Learning to Rank Comments Within Subreddit Submissions

Alex Jin Don Mai Jeff Setter

1: Introduction

Reddit is an online bulletin-board forum where users can post content, and judge the interest of the post by means of voting for the best content.

We want to examine the relationship between the *top-level comment score* and *all other attributes*, as shown in **Figure 1.1.** However, since a comment's absolute score depends on the number of active users at the time of creation, we will instead try to predict comment ranking. The inputs and outputs of our problem are as follows:

Figure 1.1: Example Reddit Post

Input = { post content, time of post,
comments' contents, time of comments }
Output = { predicted ranking of the comments }

2: Related Work

Spearman's Footrule is a metric used to evaluate predicted rankings [1]. The metric measures the difference between the actual ranking and our predicted ranking. It penalizes for relative error. We normalized the error function to be between 0 and 1.

We looked at NLP analyses [2][3] for insights to produce "intelligent" features that extracts popular keywords and meaning from a paragraph instead of just words. The suggested approach in these papers is to use multi-grams (consecutive words from a sentence). However, since users in online communities often use abbreviation and/or variants of words, removal of these words might be undesirable. Instead we looked into meme clustering (finding popular phrases) for better features.

Meme clustering is not a new topic. Previous studies [4][5] have looked at Meme Clustering albeit on a much larger scale. These two papers took the same approach as our algorithm in assuming a peak in popularity during a certain time window. Authors from [5] acknowledge that this problem is NP-hard and uses heuristics to tackle the problem. Our approach, K-means, showed a similar understanding and produced analogous results on the subreddits we tested.

3: Dataset and Features

For the subreddit that we analyzed, /r/murica, we collected metadata for 25,688 submissions and 173,875 top-level comments. After the preprocessing described below, the size of this dataset was trimmed to 21,764 submissions and 104,521 top-level comments.

To collect Reddit submissions and comments, we sent web requests to the official Reddit API. We had to make two types of API calls: one to retrieve a list of submission IDs, followed by a separate one to collect detailed post and comments metadata for each discovered submission ID. Reddit limits web requests to their API to one request/second so that users do not overwhelm their servers, so we used a wrapper request library [6] to ensure our web requests were properly rate-limited and met other API specs.

The response data was in JSON format, and we used LevelDB, a key-value store, to store the submission IDs and post and comment(s) JSONs for each submission. By choosing a clever key-naming scheme that includes the subreddit and timestamp of the post, we could keep track of what submission IDs we already sent fetch metadata API requests to so that if our data collection program experienced any problems, we could always restart the program and resume sending web requests only for unprocessed submission IDs. That is, our data collection becomes verifiable and resumable, which are helpful properties when we want to send web requests only for unprocessed submission IDs due to the time costs of request throttling.

In terms of preprocessing, we tokenized and stemmed the text of all collected submissions and comments. We used the Porter Stemming algorithm provided by a natural language processing library[7]. The tokenization got rid of common words used in the English language (e.g. articles) and removed punctuation. We also implemented our own url-cleaning procedure where we replaced urls with their hostname by stripping the url path. Otherwise, it would be difficult to have url detection features that would be activated across multiple submissions since the url path varies a lot for the each url inclusion involving the same hostname. To help our downstream algorithm learn better, we also removed duplicate comments with the same score under the same parent submission, comments where a moderator removed the text body, and submissions where the number of top-level comments was less than two.

Here is an example of a JSON comment after cleaning (some fields are omitted for conciseness):

{"body": "Doing it right patriot!\nMy apartment has 2 decorations. Both are American flags. ",
"created_utc": 1441063213,
"score": 2,
"cleanBody": { "do": 1, "right": 1, "patriot": 1, "my": 1, "apart":1, "decor":1, "both":1, "american":1, "flag": 1}
}

To help some of our machine learning algorithms converge faster, we used feature scaling on all features that did not already have a magnitude between 0 and 1. For each comment in a submission, we applied the following formula:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

where x is the original feature score, x' is the corrected feature score, $min(x)$ is the smallest value of the feature score across all comments for a particular submission, and $max(x)$ is the corresponding largest value.

From each submission, we could extract several quantities, such as the frequency of words (individual, bigrams, trigrams, urls) as well as compute properties of a comment (as listed in **Figure 3.1**). Each algorithm uses some set of these features.

