The Universal Game Rater

XGBoost and Text Mining in Review Score Predictions

Matheus Correia Ferreira

12/24/2020

Contents

1. Introduction

There is no shortage of reviews on the web. Whether they are about movies, games, books, hotels, electronics, or even supermarket products, they come from many sources, exist in many formats, and invariably elicit a myriad of intense reactions that range from total disagreement to enthusiastic agreement, especially when they are related to a hobby, such as reading or playing games. With so much information available, it is only natural that the average costumer will have trouble knowing who to trust; after all, regardless of the product being evaluated, the internet is large enough to contain both positive and negative reactions concerning it.

To solve that problem, review aggregators have become quite popular in the past few years. These websites gather reviews from relatively trustworthy and professional sources around the web in order to give both fans and costumers a neat glimpse into the overall perception that surrounds an item. From Rotten Tomatoes [1], which does so for movies, to Metacritic [2], which covers numerous fields, these websites, more than thriving on how practical they are, work on the principle that all humans are naturally biased [3] and, because of that, the aggregated opinions of critics is more accurate than the individual evaluations; an idea that, curiously, is not too distant from the Machine Learning practice of Boosting [4].

Still, biases remain, as they always will, and most of that comes from the scores themselves. Scores are, ultimately, arbitrary numbers that are attached to items in order to try to capture how one person feels in relation to them, an activity with a lot of room for error in judgment. Due to that nature, discussions on the value of scores have been constant, especially throughout the gaming media [5], with some publications abandoning scores altogether [6] and others adapting their scoring scales to simpler formats, usually attaching words that define quality (such as “Masterpiece” and “Amazing”) to the numbers [7].

The question proposed in this work is: can Machine Learning save us from this mess? These techniques have done quite a good job at detecting the sentiment expressed in texts; so much, in fact, that there is a whole field of study dedicated to it [8]. So would it be possible for a Machine Learning model to, based on the text that was written, capture the feeling contained in it and, from that point, indicate the score that should be given to a game?

Artificial Intelligence is already widely used to take away certain activities from the hands of humans and reduce the rates of errors [9]. Could attaching scores to reviews based on their enthusiasm, neutrality, or disappointment be one of them?

2. Proposal

The proposal of this work is the creation of a model that, based on the text of a review, will indicate the score that should be given to it. The conceptual idea is not the elimination of the critic’s right to award a score to an item (in the case studied here, a game), but the creation of what we will call a Universal Rater: an algorithm that will produce a global score, an alternative series of ratings that will come from measuring the sentiment in the text of reviews from multiple publications and giving them individual grades.

It is important to note that, in a way, that task is quite hard. Our model will, naturally, learn from reviews and scores produced in the past in order to try to create probabilistic associations between terms employed and grades. These reviews themselves, though, carry human bias, so our algorithm will not be able to achieve complete neutrality, as the source of its knowledge carries defined tendencies. Furthermore, Machine Learning algorithms essentially work by trying to minimize an error metric, with the ultimate goal being to achieve zero error in a real-world scenario. If our Universal Rater got to this zero error, however, it would not have reached total neutrality, but modeled the bias of every reviewer it learned from.

Still, minimizing the error is important, of course, as we do not want the Universal Rater to give a shiny grade to a game that was crushed to pieces by a reviewer or vice-versa. In the case of the concept here, however, what is most valuable is having a model that will grasp the universal scoring tendencies: the associations between words used and score. Nonetheless, given it is the nature of Machine Learning algorithms to reduce errors as much as possible, we will pursue that goal and look at the results.

3. Dataset Overview

The data that will be used in this work to construct our targeted Machine Learning model was obtained from Kaggle. Called "Metacritic Critic Games Reviews 2011-2019" [10], it contains 125,876 reviews published by professional critics between the indicated years. As it is the norm, excerpts from these reviews were published and aggregated in Metacritic with the goal of giving its users an overall idea of the reception received by these titles. Each row (review) in the dataset has the columns listed below:

- **name**: The name of the publication in which the review was posted.
- **review**: The written content of the review. Note that this is not the full text, but the excerpt that is published by Metacritic. The intent of this excerpt is to, naturally, capture the overall feelings of the writer regarding the title.
- **game**: The game being reviewed.
- **platform**: The platform where the game was played during the analysis.
- **score**: The score given to the game. In other words, the value we will try to predict.
- **date**: The date when the review was published.

Below, a small sample of the dataset is displayed.

Name	Review	Game	Platform	Score	Date
GameCritics	So do we need Portal 2? Do I need it? Maybe not, but I'm sure as hell glad it exists. The portal aspect has probably reached its zenith in Portal 2, and given the way the game ends I don't think there's much room for a Portal 3.	Portal 2	PC	100	May 8, 2011
Nintendojo	Beating each stage is pure joy, symbolizing our ascent beyond our childhood years– or perhaps how they stay with us constantly, only even more vivid than before.	Pushmo	3DS	91	Dec 22, 2011
Bit-Gamer	Yes, Limbo channels the poems of our inner teenagers about being misunderstood and lonely, but it does so bravely and beautifully, and it is a better game for it; a game that should not be missed.	LIMBO	PC	90	Aug 7, 2011

Given we want to focus on the relationship between words and score only, just two of the columns represented in the prior table will be considered in this work: review and score. The others will not be used. Finally, as far as general dataset considerations go, the reviews were split into three subsets for experimental purposes:

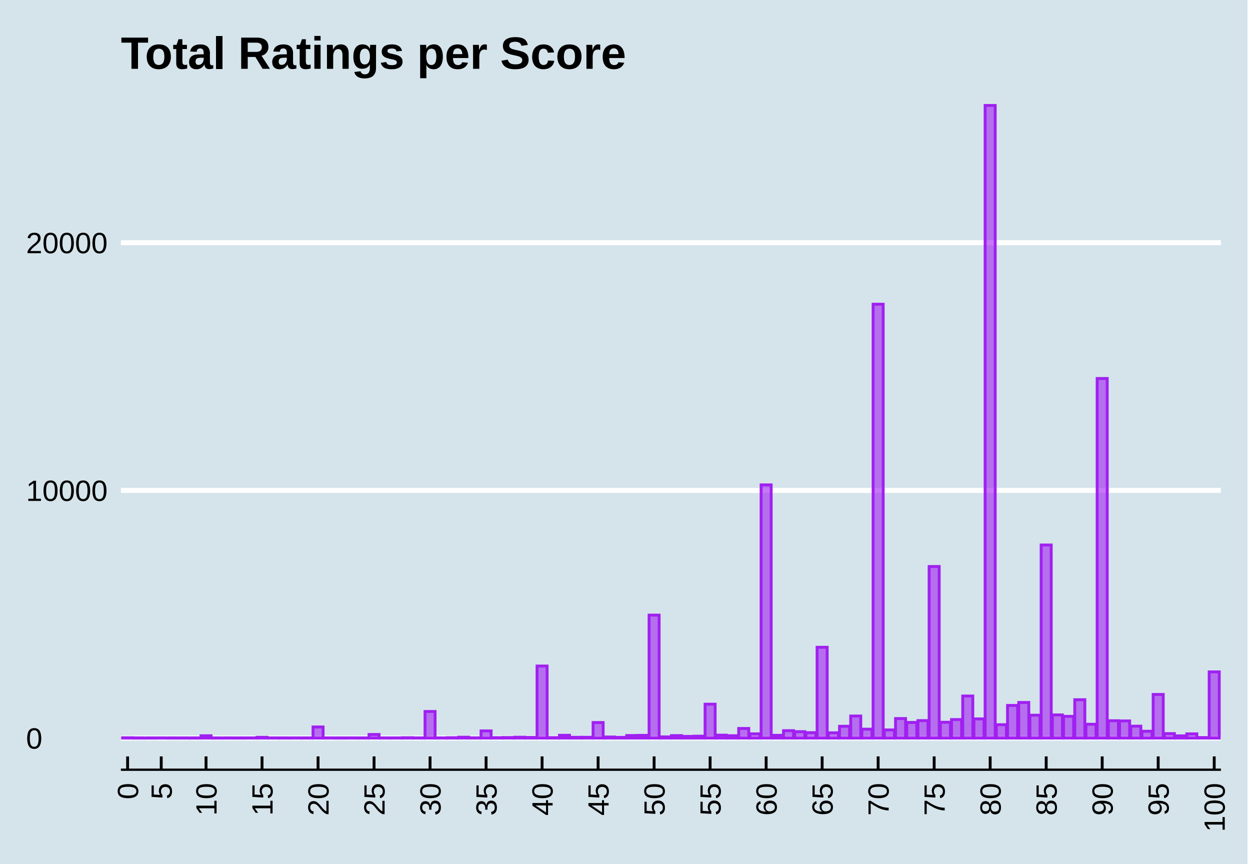
- **Training Set**: Containing 99,814 rows (that is, approximately 80% of the full dataset), it was used to build the rating models.
- **Validation Set**: Containing 11,094 rows (that is, approximately 10% of the full dataset), it was used to test the performance of the models constructed with the training set so that they could be refined.
- **Test Set**: Containing 12,325 rows (also approximately 10% of the full dataset), it was only used to get the final performance metric that will be reported in this work. In other words, it was the dataset against which our final model was used so the obtained accuracy of the scoring predictions could be reported.

Here, two observations are important. The first is that summing the amount of rows of the sets will not yield the total dataset because a few reviews had to be excluded due to reasons that will be detailed in the next section. Secondly, the percentage of data the validation set corresponds to is slightly less than 10% because of how the split was performed. To being, the full dataset, minus the eliminated reviews, was divided between training and test. Only then was the training set further broken into training and validation.

4. General Data Preparation

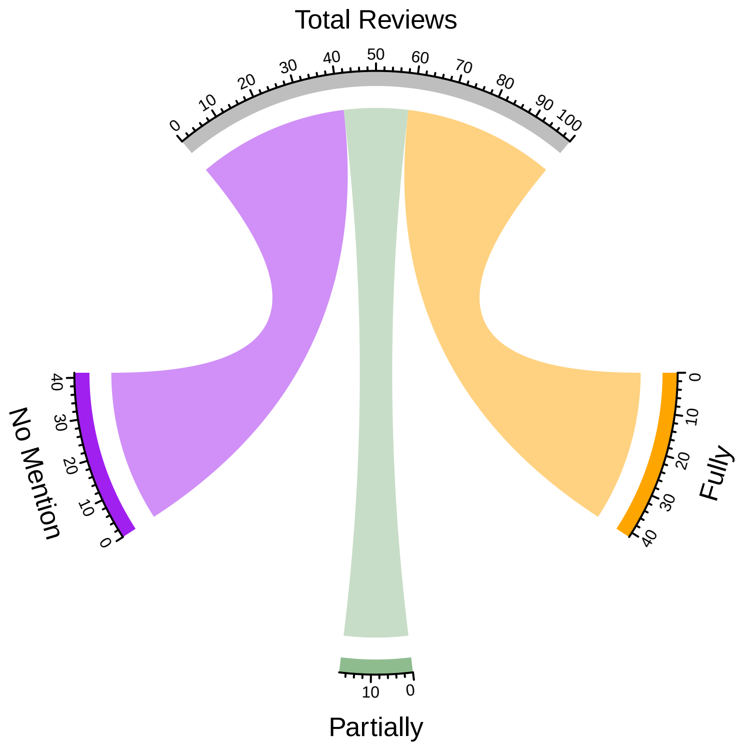
The inaugural task of data preparation was getting rid of reviews whose contents were "Quotation forthcoming". This is a placeholder text put in by Metacritic for reviews whose excerpts are yet to be extracted. Sadly, 1,078 of them made it into the dataset whether because the excerpts never came or because they had yet to show up when the extraction was done. Since such quote has no value when it comes to feelings or score, these records were eliminated. Additionally, 1,565 reviews had no score, so these were also cut.

After doing so, a second task of preparation was executed, this time over the scores. The reason behind such move can be partially explained by the plot below.



As it is possible to observe, ratings that follow a 5-point scale are far more common than those that go for a 1-point scale. Such fact can be explained by how most publications do not make use of the latter. Following the majority, our Universal Rater will do the same, producing all of its scores in increments of 5. As such, scores that do not follow such rule were were rounded to the nearest 5-point increment: for example, a 72 rating was brought down to 70 and a 73 elevated to 75.

The final preparation that will be discussed in this section is related to game names. All reviews, naturally, reference the title of the product they are analyzing, and the excerpts with which we will work are no exception.



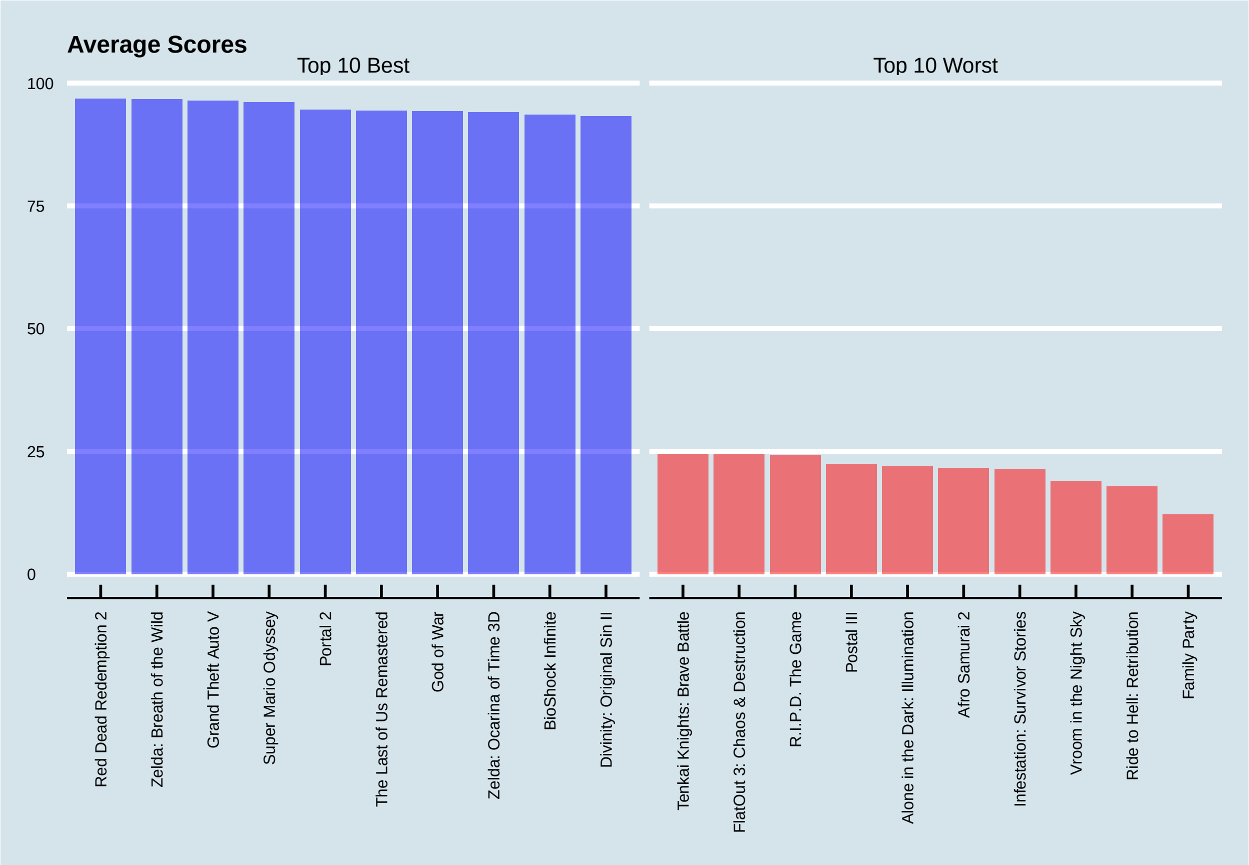
How Often Are Game Names Mentioned in Reviews?

The previous plot shows that a surprising amount of our reviews, which are excerpts, do not have the names of the games: about 40%. Yet, the other 60% either have them partially, like mentions to the word “Zelda” in reviews for “The Legend of Zelda: Ocarina of Time 3D”, or fully. This information is relevant to the task we are pursuing because we want our Universal Rater to be as unbiased as possible; we want it to give out ratings according to the feeling that is contained in the text. Leaving the game names in there would go against that goal.

After all, some games receive better scores than others, as such if the game titles remained in the text, we would create a situation in which the Universal Rater would have the score it dishes out influenced by the perception reviewers have about the game. If Zelda games tend to receive good evaluations, then a text with the word Zelda would be seen as having positive tendencies in it. That is, curiously, an effect not too different from the one noticed in the media in general, where franchise names and hype are sometimes enough to produce great scores or alleviate bad ones.

To illustrate how much game names could be influential, the next plot shows the average score for the best and worst rated games in our dataset.

It goes without saying that the gap is considerable, and that any future release that carried these names would have our Universal Rater approaching it with either a positive or a negative bias, given the terms would be heavily associated with positive or negative scores. For the sake of visualization, a couple of names were abbreviated.



5. Text Preparation

In addition to these more general steps of data preparations, a little bit of text processing was also done in order for the learning of our model to go more smoothly and, hopefully, generate better results. These steps, which are commonplace in the Natural Language Processing area, included:

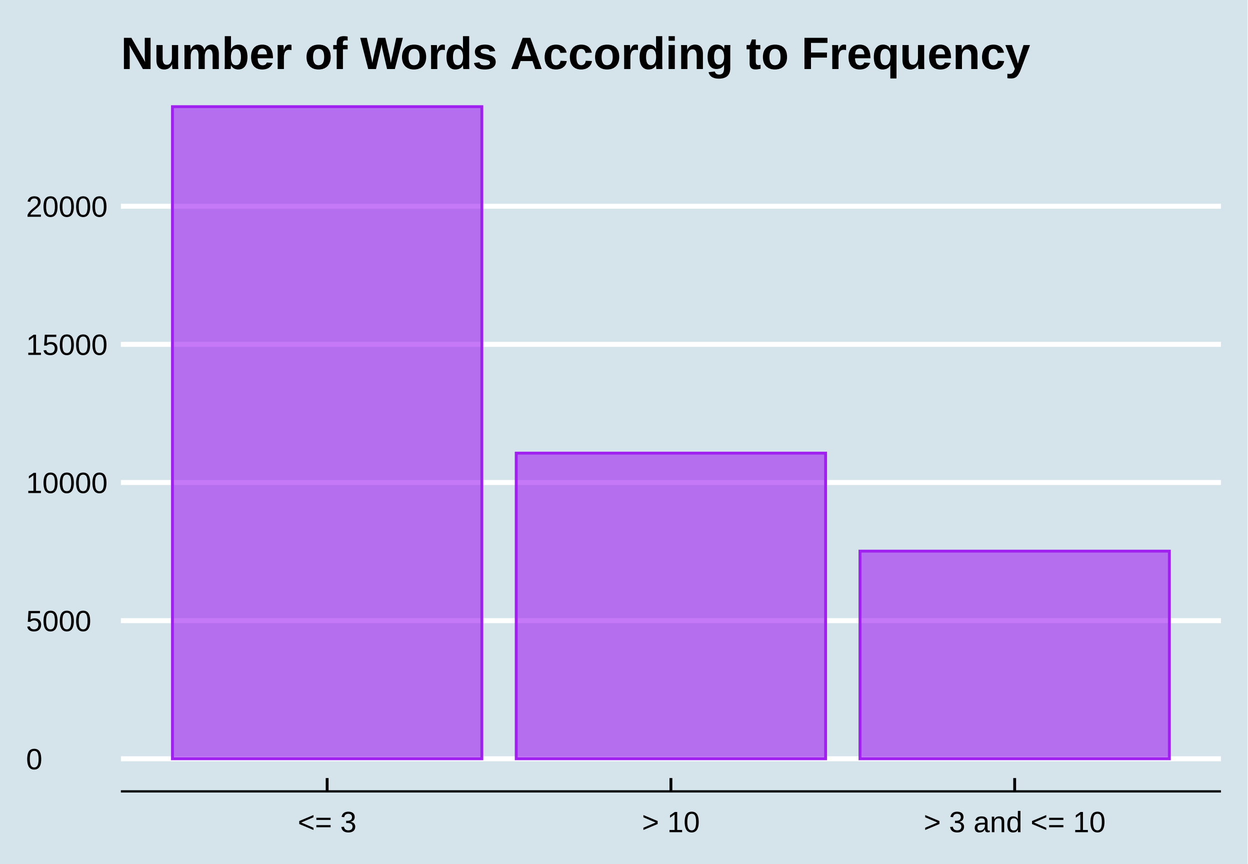
- Removal of digits, given that, intuitively, their presence does not add much in terms of value and meaning to the text. Worse yet, they may be ratings themselves, which would generate bias.

