TagRefinery: A Visual Tool for Tag Wrangling
Appendices

APPENDIX A: CALCULATING TAG QUALITY FOR THE LAST.FM DATASET

For Last.fm tags, we used the formula in Equation 1 to calculate tag quality $Q(t)$ per tag t. Over all songs s, we sum contribution of all (tag,song) pairs in the dataset to a tag's overall quality $Q(t)$. The Last.fm weight $w(t,s)$ is the percentage of taggers of a song who applied a certain tag to the song, *listeners(s)* is the number of users who listened to a song, and *playcount(s)* is the total number of times a song has been played. The overall quality $Q(t)$ can be normalized over the highest quality tag in the dataset. Therefore, the more occurrences a tag has and the more popular the songs in which it appears in are, the higher its tag quality will be. Due to the fact that highly popular songs get significantly more listens than less popular ones, *listeners(s)* and *playcount(s)* showed an extreme long-tail distribution. As such, we used logarithms in the formula to lessen the effects of this phenomenon.

$$Q(t) = \sum_{\text{all songs } s} w(t,s) \times (\log(\text{listeners}(s)) + \log(\text{playcount}(s))) \tag{1}$$

APPENDIX B : WORKFLOW IN DETAIL

1. Decomposition

The Decomposition step takes all tag/item pairs from the input dataset, changes the case of all characters to lower case and reduces multiple spaces to one space. Next, all tags are decomposed into single words [1] in order to perform spell-correction in future steps. Later on, multi-word tags are reconstructed from these words according to the initial dataset. Quality of words that appear only as part of another tag is set as the quality of the tag they are derived from. When a word appears more than once (coming from different original tags), its quality is set as the maximum of all its parents' qualities.

2. Pre-Filtering

The Pre-Filter step is intended to reduce the computation time of the work-flow by removing extremely rare words from the computation. This step is important if the dataset ranges in the hundreds of thousands of tags (so that there are tens of thousands of unique words). It is possible to add those words again after one has finished work on refining the parameters in later steps, for one long computation phase. This increases the amount of salvaged words due to the fact that a lot of spelling errors and uncommon word forms are really rare.

3. Black-listing

The user can at this phase import lists of tags, words and characters that will be removed from the dataset or replaced with others. For instance, some users might need to remove all stop-words and prepositions or special characters such as - _ : ; /.

4. White-listing

The White-list step reads all rows from the input csv and interprets all words in it as ground-truth, for the spell correction step and all multi-word tags as ground-truth for the multi-word detection step.

5. Spell-correction

This step corrects spelling errors and consolidates various forms of the same word. This is accomplished by creating a character 2-gram set for each word (rock turns into [_r, ro, oc, ck, k_]) and computing the Jaccard index between different sets. The Jaccard distance is the fraction of common 2-grams from both sets (intersection) over the number of total unique 2-grams occuring in both sets (union). If two words are similar enough, the lower quality word will be replaced by the higher quality one. This method has the advantage of complete language independence. This is especially important in complex datasets like Last.fm, where any language can be used. Thus, we used 2-grams instead of the popular language dependent method of stemming. Another common similarity measure is The Damerau-Levenshtein distance. We dropped this measure due to its very high computation-time and our requirement of real-time feedback in the interface.

There are two user customizable thresholds in this step; one dictates the required degree of Jaccard similarity between two words for a replacement to happen, and the other is a threshold on word quality, which is used for designating a subset of the words as "correct". These words, along with any words in the white-list (previous step) are safe from being replaced, and can replace other similar words. Besides, the tool provides graphical means for the user to manually reject any replacements that will be made given the two thresholds. This is one of the more time consuming steps with a complexity of $O((w+i) \times (n-i))$, where w is the number of white-listed words, i is the number of words chosen by the user as ground-truth with the word quality threshold, and n is the total number of words. To compute the similarity between a pair of words, the first step is the decomposition of the two words into 2-gram sets, an example of which is shown in Algorithm 1.

---

[1] The length of Last.fm tags can range between one and 67 words.

The Jaccard Index between these two sets is computed using Equation 2 .

```
// Words for the similarity computation
a = rock;
b = roock;
// The 2-gram sets of each of those
   words
A = { r,ro,oc,ck,k };
B = { r,ro,oo,oc,ck,k };
// The Jaccard index of the two example
   sets as shown in 2
J(A, B) = 5/6 = 0.83;
// The similarity between those two
   words is 83.3%
```

Algorithm 1: Example similarity computation

$$J(A,B) = \frac{|A \cup B|}{|A \cap B|} \tag{2}$$

More specifically, spell correction has the following procedure:

1. All single words from the imported ground truth list will be added to the correct words.
2. The user sets a word quality threshold and a similarity threshold.
3. All words above or equal to this threshold will be added to the list of correct words.
4. For each word in the correct words list, the similarity to all other words (except the words in the correct words list) in the dataset will be computed using Algorithm 1.
5. Each word which has a similarity above or equal to the threshold will be added to a replacement list.
6. If there are multiple replacements for the same word, the word with the highest word quality will be used as the correct word.
7. The replacement list will be applied to the dataset.

6. Multi-word tag detection

The Multi-Word Tag Detection step consists of two different algorithms for detecting frequent and unique multi-word tags. The frequent version counts all occurrences of a all word sequences and normalizes them to [0,1]. With this algorithm frequent groups such as "hard rock" have a high value. The unique version uses again the Jaccard Index as measure, which is computed as the sum of co-occurrences divided by the sum of all single word occurrences. This algorithm gives high values for groups where the words in the group occur mostly in this combination and not individually.

7. Post-Filtering

After multi-word tags are re-constructed, tag quality is computed again for all tags. In this step the user can set a threshold on the tag quality to reduce the total amount of tags in the final dataset, as some use cases might require smaller and higher quality sets of tags.

8. Manual polishing

This step is intended for manually editing the post-filtered tags. Here the user can remove synonyms or translate tags from different languages to one. Also the user might decide to remove words which do not fit into his use case or rename multiple words to create custom clusters.

9. Information salvaging

The last step uses the post-filtered and polished tags as a basis for finding useful tags in the rest of the dataset. The salvager uses two different equations which are the greedy method (Equation 3) and the conservative method (Equation 4). The greedy salvager is used for all cases but with single word tags of a length shorter than three characters. This is because when word length is this short, the greedy salvager might salvage wrong or misleading tags. For instance, the music genre "emo" is salvaged out of the previously removed tag "emoticon", which means wrong information is introduced into the dataset, as these two tags are not related. An example with default settings: "rock" and "emo" are in the final tag list. "emo-song", "emoticon", "tree" and "poprock" have been previously removed in Pre-Filtering. The salvager would turn "emo-song" into "emo" , "emoticon" and "tree" will be removed and "poprock" would turn into "rock".

$$Greedy\ regular\ expression : ".*TAG.*" \tag{3}$$

$$Conservative\ regular\ expression : "\backslash sTAGs\backslash" \tag{4}$$

APPENDIX C: IMPLEMENTATION

TagRefinery is built as a server/client application. The server is written in Java and has a multilayer design. As transport layer, the open source Socket.IO server implementation Netty-socketio [2] is used which provides the web socket interface the client connects to. The second layer is responsible for the workflow and keeps track of all the states in the system. Finally, the lowest layer handles the different computations needed in the system. The client is built in AngularJS [3] and uses D3 [4] for the visualisations. All computations are performed server-side and the client works only as graphical user interface. The usage of Java and Javascript makes the tool deployable on all platforms with the only dependency being Java 1.8.

---

[2] https://github.com/mrniko/netty-socketio

[3] https://angularjs.org

[4] https://d3js.org