Feature Category	Feature
Specific post words	Comment contains: gif, !, http, freedom
Relevant keywords	Overlap of words in comment and post title/body
Quality post	Punctuation, usage of caps
Histogram of word lengths	Count of words in comment of length n
Time	Comment_time – submission_time
Bigrams/trigrams	Frequency of comment bi/tri
URLs	Frequency of url domains

Figure 3.1: Features

4.1: Naive Bayes

Our first attempt at solving the stated problem was Naive Bayes. As shown in **Figure 4.1.2**, to divide the upvote_ratio (upvotes / (upvotes+downvotes)), we set a cutoff value ALPHA. We run Naive Bayes using the following inputs and outputs for different ALPHAs:

inputs = { words of a message }
classes:

good   if upvote_ratio > ALPHA
fair   if upvote_ratio = ALPHA
bad   if upvote_ratio < ALPHA

Figure 4.1.1: Naive Bayes Implementation

The results of our Naive Bayes show that the error increases as number of training examples increases. This is understandable because the Naive Bayes assumption breaks down for some of the features we ended up using. This motivated us to implement our regression algorithms.

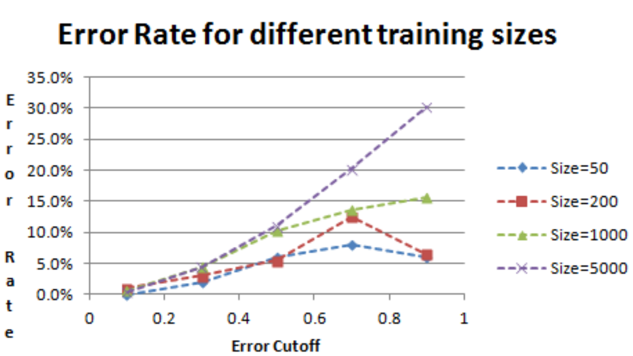

Figure 4.1.2: Naive Bayes Error Rates

4.2: Regression

Naive Bayes failed to scale well with a larger dataset size, so instead we approached the problem from a different angle that used features from each comment that would be unique values. For example, one possible feature was the total number of characters in the comment text. The features that we extracted from each comment is shown in **Figure 3.1**. The goal of our problem was to eventually rank comments based on their features. We converted the rank into a score value computed by the rank normalized with the number of comments in the submission (see **Figure 4.2.1**). This means the scores are bound between 0 and 1, and the highest comment has the lowest score. Our model then attempts to compute this score using the feature vector by computing the appropriate weights for each feature ($h_\theta(x^{(i)}) = \theta^T x^{(i)}$).

$$y^{(i)} = \frac{rank}{length(comments)}$$

Figure 4.2.1: Regression score computation

To train our model, we input submissions from a subreddit, and try to find optimal weights to solve the target regression problem. We used two different models, linear regression and Support Vector Regression, both of which attempt to minimize the distance between the score and the predicted score (see **Figure 4.2.2**). For linear regression, we created our own code that used stochastic gradient descent, while we used the scikit[8] implementation of SVR. However, we found that SVR had similar accuracies, but was more consistent, so only SVR is shown.

$$\begin{array}{ll} \min_{w,b} & \frac{1}{2}\|w\|^2 \\ \text{s.t.} & y^{(i)} - w^T x^{(i)} - b \leq \epsilon \\ & w^T x^{(i)} + b - y^{(i)} \leq \epsilon \end{array}$$

Figure 4.2.2: SVR optimization equations

Evaluation of each of the results from each model used a normalized version of Spearman's Footrule. Given a predicted ranking and an correct ranking, Spearman's Footrule is the sum of absolute differences of each individual item. Our normalized version of this (see **Figure 4.2.3**) divides the error by the maximum possible, which is computed using the items in the reverse ranking. The normalized version of this error is thus bounded between 0 and 1, where 1 denotes the worst possible predicted ranking. Note that this error distribution varies immensely based on the number of comments in a submission. For example, a submission with only 2 posts can have only errors of 0 or 1, while a submission with many comments can have many different values in the range. Furthermore, the average error rate for a random guess is not 0.5. For example, a random guess for three comments has a ⅙ probability of 0 error, ⅓ probability of 0.5 error, and ½ probability of 1 error for an average of 0.67. Thus, depending on the how many comments the submission have, a random guess can have different accuracies. The distribution of errors for a random guess in our dataset is shown in **Figure 4.2.4**.

