simple features from the texts to be classified. That is, we will simply use the frequencies of words in a text as its features. More formally, the matrix $\mathbf{X}$ will have as many columns as words in the vocabulary. Each cell x_{ij} corresponds to the number of times the word at column j occurs in the text stored at row i. For example, the text *This is a great great buy* will produce a feature corresponding to the word *buy* with value 1, one for the word *great* with value 2, etc., while the features corresponding to all the other words in the vocabulary that do not occur in this document receive a value of 0.

2.3 The Perceptron

The Perceptron algorithm was invented by Frank Rosenblatt in 1958. Its aim is to mimic the behavior of a single neuron [Rosenblatt 1958]. Figure 2.3 shows a depiction of a biological neuron,[1] and Rosenblatt's computational simplification, the Perceptron. As the figure shows, the Perceptron is the simplest possible artificial neural network. We will generalize from this single-neuron architecture to networks with an arbitrary number of neurons in Chapter 4.

The Perceptron has one input for each feature of an example $\mathbf{x}$, and produces an output that corresponds to the label predicted for $\mathbf{x}$. Importantly, the Perceptron has a weight vector $\mathbf{w}$, with one weight w_i for each input connection i. Thus, the size of $\mathbf{w}$ is equal to the number of features, or the number of columns in $\mathbf{X}$. Further, the Perceptron also has a bias term, b, that is scalar (we will explain why this is needed later in this section). The Perceptron outputs a binary decision, let us say Yes or No (e.g., Yes, the text encoded in $\mathbf{x}$ contains a positive review for a product, or No, the review is negative), based on the decision function described in Algorithm 1. The $\mathbf{w} \cdot \mathbf{x}$ component of the decision function is called the *dot product* of the vectors $\mathbf{w}$ and $\mathbf{x}$. The dot product of two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=0}^{n} x_i y_i \tag{2.1}$$

[1] By BruceBlaus – Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=28761830

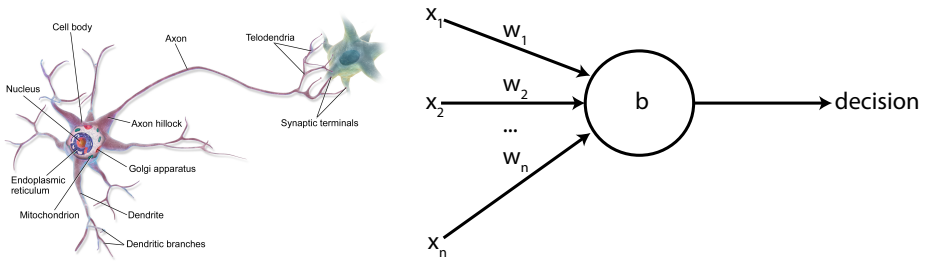

Figure 2.3 A depiction of a biological neuron, which captures input stimuli through its dendrites and produces an activation along its axon and synaptic terminals (left), and its computational simplification, the Perceptron (right).

Algorithm 1: The decision function of the Perceptron.

```
if w · x + b > 0 then
    return Yes
else
    return No
end
```

where n indicates the size of the two vectors. Intuitively, in the case of the Perceptron, the dot product of $\mathbf{x}$ and $\mathbf{w}$ is the weighted sum of the feature values in $\mathbf{x}$, where each feature i is weighted by w_i. If this sum (offset by the bias term b, which we will discuss later) is positive, then the decision is Yes. If it is negative, the decision is No.

Sidebar 2.1 The dot product in linear algebra

In linear algebra, the dot product of two vectors $\mathbf{x}$ and $\mathbf{y}$ is equivalent to $\mathbf{x}^T\mathbf{y}$, where T is the transpose operation. However, in this book we rely on the dot product notation for simplicity.

Sidebar 2.2 The sign function in the Perceptron

The decision function listed in Algorithm 1 is often shown as $\text{sign}(\mathbf{w}\cdot\mathbf{x}+b)$, where the $+$ sign is used to represent one class, and the $-$ sign the other.

There is an immediate parallel between this decision function and the story of the little monkey. If we consider the Yes class to be `isMyMom`, then we would like the weights of

Algorithm 2: Perceptron learning algorithm.

```
1  w = 0
2  b = 0
3  while not converged do
4      for all training examples x_i ∈ X do
5          d = decision(x_i, w, b)
6          if d == y_i then
7              continue
8          else if y_i == Yes and d == No then
9              b = b + 1
10             w = w + x_i
11         else if y_i == No and d == Yes then
12             b = b − 1
13             w = w − x_i
14     end
15 end
```

the features that belong to the mom (e.g., `hasColorBrown`) to have positive values, so the dot product between $\mathbf{w}$ and the $\mathbf{x}$ vector corresponding to the mom turns out positive, and the features specific to other animals (e.g., `hasTrunk`) to receive negative weights, so the corresponding decision is negative. Similarly, if the task to be learned is review classification, we would like positive words (e.g., *good*, *great*) to have positive weights in $\mathbf{w}$, and negative words (e.g., *bad*, *horrible*) to have negative weights.

In general, we call the aggregation of the parameters of a learning algorithm or classifier ($\mathbf{w}$ and b for the Perceptron) a *model*. All classifiers aim to learn these parameters to optimize their predictions over the examples in the training dataset.

The key contribution of the Perceptron is a simple algorithm that learns these weights (and bias term) from the given training dataset. This algorithm is summarized in Algorithm 2. Let us dissect this algorithm next. The algorithm starts by initializing the weights and bias term with 0s. Lines 3 and 4 indicate that the learning algorithm may traverse the training dataset more than once. As we will see in the following example, sometimes this repeated exposure to training examples is necessary to learn meaningful weights. Lines 6 and 7 indicate that the Perceptron simply skips over training examples that it already knows how to classify, i.e., its decision d is equal to the correct label y_i. This is intuitive: if the Perceptron has already learned how to classify an example, there is limited benefit in learning it again. In fact, the opposite might happen: the Perceptron weights may become too tailored for the particular examples

seen in the training dataset, which will cause it to overfit. Lines 8 – 10 address the situation when the correct label of the current training example $\mathbf{x_i}$ is Yes, but the prediction according to the current weights and bias is No. In this situation, we would intuitively want the weights and bias to have higher values such that the overall dot product is more likely to be positive. To move towards this goal, the Perceptron simply *adds* the feature values in $\mathbf{x}$ to the weight vector $\mathbf{w}$, and adds 1 to the bias. Similarly, when the Perceptron makes an incorrect prediction for the label No (lines 11 – 13), it decreases the value of the weights and bias by *subtracting* $\mathbf{x}$ from $\mathbf{w}$, and subtracting 1 from b.

Sidebar 2.3 Error driven learning

The class of algorithms such as the Perceptron that focus on "hard" examples in training, i.e., examples for which they make incorrect predictions at a given point in time, are said to perform *error driven learning*.