- Removal of punctuation. The reasoning here was the same one behind the exclusion of digits: the fact they don't add meaning to the text.

- Transformation of the text to lower case letters. After all, whether its first character has been capitalized or not, the word is usually the same and we want our algorithm to see them as such.

More important than those were another two tasks that were performed (or, in the case of the second, not performed) over the texts of our review excerpts. The first had to do with the frequency of words. In problems of this sort (that is, tasks that involve Machine Learning and text-based datasets) it is common for one to fall victim to the so-called Curse of Dimensionality [11], a problem that arises - in these cases - from having texts with too many unique terms. As a consequence, models suffer to learn not only because the processing they need to do is heavier, but also because achieving generalizations in higher dimensions (with more words) is simply harder, which may affect their precision.

With 123,233 review excerpts, when the previously discussed exclusions are accounted for, the dataset at hand is one that could suffer from that problem, especially given the hardware limitations involved in performing these tests and the numerous experiments that were executed, which could lead to prolonged processing times. And, in fact, a count done only in the training dataset revealed the presence of 42,186 unique words. One detail about their frequency, demonstrated in the next plot, is quite relevant.



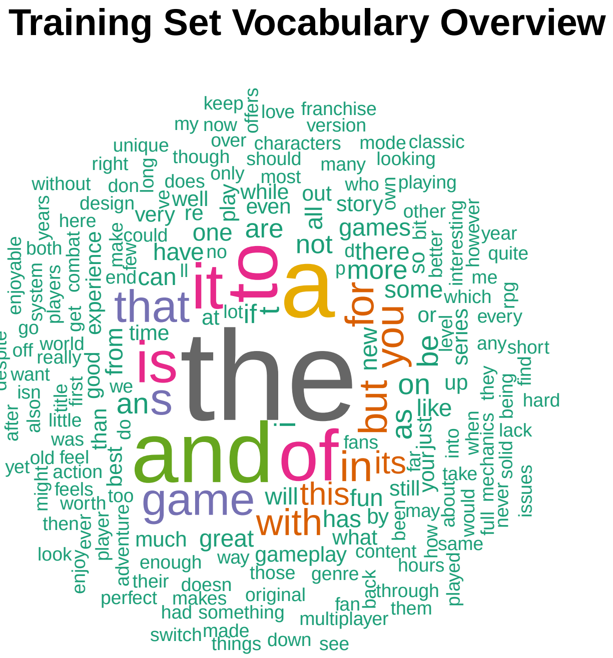
Out of the 42,186 unique words of our review excerpts, more than half of them appear three times or less, meanwhile those that show up over 10 times are slightly over 10,000 and those between 3 and 10 are even rarer. Given this scenario, a choice was made to eliminate from reviews all words whose frequencies were lower than 3; in addition, a few terms with only one character (mostly leftovers from the previous data-cleaning procedures) were also removed. This process reduced our total vocabulary to 18,544 words.

It is important to highlight that the counts in the plot above refer only to the training dataset. Words from the validation and test sets were not counted here. This is because considering those counts would have led to data leakage; that is, the use of information outside of the training set to make decisions about how to process its data. Words that were cut in this step could have been retained in case they appeared in those other two sets one or two times, meaning they would be saved from this small purge due to the consideration of data that should not have been taken into account.

The last important note about the textual pre-processing that was done is related to stopwords: terms that have little value and that are used quite frequently in texts, such as pronouns, connectives, and articles. These terms are indeed, naturally, quite abundant in the reviews we are using. The Word Cloud below, which highlights the most frequent words in our training vocabulary, makes it quite clear.

Aside from “game”, which is to be expected, all of the most frequent words are stopwords. Despite that, we opted not to remove them from the reviews.

For starters, because there is some debate regarding whether or not taking stopwords out of a text actually improves the precision of models [12]. Furthermore, because to count the words in each review we will use a method, TF-IDF [13], that already reduces the weight of these terms. Lastly, by simply looking at the training reviews pre and post stopwords removal it was possible to see that this process could lead them to lose meaning. Lists of stopwords vary depending on the source, but the ones that were evaluated for this work contained terms such as “but”, “however”, “best”, “good”, and other words that bring meaning to reviews.



Training Set Vocabulary Overview

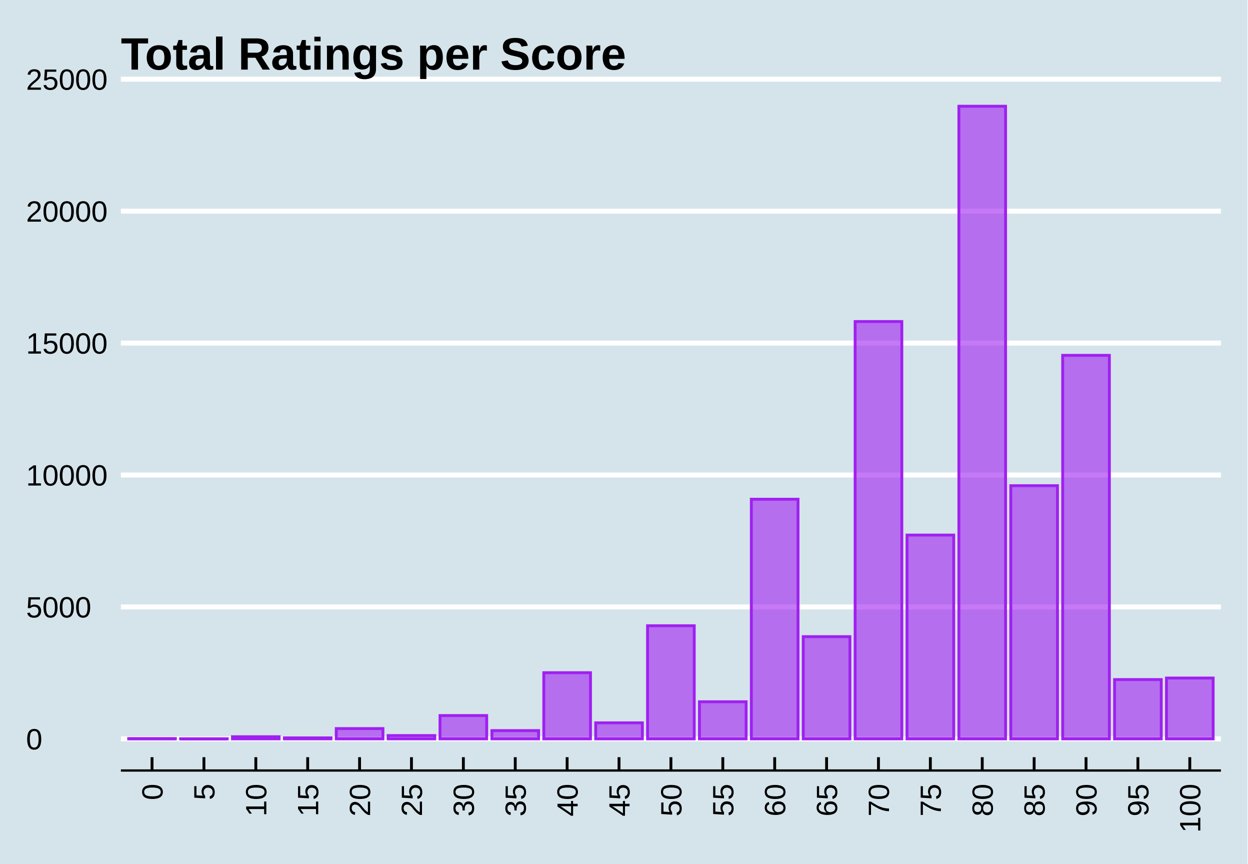
To illustrate the final result of these treatments, which bring the texts to its final written format (the one that will be transformed and fed into the model), a few selected reviews are shown here both in their initial condition, as obtained from the Kaggle dataset, and in the state that was seen after all the processing.

Before	**After**
So do we need Portal 2? Do I need it? Maybe not, but I’m sure as hell glad it exists. The portal aspect has probably reached its zenith in Portal 2, and given the way the game ends I don’t think there’s much room for a Portal 3.	so do we need do need it maybe not but sure as hell glad it exists the portal aspect has probably reached its zenith in and given the way the game ends don think there much room for
What’s remarkable about the game is that it’s aged far better than most of its contemporaries. It’s still fun, it’s still relevant, and it’s still Zelda.	what remarkable about the game is that it aged far better than most its contemporaries it still fun it still relevant and it still
Everyone wanted to know where Mario would go after conquering the universe (twice) on the Wii, and with this new adventure, Nintendo gave the answer. To alternate dimensions? No. To home.	everyone wanted to know where would go after conquering the universe twice on the wii and with this new adventure nintendo gave the answer to alternate dimensions no to home
Dead Rising 4 maintains the series’ tradition of superficial yet entertaining mayhem by adapting ideas from every previous iteration.	maintains the series tradition of superficial yet entertaining mayhem by adapting ideas from every previous iteration

6. Data Analysis

Previously, we looked at the ratings of the whole dataset and noticed that scores with one-point increments were much rarer than those with five-point increments on account of the fact many publications do not use such a precise scale; in fact, many of them do not even use ratings that go up to 10, but Metacritic already adjusts for them. We went one step further and turned all scores that were not in 5-point increments into numbers ending in 0 or 5 by rounding them to their closest neighboring value of that sort.

Now, we look at how our scoring distribution stands after that adjustment.

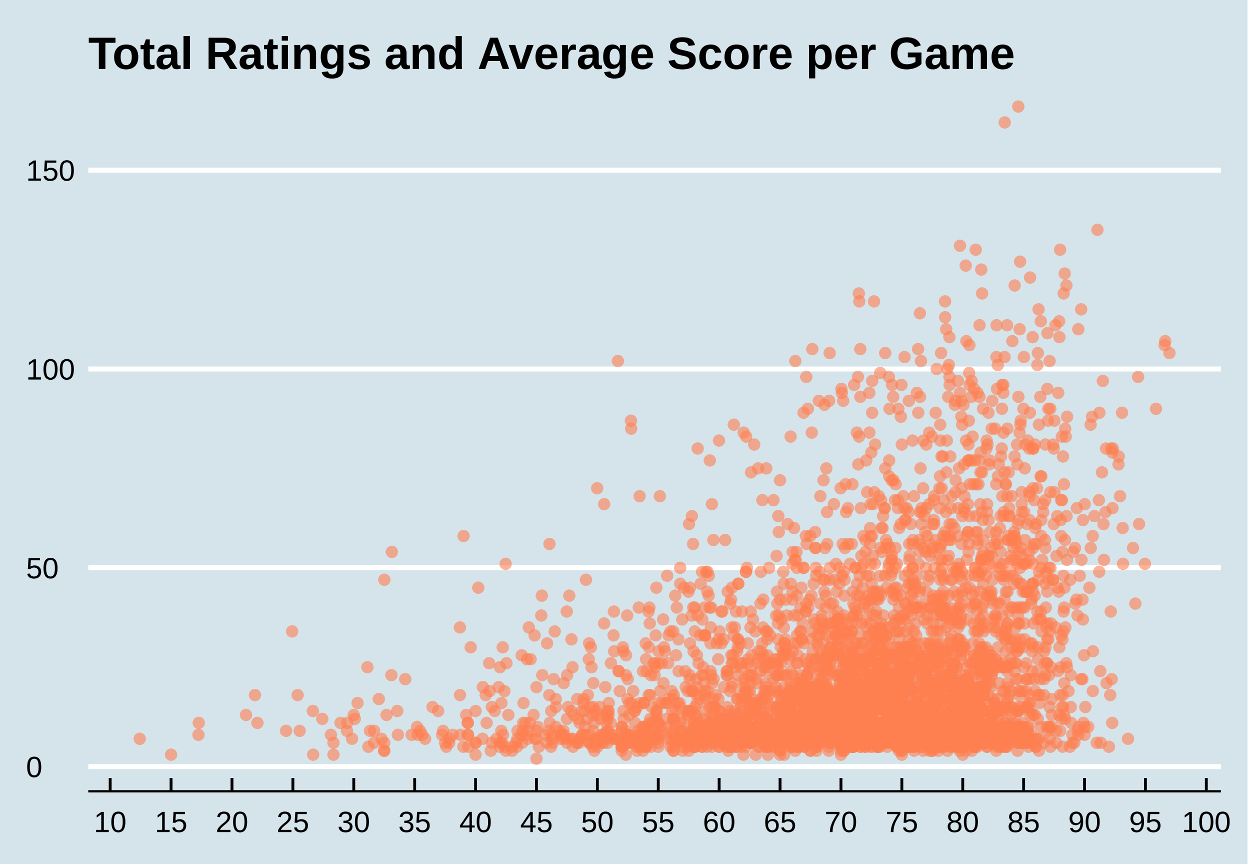


Note that the numbers shown here are those of the training dataset. In fact, from this point onward, any analysis that is done will be focused exclusively on that data. Reviews from the validation and test sets will only be looked at when we check the results of our rating models, with the test set in particular only re-emerging after we have chosen a final model.

What we see in the previous image is a characteristic that could already be noticed in the first plot of this document: that this is an imbalanced dataset. It seems that between the period of 2011 and 2019 the gaming industry released far more games that were received positively than negatively.

A lot of that imbalance also has to do with the fact highly rated games, as seen in the next plot, simply receive more reviews from specialized critics; these titles are, after all, the most popular and hyped ones.

In a way, this is not very good news for our Universal Rater, given it will have fewer examples to learn from when it comes to negative scores. Moreover, it can become slightly biased towards positive ratings since it will naturally diminish the error it is trying to minimize while learning if it hands out grades that fall into the most popular part of the scale.

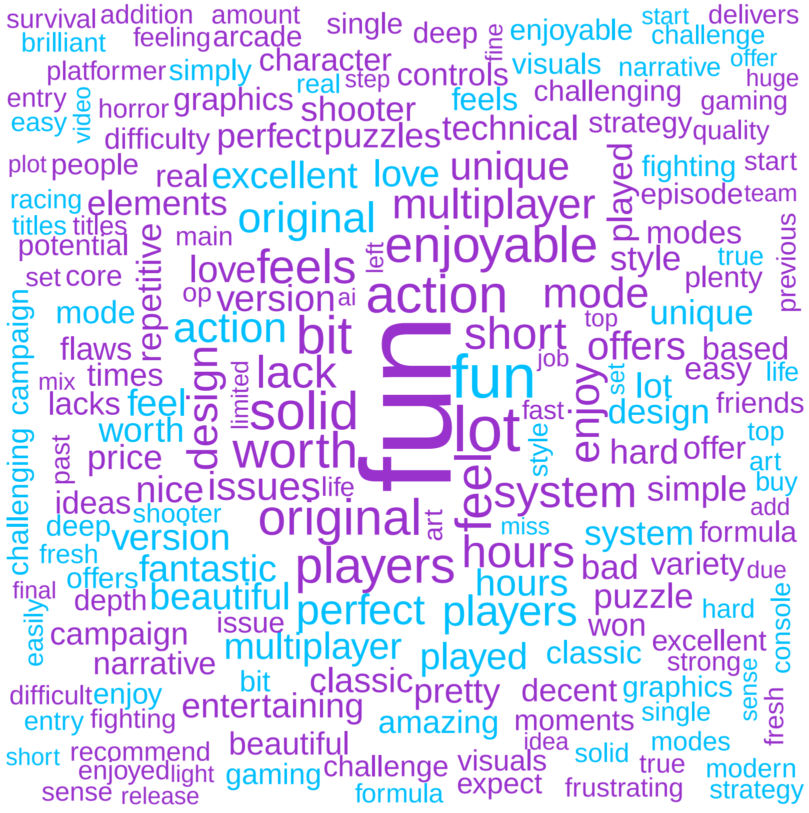


The ultimate question here, though, is if there is enough distinction between the reviews that produced different scores for our Universal Rater to learn what separates a 90 from a 60, and hopefully even an 85 from an 80. We will have the answer for that question when the model is trained over the excerpts we are working with, but a way to get a glimpse into the future is by, of course, looking at the words that are most commonly used according to each grade. Given stopwords were not removed, Word Clouds for different ratings would all be dominated by them; the following images, however, were produced by counting the vocabulary found in reviews that yielded certain scores and eliminating the stopwords so that the visualization of potential distinctions could be possible.

The Word Cloud below compares two ranges of scores that are relatively tight. Words from reviews that got either 95 or 90 can be seen it blue. Words from reviews that got either 75 or 70 are in purple. The size of terms indicates how frequently they appear.

In the case of the two rating ranges we observe, the word “fun” seems to be quite popular, appearing more frequently in reviews from the 70-75 range simply because there are more of those in our data, and there are a lot of positive words floating around. Notice, however, that reviews from the 70-75 range (in purple) seem to include some words that denote problems: “lack”, “bad”, “frustrating”, “repetitive”, and “flaws”. This indicates, of course, that these games are good, but showcase some issues. Titles that fell in the 90-95 range (in blue), meanwhile, seem to carry no such terms with negative tones, which is consistent with what one would expect out of them.

As a consequence, of these gaps when it comes to the terms employed, it looks like a model could learn the distinctions between the tones of these review excerpts to deliver the appropriate scores.