$$\hat{\sigma} = \frac{\sum_i^m |h(x^{(i)}) - y^{(i)}|}{\max_{err} x}$$

Figure 4.2.3: Normalized Spearman's Footrule

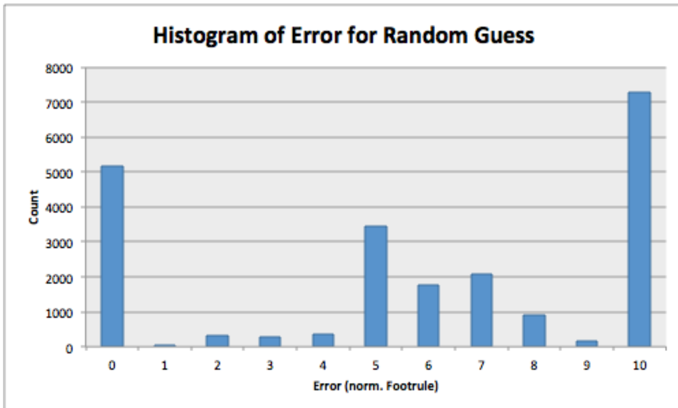

Figure 4.2.4: Histogram of errors for Random

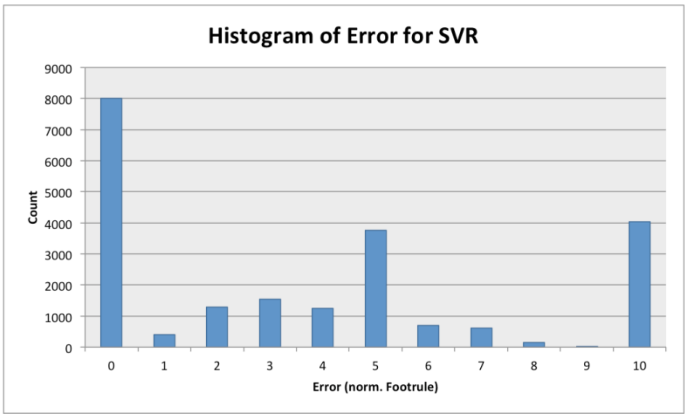

Figure 4.2.5: Histogram of errors for SVR

After training our model on all of the data with our set of features, the error is reduced from a random model error of 0.613 to the full set of features giving 0.377. The histogram of errors using all of the features is shown in **Figure 4.2.5**. Note that we still have peaks at 0 and 1 error due to the large number of submission with only two comments.

Once we had our model and features, we evaluated our features by doing a feature component analysis. Starting with no features, we trained our model and gradually added features, recording the error associated with each set of features (see **Figure 4.2.6**). This allowed us to evaluate the effectiveness of each of the features created. The most significant feature is time, which reduces the error from 0.546 to 0.377. The effectiveness of this feature makes sense, because comments that are posted earlier have more exposure time than other comments, meaning it has a greater opportunity to accumulate the necessary votes to make it a top comment. Each of the other features add a small benefit, since each captures just a small amount of how users might react to the comments. The next most significant feature, specific words, is able to capture a general trend that certain words may inherently cause a comment to be favored.

The final analysis of our model was a learning curve, which looks at the testing and training error over different sizes of datasets (see **Figure 4.2.7**). We find that although we have 18,000 submissions in our entire dataset, after only 1,000 posts our training and testing error converge at a value of around 0.38. Because the errors converge, and even cross each other, it means that our model is

Features	Test Error
random (no features)	0.613
+ specific post words	0.597
+ relevant keywords	0.581
+ good formatting	0.578
+ quality post	0.560
+ histogram of word lengths	0.546
+ time	0.377