Figure 2.4 shows an intuitive visualization of this learning process.[2] In this figure, for simplicity, we are ignoring the bias term and assume that the Perceptron decision is driven solely by the dot product $\mathbf{x} \cdot \mathbf{w}$. Figure 2.4 (a) shows the weight vector $\mathbf{w}$ in a simple two-dimensional space, which would correspond to a problem that is represented using only two features.[3] In addition of $\mathbf{w}$, the figure also shows the *decision boundary* of the Perceptron as a dashed line that is perpendicular on $\mathbf{w}$. The figure indicates that all the vectors that lie on the same side of the decision boundary with $\mathbf{w}$ are assigned the label Yes, and all the vectors on the other side receive the decision No. Why is this? To answer this question, we need to introduce a new formula that measures the cosine of the angle between two vectors, *cos*:

$$cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}||||\mathbf{y}||} \tag{2.2}$$

where $||\mathbf{x}||$ indicates the length of vector $\mathbf{x}$, i.e., the distance between the origin and the tip of the vector's arrow, measured with a generalization of Pythagora's theorem: $||\mathbf{x}|| = \sqrt{\sum_{i=1}^{N} x_i^2}$. The cosine similarity, which ranges between -1 and 1, is widely used in the field of information retrieval to measure the similarity of two vectors [Schütze et al. 2008]. That is, two perfectly similar vectors will have an angle of 0° between them, which has the largest possible cosine value of 1. Two "opposite" vectors have an angle of 180° between them, which has a cosine of -1. We will extensively use the cosine similarity formula starting with the next chapter. But, for now, we will simply observe that the cosine similarity value has the

[2] This visualization was first introduced by Schütze et al. [2008].

[3] This simplification is useful for visualization, but it is highly unrealistic for real-world NLP applications, where the number of features is often proportional with the size of a language's vocabulary, i.e., it is often in the hundreds of thousands.

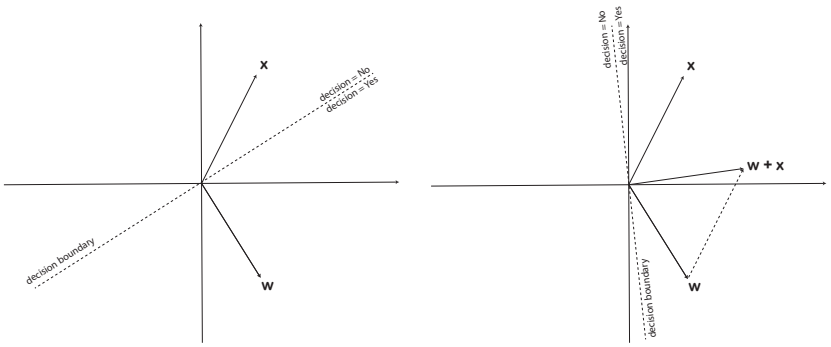

Figure 2.4 Visualization of the Perceptron learning algorithm: (a) incorrect classification of the vector **x** with the label Yes, for a given weight vector **w**; and (b) **x** lies on the correct side of the decision boundary after **x** is added to **w**.

same sign with the dot product of the two vectors (because the length of a vector is always positive). Because vectors on the same side of the decision boundary with **w** have an angle with **w** in the interval $[-90°, 90°]$, the corresponding cosine (and, thus, dot product value) will be positive, which yields a Yes decision. Similarly, vectors on the other side of the decision boundary will receive a No decision.

Sidebar 2.4 Hyper planes and Perceptron convergence

In a one-dimensional feature space, the decision boundary for the Perceptron is a dot. As shown in Figure 2.4, in a two-dimensional space, the decision boundary is a line. In a three-dimensional space, the decision boundary is a plane. In general, we call the decision boundary of the Perceptron a *hyper plane*. If such a hyper plane that separates the labels of the examples in the training dataset exists, it is guaranteed that the Perceptron will find it, or will find another hyper plane with similar separating properties [Block 1962, Novikoff 1963]. We say that the learning algorithm has *converged* when such a hyper plane is found, which means that all examples in the training data are correctly classified.

Figure 2.4 (a) shows that, at that point in time, the training example **x** with label Yes lies on the incorrect side of the decision boundary. Figure 2.4 shows how the decision boundary is adjusted after **x** is added to **w** (line 10 in Algorithm 2). After this adjustment, **x** is on the correct side of the decision boundary.

To convince ourselves that the Perceptron is indeed learning a meaningful decision boundary, let us go trace the learning algorithm on a slightly more realistic example. Table 2.2 shows

Table 2.2 The feature matrix **X** (left table) and label vector **y** (right table) for a review classification training dataset with three examples.

#	*good*	*excellent*	*bad*	*horrible*	*boring*
#1	1	1	1	0	0
#2	0	0	1	1	0
#3	0	0	1	0	1

Label
Positive
Negative
Negative

Table 2.3 The Perceptron learning process for the dataset shown in Table 2.2, for one pass over the training data. Both **w** and b are initialized with 0s.

Example seen: #1
$\mathbf{x} \cdot \mathbf{w} + b = 0$
Decision = Negative
Update (add): $\mathbf{w} = (1,1,1,0,0)$, $b = 1$
Example seen: #2
$\mathbf{x} \cdot \mathbf{w} + b = 2$
Decision = Positive
Update (subtract): $\mathbf{w} = (1,1,0,-1,0)$, $b = 0$
Example seen: #3
$\mathbf{x} \cdot \mathbf{w} + b = 0$
Decision = Negative
Update: none

the matrix **X** and label vector **y** for a training dataset that contains three examples for a product review classification task. In this example, we assume that our vocabulary has only the five words shown in **X**. For example, the first example in this dataset is a positive review that contains the words *good*, *excellent*, and *bad*.

Table 2.3 traces the learning algorithm as it iterates through the training examples. For example, because the decision function produces the incorrect decision for the first example (No), this example is added to **w**. Similarly, the second example is subtracted from **w**. The third example is correctly classified (barely), so no update is necessary. After just one pass over this training dataset, also called an *epoch*, the Perceptron has converged. We will let the reader convince herself that all training examples are now correctly classified. The final weights indicate that the Perceptron has learned several useful things. First, it learned that *good* and *excellent* are associated with the Yes class, and has assigned positive weights to them. Second, it learned that *bad* is not to be trusted because it appears in both positive and

Table 2.4 The feature matrix $\mathbf{X}$ (left table) and label vector $\mathbf{y}$ (right table) for a review classification training dataset with four examples. In this example, the only feature available is the *total* number of positive words in a review.

#	Number of positive words
#1	1
#2	10
#3	2
#4	20

Label
Negative
Positive
Negative
Positive

negative reviews, and, thus, it assigned it a weight of 0. Lastly, it learned to assign a negative weight to *horrible*. However, it is not perfect: it did not assign a non-zero weight to *boring* because of the barely correct prediction made on example #3. There are other bigger problems here. We discuss them in Section 2.6.