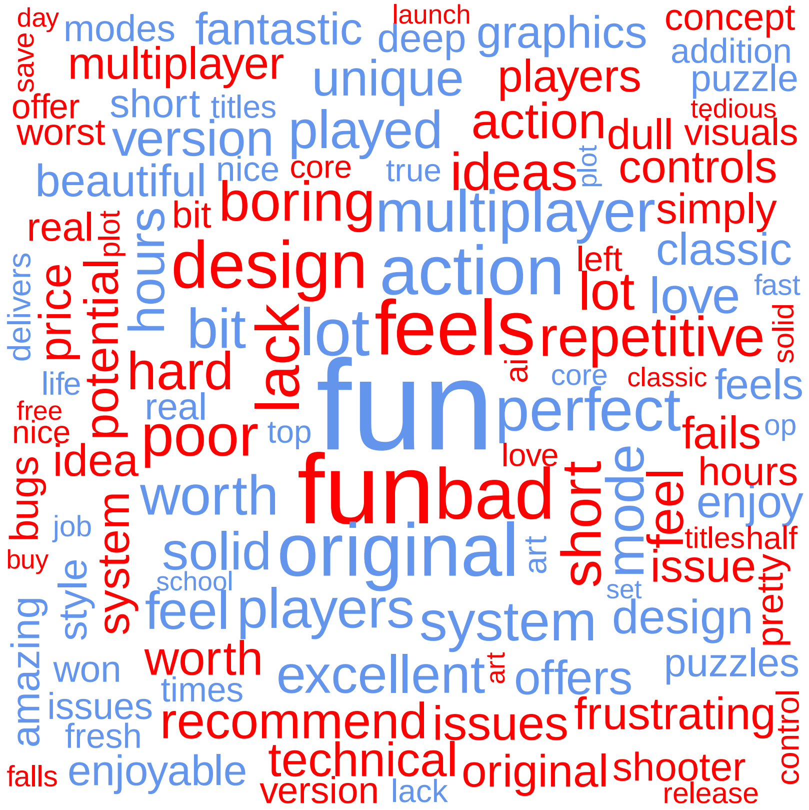
Before we move on, let's look at two more comparisons between the words contained in reviews that fall into different scoring ranges: one in which it will be easy to spot the differences in terms and another where that task will be much harder for our human eyes. For starters, we will see what happens when we create a Word Cloud with terms found in reviews that had a final score of 85 (in blue), and those that had negative grades, namely scores of 50 and below (in red).

Here, the difference is more prominent, as expected. Games that got 85s received words of praise, with terms such as "beautiful", "original", "excellent", and "unique" appearing with elevated frequency.

On the more negative side of the spectrum we are evaluating, titles that got a bad reception from the media are accused of being "bad", "repetitive", "shallow", "boring", "buggy", "flawed", and much more. Still, for both groups, the word "fun" appears to be quite popular, perhaps because titles that received less than 50 were labeled as being "no fun" or "not fun", a conjunction of terms that is not analyzed by our Word Clouds given they work exclusively with individual terms.

As a consequence of this large difference, maybe our model can overcome the imbalanced nature of our dataset and accurately predict when games are called out as being bad. Of course, there are a lot of bins for the range of negativity we are exploring here, as from 0 to 50 there are 11 possible scores.

Nevertheless, it is possible there are different degrees of negativity in the text, which will allow the Universal Rater to pull off its job with some accuracy, or at least that is what we hope will happen when the final results are evaluated and we look at how the model performed on the test set.



To wrap up this sequence of Word Clouds, let's observe the case when differences in vocabulary are much tighter; more specifically, in the case where the gap between the scores is the minimum value for an increment in our scale: 5 points.

In the following image, we display the words that are more commonly used in reviews that produced scores of 60 and 65. The first group is in orange, while the second appears in pink.

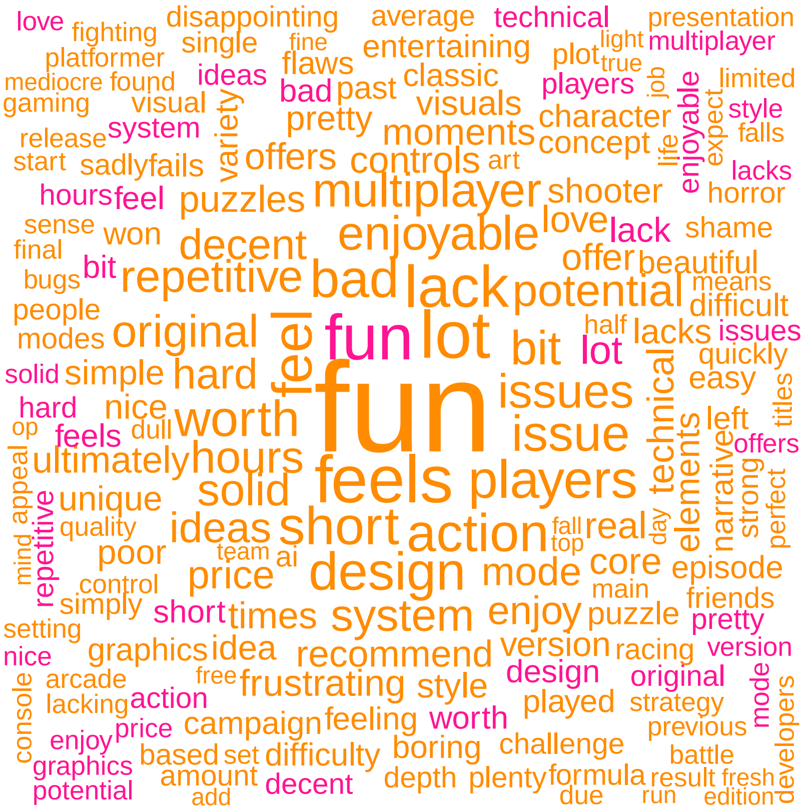
A detail that emerges, one that is there for reasons already explored, is that orange words outnumber those in pink. The cause for that phenomenon is, of course, that reviews where the score was 65 are much rarer than those where the final grade was 60. What we see when we observe the words belonging to each group is that there is not much difference between what these evaluations are saying, a fact that confirms our intuition.

Games in the 60-65 range are decent, but problematic in many forms, at least according to the rating scale used by many publications. As a consequence, the words they contain are a mixed bag. "Enjoyable", "original", "nice", and "solid" are all in there; the same can be said for the omnipresent term "fun", which once again comes out as the winner in both cases as far as sheer frequency is concerned.

At the same time, though, we get words that likely refer to problems these games contain, like "bad", "lack", "fails", "bland", "disappointing", "dull" and a few others.

The difficulty here lies in finding a visible difference between the two groups. How are games that got 65 different from those that got 60? Are there actually any distinctions between them or is this a case that proves that scores are stupid [5] and arbitrary?

It is hard to say for now, but we will see how our model will perform in these situations.



The final analysis that will be done here has to do with one path of sentiment analysis that is not explored in this work and that can be used as another way to predict scores.

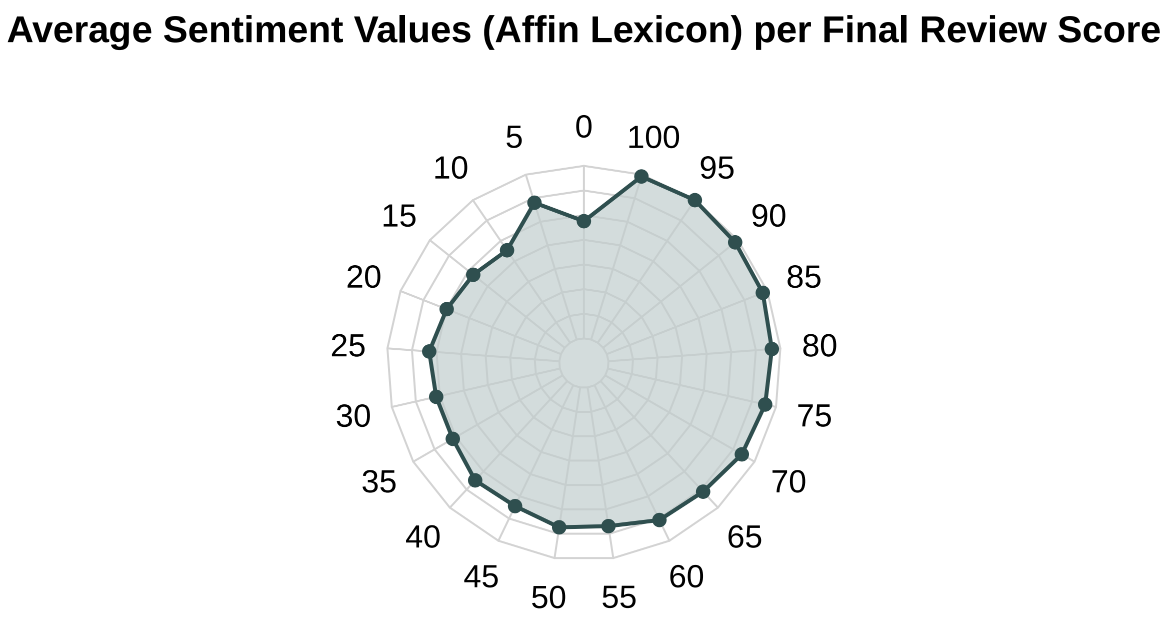
In R, there are a few libraries that present lists where every word is related to a number, and this value tries to capture how positive or negative the term in question is. Given that feature, it is possible to assume that the more positive a review is, the higher the total value of its words will be. As such, this value could be indicative of score, at least theoretically.

As mentioned, the goal of doing this here is not to predict the rating, as that will be done by a Machine Learning model, but by performing this analysis we want to know if there is any correlation between the score given by a reviewer and the value of the words as calculated by one of these dictionaries. If that is the case, maybe the chances our classifier has of being precise in its job will be higher, as that would mean words and scores are linked in a smooth comprehensible way.

For this next plot, we use the AFINN lexicon, found in the *tidyr* package. This lexicon attaches to each word a value between -5 and 5. We process each word of each review through this dictionary and calculate the final value of each excerpt averaged according to the amount of words contained in the text. After that, we calculate the average sentiment present in the reviews of each score.

Note that, for the sake of having a scale that starts at 0, we add five to each value indicated by the lexicon, hence turning -5s into 0 and 5s into 10s.

What we see as the results is that outliers exist, but the correlation is there. The average sentiment present in reviews of rating 5 is higher than those observed in the entire range from 10 to 55; an error that might have to do with the fact 5s are quite rare. Likewise, reviews that generated 0s have more positive feelings than those of excerpts coming from texts that yielded 10s and 15s; 40s are more positive than 45s, even if the difference is quite tight; 50s are more positive than 55s, by another close margin; and the scoring average which produced the best sentiment was actually the one for 95s, not 100s.



Average Sentiment Values (Affin Lexicon) per Final Review Score

Intuitively, these little correlation failures are partially expected in an area that is so subjective, but they give hope for our model, especially when it comes to detecting gaps between grades that are 5 points away from each other; differences that were not very visible when we looked at the Word Clouds. As far as our sentiment evaluation here goes, 85s are indeed slightly more positive than 80s, 65s overcome 60s; 95s beat 90s; among a few others.

7. Experiments

Throughout this section, the experiments that were performed with our selected models will be described. Before getting into actual results, however, we will look at how the text will be further prepared in order for the learning to take place, how we will measure the performance of our raters, and what models will be used.

Before advancing, it is important to mention all metrics reported were produced by evaluating the trained models on the validation set. This task was not done via cross-validation because doing so demands more processing time, which would have reduced the scope of the experiments.

Besides, it is also worth noting that, before all values for RMSE were calculated, the scores yielded by the trained models were adjusted to fall into our scale of 5-point increments, so they were all rounded to their closest neighbor that respected the scale.

7.1 From Text to Vector

Given they are mathematical algorithms, models cannot learn from words themselves. For that reason, as it is always the case when either classification or regression is done based on text, documents need to be turned into vectors. In this experiment, that task is performed by TF-IDF, the acronym for Term Frequency–Inverse Document Frequency [13]. Through it, our train dataset will become a large matrix where every row corresponds to a document and every column to a word; more precisely, to every word of our training vocabulary. As such, we end up with a matrix that has 99,814 rows and 18,544 columns

Essentially, every cell of the matrix will contain the number of times a given word appears in a given document. TF-IDF, though, brings a twist to this calculation, one that is shown in the following formula.

$$w_{i,j} = tf_{i,j}log(\frac{N}{df_i})$$

The weight (the value of the cell) of term i in document j is given by how often that term appears in that document (tf), times the log of number of documents N divided by the number of documents that contain the term, *df*.

The options of methods that turn documents into vectors are many: TF-IDF itself has numerous variations [14] and more advanced techniques such as Word2Vec [15] have emerged in recent years. TF-IDF, however, was picked not just due to its widespread usage and good results, but also because we opted not to remove stopwords from the text. TF-IDF, by taking into account how frequent the word in question is throughout the dataset, greatly helps reduce the weight these terms have.

In R terms, this process from text to vector was executed via the *text2vec* package [16]. The code below, fully commented with explanations of all steps, details how this task was performed whenever a model was about to be trained.

```r
# Now we are ready to train our first model. Before doing so, some extra text
# preparation has to be done. The first part is to tokenize the text (separate
# it into individual words). Here, the function that will do the tokenizing
# is instanced.
tokenizer_function <- word_tokenizer

# The function tokenizes the text of the training reviews.
train_tokens <- tokenizer_function(train$review)

# Next, we create an iterator that will read these tokens one by one and
# transform them. We also tell it the column that uniquely identifies all
# reviews so it knows which transformed results belong to whom.
iterator_train <- itoken(train_tokens,
                         ids = train$reviewId,
                         progressbar = FALSE)

# The iterator is used to create a vocabulary with all words that are present in
# all the training reviews.
vocabulary <- create_vocabulary(iterator_train)

# The vocabulary is then fed into a vectorizer. This guy will turn the tokenized
# texts into vectors.
vectorizer <- vocab_vectorizer(vocabulary)

# Here we get our Document-Term matrix. Each review is a row, and each column
# represents a word. Each row-column is filled with the number of times
# the word appears in the review.
document_term_matrix_train <- create_dtm(iterator_train, vectorizer)
```

```r
# We, however, do not want to simply get the counts of when all words appear
# in all reviews. We want a normalized number that will be influenced by how
# frequent the word is in all texts. TF-IDF will give us that, so we run it.
model_tf_idf <- TfIdf$new()
document_term_matrix_train <- model_tf_idf$fit_transform(document_term_matrix_train)
```

7.2 Measuring Performance

Here, it is important to reiterate that the matter of measuring the performance of our Universal Rater is troublesome. Given what we set out to accomplish, we want our model to be as unbiased as possible: we want it to rate reviews exclusively on the content they carry (that is, their words). As such, building a model that achieved absolute perfection according to a certain metric would not correspond to our goal. After all, if it did so, the Universal Rater would not be unbiased; it would have merely learned to model the biases of every single reviewer.

Yet, Machine Learning models work by trying to minimize error measures, and consequently there is no choice but pick one metric, work towards reducing it as much as possible, and measuring the ultimate success of our model through it.

Given this is a regression task, because we are trying to calculate a number based on a bunch of features (words), the metric that was chosen was RMSE, defined by the formula below.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(predicted_i - actual_i)^2}{N}}$$

Basically, it is the squared average of the differences (to the second power) between the predicted and the actual ratings, and it will be the one used throughout the experiments to select the best models.

7.3 Evaluated Models

The models selected to learn from our dataset of review excerpts were two:

- A tree-based model;
- A linear function.

Both of them, however, were executed through the XGBoost framework. Introduced in 2016 by Tianqi Chen [17], this method quickly grew to fame by winning multiple Machine Learning competitions [18], especially in its tree-based variant, and it has been used extensively ever since, including in text-related problems [19].

Boosting is a technique that has been around for quite a while [20], and it essentially consists of combining multiple weak models in order to build a strong one by aggregating their predictions. Rather than being built at once, these models are produced iteratively: at every round, a weak learner is added to the solution and tries to make predictions on the input data. This data, however, is re-weighted at every iteraction according to the results of the previous one. Records that generated errors gain weight, while those that were accurately evaluated lose weight.

As such, every model that is added to the pile is different thanks to how they learn from re-weighted data, and it is their combined effort that produces the final results. After the iterations are over and the model is trained, new data comes and each individual learner predicts what class or number (in our case, the score) the data should have. The final result is given by aggregating the votes of all models, usually via some mechanism of weighted voting based on the accuracy each learner presented during training.

XGBoost is part of the family of Gradient Boosting, hence the G in its name. Boosting algorithms of that kind try to minimize the error (in our case, measured by RMSE) by, at every iteration, picking a function that points in the negative direction of the error curve, something that is achieved through calculating a derivative.

Meanwhile, the *X* seen in the name of XGBoost comes from the word "extreme". The boosting algorithm it implements comes with numerous changes over the traditional methods of Gradient Boosting [21], which boil down to:

- Using the second derivative, rather than the first, to make a decision on which path to follow along the gradient curve;

- Combining L1 and L2 regularization to avoid overfitting and cut down on model complexity;

- Customizing its loss function so that it punished complexity.

Other than its performance, the choice of XGBoost here was also influenced by its light weight, as it is known to be quite efficient and run fast [22]. Since there were hardware limitations to performing these experiments and the dataset was relatively large, XGBoost was a perfect fit.

In its R implementation [23], XGBoost has multiple solvers (that is, the model it builds) and parameters. The tree was picked because it is generally accepted to be the strongest of the options; meanwhile, the linear function suited the regression problem at hand. Both have multiple parameters, but the description of those will be left to the tuning experiments, in which the ones that we tried to adjust will be discussed.

7.4 Experiment 1 - Building the Baselines

The first experiment conducted was also the simplest one. The XGBoost Tree and the XGBoost Linar Model were setup with their standard parameters, as seen in the R implementation of the algorithm [23], and trained with the appropriate dataset. The only additional configuration that was done had to do with setting up the parameter *Nround*. It is through it that we indicate how many weak models the boosting algorithm will produce; in other words, it dictates the number of learning iterations that are performed.

For this experiment, *Nround* was set to 500. After being trained, the models were executed over the validation dataset, and RMSE was reported by comparing their predictions with the actual scores contained in there.