Figure 4.2.6: Feature Analysis for SVR

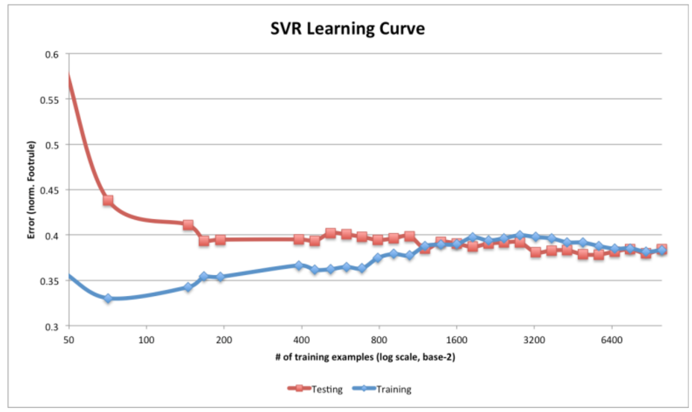

Figure 4.2.7: Learning Curve for SVR

underfitting the data and has high bias. This interpretation makes sense, because our 20 or so features could not be expected to capture all of the complexities of the thousands of submissions. Therefore, a reasonable approach to improving our model would be to add features.

4.3: Meme Clustering

The Meme Clustering algorithm was our attempt to address the issue of lack of "good" features. Our definition of Meme is a phrase or a variant of that phrase, which become very popular occur repeatedly during a given time period. With this definition in mind, we implemented the following algorithm using K-means model in **Figure 4.3.2**. A proof of the validity of this algorithm is attached in **Appendix 6.1**.

Some human-readable sample outputs of this are as follows (we de-stemmed the words to make them more human-readable) are shown in **Figure 4.2.7**.

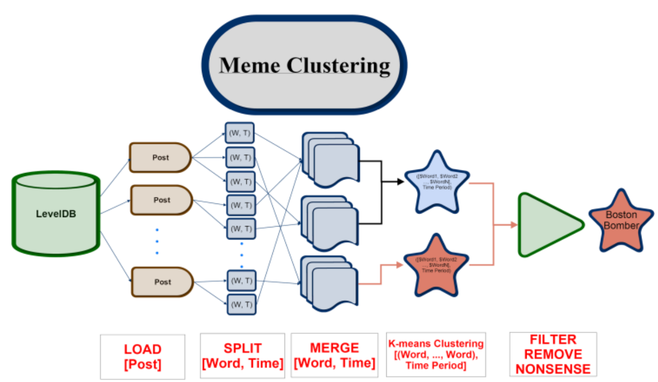

Figure 4.3.1: Implementation of Clustering

```
WT = [] # WT means word_time_pairs
For each post in posts and comments:
  For each word in post:
    skip if word is too common (such as "the", "a")
    if word occurrence > threshold occurrence:
      put (word, t_start, t_end) in word_time_pairs
initialize n clusters
run K-means with distance metric D
filter clusters with < LB WT or > UB WT
  (we fixed LB = 2 and varied UB in {6,7,8,9,10})
define D(WT wt1, WT wt2):
  assign smaller distance based on the following priority:
    1. percentage of time_period overlap
    2. nearness of occurring frequencies
```

Figure 4.3.2: Pseudocode for Clustering

Centroid	Time Stamp
['Boston', 'bomber']	Nov/06/2014
['Amsterdam', 'train', 'saving', 'American']	Aug/21/2015
['Paris', 'AK', '47']	Aug/21/2015
['handsome', 'criminal', 'sentenced']	May/24/2015

Figure 4.2.7: Example Clusters

We can see from **Figure 4.2.7** that we have learned some useful memes corresponding to news events of the given timestamp. The results show that memes do exist in reddit posts and is an interesting application by itself already. However, we were not able to find enough memes (more than 10% of the total number of posts would have been ideal) to have a significant impact on our regression algorithms.

4.4: Logistic Regression

The clustering implementation attempted to achieve capture some semantics of the comments by looking at what words were used together. By looking at which words were associated with each other, we could generate features based on these pairs of words, instead of single words like Naive Bayes does. However, after looking at the meaningful groups created by our clustering algorithm, we found that the majority of these clusters were simply words that were placed next to each other. This makes sense, because adjacent words are likely to be associated with each other. Therefore, to simplify our addition of word clusters, we decided to simply add groups of words next to each other (bigrams and trigrams).