This example as well as Figure 2.4 seem to suggest that the Perceptron learns just fine without a bias term. So why do we need it? To convince ourselves that the bias term is useful let us walk through another simple example, shown in Table 2.4. The Perceptron needs four epochs, i.e., four passes over this training dataset, to converge. The final parameters are: $\mathbf{w} = \langle 2 \rangle$ and $b = -5$. We encourage the reader to trace the learning algorithm through this dataset on her own as well. This parameters indicate that the hyper plane for this Perceptron, which is a dot in this one-dimensional feature space, is at 2.5. That is, in order to receive a Yes decision, the feature of the corresponding example must have a value ≥ 2.5, i.e., the review must have at least three positive words. This is intuitive, as the training dataset contains negative reviews that contain one or two positive words. What this shows is that the bias term allows the Perceptron to shift its decision boundary *away* from the origin. It is easy to see that, without a bias term, the Perceptron would not be able to learn anything meaningful, as the decision boundary will always be in the origin. In practice, the bias term tends to be more useful for problems that are modeled with few features. In real-world NLP tasks that are high-dimensional, learning algorithms usually find good decision boundaries even without a bias term (because there are many more options to choose from).

Sidebar 2.5 Implementations of the bias term

Some machine learning software packages implement the bias term as an additional feature in $\mathbf{x}$ that is always active, i.e., it has a value of 1 for all examples in $\mathbf{X}$. This simplifies the math a bit, i.e., instead of computing $\mathbf{x} \cdot \mathbf{w} + b$, we now have to compute just $\mathbf{x} \cdot \mathbf{w}$. It is easy to see that modeling the bias as an always-active feature has the same functionality as the explicit bias term in Algorithm 2. In this book, we will maintain an explicit bias term for clarity.

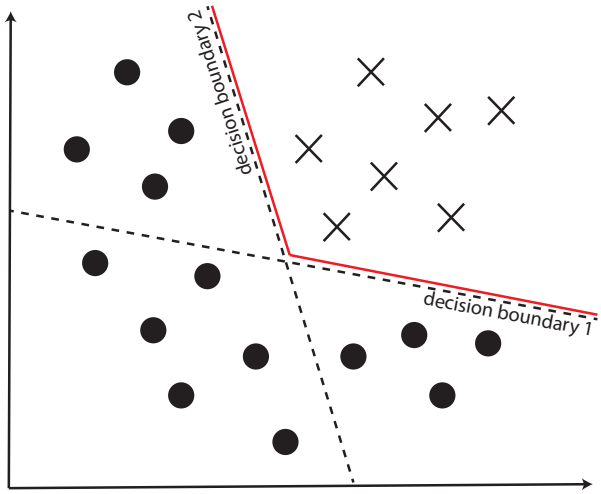

Figure 2.5 An example of a binary classification task, and a voting Perceptron that aggregates two imperfect Perceptrons. The voting algorithm classifies correctly all the data points by requiring two votes for the $\times$ class to yield a $\times$ decision. The decision boundary of the voting Perceptron is shown in red.

2.4 Voting Perceptron

As we saw in the previous examples, the Perceptron learns well, but it is not perfect. Often, a very simple strategy to improve the quality of classifier is to use an *ensemble model*. One such ensemble strategy is to *vote* between the decisions of multiple learning algorithms. For example, Figure 2.5 shows a visualization of such a voting Perceptron, which aggregates two individual Perceptrons by requiring that both classifiers label an example as $\times$ before issuing the $\times$ label.[4]

The figure highlights two important facts. First, the voting Perceptron performs better than either individual classifier. In general, ensemble models perform better than the individual (or base) classifiers that are part of the ensemble [Dietterich 2000]. Second, and more important for our discussion, the voting Perceptron is a *non-linear classifier*, i.e., its decision boundary is no longer a line (or a hyper plane in n dimensions). In figure 2.5, the decision boundary for the voting Perceptron is shown with the red lines.

While the voting approach is an easy way to produce a non-linear classifier that improves over the basic Perceptron, it has drawbacks. First, we need to produce several individual Perceptron classifiers first. This can achieved in at least two distinct ways. For example, instead of initializing the $\mathbf{w}$ and b parameters with 0s (lines 1 and 2 in Algorithm 2), we

[4] This example taken from Erwin Chan's Ling 539 course at University of Arizona.

initialize them with random numbers (typically small numbers centered around 0). For every different set of initial values in **w** and b, the resulting Perceptron will end up with a different decision boundary, and, thus, a different classifier. The drawback of this strategy is that the training procedure must be repeated for each individual Perceptron. A second strategy for producing multiple individual Perceptron that avoids this training overhead is to keep track of all **w**s and bs that are produced during the training of a single Perceptron. That is, before changing the b and **w** parameters in Algorithm 2 (lines 9 and 12), we store the current values (before the change) in a list. This means that at the end of training procedure, this list will contain as many individual Perceptrons as the number of updates performed in training. We can even sort these individual classifiers by their perceived quality: the more iterations a specific b and **w** combination "survived" in training, the better the quality of this classifier is likely to be. This indicator of quality can be used to assign weights to the "votes" given to the individual classifiers, or to filter out base models of low quality (e.g., remove all classifiers that survived fewer than 10 training examples).

The second drawback of the voting Perceptron is its runtime overhead at evaluation time. When the voting Perceptron is applied on a new, unseen example, it must apply all its individual classifiers before voting. Thus, the voting Perceptron is N times slower than the individual Perceptron, where N is the number of individual classifiers used. To mitigate this drawback, we will need the average Perceptron, discussed next.

2.5 Average Perceptron

The average Perceptron is a simplification of the latter voting Perceptron discussed previously. The simplification consists in that, instead of keeping track of *all* **w** and b parameters created during the Perceptron updates like the voting algorithm, these parameters are averaged into a *single* model, say **avgW** and *avgB*. This algorithm is summarized in Algorithm 3. This algorithm has a constant runtime overhead for computing the average model, i.e., the only additional overhead compared to the regular Perceptron are the additions in lines 12 – 14 and 18 – 20, and the divisions in lines 25 and 26. Further, the additional memory overhead is also constant, as it maintains a single extra vector (**avgW**) and a single bias term (*avgB*). After training, the average Perceptron uses a decision function different from the one used during training. This function has a similar shape to the one listed in Algorithm 1, but uses **avgW** and *avgB* instead.