The following chunk of code, fully commented with explanations, indicates what was done.

```
# Here, we train the first of our baseline models via the library XGBoost.
# It is a linear model, but with a twist, as it uses an extreme boosting
# technique (hence the name) to create a lot of weak models and combine their
# results into a - hopefully - good one. We also set the seed so the experiment
# can be reproduced and yield the same results. The metric we are trying to
# minimize is RMSE.
set.seed(1, sample.kind="Rounding")
linear_model <- xgboost(data = document_term_matrix_train,
                        label = train$score,
                        booster = 'gblinear',
                        nround = 500,
                        objective = "reg:squarederror")
```

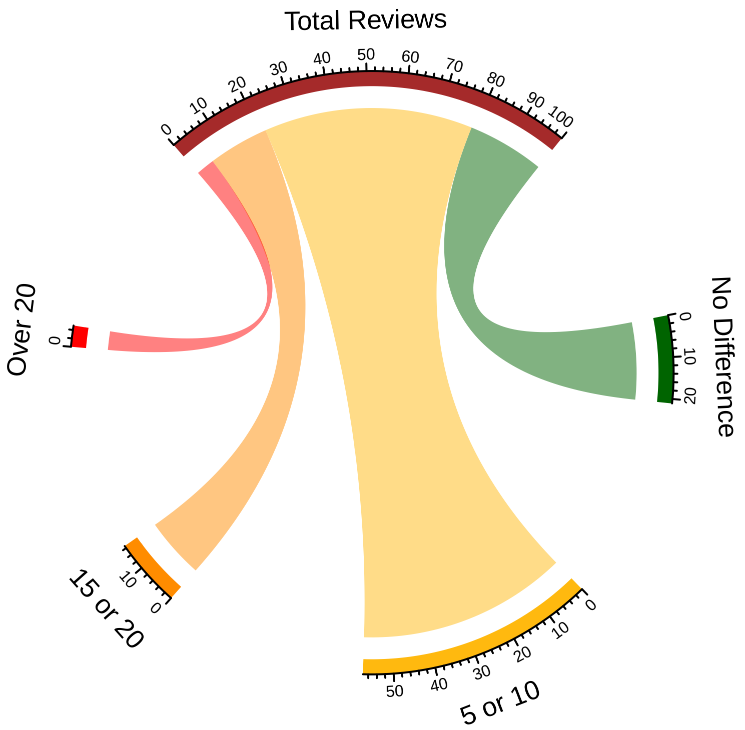
```
# We set the seed again and train our second baseline model. A regression tree
# (actually many regression trees), that are being built via the XGBoost technique.
set.seed(1, sample.kind="Rounding")
tree_model <- xgboost(data = document_term_matrix_train,
                      label = train$score,
                      booster = 'gbtree',
                      nround = 500,
                      objective = "reg:squarederror")

# Before we do the validation (that is, applying the models built in the
# previous section to the validation dataset to see how well they will
# perform), we need to tokenize/vectorize the text of the validation reviews.
# So the same process done for the train set is executed here.
validation_tokens <- tokenizer_function(validation$review)

iterator_validation <- itoken(validation_tokens,
                              ids = validation$reviewId,
                              progressbar = FALSE)

document_term_matrix_validation <- create_dtm(iterator_validation, vectorizer)

# There is one difference, though, and this is it. The TF-IDF model that was fit
# to our train set is reused. This is done so the rows (reviews) of the validation
# set have the same columns (words) as the ones in the training set. Otherwise,
# the models would not work, since the dimensions of the training and validation
# matrices would be different. Plus, reusing this TF-IDF here allows information
# on the frequency of words in training data to be used to normalize validation data.
document_term_matrix_validation <-
  model_tf_idf$fit_transform(document_term_matrix_validation)

# The matrix is ready, so we generate the predictions with the linear and tree model.
predictions_linear <- predict(linear_model, document_term_matrix_validation)
predictions_tree <- predict(tree_model, document_term_matrix_validation)

# We adjust the generated scores so they are all between 0-100 and also
# respect our scale of 5-point increments. This is done because our models
# generate continuous numbers, and we want our rater to give scores with
# 5-point increments.
predictions_linear <- sapply(predictions_linear, adjust_scores)
predictions_tree <- sapply(predictions_tree, adjust_scores)

# We obtain RMSE.
rmse_linear <- RMSE(predictions_linear, validation$score)
rmse_tree <- RMSE(predictions_linear, validation$score)
```

The overall results of the experiment can be seen in the table below. As it is possible to observe, the tree-based model outperformed the linear one by a margin of approximately 0.8 as measured by RMSE.

Model	**RMSE**
XGBoost Linear Model	12.35462
XGBoost Tree	11.25177

Meanwhile, in the chart below we report just how accurate we were when it comes to predicting ratings in the validation dataset. To create it, the predictions of the tree-based model were used.

Margins Between Actual Scores and Predicted Scores − Validation

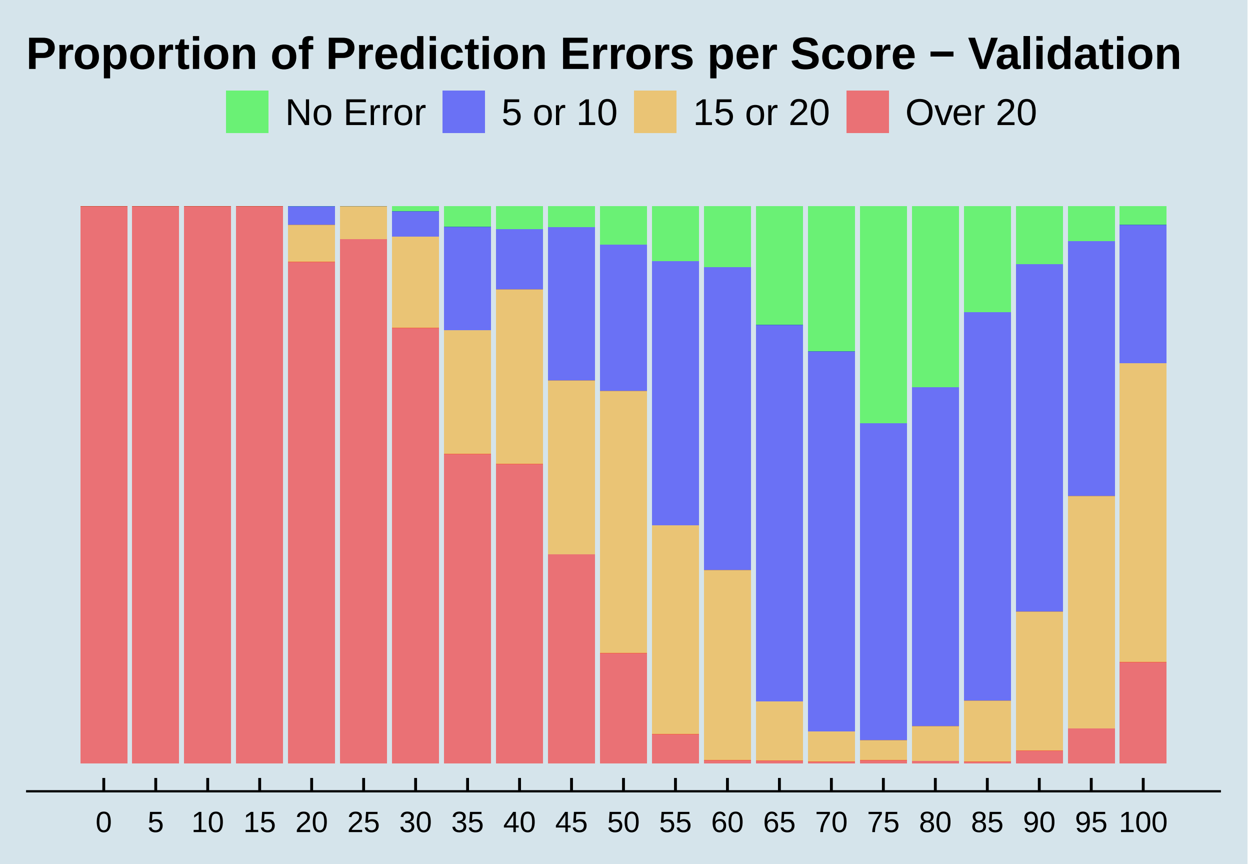


It is possible to see total accuracy was achieved for about 20% of the validation dataset, which contains 11,094 rows. Meanwhile, about 55% of our model's predictions feel within 5 or 10 points away from the actual score. The remaining predictions contained errors that were 15 points or more off the mark; fortunately, only about 5% of these missed by more than 20, meaning that the majority of our larger errors were within a relatively small scale.

One important matter to look into here is the nature of the reviews where our predicted ratings presented mistakes. Our dataset is, after all, imbalanced, as it contains far more positive reviews than negative ones. Were our model's mistakes possibly related to that lack of negative reviews or did it have similar error rates regardless of the positivity or negativity of the text?

If the previous plot had made it all look pretty good for our model, since it is only missing by more than 20 points for about 5% of the reviews, the next plot reveals its struggle with negative texts and shows the imbalanced nature of our dataset has affected it. The image shows the proportion of types of misses for each possible rating, and for all scores that are below 45, it is more common for our Universal Rater to miss by over 20 than not to do so.

From 45 onward, the reality shifts and the rater starts becoming more precise, to the point where there are almost no misses by more than twenty in the whole range from 60 to 85. As such, while it is quite good that our model is getting 75% of the validation dataset either right on the money or within 10 points, the fact it struggles with very negative reviews, a type of text that is not very present in our dataset, cannot be overlooked.



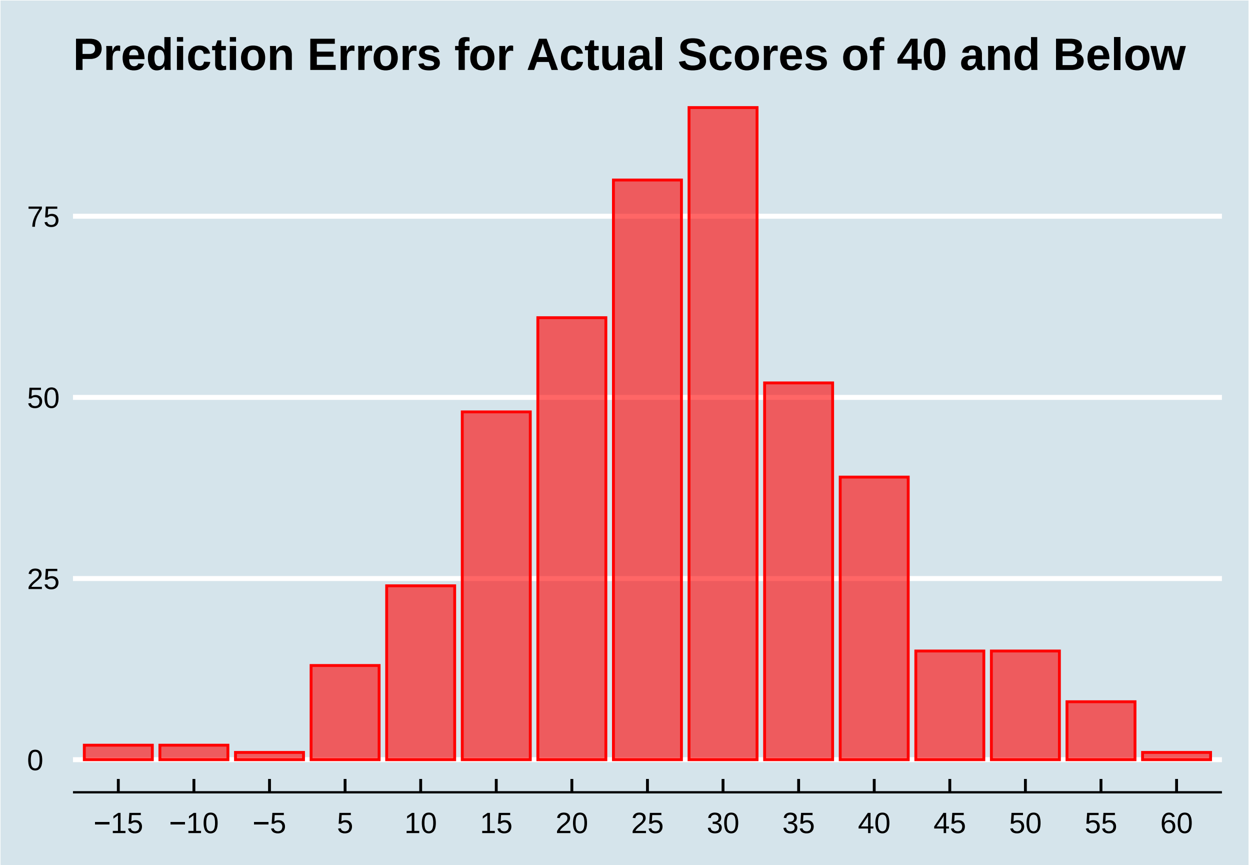
Given the data above, one question that can be raised about the mistakes our model is making relates to the direction of those errors; in other words, when analyzing negative reviews, is our model overrating bad games or underrating them? Theoretically, one could even bet that it is likely to be giving grades that are higher than the actual ones; after all, our dataset has an imbalance towards the positive so our model may have acquired a similar type of bias.

The next and final chart of this experiment looks into this matter. And, in the end, it indeed confirms intuition. In it, we plot the value of the errors that were made for reviews whose actual scores were 40 or less, and count how often they happened.

As it turns out, when it makes a mistake with these very negative reviews, the most common decision by our Universal Rater is to overrate these games by 30, with this value of error showing up more than 75 times. As seen in the previous chart, for negative reviews tight misses are rare, and those that range from -10 to 10 are indeed an uncommon sight.

Perhaps reviewers are not being verbally harsh enough with those games to let the negativity seep through, but that is really an unlikely scenario since our analysis of the values attributed by the AFFIN lexicon to each of our scoring bins showed that a gap exists. It is probable that our Universal Rater is simply biased towards positivity.

Can we somehow improve on that or will our model simply keep having trouble with negative reviews? In the next two experiments, we will refine it and see if that tendency can be controlled.



7.5 Experiment 2 - Feature Selection

Aside from being imbalanced, one of the main traits of our dataset is that it has many features. Originally, we started with 42,186 words, but after noticing more than half of them were not too common, we opted to cut those that appeared three times or less, leaving us with a total vocabulary of 18,544 terms.

It is hard to say what number of features (in our case, words) qualifies as too many [24], but could we possibly get better performance by diminishing them? Is it possible to work with fewer words and achieve better results?

Occasionally, the benefits of feature selection are debated [25], but at the same time, the list of benefits can be numerous [26], from improving accuracy to solving problems with imbalanced datasets [26]. In this second experiment, we try to improve on the results found in the previous one by removing features from our data.

There are numerous methods through which one can execute feature selection, from the chi-square test [28] to other statistical evaluations of the kind. At their heart, the idea is the same: identify which features are not important to distinguish between the classes or values a model is trying to predict, and cut them out so that the algorithms can work with a leaner set of features. In this experiment, we take advantage of the values returned by our models themselves.

After a set of training iterations, both the XGBoost Linear Model and Tree return lists with how important each feature (word) was to their regression task: the first does so via the weights it attributes to each word and the latter gives us that information through the Gain statistic [29], which essentially measures how much doing splits of nodes through that feature improved on the purity of the generated sets.

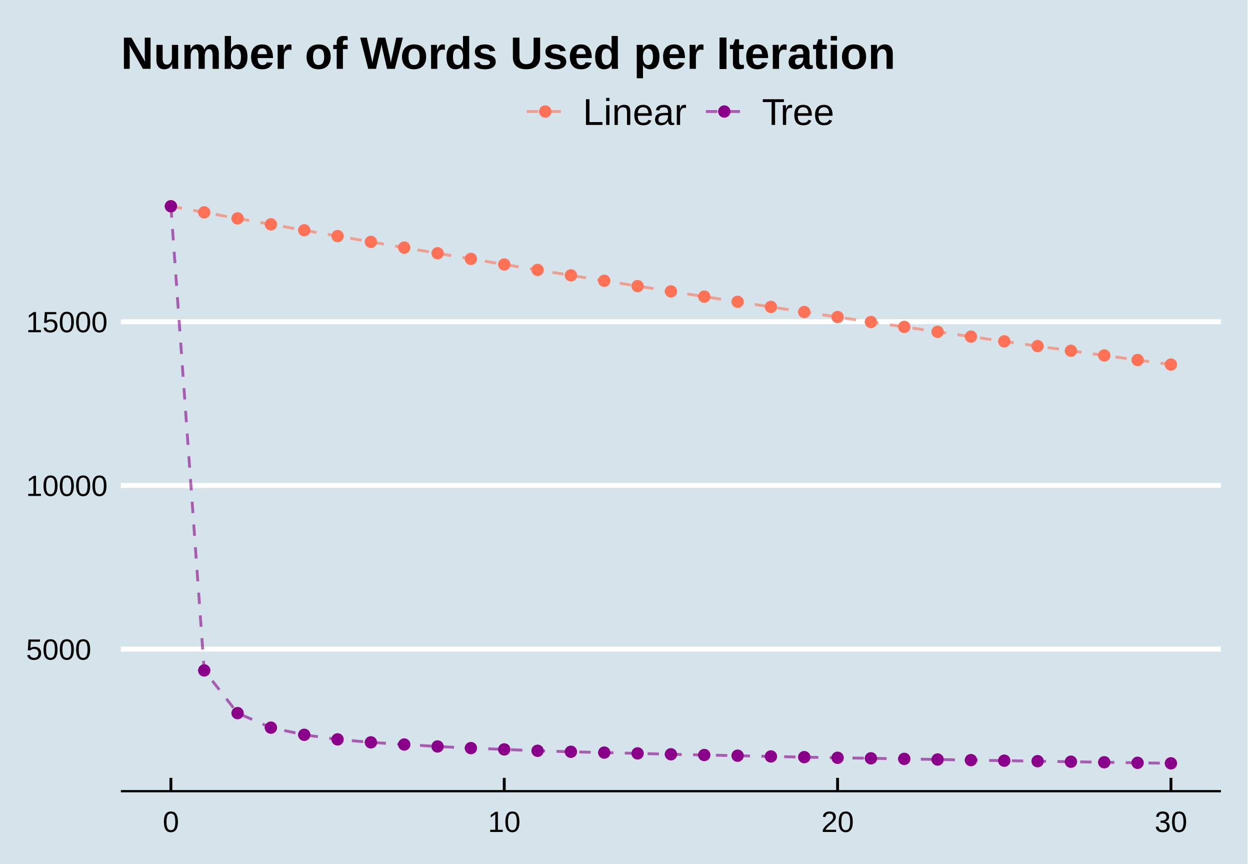
With those details established, we can begin to look at the second experiment.

The processing steps consisted of the following and were repeated thirty times:

- Training the two models with the standard parameters and 500 iterations.
- Obtaining the list of features (words) according to their importance: the weights of the linear model and the gain of the tree.
- Evaluating the models on the validation set and recording their achieved RMSE.
- Excluding the bottom 1% features as ranked by importance, and running a new training and validation round with the reduced vocabulary.

In a way, the first iteration of this process happened in Experiment 1 itself, when the models were executed with the full vocabulary: 18,544 words. When that was done, the list of features by importance was obtained. As such, this experiment starts with that complete vocabulary minus the bottom 1% terms.

Two details are important to highlight here. First, each model determined what its bottom 1% features were; as such, in this experiment, the two models are working with distinct vocabulary sets. After all, a word listed by the linear algorithm as unimportant might have been evaluated in a brighter light by the tree-based model. Secondly, while the linear model returns weight values for all terms, the tree-based one actually just returns the Gain statistic for the terms it used. As such, while in the case of the former only the 1% least important features were cut with every iteration, in the case of the latter we opted to exclude both the features that were not listed and the bottom 1% of those that were. Because of that, especially at first, the cutting of features was far more drastic for the tree-based model.