Another issue that we found in regression was attempts to add bigrams and trigrams failed. This is because the size of features became too large for an SVR to compute, since the memory required because extremely large. From this point, it made sense to keep our same goal, to compare comments and find the better comment, but now change it so that we only compare comments. Note that one could think of the original ranking problem as simply repeating these comparisons sequentially. Therefore, our new problem is simply a classification problem where each submission consists of a good comment and a poor comment. The algorithm simply must correctly label the comments based on the features. For the classification problem, we used logistic regression (see **Figure 4.4.1**).

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}},$$

Figure 4.4.1: Logistic Regression

Using logistic regression, we are able to get achieve a testing error of 0.335 using all of the features, including measurements on the posts, time, and bigram/trigram frequencies. In **Figure 4.5.2**, the learning curve for this algorithm shows that the error does not converge even after using all 18,000 submissions. This means that we are overfitting the data. This makes sense, because the number of features increases with each submission, since we introduce new bigrams and trigrams each comment. We would never expect to be able to utilize all of these features without some overfitting. However, since the testing error with these features decreases, we know that some of bigrams/trigrams help our prediction.

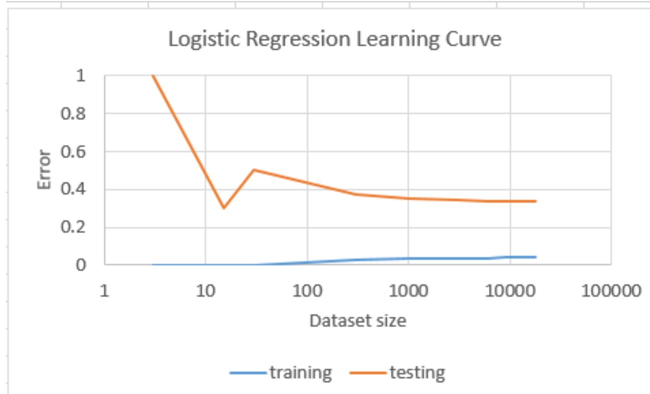

Figure 4.4.2: Logistic Regression Learning Curve

Another analysis looks at the precision and recall of an algorithm. The precision specifies how well the algorithm performs at correctly identifying a comment as "good," while the recall is how many of the "good" posts were correctly found. By modifying the threshold score necessary to consider a comment as "good," we can move along this precision-recall tradeoff (see **Figure 4.4.3**). From this graph, we see that for perfect recall, we end up having a 50% accuracy. This makes sense, since exactly half of our comments will be labeled "good." Also, if we want to increase the threshold, we limit our analysis to posts that we are very sure are "good," since they score so high. In the most extreme case, we have an 80% accuracy/precision, but also classify a vast majority of the "good" posts as bad, meaning low recall. We use a balance of these two extremes (i.e., we do not have a preference between false positives and false negatives) by choosing the point at the knee of the graph with 0.66 precision and 0.67 recall. This data point corresponds to a threshold of 0.5, which verifies our threshold choice for the learning rate. The table of true/false positive/negatives are shown in **Figure 4.4.4** for a threshold of 0.5.

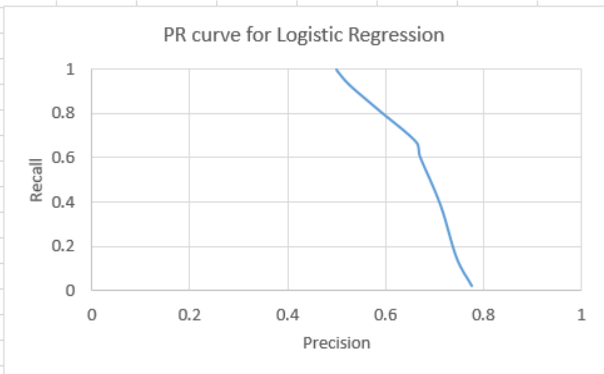

Figure 4.4.3: PR Curve for Logistic Regression

	TRUE	FALSE
Positive	2248	1145
Negative	2188	1085

Figure 4.4.4: Classification for Threshold=0.5

5. Conclusion and Future Work

In this project, we designed several different machine learning algorithms to rank comments. The Naïve Bayes algorithm was not able to extract sufficient information out of large datasets, because of the NB assumption. We solved this by SVR implementation that could rank groups of comments with an error of 0.377. However, it lacked the semantic features found in clustering. By including bigrams/trigrams, our logistic regression could classify a modified version of our original problem with an error of 0.355.