Despite its simplicity, the average Perceptron tends to perform well in practice, usually outperforming the regular Perceptron, and approaching the performance of the voting Perceptron. But why is the performance of the average Perceptron so good? After all, it remains a linear classifier just like the regular Perceptron, so it must have the same limitations. The intuitive explanation for its good performance is the following. When the Perceptron is exposed to unreliable features during training, these features will receive weight values in the intermediate Perceptron model (the **w** vector) that are all over the place, sometimes positive

Algorithm 3: Average Perceptron learning algorithm.

```
w = 0
b = 0
numberOfUpdates = 0
avgW = 0
avgB = 0
while not converged do
    for all training examples x_i ∈ X do
        d = decision(x_i, w, b)
        if d == y_i then
            continue
        else if y_i == Yes and d == No then
            numberOfUpdates = numberOfUpdates + 1
            avgB = avgB + b
            avgW = avgW + w
            b = b + 1
            w = w + x_i
        else if y_i == No and d == Yes then
            numberOfUpdates = numberOfUpdates + 1
            avgB = avgB + b
            avgW = avgW + w
            b = b − 1
            w = w − x_i
    end
end
avgB = avgB/numberOfUpdates
avgW = avgW/numberOfUpdates
```

and sometimes negative. All these values are averaged in the average vector, and, thus, the average weight value for these unreliable features will tend to be squished to zero. The effect of this squishing to zero is that the decision function of the average Perceptron will tend to not rely on these features (because their contribution to the dot product in the decision function will be minimal). This differs from the regular Perceptron, which does not benefit from this averaging process that reduces the weights of unimportant features. In general, this process of squishing to zero the weights of features that are not important is called *regularization*. We

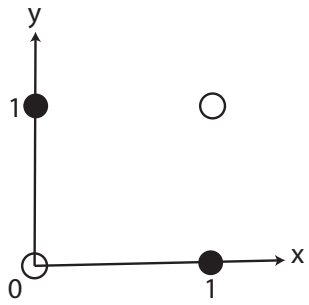

Figure 2.6 Visualization of the XOR function operating over two variables, x and y. The dark circles indicate that the XOR output is 1; the clear circles stand for 0.

will see other regularization strategies starting with the logistic regression classifier, in the next chapter.

2.6 Drawbacks of the Perceptron

The Perceptron algorithm and its variants are simple, easy to customize for other tasks beyond text classification, and they perform fairly well (especially in the voting and average form). However, they also have important drawbacks. We discuss these drawbacks here, and we will spend a good part of this book discussing solutions that address them.

The first obvious limitation of the Perceptron is that, as discussed in this chapter, it is a linear classifier. Yes, the voting Perceptron removes this constraint, but it comes at the cost of maintaining multiple individual Perceptrons. Ideally, we would like to have the ability to learn a single classifier that captures a non-linear decision boundary. This ability is important, as many tasks require such a decision boundary. A simple example of such a task was discussed by Minsky and Papert as early as 1969: the Perceptron cannot learn the XOR function [Minsky and Papert 1969]. To remind ourselves, the XOR function takes two binary variables, i.e., numbers that can take only one of two values: 0 (which stands for False) or 1 (or True), and outputs 1 when exactly one of these values is 1, and 0 otherwise. A visualization of the XOR is shown in Figure 2.6. It is immediately obvious that there is no linear decision boundary that separates the dark circles from the clear ones. More importantly in our context, language is beautiful, complex, and ambiguous, which means that, usually, we cannot model tasks that are driven by language using linear classifiers. We will address this important limitation in Chapter 4, where we will introduce neural networks that can learn non-linear decision boundaries by combining multiple layers of "neurons" into a single network.

A second more subtle but very important limitation of the Perceptron is that it has no "smooth" updates during training, i.e., its updates are the same regardless of how incorrect

the current model is. This is caused by the decision function of the Perceptron (Algorithm 1), which relies solely on the *sign* of the dot product. That is, it does not matter how large (or small) the value of the dot product is; when the sign is incorrect, the update is the same: adding or subtracting the *entire* example $\mathbf{x_i}$ from the current weight vector (lines 10 and 13 in Algorithm 2. This causes the Perceptron to be a slow learner because it jumps around good solutions. One University of Arizona student called this situation "Tony Hawk-ing the data". On data that is linearly separable, the Perceptron will eventually converge [Novikoff 1963]. However, real-world datasets do not come with this guarantee of linear separation, which means that this "Tony Hawk-ing" situation may yield a Perceptron that is far from acceptable. What we would like to have is a classifier that updates its model *proportionally* with the errors it makes: a small mistake causes a small update, while a large one yields a large update. This is exactly what the logistic regression does. We detail this in the next chapter.

The third drawback of the Perceptron is that it relies on explicit features that must be designed and implemented by the machine learning developer. For example, in the text classification use case introduced in Section 2.2, we mentioned that we rely on features that are simply the words in each text to be classified. Unfortunately, in real-world NLP applications feature design gets complicated very quickly. For example, if the task to be learned is review classification, we should probably capture negation. Certainly the phrase *great* should be modeled differently than *not great*. Further, maybe we should investigate the syntactic structure of the text to be classified. For example, reviews typically contain multiple clauses, whose sentiment must be composed into an overall classification for the entire review. For example, the review *The wait was long, but the food was fantastic.* contains two clauses: *The wait was long* and *but the food was fantastic*, each one capturing a different sentiment, which must be assembled into an overall sentiment towards the corresponding restaurant. Further, most words in any language tend to be very infrequent [Zipf 1932], which means that a lot of the hard work we might invest in feature design might not generalize enough. That is, suppose that the reviews included in a review classification training dataset contain the word *great* but not the word *fantastic*, a fairy similar word in this context. Then, any ML algorithm that uses features that rely on explicit words will correctly learn how to associate *great* with a specific sentiment, but will not know what to do when they see the word *fantastic*. Chapter 5 addresses this limitation. We will discuss methods to transform words into a numerical representation that captures (some) semantic knowledge. Under this representation, similar words such as *great* and *fantastic* will have similar representations, which will improve the generalization capability of our ML algorithms.

Lastly, in this chapter we focused on text classification applications such as review classification that require a simple ML classifier, which produces a single label for an input text, e.g., positive vs. negative review. However, many NLP applications require the extraction of *structured* output. For example, a part-of-speech tagger, which identifies which words are nouns, verbs, etc., must produce the *sequence* of part of speech tags for a given sentence. Similarly, a

syntactic parser identifies syntactic structures in a given sentence such as which phrase serves as subject for which verb, which are typically represented in a *tree* structure. The type of ML algorithms that produce structures rather than individual labels are said to perform *structured learning*. We will begin discussing structured learning in Chapter 8.

2.7 References and Further Readings

TODO: To do

3 Logistic Regression

As mentioned in the previous chapter, the Perceptron does not perform smooth updates during training, which may cause it to "Tony Hawk the data", which, in turn, may slow down learning, or cause it to miss good solutions entirely in real-world situations. In this chapter, we will discuss logistic regression (LR), a machine learning algorithm that elegantly addresses this problem.

3.1 The Logistic Regression Decision Function and Learning Algorithm

As we discussed, the lack of smooth updates in the training of the Perceptron is caused by its reliance on a discrete decision function driven by the sign of the dot product. The first thing LR does is replace this decision function with a new, *continuous* function, which is:

$$\text{decision}(\mathbf{x}, \mathbf{w}, b) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}} \tag{3.1}$$

The $\frac{1}{1+e^{-x}}$ function is known as the logistic function, hence the name of the algorithm. The logistic function belongs to a larger class of functions called sigmoid functions because they are characterized by an S-shaped curve. Figure 3.1 shows the curve of the logistic function. In practice, the name sigmoid (or σ) is often used instead of logistic, which is why the LR decision function is often summarized as: $\sigma(\mathbf{w} \cdot \mathbf{x} + b)$. For brevity, we will use the σ notation in our formulas as well.