The previous plot shows just how extreme that drop was. From iteration zero, which happened in Experiment 1 with the full vocabulary for the two models, to the first training round with a reduced number of words, the tree-based model goes from working with 18,544 terms to training with less than 5,000: most of those words simply did not appear in the list of the Gain statistic because at no point were they used to split the tree nodes. In fact, through the first few iterations it seems the tree model actually cuts more than 1% of the words. Meanwhile, as previously discussed, the linear XGBoost cuts its vocabulary steadily all the way down, continuously throwing out 1% of the words until the iterations end.

For the sake of transparency, the commented code for this elimination process can be found below, with the steps explained one by one. Note that this is just part of the experiment's code. For brevity, we only showcase the portion concerned with eliminating the words from each model's vocabulary, given the training and validation processes are not too different from the ones shown in Experiment 1.

```r
# In order to select the most valuable features (terms) used by the model to
# make its decision, we will need to pull the "importance" rates out of them.
# This function does so for both models that will be used, the Linear one, which
# measures such importance in "Weights", and the Tree Model, which measures
# such importance in Gain.
obtain_importance_terms <- function(model, vocabulary_model) {

  # The importance of all used terms is obtained via the xgb.importance function.
  importance <- xgb.importance(feature_names = vocabulary_model$term, model = model)

  # The value of the importance will be given either by the "Weight" or "Gain"
  # column, depending on the model. Here, that column is renamed "value".
  importance <- importance %>%
    mutate(value = {if("Weight" %in% names(.)) abs(Weight) else Gain})

  # The column Feature is renamed to term.
  importance <- importance %>% mutate(term=Feature) %>%
    select(term, value)

  # The list of terms x importance is returned.
  importance
}

# Here, we eliminate words from the vocabulary that will be used by the models.
# Given the two models (trained with all the terms in the previous experiment)
# return the importance of the terms for the decision they made, we order
# (descending) by those values (Gain for the tree and Weight for
# the linear regression). We then calculate how many terms correspond to 1% of
# the vocabulary and take them out.
# It is, however, important to note that while the linear model returns
# weights for all terms used, the tree reports Gain only for terms that
# were employed in creating the branches. As such, for the trees, the
# elimination  being done is far more aggressive, as it cuts out not just the
# bottom 1% but also all terms that were not used.
vocabulary_to_keep_linear <- importance_terms_linear %>% arrange(-value)
threshold_index <- nrow(vocabulary_to_keep_linear) - (nrow(vocabulary_to_keep_linear)/100)
vocabulary_to_keep_linear <- vocabulary_to_keep_linear[1:floor(threshold_index),]

vocabulary_to_keep_tree <- importance_terms_tree %>% arrange(-value)
threshold_index <- nrow(vocabulary_to_keep_tree) - (nrow(vocabulary_to_keep_tree)/100)
vocabulary_to_keep_tree <- vocabulary_to_keep_tree[1:floor(threshold_index),]
```

```r
# Let's see how the models do with 1% (or more, in the case of the tree) less
# vocabulary than the last time around. We are creating new tokenizer and
# vectorizer functions here. It is the same procedure done in Experiment 1.
tokenizer_function <- word_tokenizer
train_tokens <- tokenizer_function(train$review)
iterator_train <- itoken(train_tokens,
                         ids = train$reviewId,
                         progressbar = FALSE)
vocabulary <- create_vocabulary(iterator_train)

# Here comes the difference. At this point, the vocabulary variable (which
# will determine the columns of our Document-Term matrix) has all words
# of the train dataset (that is, the full vocabulary). We want to cut terms
# that are in the bottom 1% or, in the case of the tree, those in the
# bottom 1% plus those that were not used. Here, that exclusion is done.
vocabulary_linear <- vocabulary[(vocabulary$term %in% vocabulary_to_keep_linear$term), ]
vocabulary_tree <- vocabulary[(vocabulary$term %in% vocabulary_to_keep_tree$term), ]

# We vectorize with the new, reduced, vocabulary.
vectorizer_linear <- vocab_vectorizer(vocabulary_linear)
vectorizer_tree <- vocab_vectorizer(vocabulary_tree)

# We create the Document x Term matrices. Since the vocabularies of the models
# are, at this point, different, we need two of everything.
document_term_matrix_train_linear <- create_dtm(iterator_train, vectorizer_linear)
document_term_matrix_train_tree <- create_dtm(iterator_train, vectorizer_tree)

# For example, we also need two TF-IDF models, because the normalization will
# be different.
model_tf_idf_linear <- TfIdf$new()
document_term_matrix_train_linear <-
  model_tf_idf_linear$fit_transform(document_term_matrix_train_linear)

model_tf_idf_tree <- TfIdf$new()
document_term_matrix_train_tree <-
  model_tf_idf_tree$fit_transform(document_term_matrix_train_tree)
```

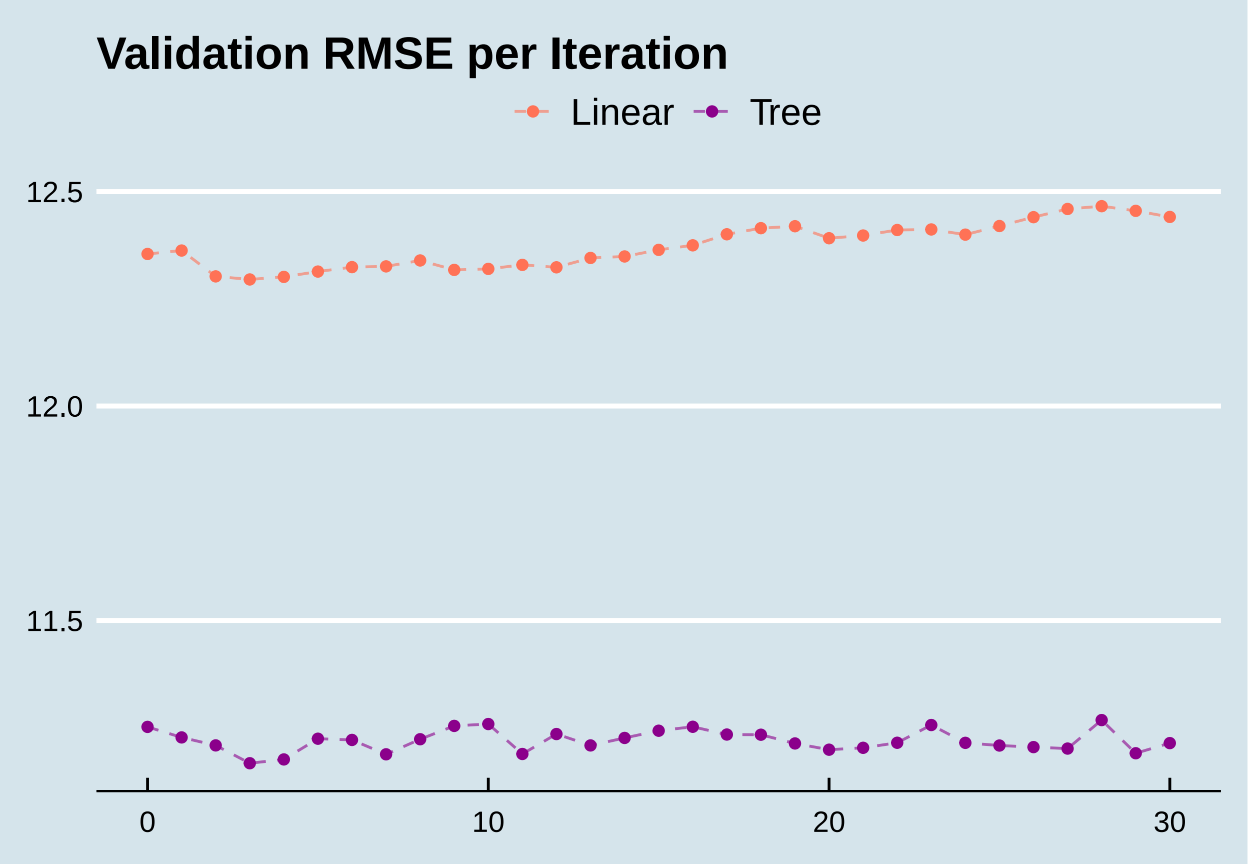
After training the models, we obtain the most important features of each by calling:

```r
importance_terms_linear <- obtain_importance_terms(linear_model, vocabulary_linear)

importance_terms_tree <- obtain_importance_terms(tree_model, vocabulary_tree)
```

With so many words being cut, how did the tree-based model do? And what about the linear one? Since its feature removal was lighter, was it able by any chance to close the gap to its most popular brother and maybe even overcome it? The following plot answers those questions, and, when looking at it, it is important to remember that - for both - the value of iteration zero corresponds to the outcome of Experiment 1, when all of the initial terms were available for training.

What we see is that, as far as RMSE goes, no brutal changes occurred. For the linear model, there is actually a tendency towards better performance at first, but as iterations go along and vocabulary is cut, its precision starts to decay and the tendency observed is the opposite. For the tree model, the scenario is a bit foggier, as it oscillates between finding better performances than that of Experiment 1 and worse ones as well.

However, for the two models we can conclude that, even if no considerable improvements can be seen, at some point both were able to get to a lower RMSE threshold with fewer terms, which shows that even if it doesn't necessarily reduce error, feature selection can help build lighter models when the situation calls for it.



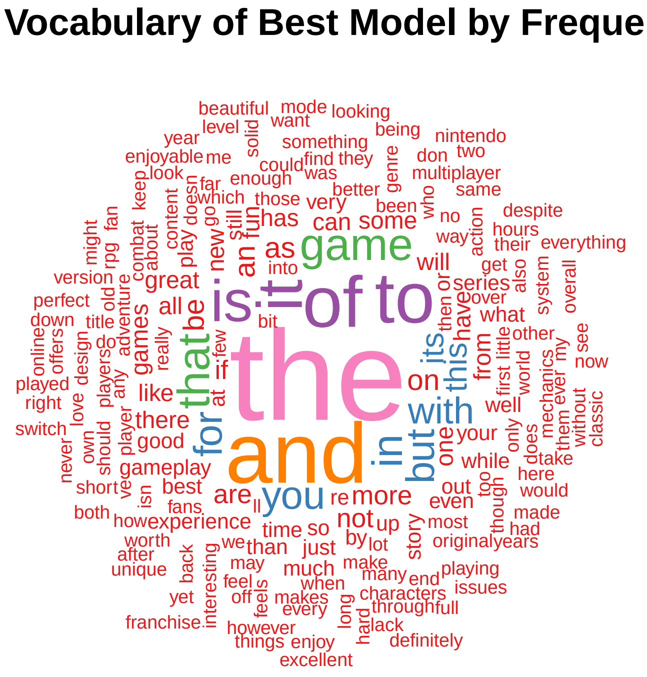
The next table brings a neat summary of the results encountered in Experiment 2, showcasing at which point each model did best, worst, and with how many terms those outcomes happened. Some similarities can be pointed out here. Curiously the best and worst results of each model came at the same iterations: 3 and 28 respectively. Likewise, their worst performances yielded RMSE values that were above those of Experiment 1, meaning that at some point the removal of words was detrimental to the performance, even if not by much.

Most importantly, we can see that the tree-based model achieved slightly better results than it did in Experiment 1 with far fewer words. Dropping down from the original RMSE value of 11.25177 with 18,526 terms, it was able to now reach 11.16703 with only 2,603. As such, although the results here are not significant in terms of RMSE improvement, at least they are quite visible when it comes to feature exclusion.

	XGBoost Linear Model	**XGBoost Tree**
RMSE Experiment 1	12.35462	11.25177
Original Vocabulary Size	18,526	18,526
Best RMSE Experiment 2	12.29519	11.16703
Best Vocabulary Size	17,974	2,603
Best Iteration	3	3
Worst RMSE Experiment 2	12.46602	11.26768
Worst Vocabulary Size	13,971	1,575
Worst Iteration	28	28

As the final observation of this experiment, we will not look too deeply into how the model with less features did on the validation set: we will leave that performance neatly summarized to RMSE given we have yet to tune any parameters. However, it is worth taking a look at how our Word Cloud of global training set vocabulary looks after all of these removals. What words remain? What are the most common ones?

Curiously, what we see is that among the 2,603 words employed by our best model so far, the stopwords still appear. Intuition would lead one to believe these terms have no value and, therefore, would be cut by our models, especially from tree-based one, which had - thanks to its nature - a more aggressive rate of term exclusion. Yet, these apparently meaningless words remain in there, showing that sometimes out intuition is deceiving and algorithms can see value where we do not see any.

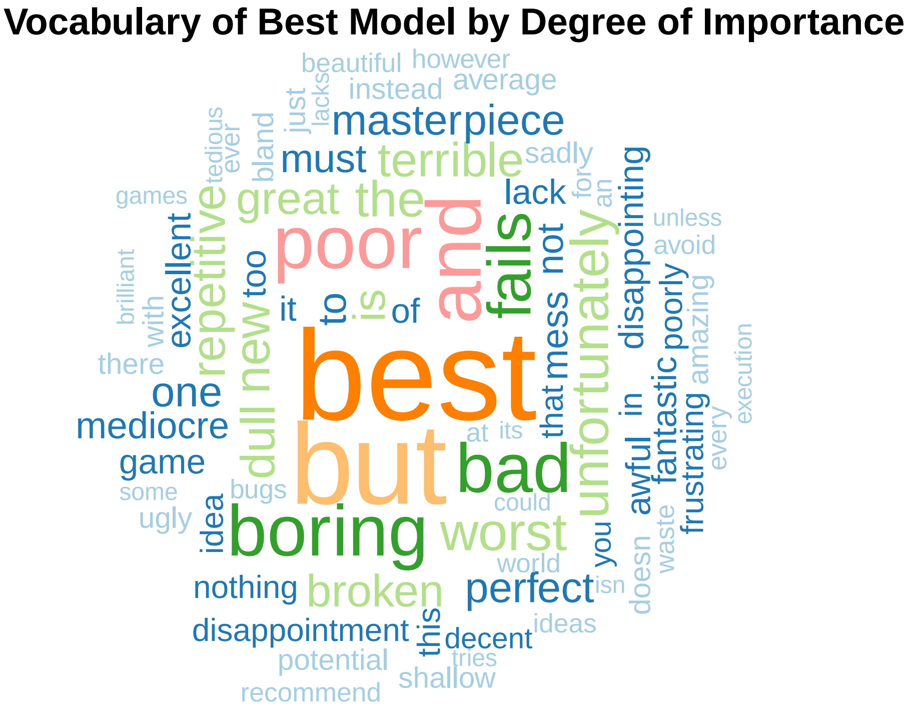


Vocabulary of Best Model by Frequency

As an implementation comment, it is worth noting words seen by the model as not useful for the regression process were not actually excluded from the training dataset. Even after selecting only the best ones, they remained in the treated review excerpts. After this moment, what was done was that the vocabulary of our best model was, instead, passed to the tokenizer and iterator of the training model whenever the Document x Term matrix was being created. Consequently, terms that did not belong to the vocabulary were merely ignored.

At last, when it comes to how important the words in this refined vocabulary were to the model, we present a final Word Cloud. In it, rather than the usual procedure of plotting words so that their size matches their frequency, we setup the image so the sizes actually correspond to their importance as dictated by the Gain statistic obtained from the tree-based model, and we limit the image to the 80 most important words.

Surprisingly, “and” as well as “the”, which are a stopwords whose natures appear to be neither negative nor positive, have considerable importance for our model. “But” is another stopword that has high value; however, in its case, it is possible to see there could be a negative connotation to the term, one that allows, for instance, the rater to separate between 100s and 90s. Nevertheless, this particular cloud reveals that, ultimately, words loaded with negative or positive meaning rise to the top and are key in indicating the rating a game will get.


Vocabulary of Best Model by Degree of Importance

Finally, from this point onward, as we move on to tuning our model, a choice was made to drop the linear algorithm due to its performance. Since we are going to look for the individual parameters that generate the best results, it was concluded that efforts would be better used if focused solely on the tuning opportunities that exist in the tree-based model.

Furthermore, since this experiment showed that, although slightly, the tree-based model reached its best results when working with a vocabulary of 2,603 words, it is with this set of words that we move on to the next task. As such, looking to optimize the rating model as much as possible, we will seek the best parameters for the tree algorithm using 2,603 words.

7.6 Experiment 3 - Tuning

In this third, and final, experiment, we go through the process of tuning the tree-based model. As previously mentioned, there are a number of parameters [23] that can be adjusted in the search to get the best performance out of the algorithm and, as such, there are various orders in which the process could be carried out. A choice was made, though, to follow the order described in this guide [30]. As mentioned by it, a good rule of thumb to follow when tuning not only XGBoost but also other models is to first take care of the parameters that relate to the general learning procedure, then attack the ones that tend to yield the largest gains, only to then fine-tune those that concern minor, yet still relevant, details.

Consequently, the parameters that will be tuned in this experiment, in the order in which they will be evaluated, are:

- **Nround**: As already mentioned, this parameter controls the number of iterations the boosting algorithm goes through; in other words, the amount of weak learners it creates. We had originally set it to 500, but will try distinct values here.

- **Max_Depth** and **Min_Child_Weight**: Evaluated in conjunction, these parameters dictate the structure of the tree. *Max_Depth* determines the maximum depth the tree can have, which is the number of nodes from the root to the leaf. Meanwhile, *Min_Child_ Weight* establishes the minimum number of instances needed in each node; as such, if the partitioning of a node results in other nodes with less items (in our case, reviews) than the set number, it simply gives up further partitioning. In a way, these two parameters, respectively, control how complex and conservative our tree will be.

- **Gamma**: This is yet another parameter related to whether the trees generated by XGBoost will be more or less conservative. In the case of *Gamma*, it establishes the reduction in loss that must be achieved for a node to be partitioned. As such, the larger it is, the harder it is for the algorithm to justify partitioning a node; it has to show some pretty considerable gain for that to happen.

- **Subsample** and **Colsample_Bytree**: These parameters will also be defined in conjunction, as they are related to the same aspect, which is how much of the data will be employed when the trees are created. For every boosting iteration, the data that is used to train the trees is not the full training dataset, but only part of it. This is done to avoid overfitting. Consequently, while *Subsample* indicates what percentage of the records in the dataset will be used for that, *Colsample_Bytree* determines the rate of columns (in our case, words) that will be considered

- **Lambda**: It is the regularization parameter applied on the weights. The higher it is, the less the values of the columns (as determined by TF-IDF) will have effect on the model, especially if they display a noisy far-from-the-average nature.