To improve our ranking algorithm, our hypothesis that the inclusion of certain keywords, in the form of bigrams/trigrams/urls, was significantly correlated with the ranking turned out to be invalid. However, by improving our clustering algorithm or researching alternative means to group together words, we can instead start ranking posts based on their content. In general, for subreddits where the users are likely to upvote based on the meaning of a comment, which suggests these new features would perform well.

6.1: Proof of Meme Clustering

```
WT = [] # WT means word_time_pairs
For each post in posts and comments:
  For each word in post:
    skip if word is too common (such as "the", "a")
    if word occurrence > threshold occurrence:
      put (word, t_start, t_end) in word_time_pairs
initialize n clusters
run K-means with distance metric D
filter clusters with < LB WT or > UB WT
  (we fixed LB = 2 and varied UB in {6,7,8,9,10})
define D(WT wt1, WT wt2):
  assign smaller distance based on the following priority:
    1. percentage of time_period overlap
    2. nearness of occurring frequencies
```

The above is a portion of the Meme Clustering algorithm. I will prove by showing that 1. words from a meme are passed into the clusters, 2. others are not, and 3. words from a meme will be moved to the same cluster:

1. suppose a meme M = w_1, w_2, ..., w_n

By definition, all w_i's will occur repeatedly during a certain given period, t. For these w_i's, word occurrence > threshold occurrence. Hence they will be added to the clusters.

2. suppose a non-meme consists of words, v_1, v_2, ..., v_m

We simply negate the definition of a meme and see that the condition, word occurrence > threshold occurrence, for all such v_i's will not be true (unless it is the same as one of the w_i's in which case it will be joined by K-means)

3. since the distance metric gives higher priority to percentage of time_period overlap, all w_i's will necessarily be closer to each other since they came from the same time_period.

Q.E.D

Remark: the drawback to this algorithm is that single-word-memes or several memes occurring during the same time will be clustered into a centroid. We simply ignore these cases as they are rare.

6.2: Proof of normalization of Spearman's Footrule

Need to proof: 0<=F'(sigma)<=1, where sigma is any permutation of a ranking vector v and F' is our normalized version of Spearman's Footrule.

We note that it suffices to show 0<=F(sigma)<=F(reverse)=max(F(sigma)), where reverse is simply a reverse ordering of v.

0<=F(sigma)

This is obvious since F is a summation of absolute values with equality when sigma is the identity.

F(reverse)=max(F(sigma))

let v=(1,2,...,n)

sigma(v)=(sigma(1),sigma(2),...,sigma(n))

We note that any permutation of length k can be expressed as a product of permutations of length 2

Therefore, we can write sigma=sigma_1*sigma_2*...*sigma_m where all sigma_i's are permutations of length 2

Hence we have sigma(v)=delta(sigma_m(v)) where delta=sigma_1*sigma_2*...*sigma_(m-1).

This is a recursive equation and we can solve it by greedy algorithm, the sigma_m that gives the largest value is clearly the swapping of 1 and n, (1,n).

By induction on n, we are done.

Q.E.D

7: References

[1] Diaconis, Persi, and R. L. Graham. “Spearman's Footrule as a Measure of Disarray”. *Journal of the Royal Statistical Society. Series B (Methodological)* 39.2 (1977): 262–268. Web...

[2] Clarkson, Philip, and Ronald Rosenfeld. "Statistical language modeling using the CMU-cambridge toolkit." *Eurospeech*. Vol. 97. 1997.

[3] Balchandran, Rajesh, and Linda Boyer. "Identification and rejection of meaningless input during natural language classification." U.S. Patent No. 7,707,027. 27 Apr. 2010.

[4] Fetterly, Dennis, Mark Manasse, and Marc Najork. "Detecting phrase-level duplication on the world wide web." *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2005.

[5] Leskovec, Jure, Lars Backstrom, and Jon Kleinberg. "Meme-tracking and the dynamics of the news cycle." *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009.

[6] Senior, Trevor. *Snoocore*. https://github.com/trevorsenior/snoocore. GitHub, 2015.

[7] Umbel, Chris. *Natural*. https://github.com/NaturalNode/natural. GitHub, 2015.

[8] *Scikit Learn*. https://github.com/scikit-learn/scikit-learn. GitHub, 2015.