Figure 3.1 shows that the logistic function has values that monotonically increase from 0 to 1. We will use this property to implement a better learning algorithm, which has "soft" updates that are proportional with how incorrect the current model is. To do this, we first arbitrarily associate one of the labels to be learned with the value 1, and the other with 0. For example, for the review classification task, we (arbitrarily) map the positive label to 1, and the negative label to 0. Intuitively, we would like to learn a decision function that produces values close to 1 for the positive label, and values close to 0 for the negative one. The difference between the value produced by the decision function and the gold value for a training example will quantify the algorithm's confusion at a given stage in the learning process.

Algorithm 4 lists the LR learning process that captures the above intuitions. We will discuss later in this chapter how this algorithm was derived. For now, let us make sure that this algorithm does indeed do what we promised.

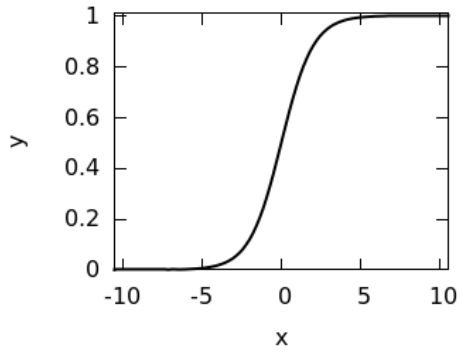

Figure 3.1 The sigmoid curve.

Algorithm 4: Logistic regression learning algorithm.

1 $\mathbf{w} = 0$
2 $b = 0$
3 **while** *not converged* **do**
4 **for** *all training examples* $\mathbf{x_i} \in \mathbf{X}$ **do**
5 $d = \text{decision}(\mathbf{x_i}, \mathbf{w}, b)$
6 $\mathbf{w} = \mathbf{w} + \alpha(y_i - d)\mathbf{x_i}$
7 $b = b + \alpha(y_i - d)$
8 **end**
9 **end**

Note that the only new variable in this algorithm is α, known as the learning rate. The learning rate takes a positive value that scales up or down the values used during the update. We will revisit this idea later on in this chapter. For now, let us assume $\alpha = 1$.

It is easy to see that, at the extreme (i.e., when the prediction is perfectly correct or incorrect), this algorithm reduces to the Perceptron learning algorithm. For example, when the prediction is perfectly correct (say $y_i = 1$ for the class associated with 1), y_i is equal to d, which means that there is no weight and bias update in lines 6 and 7. This is similar to the Perceptron (lines 6 and 7 in Algorithm 2). Further, when a prediction is perfectly incorrect, say, $y_i = 1$ (Yes) when $d = 0$ (No), this reduces to adding $\mathbf{x_i}$ to $\mathbf{w}$ and 1 to b (similar to the Perceptron update, lines 8 – 10 in Algorithm 2). When $y_i = 0$ when $d = 1$, the algorithm reduces to subtracting $\mathbf{x_i}$ from $\mathbf{w}$ and 1 from b (similar to lines 11 – 13 in Algorithm 2).

The interesting behavior occurs in the majority of the situations when the LR decision is neither perfectly correct nor perfectly incorrect. In these situations, the LR performs a soft update that is proportional with how incorrect the current decision is, which is captured by $y_i - d$. That is, the more incorrect the decision is, the larger the update. This is exactly what we would like a good learning algorithm to do.

Once the algorithm finishes training, we would like to use the learned weights ($\mathbf{w}$ and b) to perform binary classification, e.g., classify a text into a positive or negative review. For this, at prediction time we will convert the LR decision into a discrete output using a threshold τ, commonly set to 0.5.[1] That is, if decision$(\mathbf{x}, \mathbf{w}, b) \geq 0.5$ than the algorithm outputs one class (say, positive review); otherwise it outputs the other class.

3.2 The Logistic Regression Cost Function

The next three sections of this chapter focus on deriving the LR learning algorithm shown in Algorithm 4. The reader who is averse to math, or is satisfied with the learning algorithm and the intuition behind it, may skip to Section 3.5. However, we encourage the reader to try to stay with us through this derivation. These sections introduce important concepts, i.e., cost functions and gradient descent, which are necessary to thoroughly understand the following chapters in this book. We will provide pointers to additional reading, where more mathematical background may be needed.

The first observation that will help us formalize the training process for LR is that the LR decision function implements a conditional probability, i.e., the probability of generating a specific label given a training example and the current weights. More formally, we can write:

$$p(y = 1|\mathbf{x}; \mathbf{w}, b) = \sigma(\mathbf{x}; \mathbf{w}, b) \tag{3.2}$$

The left term of the above equation can be read as the probability of generating a label y equal to 1, given a training example $\mathbf{x}$ and model weights $\mathbf{w}$ and b (the vertical bar "|" in the conditional probability formula should be read as "given"). Intuitively, this probability is an indicator of confidence (the higher the better). That is, the probability approaches 1 when the model is confident that the label for $\mathbf{x}$ is 1, and 0 when not. Similarly, the probability of y being 0 is:

$$p(y = 0|\mathbf{x}; \mathbf{w}, b) = 1 - \sigma(\mathbf{x}; \mathbf{w}, b) \tag{3.3}$$

These probabilities form a probability distribution, i.e., the sum of probabilities over all possible labels equals 1. Note that while we aim to minimize the use of probability theory

[1] Other values for this threshold are possible. For example, for applications where it is important to be conservative with predictions for class 1, τ should would take values larger than 0.5.

in this section, some of it is unavoidable. The reader who wants to brush up on probability theory may consult other material on this topic such as [Griffiths 2008].

To simplify notations, because in many cases it is obvious what the model weights are, we will skip them and use simply $p(y = 1|\mathbf{x})$ and $p(y = 0|\mathbf{x})$. Further, we generalize the above two formulas to work for any of the two possible labels with the following formula:

$$p(y|\mathbf{x}) = (\sigma(\mathbf{x};\mathbf{w},b))^y(1-\sigma(\mathbf{x};\mathbf{w},b))^{1-y} \tag{3.4}$$

It is trivial to verify that this formula reduces to one of the two equations above, for $y = 1$ and $y = 0$.

Intuitively, we would like the LR training process to maximize the probability of the correct labels in the entire training dataset. This probability is called the *likelihood of the data* (L), and is formalized as:

$$L(\mathbf{w},b) = p(\mathbf{y}|\mathbf{X}) \tag{3.5}$$

$$= \Pi_{i=1}^{m} p(y_i|\mathbf{x_i}) \tag{3.6}$$

where $\mathbf{y}$ is the vector containing all the correct labels for all training examples, $\mathbf{X}$ is the matrix that contains the vectors of features for all training examples, and m is the total number of examples in the training dataset. Note that the derivation into the product of individual probabilities is possible because we assume that the training examples are independent of each other, and the joint probability of multiple independent events is equal to the product of individual probabilities [Griffiths 2008].