Since, for the first two experiments, the standard XGBoost values for these parameters were used, with the exception of *Nround*, for which a number has to be informed, the configuration for the parameters that will be evaluated here was, up to this point, the following:

Nround	**Max_Depth**	**Min_Child_Weight**	**Gamma**	**Subsample**	**Colsample**	**Lambda**
500	6	1	0	1	1	1

For both *Subsample* and *Colsample_Bytree*, it is important to note that 1 is equivalent to 100% of the data and columns.

In all cases of this experiment, the process of tuning was the same:

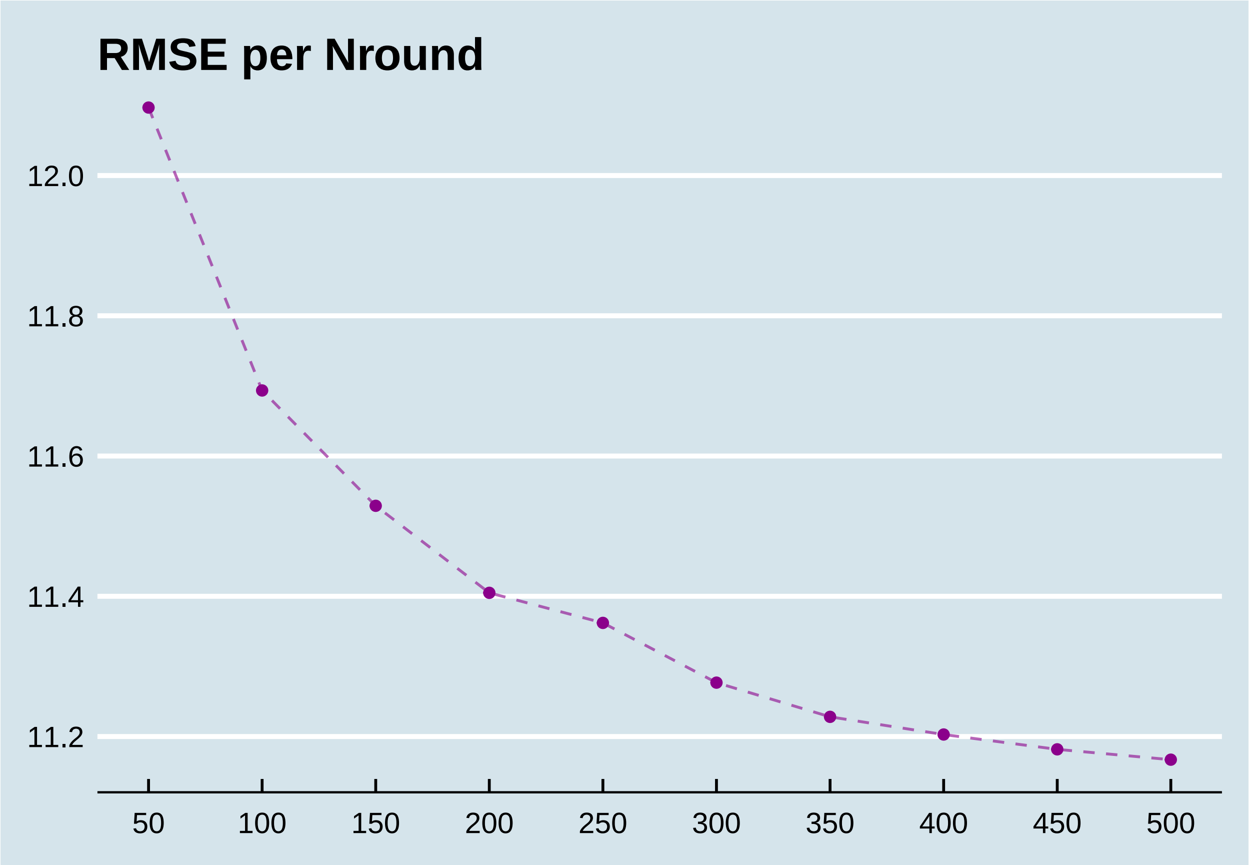
- A certain number of values to be tested was specified for the parameter or set of parameters;

- The model was trained (over the training dataset only) by using all of the values;

- For each value, predictions with the trained model were done on the validation dataset and RMSE was reported;

- The value that generated the best RMSE was selected and, from that point forward, the parameter evaluated was always configured with that winning value.

Given there are five parameters or sets of parameters to be evaluated in this experiment, we will show five charts to illustrate the performances obtained by the tested values.

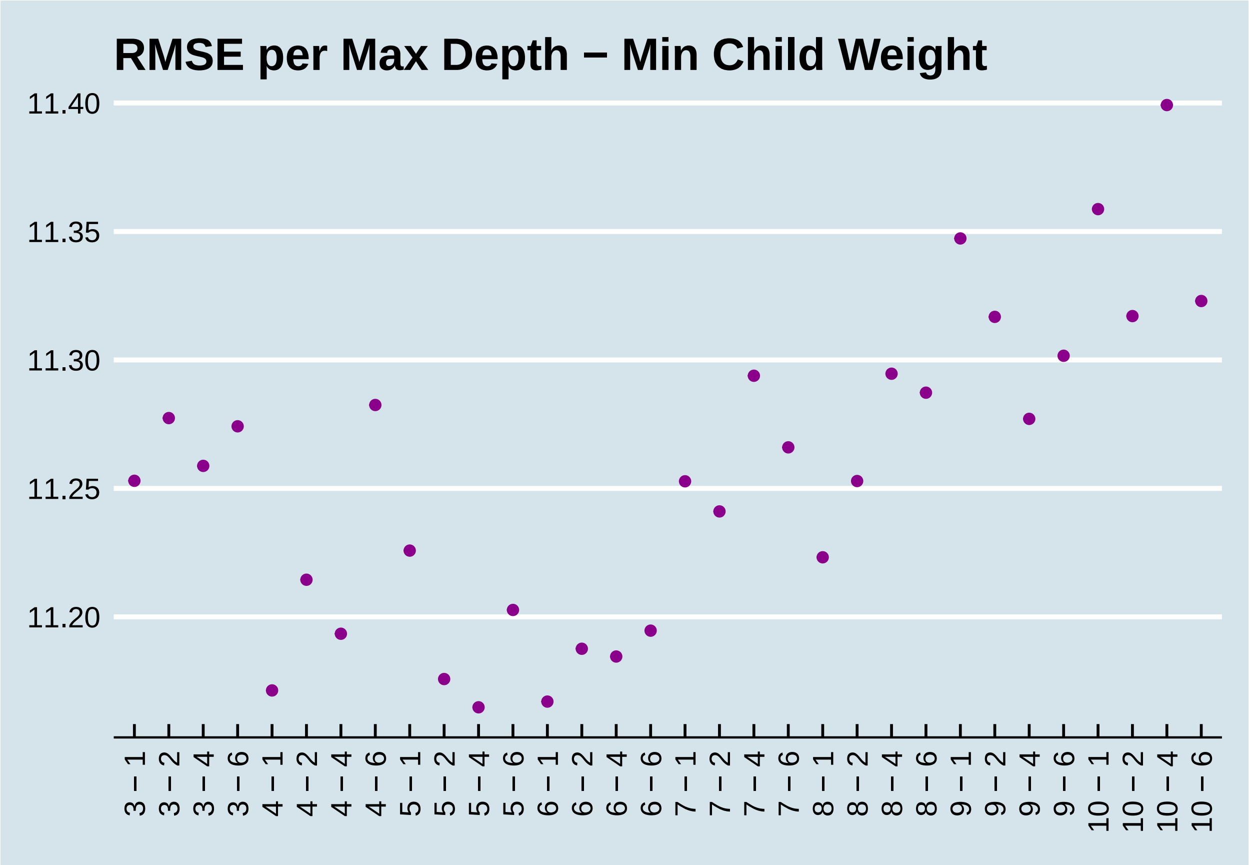


The first can be seen above, and it shows how RMSE changed as *Nround* increased. With only a few iterations and 50 trees, the algorithm has trouble making good predictions on the data, but as the number of iterations and learners created go up, the fit gets better and RMSE is reduced. As it can be observed, the tested values of *Nround* were between 50 and 500, with increments of 50, and it seems that the initial decision to set it to 500 was solid; after all, it is the one that produced the best results here and it will be the one we will continue to use as we move forward.

An observation, however, can be made about that. It is somewhat natural that the more learners are created, the better the algorithm will perform, and it could be argued that had we gone past the 500 value here, results would have continued to get better. Doing so, though, could lead to overfitting, as the model would adjust so well to the training data that it would basically memorize it, showing a degradation of performance when it comes to the test set.

One common argument made in this case is that the number of learners should stop rising when the validation performance begins to display a lack of significant improvement; a recommendation that, if followed, could support the idea that 300 or 350 were better choices. What significant improvement means, though, is highly subjective and open for debate, as such 500 was picked because it seems to be the point when the model stabilizes the gains it has with every iteration.

The chart of the second part of this experiment is a bit different from the first, and that is because there are two values being tested in conjunction: *Max_Depth* and *Min_Child_Weight*. Due to that, the *X* axis now represents the combinations of these parameters. As it is possible to see, values tested for *Max_Depth* ranged from 3 to 10, while those for *Min_Child_Weight* were 1, 2, 4, and 6, amounting to a total of 32 tested combinations, including the standard value of 6 and 1.



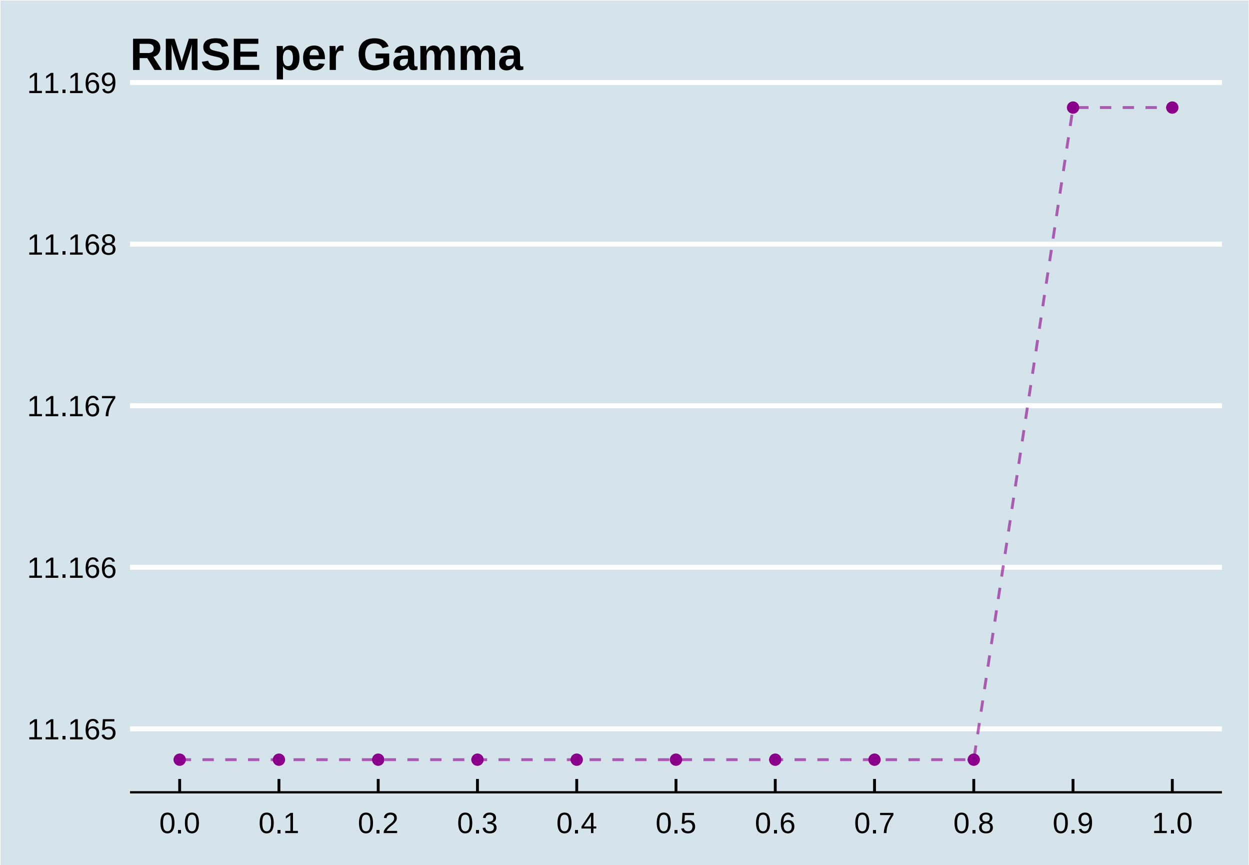
By observing the chart, it is possible to see that, once more, the default values for the parameters of the XGBoost package (6 - 1) did quite well. This time around, though, they were beaten by the combination of 5 to *Max_Depth* and 4 to *Min_Child_Weight*, which yielded the best RMSE with 11.16481: a smudge better than the best one we had encountered so far, which was 11.16703, but not a big improvement.

As we move on to the next experiment, then, we set *Max_Depth* to 5 and *Min_Child_Weight* to 4, alongside the already established value for *Nround*, which is - as it has always been - tuned to 500.

The next plot refers to the parameter *Gamma*, which - as discussed - establishes the reduction in loss that must be achieved for a node to be partitioned. Its default value, the one that has been used so far, is 0, which means that partitions can be made as long as any reduction of loss is achieved. *Gamma* can be set to any number between 0 and infinity, but here we opted for a more reduced scale, one that goes from 0 to 1 in increments of 0.1. As such, 11 values were tested and they can be seen in the next plot.

In the case of this plot, it is not that the differences between the RMSEs produced by distinct values of *Gamma* are so smalll they cannot be spotted; this is actually a case in which they simply did not change all the way from 0 to 0.8. In all of those observations, RMSE was equal to 11.16481, which is the best number we have achieved so far: the one that was reached when we set *Max_Depth* to 5 and *Min_Child_Weight* to 4.

After 0.8, what we see is a drop in performance. It is not a big one, as it is of the scale of 0.004, but it shows that making our trees more conservative and removing their ability to split nodes when gains are not considerable is not helping much as far as sheer RMSE goes. Due to that, once more, we select the default value of 0. Any of the numbers between it and 0.8 could have been selected, but by choosing 0 we opt not to limit our trees' capacity to split nodes and achieve gains.

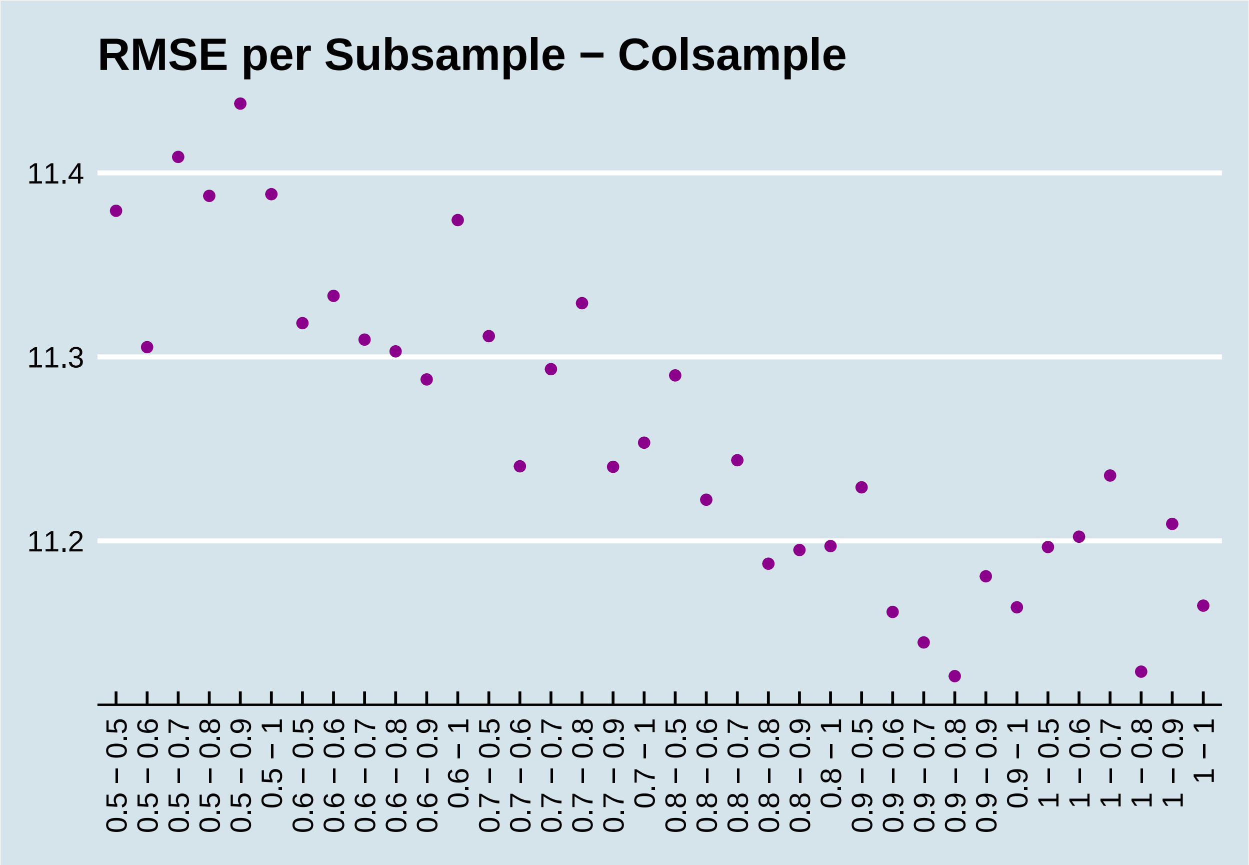


In a way, it could be argued that perhaps our *Gamma* should be higher here, especially considering what was observed in the detailed validation results of Experiment 1, when our model showed it was quite good at handling positive reviews, but very inaccurate when trying to rate negative ones. Perhaps holding it back from further splits could stop it from fitting the positive data so well and the negative data so badly, and make it look for some sort of balance.

Reducing its capacity to split nodes, however, is unlikely to be the solution in that situation; in fact, it might make it worse, for a negative review that falls into a node with a lot of positive ones (a situation that is not unlikely given they are strongly outnumbered) may end up staying there if our *Gamma* is too high. After all, there will not be much gain obtained in separating it, so our model may be blocked from doing so by a *Gamma* value that is higher than 0.

For the next part of the experiment, and the following plot, we look at the combination of the parameters related to the sampling technique employed in the construction of our trees. Originally, both *Subsample* and *Colsample_Bytree* are set to 1, which means all trees are constructed with the full training set at hand and all possible columns (the words in our reduced vocabulary). These values were included in the test that was performed, as we evaluated all possible combinations with the two parameters varying from 0.5 to 1. As a consequence, a total of 36 possibilities were tried.

As it happened with *Max_Depth* and *Min_Child_Weight*, the chart produced to illustrate this portion of the experiment uses the tested combinations for the *X* axis. In the results displayed, a tendency emerges: the more of the rows and columns are used, the better our achieved value of RMSE seems to get. Nearly all of the worst-performing points have one of the two parameters standing either at or quite close to the lowest evaluated value of 0.5.