A common convention in machine learning is that instead of maximizing a function during learning, we instead aim to minimize a *cost* or *loss* function C, which captures the amount of errors in the model. By definition, C must take only positive values. That is, C will have large values when the model does not perform well, and is 0 when the learned model is perfect. Using L, we write the LR cost function as:

$$C(\mathbf{w},b) = -\log L(\mathbf{w},b) \tag{3.7}$$

$$= -\sum_{i=1}^{m}(y_i \log \sigma(\mathbf{x_i};\mathbf{w},b) + (1-y_i)\log(1-\sigma(\mathbf{x_i};\mathbf{w},b))) \tag{3.8}$$

It is easy to see that C satisfies the constraints of a cost function. First, it is always positive: the logarithm of a number between 0 and 1 is negative; the negative sign in front of the sum turns the value of the sum into a positive number. Second, the cost function takes large values when the model makes many mistakes (i.e., the likelihood of the data is small), and approaches 0 when the model is correct (i.e., the likelihood approaches 1).

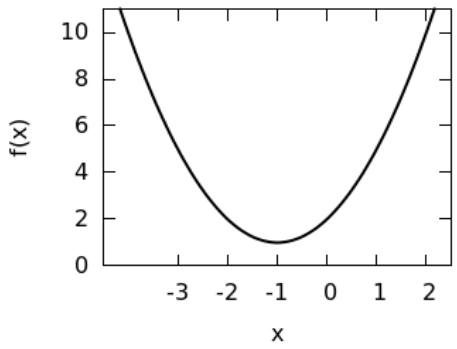

Figure 3.2 Plot of the function $f(x) = (x+1)^2 + 1$.

Thus, we can formalize the goal of the LR learning algorithm as minimizing the above cost function. Next we will discuss how we do this efficiently.

3.3 Gradient Descent

The missing component that connects the cost function just introduced with the LR training algorithm (Algorithm 4) is gradient descent. Gradient descent is an iterative method that finds the parameters that minimize a given function. In our context, we will use gradient descent to find the LR parameters ($\mathbf{w}$ and b) that minimize the cost function C.

However, for illustration purposes, let us take a step away from the LR cost function and work with a simpler example in the beginning: let us assume we would like to minimize the function $f(x) = (x+1)^2 + 1$, which is plotted in Figure 3.2. Clearly, the smallest value this function takes is 1, which is obtained when $x = -1$. Gradient descent finds these values by taking advantage of the function slope, or derivative of $f(x)$ with respect to x, i.e., $\frac{d}{dx}f(x)$. Note: if the reader needs a refresher of what function derivatives are, and how to compute them, now is a good time to do so. Any calculus textbook or even the Wikipedia page for function derivatives[2] provide sufficient information for what we need in this book.

One important observation about the slope of a function is that it indicates the function's direction of change. That is, if the derivative is negative, the function decreases; if it is positive, the function increases; and if it is zero, we have reached a local minimum or maximum for the function. Let us verify that is the case for our simple example. The derivative of our function $\frac{d}{dx}((x+1)^2+1)$ is $2(x+1)$, which has negative values when $x < -1$, positive values when $x > 1$, and is 0 when $x = -1$. Intuitively, gradient descent uses this observation to take small steps towards the function's minimum in the direction indicated by the slope. More formally, gradient descent starts by initializing x with some random value, e.g., $x = -3$, and

[2] https://en.wikipedia.org/wiki/Derivative

then repeatedly subtracts a quantity proportional with the derivative from x, until it *converges*, i.e., it reaches a derivative of 0 (or close enough so we can declare success). That is, we repeatedly compute:

$$x = x - \alpha \frac{d}{dx} f(x) \tag{3.9}$$

until convergence. α in the above equation is the same learning rate introduced before in this chapter. Let us set $\alpha = 0.1$ for this example. Thus, in the first gradient descent iteration, x changes to $x = -3 - 0.1 \times 2(-3+1) = -2.6$. In the second iteration, x becomes $x = -2.6 - 0.1 \times 2(-2.6+1) = -2.28$. And so on, until, after approximately 30 iterations, x approaches -1.001, a value practically identical to what we were looking for.

This simple example also highlights that the learning rate α must be positive (so we don't change the direction indicated by the slope), and small (so we do not "Tony Hawk" the data). To demonstrate the latter situation, consider the situation when $\alpha = 1$. In this case, in the first iteration x becomes 1, which means we already skipped over the value that yields the function's minimum ($x = -1$). Even worse, in the second iteration, x goes back to -3, and we are now in danger of entering an infinite loop! To mitigate this situation, α usually takes small positive values, say, between 0.1 and 0.00001. In Chapter 6 we will discuss other strategies to dynamically shrink the learning rate as the learning advances, so we further reduce our chance of missing the function's minimum.

The gradient descent algorithm generalizes to functions with multiple parameters: we simply update each parameter using its own partial derivative of the function to be minimized. For example, consider a new function that has two parameters, x_1 and x_2: $f(x_1, x_2) = (x_1 + 1)^2 + 3x_2 + 1$. For this function, in each gradient descent iteration, we perform the following updates:

$$x_1 = x_1 - \alpha \frac{d}{dx_1} f(x) = x_1 - 0.1(2x_1 + 2)$$
$$x_2 = x_2 - \alpha \frac{d}{dx_2} f(x) = x_2 - 0.3$$

or, in general, for a function $f(\mathbf{x})$, we update each parameter x_i using the formula:

$$x_i = x_i - \alpha \frac{d}{dx_i} f(\mathbf{x}) \tag{3.10}$$

One obvious question that should arise at this moment is why are we not simply solving the equation where the derivative equals 0, as we were taught in calculus? For example, for the first simple example we looked at, $f(x) = (x+1)^2 + 1$, zeroing the derivative yields immediately the exact solution $x = -1$. While this approach works well for functions with a single parameter or two, it becomes prohibitively expensive for functions with four or more

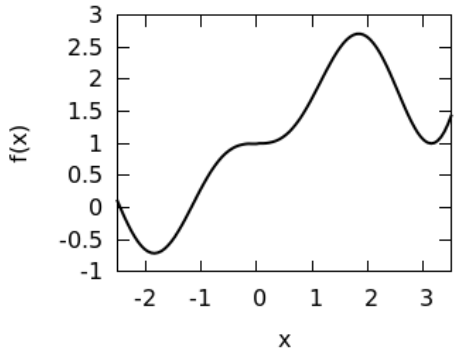

Figure 3.3 Plot of the function $f(x) = x\sin(x)^2 + 1$.

parameters. Machine learning in general falls in this latter camp: it is very common that the functions we aim to minimize have thousands (or even millions) of parameters. In contrast, as we will see later, gradient descent provides a solution whose runtime is linear in the number of parameters times the number of training examples.

It is important to note that gradient descent is not perfect. It does indeed work well for convex functions, i.e., functions that have exactly one minimum and are differentiable at every point such as our simple example, but it does not perform so well in more complex situations. Consider for example the function shown in Figure 3.3.[3] This functions has two minima (around $x = 3$ and $x = -2$). Because gradient descent is a "greedy" algorithm, i.e., it commits to a solution relying only on local knowledge without understanding the bigger picture, it may end up finding a minimum that is not the best. For example, if x is initialized with 2.5, gradient descent will follow the negative slope at that position, and end up discovering the minimum around $x = 3$, which is not the best solution. However, despite this known limitation, gradient descent works surprisingly well in practice.

Now that we have a general strategy for finding the parameters that minimize a function, let us apply it to the problem we care about in this chapter, that is, finding the parameters $\mathbf{w}$ and b that minimize the cost function $C(\mathbf{w}, b)$ (Equation 3.8). A common source of confusion here is that the parameters of C are $\mathbf{w}$ and b, not $\mathbf{x}$ and y. For a given training example, $\mathbf{x}$ and y are known and constant. That is, we know the values of the features and the label for each given example in training, and all we have to do is compute $\mathbf{w}$ and b. Thus, the training process of LR reduces to repeatedly updating each w_j in $\mathbf{w}$ and b features by the corresponding partial derivative of C:

[3] This example of a function with multiple minima taken from https://en.wikipedia.org/wiki/Derivative.

$$w_j = w_j - \alpha \frac{d}{dw_j} C(\mathbf{w}, b) \tag{3.11}$$

$$b = b - \alpha \frac{d}{db} C(\mathbf{w}, b)) \tag{3.12}$$

Assuming a sufficient number of iterations, and a learning rate α that is not too large, $\mathbf{w}$ and b are guaranteed to converge to the optimal values because the logistic regression cost function is convex.[4] However, one problem with this approach is that computing the two partial derivatives requires the inspection of *all* training examples (this is what the summation in Equation 3.8 indicates), which means that the learning algorithm would have to do many passes over the training dataset before any meaningful changes are observed. Because of this, in practice, we do not compute C over the whole training data, but over a small number of examples at a time. This small group of examples is called a *mini batch*. In the simplest case, the size of the mini batch is 1, i.e., we update the $\mathbf{w}$ and b weights after seeing each individual example i, using a cost function computed for example i alone:

$$C_i(\mathbf{w}, b) = -(y_i \log \sigma(\mathbf{x_i}; \mathbf{w}, b) + (1 - y_i) \log(1 - \sigma(\mathbf{x_i}; \mathbf{w}, b))) \tag{3.13}$$

This simplified form of gradient descent is called *stochastic gradient descent* (SGD), where "stochastic" indicates that we work with a stochastic approximation (or an estimate) of C. Building from the last three equations above, we can write the logistic regression training algorithm as shown in Algorithm 5. The reader will immediately see that this formulation of the algorithm is similar to Algorithm 4, which we introduced at the beginning of this chapter. In the next section, we will demonstrate that these two algorithms are indeed equivalent, by computing the two partial derivatives $\frac{d}{dw_j} C_i(\mathbf{w}, b)$ and $\frac{d}{db} C_i(\mathbf{w}, b)$.

3.4 Deriving the Logistic Regression Update Rule

Here we will compute the partial derivative of the cost function $C_i(\mathbf{w}, b)$ with respect to each feature weight w_j and bias term b. For these operations we will rely on several rules to compute the derivatives of a few necessary functions. These rules are listed in Table 3.1.

Let us start with the derivative of C with respect to one feature weight w_j:

$$\frac{d}{dw_j} C_i(\mathbf{w}, b) = \frac{d}{dw_j} (-y_i \log \sigma(\mathbf{x_i}; \mathbf{w}, b) - (1 - y_i) \log(1 - \sigma(\mathbf{x_i}; \mathbf{w}, b)))$$

[4] Demonstrating that the LR cost function is convex is beyond the goal of this book. The interested reader may read other materials on this topic such as http://mathgotchas.blogspot.com/2011/10/why-is-error-function-minimized-in.html.

Algorithm 5: Logistic regression learning algorithm using stochastic gradient descent.

```
w = 0
b = 0
while not converged do
    for all training examples x_i ∈ X do
        d = decision(x_i, w, b)
        for each w_j in w do
            w_j = w_j − α (d/dw_j) C_i(w, b)
        end
        b = b − α (d/db) C_i(w, b)
    end
end
```

Let us use σ_i to denote $\sigma(\mathbf{x_i}; \mathbf{w}, b)$ below, for simplicity:

$$= \frac{d}{dw_j}(-y_i \log \sigma_i - (1-y_i)\log(1-\sigma_i))$$

After applying the chain rule on the two logarithms:

$$= -y_i \frac{d}{d\sigma_i} \log \sigma_i \frac{d}{dw_j}\sigma_i - (1-y_i)\frac{d}{d(1-\sigma_i)}\log(1-\sigma_i)\frac{d}{dw_j}(1-\sigma_i)$$

After applying the derivative of the logarithm:

$$= -y_i \frac{1}{\sigma_i}\frac{d}{dw_j}\sigma_i - (1-y_i)\frac{1}{1-\sigma_i}\frac{d}{dw_j}(1-\sigma_i)$$

After applying the chain rule on $\frac{d}{dw_j}(1-\sigma_i)$:

$$\begin{aligned}
&= -y_i \frac{1}{\sigma_i}\frac{d}{dw_j}\sigma_i + (1-y_i)\frac{1}{1-\sigma_i}\frac{d}{dw_j}\sigma_i \\
&= (-y_i \frac{1}{\sigma_i} + (1-y_i)\frac{1}{1-\sigma_i})\frac{d}{dw_j}\sigma_i \\
&= \frac{-y_i(1-\sigma_i) + (1-y_i)\sigma_i}{\sigma_i(1-\sigma_i)}\frac{d}{dw_j}\sigma_i \\
&= \frac{\sigma_i - y_i}{\sigma_i(1-\sigma_i)}\frac{d}{dw_j}\sigma_i
\end{aligned}$$

Table 3.1 Rules of computation for a few functions necessary to derive the logistic regression update rules. In these formulas, f and g are functions, a and b are constants, x is a variable.