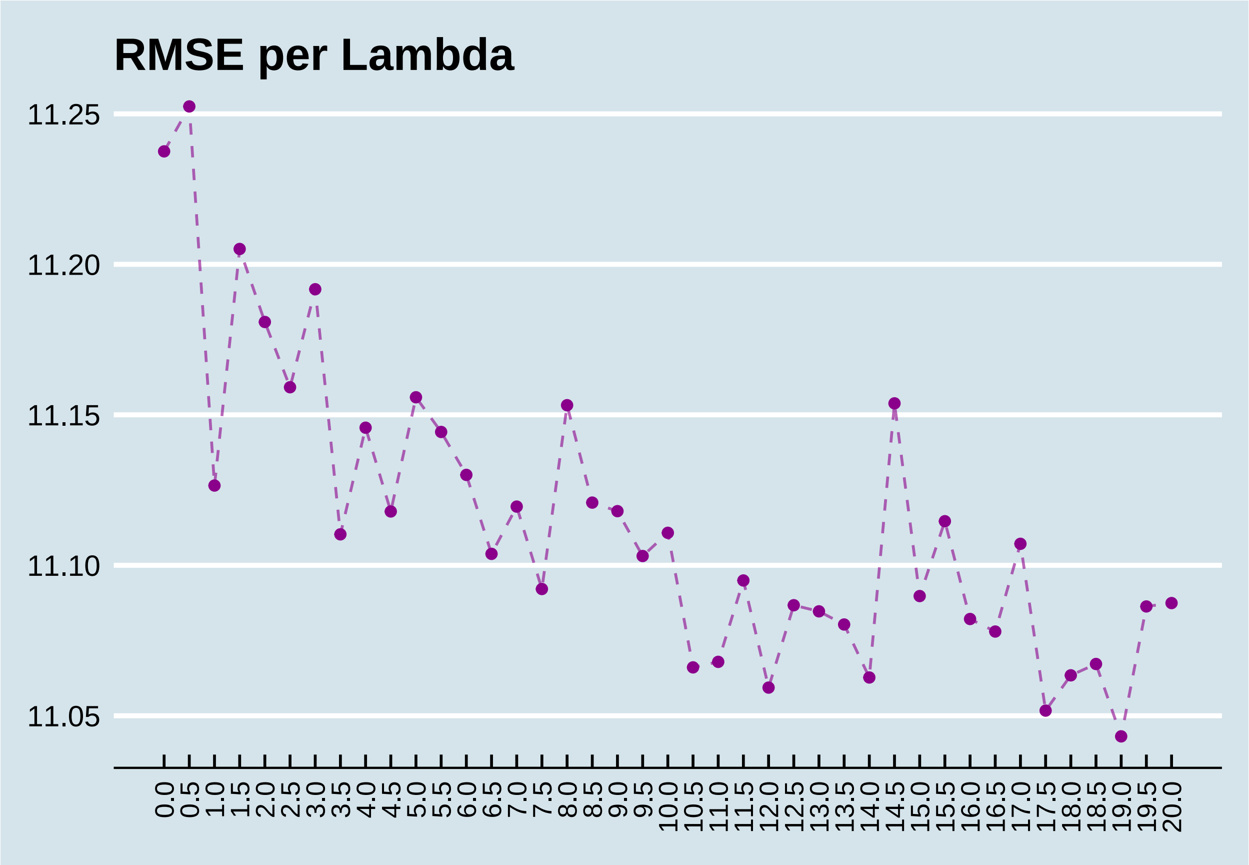
In spite of that tendency, though, the best performance was not achieved by the default values of 1. In fact, that combination was surpassed by five others. The one that produced the best RMSE came up when *Subsample* was set to 0.9 and *Colsample_Bytree* to 0.8, and those are the values, therefore, that will be used from this point forward. The RMSE found in that case was equal to 11.12650, slightly better than the previous best, which was the 11.16481 encountered when *Max_Depth* and *Min_Child_Weight* were tested.

The last test of this experiment concerns the regularization parameter *Lambda*. Before getting into the results, it is worthy pointing out that the XGBoost implementation available on R offers two choices as far as regularization goes: L1 and L2. *Lambda* is the parameter of the L2 regularization, which means that we opted for that type of strategy.

The reason behind it was that L1 regularization, which is tuned through the parameter *Alpha*, is the more aggressive of the pair, as it has the capacity to remove the influence certain features (words) have over the linear regression we are performing. Given this was a task that, in a way, had already been carried out when we evaluated and implemented feature selection, in Experiment 2, L2 was thereby picked, as the most it can do is strongly reduce the effect a term has over the score.

For this parameter in particular, the relatively large range between 0 (which means no regularization at all) and 20 was tested, in increments of 0.5. As such, 41 values were tried, including the default of 1. The results can be seen in the next plot and it seems that, in our case, regularization was good.

Once more, the improvement from the best value of RMSE that had been found up to this point (11.12650) was not considerable, as the lowest one reached here was 11.04315: a decrease of 0.08. However, it is possible to observe that many were the values that surpassed the library's default, and the one that produced the best RMSE was 19: close to our upper limit of 20. As a whole, the chart shows a good deal of oscillation between improvements and steps back, but in general there is a clear downwards tendency as the iterations moved along and the values of *Lambda* were raised.



With the experiment concluded and all selected parameters tuned, it is worth having a look at a table that summarizes what was done. In it, we see the evaluated parameters, their values (accompanied by the increments, if they apply), the ones that produced the best result, and - of course - the achieved value of RMSE for each step of the tuning process.

Parameters	**Tested Values**	**Best Values**	**Best RMSEs**
Nround	50-500, +50	500	11.16703
Max_Depth and Min_Child_Weight	3-10 and 1,2,4,6	5 and 4	11.16481
Gamma	0-1, +0.1	0	11.16481
Subsample and Colsample_Bytree	0.5-1, +0.1	0.9 and 0.8	11.12650
Lambda	0-20, +0.5	19	11.04315

8. Final Results

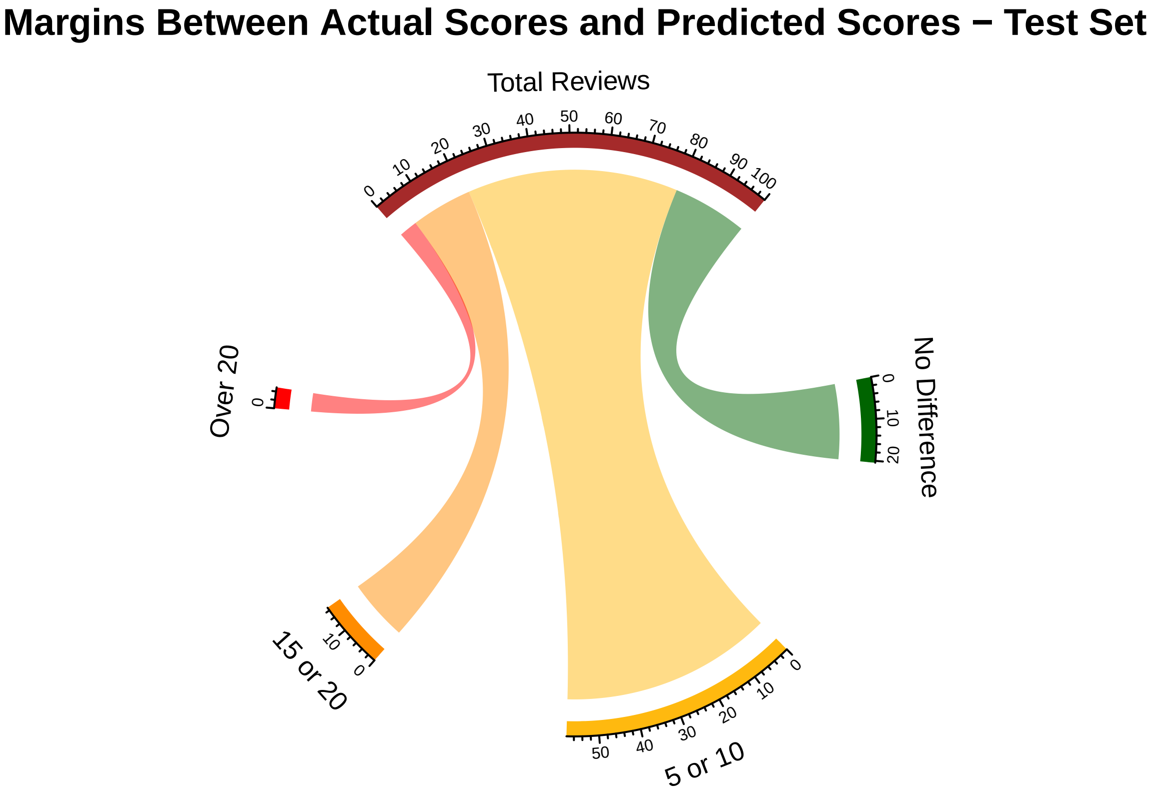
With the best of our two models selected (Experiment 1), its vocabulary trimmed down to the threshold in which it showed the best performance (Experiment 2), and its parameters tuned (Experiment 3), we apply it to the test set and to look at the final results.

Here, naturally, rather than training it solely with the training data, the validation set is added to the pile. After all, we are not going to use it to tune the model any further and the observation of the final results will be done with the test dataset. As such, we might as well take advantage of all the data we have at our disposal.

The training dataset in this section, then, actually has 110,908 reviews. Meanwhile, the test dataset, which we set aside before doing any sort of analysis, contains 12,325 rows. By using the former, we trained the model according to a configuration that can be seen in the table below.

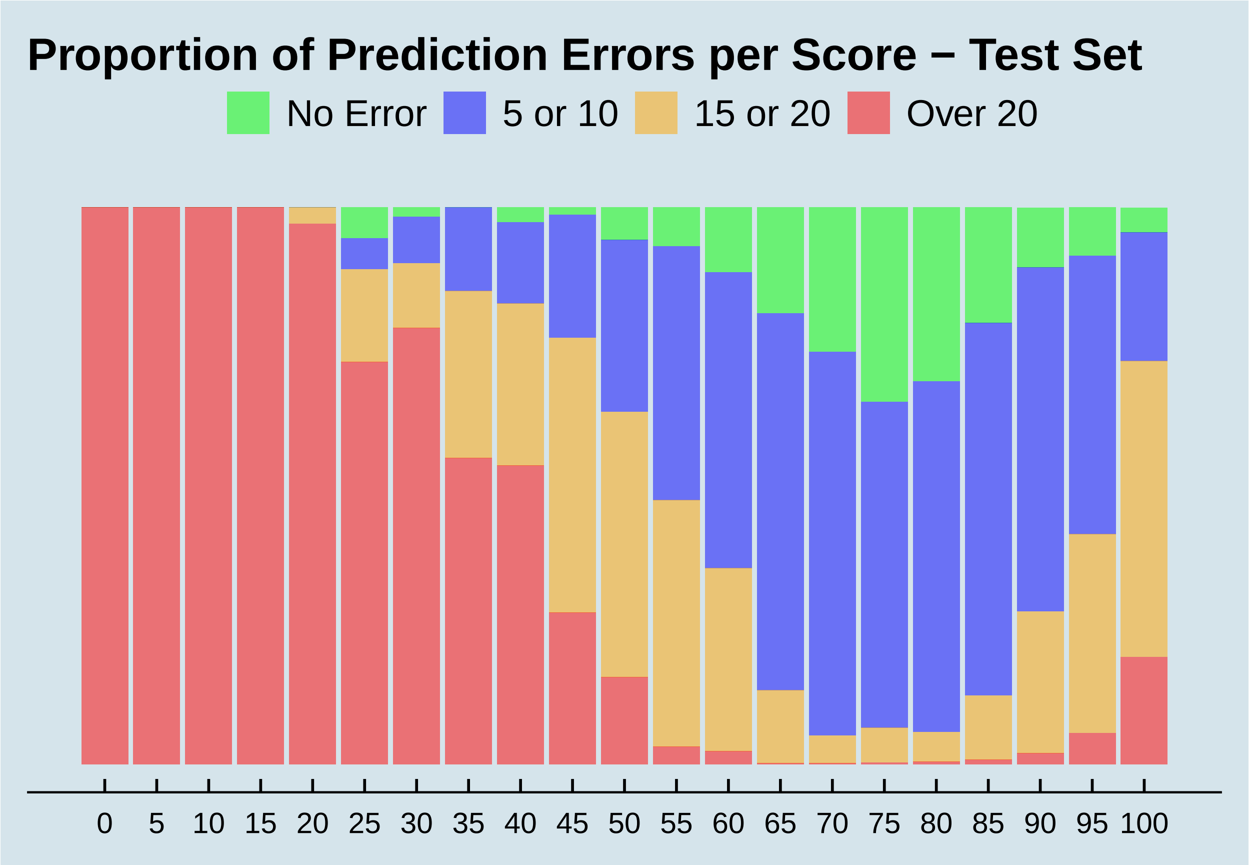
Model	XGBoost Tree
Vocabulary Size	2,603 Words
Nround	500
Max Depth	5
Min Child Weight	4
Gamma	0
Subsample	0.9
Colsample Bytree	0.8
Lambda	19

For the sake of comparison, the results achieved in the test set will be reported in a similar way to the detailed results of the validation set exposed in Experiment 1. We hope that such a configuration will allow any evolution between the baseline model and this final one - if it happened - to be observed with more ease, even if, ultimately, the validation and test set are not equal. We begin, then, by once again plotting the gaps between predicted and actual scores. It is worth remembering both the scores of the test set and those generated by the model were adjusted to our scale of 5-point increments, as previously explained.


Margins Between Actual Scores and Predicted Scores − Test Set

Differences between this chart and the one from Experiment 1 are in fact so tight they appear not to exist. There were only slight variations in the percentages of rating error margins. When it comes to those that were right on target, the rate diminished from 21.06% in the validation set to 20.45% in the test set. Errors between 5 and 10 increased from 57.26% to 57.77%. Those that fell between 15 and 20 raised from 16.58% to 16.76%. And, finally, the worse errors, those of 20 or more, actually decreased from 5.08% to 5%.

Once again, given the imbalanced nature of the dataset, it is only natural we ask ourselves just how well our model did when it comes to the different rating bins. Did its tendency to get more positive reviews right and more negative reviews wrong continue? The next chart, which is also built identically to the one from the results of Experiment 1, comes to answer that question.



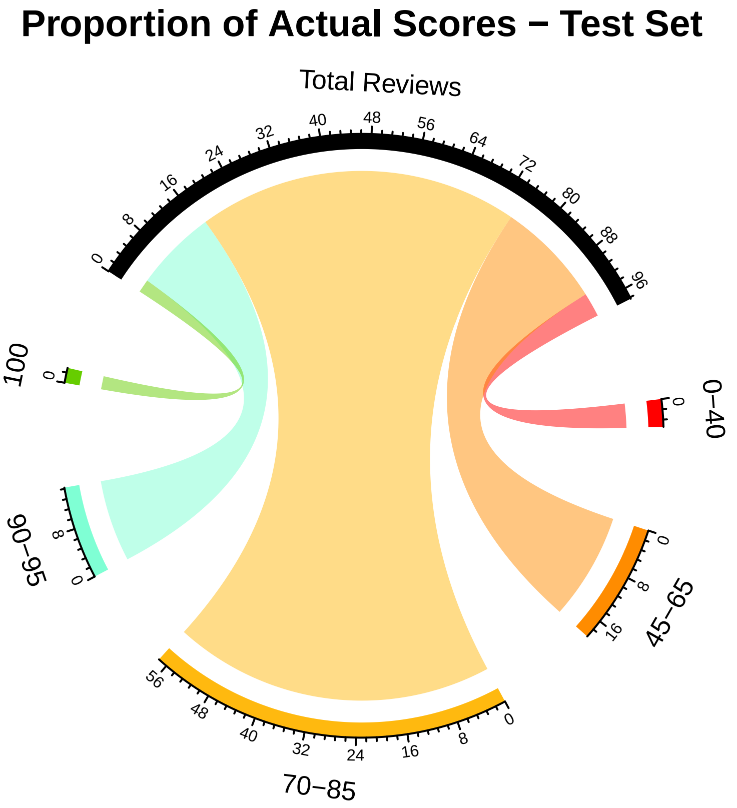
Once more, differences between the charts are so tight it is hard to tell how they are unique, which means that we did one good job at predicting test outcomes based on the validation dataset. The pattern is the same. For scores between 65 and 90, our Universal Rater does well: there are almost no reviews in that range where it misses by 20 or more. And, most of the time, it is either totally correct or close to the real rating by 5 or 10 points.

As we move out of that interval, the errors get worse, especially the ones that go over 20 points. The situation is quite dire for reviews in the 0 to 20 range; in there, virtually all of them get scores that are 20 points away from reality.

It is good to point out that, even if errors of the kind are a problem that eventually needs to be solved, reviews in that area are not that numerous, which explains why, when it is all said and done and as seen in the first plot of this section, our Universal Rater gets approximately 75% of the ratings either right on target or with´in 5 or 10 points.

To illustrate, given that, up to now, we had not looked exclusively at the test dataset with any sort of detail, the next plot shows the amount of reviews per rating contained in it. As one can observe, the interval from 0-40, which is actually the largest one in the previous image in absolute values covered, only has more reviews than the group that includes solely those with perfect scores. By a large margin, the most popular scoring range is between 70 and 85, a group that only covers - according to the scale in which we are working - four possible grades.

In a way, this puts the result of the plot above in perspective, because the range in which our model more often than not makes mistakes of over 20 points is also one with a very small rate of samples. As such, huge errors are, in proportion to the overall dataset, not frequent; yet, once more, it is clear our model struggles with getting the rating right when reviews are negative, an effect that is being quite likely caused by the set's imbalanced nature.