Description	Formula
Chain rule	$\frac{d}{dx}f(g(x)) = \frac{d}{dg(x)}f(g(x))\frac{d}{dx}g(x)$
Derivative of summation	$\frac{d}{dx}(af(x)+bg(x))) = a\frac{d}{dx}f(x)+b\frac{d}{dx}g(x)$
Derivative of natural logarithm	$\frac{d}{dx}\log(x) = \frac{1}{x}$
Derivative of sigmoid	$\frac{d}{dx}\sigma(x) = \frac{d}{dx}(\frac{1}{1+e^{-x}}) = -\frac{1}{(1+e^{-x})^2}(-e^{-x}) = \sigma(x)(1-\sigma(x))$
Derivative of dot product between vectors $\mathbf{x}$ and $\mathbf{a}$ with respect to x_i	$\frac{d}{dx_i}(\mathbf{x}\cdot\mathbf{a}) = a_i$

After applying the chain rule on σ_i:

$$= \frac{\sigma_i - y_i}{\sigma_i(1-\sigma_i)}\frac{d}{d(\mathbf{w}\cdot\mathbf{x_i}+b)}\sigma_i\frac{d}{dw_j}(\mathbf{w}\cdot\mathbf{x_i}+b)$$

After the derivative of the sigmoid:

$$= \frac{\sigma_i - y_i}{\sigma_i(1-\sigma_i)}\sigma_i(1-\sigma_i)\frac{d}{dw_j}(\mathbf{w}\cdot\mathbf{x_i}+b)$$
$$= (\sigma_i - y_i)\frac{d}{dw_j}(\mathbf{w}\cdot\mathbf{x_i}+b)$$

Lastly, after applying the derivative of the dot product:

$$= (\sigma_i - y_i)x_{ij} \tag{3.14}$$

where x_{ij} is the value of feature j in the feature vector $\mathbf{x_i}$.

Following a similar process, we can compute the derivative of C_i with respect to the bias term as:

$$\frac{d}{db}C_i(\mathbf{w},b) = \frac{d}{db}(-y_i\log\sigma(\mathbf{x_i};\mathbf{w},b) - (1-y_i)\log(1-\sigma(\mathbf{x_i};\mathbf{w},b))) = \sigma_i - y_i \tag{3.15}$$

One can immediately see that applying Equation 3.15 in line 9 of Algorithm 5 transforms the update of the bias into the form used in Algorithm 4 (line 7). Similarly, replacing the partial derivative in line 7 of Algorithm 5 with its explicit form from Equation 3.14 yields

an update equivalent with the update used in Algorithm 4. The superficial difference between the two algorithms is that Algorithm 5 updates each feature weight w_j explicitly, whereas Algorithm 4 updates *all* weights at once by updating the entire vector $\mathbf{w}$. Needless to say, these two forms are equivalent. We prefer the explicit description in Algorithm 5 for clarity. But, in practice, one is more likely to implement Algorithm 4 because vector operations are efficiently implemented in most machine learning software libraries.

3.5 Drawbacks of Logistic Regression

The logistic regression algorithm solves the lack of smooth updates in the Perceptron algorithm through its improved update functions on its parameters $\mathbf{w}$ and b. This seemingly small change has an important practical impact: in most NLP applications, logistic regression tends to outperform the Perceptron. However, the other drawbacks observed with the Perceptron still hold. Logistic regression is also a linear classifier because the decision that informs the separating hyperplane is formed by taking a linear combination of its features (i.e., $\mathbf{w} \cdot \mathbf{x} + b$). Similar to the Perceptron, LR relies on explicit features, which, as discussed in the previous chapter, may be cumbersome to generate and may generalize poorly. Lastly, LR also focuses on individual predictions rather than structured learning. We will address all these limitations in the following chapters. We will start by introducing non-linear classifiers in the next chapter.

3.6 References and Further Readings

TODO: To do

4 Feed Forward Neural Networks

TODO: Feed forward networks for binary classification. The equations for backpropagation.

5 Distributional Similarity and Representation Learning

TODO: Traditional distributional similarity: co-occurrence matrices + SVD. Word embeddings. Word2vec.

6 Best Practices in Deep Learning

TODO: Catchall for issues we did not discuss before: multiclass classification, softmax; regularization; other cost functions; optimizers

7 Implementing Feed Forward Networks in DyNet

8 Sequence Models

TODO: CNNs, RNNs

9 Implementing Sequence Models in DyNet

10 Sequence-to-sequence Methods

TODO: Seq2seq without and with attention

11 Transformer Networks

TODO: Transformer networks; BERT; BERT variants

12 Domain Transfer

TODO: Mixing source and destination datasets; neural forms of Hal Daumé's frustratingly easy algorithm

13 Semi-supervised Learning

TODO: Traditional bootstrapping. One-shot algorithms, e.g., Valpola's mean teacher

Bibliography

H.-D. Block. 1962. The perceptron: A model for brain functioning. i. *Reviews of Modern Physics*, 34(1): 123.

D. Dickerson. 2019. How i overcame impostor syndrome after leaving academia. *Nature*. https://www.nature.com/articles/d41586-019-03036-y. DOI: 10.1038/d41586-019-03036-y.

T. G. Dietterich. 2000. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pp. 1–15. Springer.

P. Domingos. 2015. *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. Basic Books.

J. Donaldson and A. Scheffler. 2008. *Where's My Mom?* Dial Books.

J. R. Firth. 1957. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*.

D. Griffiths. 2008. *Head first statistics*. O'Reilly Germany.

K. Knight, 2009. Bayesian inference with tears. https://www.isi.edu/natural-language/people/bayes-with-tears.pdf. Accessed: 2019-10-19.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105.

C. D. Manning. 2015. Computational linguistics and deep learning. *Computational Linguistics*, 41(4): 701–707.

G. Marcus and E. Davis. 2019. *Rebooting AI: Building Artificial Intelligence We Can Trust*. Penguin Random House.

M. Minsky and S. Papert. 1969. Perceptron: an introduction to computational geometry. *The MIT Press, Cambridge, expanded edition*, 19(88): 2.

A. B. Novikoff. 1963. On convergence proofs for perceptrons. Technical report, Stanford Research Institute.

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

H. Schütze, C. D. Manning, and P. Raghavan. 2008. Introduction to information retrieval. In *Proceedings of the international communication of association for computing machinery conference*, p. 260.

D. R. Swanson. 1986. Undiscovered public knowledge. *The Library Quarterly*, pp. 103–118.

M. A. Valenzuela-Escárcega, O. Babur, G. Hahn-Powell, D. Bell, T. Hicks, E. Noriega-Atala, X. Wang, M. Surdeanu, E. Demir, and C. T. Morrison. 2018. Large-scale automated machine reading discovers new cancer driving mechanisms. *Database: The Journal of Biological Databases and Curation*. http://clulab.cs.arizona.edu/papers/escarcega2018.pdf. DOI: 10.1093/database/bay098.

K. Z. Vardakas, G. Tsopanakis, A. Poulopoulou, and M. E. Falagas. 2015. An analysis of factors contributing to pubmed's growth. *Journal of Informetrics*, 9(3): 592–617.

T. Young, D. Hazarika, S. Poria, and E. Cambria. 2018. Recent trends in deep learning based natural language processing. *IEEE Computational intelligence magazine*, 13(3): 55–75.

G. K. Zipf. 1932. Selected studies of the principle of relative frequency in language.

Author's Biography

Your Name

Your Name began life as a small child ...