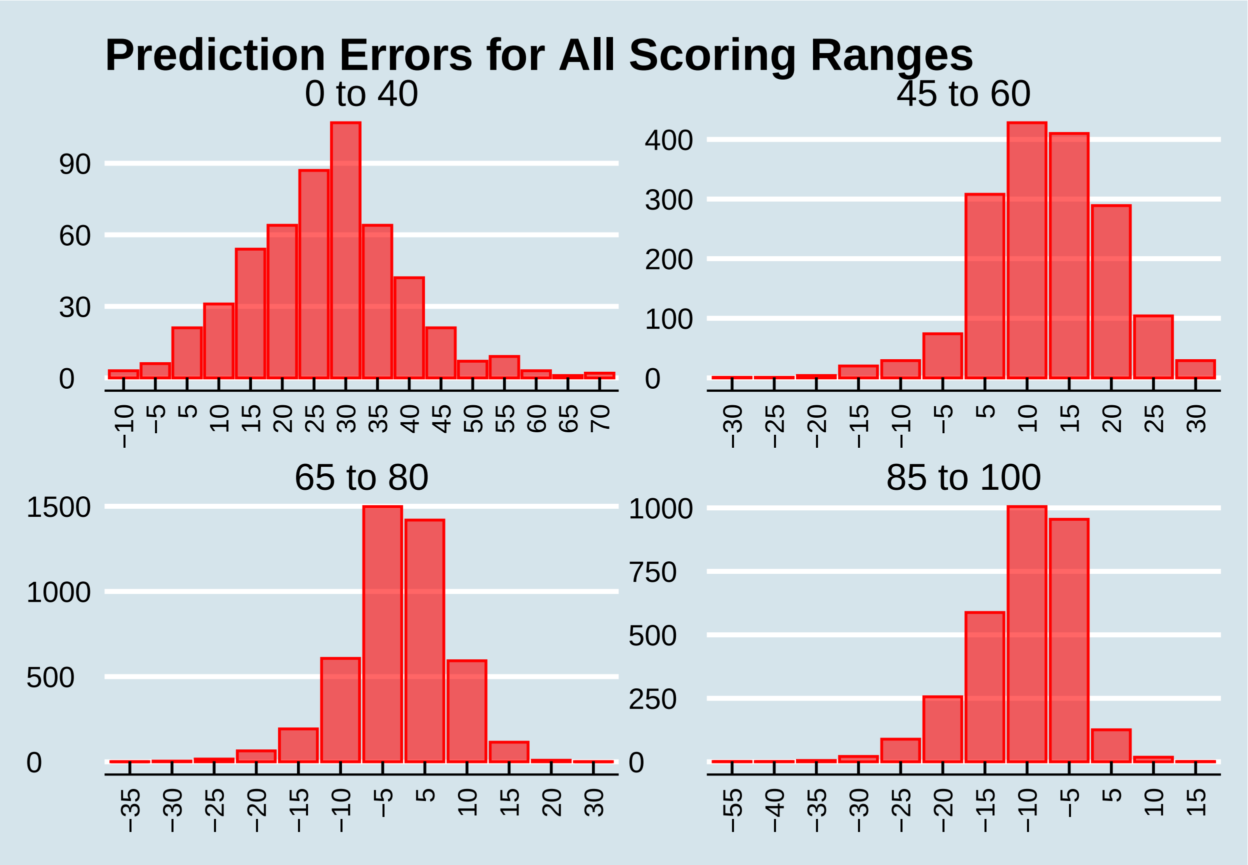
Again repeating the procedure executed in Experiment 1, given the scenario of errors we are encountering it is only natural to wonder exactly what kinds of mistakes are occurring. Is our model generally underrating or overrating games?

The answer is it depends. The next plot counts our errors by their value and then groups them according to a quartet of scoring ranges. Obviously, since we are working with values between 0 and 100, the model can neither underrate a title that got the former score nor overrate one that achieved the latter. Nevertheless, these ratings were added to the pile so we could have a complete view of the mistakes.

When it comes to actual scores that fell between 0 and 40, our model tends to overrate them, with 30-point mistakes being the most common. In that range, underrating rarely occurs, with a few instances appearing in which our model went 5 or 10 points below the score. For ratings in the range from 45 to 60, the tendency is similar, but the most frequent errors are much lighter, usually ranging between 10 and 15 points.

The scenario changes with scores from the other intervals. With those between 65 and 80, we see a curve that strongly resembles a normal distribution, as there is a nigh equal quantity of mistakes both above and below the ratings we are trying to predict: those that miss by 5 (be it by more or less) are more usual than the ones that miss by 10, and so forth. With scores between 85 and 100, the tendency is underrating, maybe because there is not much room to overrate in that group. But, once again, the most common mistakes are the smallest ones.

Here, it is important to look at exactly what the word mistake means. When we set out to build a Universal Rater, our goal was - ultimately - not to construct a model that would achieve error 0, even if that is what Machine Learning algorithms strive for; we wanted to create some sort of universal scale for rating games. The idea was that, when reviewers went about their business of analyzing new releases, they would still hand out their own personal rating. Our model would, however, also come up with a number of its own, which would be based on the general feeling present in the text.



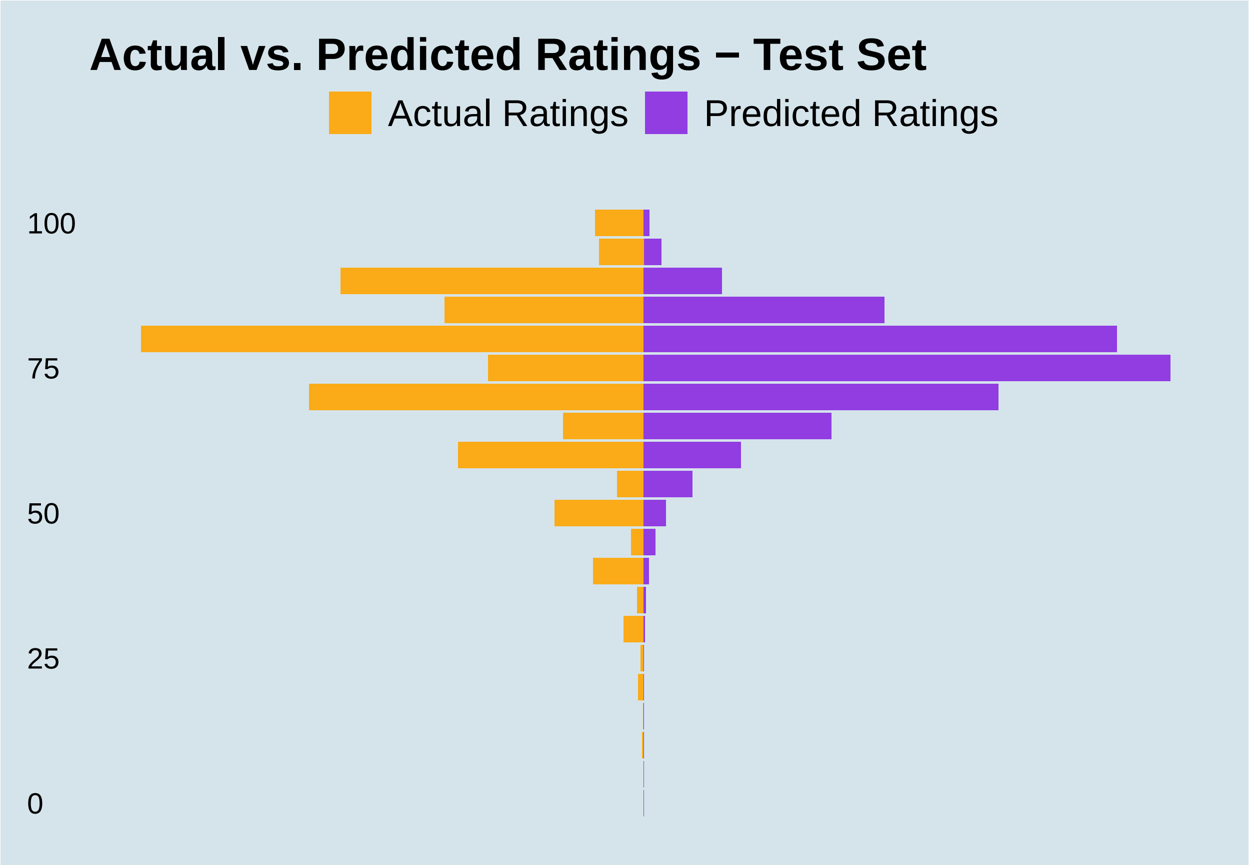
By doing so, we wanted to avoid the general biases that come with the task of giving a somewhat arbitrary number to a written evaluation. Maybe the review was more negative or positive than its final human-given score indicated, but the writer - a victim to the inherent human biases - failed to capture that idea.

In that case, the question that can be posed about the mistakes our model is making is how many of its errors are a result of modeling matters and how many of them are a result of actually doing a better job at rating than the human who is at the wheel. In a way, it is an impossible question to answer, because to do so, there would have to be a number out there that represented the totally unbiased score of each review, and that is not something that is available. In fact, it is what we are conceptually looking for here.

When we looked at the average of feelings present in the reviews of each scoring bin by using the AFFIN lexicon, the numbers indicated some distortions, but overall there was indeed a definite tendency that the more the ratings went up the more positive the general feelings in the reviews appeared to be, which indicates the mistakes we make are a source of good old modeling details. But to try to give a deeper look into this matter, we will close out this section with a pair of analyses.

In the first, we have a look at the proportion of ratings handed out by actual reviewers (the real scores of our test set) and by our model (the predicted scores). This comparison can be seen in the next plot, and the question we are asking is: does our model's tendency to overrate negative reviews means it is more generous than the professionals who dedicate their lives to analyzing games?

Again, the answer is that it depends. Our model is actually far less likely to hand out 100s, 95s, and 90s than humans, it seems. Contrarily, it awards a much larger number of 75s and 65s. Towards the bottom of the barrel, given what we have already seen, the tendency is to rescue bad games from terrible scores. In fact, while the test set has some 0s, 5s, and 10s, the lowest grade our model produced was a 15. Besides, there are 533 actual scores below or equal to 40, and the Universal Rater only went that low 55 times. It's so litle most of those scores can barely be seen in our chart.



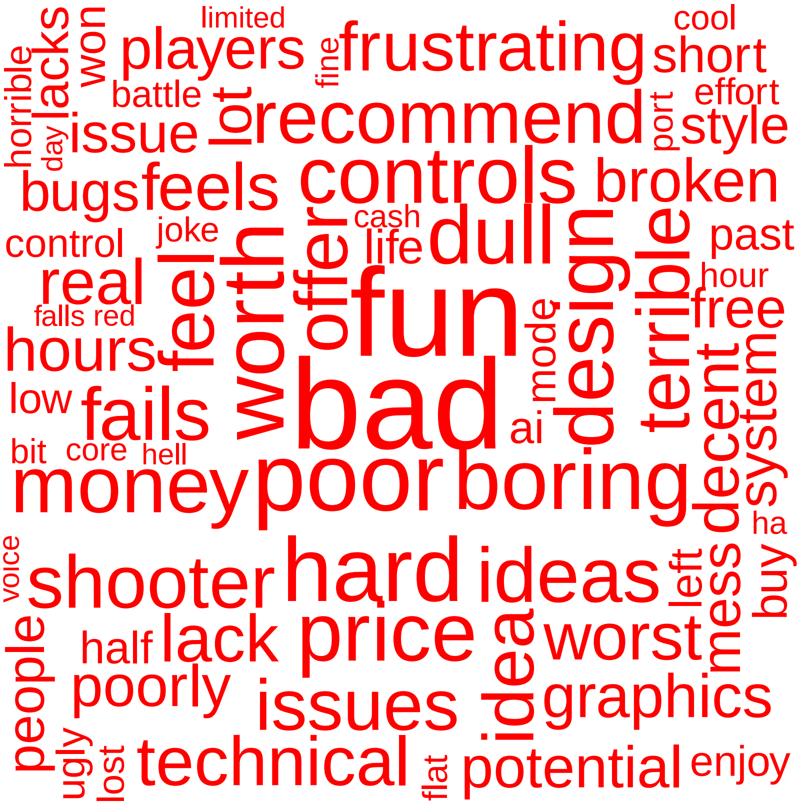
The last analysis that is worth making is related to the general vocabulary used by reviewers who found themselves in the test set. What we want to check, just to be sure, is if they sounded negative enough in their analyses to justify the score. If they are not sufficiently harsh, perhaps our model could not possibly have captured the negativity. After all, if the words are not there, it will be hard for it to tell a writer did not like a game.

Again, it is important to remember that, according to the evaluation done through the AFFIN lexicon, there was a general link between the feelings present in the excerpts and the ratings they generated, even if a few outliers appeared here and there. Just to have an idea of how big the negativity is in the scores of 40 and below, we produce a Word Cloud with the most frequently used terms of the test dataset for reviews that fall within that range.

Note that, for the construction of this image, the text was processed in two ways: stopwords were removed, otherwise they would have dominated the scene, and we only considered words that were part of our model's vocabulary, just to make sure that we did not drop negative words while we were doing feature selection.

As it is possible to see, the negativity is certainly there. A few words exist that, on their own, could indicate to our model that these are not such bad games, like "recommend", "buy", "cool", "love", and even the omnipresent term "fun".

However, that is undeniably one large Word Cloud of negativity: reviewers are not letting these bad games go by without verbal punishment. As such, it is indeed our model that has trouble capturing these tones. Certainly, many of these terms appear in reviews that are more average, because games that get ratings in the 60s and 70s tend to come with a bunch of caveats attached to them, as we have observed in previous Word Clouds that covered these mid-ranges. However, they are certainly more prominent in these cases.



To wrap it up, we report the final RMSE that was obtained by our model when executed against the test set. Its value was 11.27725, not too far off the mark achieved by that very same model by the end of Experiment 3, when it was tuned to its final configuration and got to an RMSE that was equal to 11.04315 over the validation set.

9. Limitations and Considerations

It goes without saying that the work here presented does not aim to be, in any way, a complete exploration of the dataset chosen. The areas of Machine Learning and Data Science contain infinite possibilities that could be endlessly pursued in order to improve the model that was built. By being within 10 points for about 75% of the ratings present in the dataset, it is possible to state that good results were achieved, especially given the already discussed subjective nature of scores.

Yet, problems remained and so did opportunities for future improvement. Throughout the experiment, with the objective of maintaining dimensionality low so that the processing of the full dataset would be possible, only unigrams were considered. Since they have been successful in tasks such as text classification [31], bigrams or perhaps even further combinations could have been tried in order to increase the extraction of feelings from the text, as words like “fun”, for instance, which are present in negative reviews as we have observed, could in those cases be accompanied by terms such as “no” or “not”, which are likely to have been attached to that usually positive term in many of our negative reviews. Furthermore, more advanced techniques such as the already mentioned Word2Vec [15] could have been tried.

Regarding the problem with imbalanced data that this work came across, common strategies to deal with that issue could be experimented, like SMOTE [32] as well as the simpler methods of oversampling and undersampling [33], with the latter dictating reviews of scoring bins that are too common could be simply left out of the split in order to attempt to solve the issue.

Finally, other Machine Learning models could be experimented to see how they react to the full dataset; a different rating scale, one that also removed grades ending in five from the equation, could be tried; and even lexicon-based methods of rating predictions could be evaluated.

The last limitation, and one that has yet to be discussed, has to do with the nature of the dataset itself. Metacritic only contains part of the review: an excerpt that tries to summarize the feelings of the writer about a game. Since that was the data made available, it was the one that was used in order to try to predict the score. Naturally, the full text of the review contains far more nuances than the excerpts. That is not to say using the complete review would guarantee better performance; it is hard to know which way the model would go. But, ideally, a Universal Rater such as the one proposed here would have to be trained with total texts in order for it to accurately be able to capture which moods and feelings separate 100s from 95s, and even 80s from 40s.

10. Conclusion

In this work, we set out to construct a Universal Rater: a model that would, upon reading a review, assign a score that corresponds to the real feeling contained within the text. Ultimately, the theoretical goal was that, rather than falling victim to the biases of humans, this algorithm would provide a global rating to reviews in a scale that avoids external influences such as hype and personal tendencies.

From the get go, it was understood that the goal was almost impossible. Machine Learning algorithms, after all, learn from data, and they do so by trying to minimize a specific error rate. If we reached a model with total precision, then, we would not have built an unbiased rating scale, but one that simply modeled the tendencies of each and every reviewer. Nevertheless, we proceeded centered in the question of whether or not ratings could be accurately extracted from words, a task that was itself still quite hard given the relatively arbitrary nature of scores.

In order to build the Universal Rater, we used XGBoost, a model that gained fame in recent years due to its excellent performance in multiple Machine Learning competitions. Upon trying both the linear and tree-based versions of the model on a first experiment, we opted to stick to the latter due to its superior precision according to our metric of choice, RMSE. From that point, we started to refine our rater by seeing if we could get a better performance with a smaller vocabulary (to which the answer was yes) and also do the same by selecting the best values for its parameters. With both tasks concluded, this final model was trained and applied in the test dataset, with its results fully reported.

What could be observed was that, in sheer proportion, the performance of the model was good. About 78% of the reviews were graded either right on the mark given by the professional writer who published it or within a range of 5 to 10 points. However, a fact that was noticed early on (the imbalance of the dataset) caused the model to struggle with negative reviews. These were far more uncommon than the average to positive ones, as such while much of the success of our rater came from texts that produced scores between 65 and 95, much of its failures originated in reviews that were more negative. Through analysis, it was shown that negative reviews did not lack in a negative tone: they had plenty of terms of that kind. But our Universal Rater had difficulty in identifying them and handing out the appropriate score.

Although by no means perfect or a huge success, a model that is, in a field as subjective as reviews, able to give relatively close scores to 78% of the data, while keeping 16% of the remaining 22% within a margin of error of 15 and 20 shows that the building of a universal rating scale may be possible. And although algorithms will likely, not for the foreseeable future at least, replace critics as the ultimate judges on the quality of a product, they may be able to provide a helping hand in creating a rating scale that, even if not as interesting or incendiary as the one made up of human opinions, is at least more free of biases.

References

1. RottenTomatoes

2. Metacritic

3. Psychology Today - Can You Escape Bias?

4. Towards Data Science - What is Boosting in Machine Learning?

5. GamesBeat - Metacritic is stupid, but only because review scores are also stupid

6. Eurogamer - Eurogamer has dropped review scores

7. IGN - Announcement: IGN's Review Scale Just Got Simpler

8. MonkeyLearn - Sentiment Analysis: A Definitive Guide

9. Hello Future - AI could reduce human error rate

10. Kaggle - Metacritic critic games reviews 2011-2019

11. Towards Data Science - The Curse of Dimensionality

12. Why is removing stop words not always a good idea

13. Towards Data Science - TF-IDF

14. Wikipedia - TF-IDF

15. Towards Data Science - Text Classification with NLP: Tf-Idf vs Word2Vec vs BERT

16. Text2Vec

17. XGBoost: A Scalable Tree Boosting System

18. Tree Boosting With XGBoost - Why Does XGBoost Win "Every" Machine Learning Competition

19. Text Classification by XGBoost & Others: A Case Study Using BBC News Articles

20. The Strength of Weak Learnability

21. What makes "XGBoost" so Extreme?

22. XGBoost: eXtreme Gradient Boosting

23. XGBoost Parameters

24. Why, How and When to apply Feature Selection

25. Is feature selection worth the effort? Assessing the impact on Random Forest and SVM accuracy and computation time

26. Machine Learning Mastery - Feature Selection to Improve Accuracy and Decrease Training Time

27. Feature selection for imbalanced data based on neighborhood rough sets

28. The chi-square test of independence

29. Information gain in decision trees

30. Complete Guide to Parameter Tuning in XGBoost with codes in Python

31. The Use of Bigrams to Enhance Text Categorization

32. Machine Learning Mastery - SMOTE for Imbalanced Classification with Python

33. Machine Learning Mastery - Random Oversampling and Undersampling for Imbalanced Classification