Emojification: Putting a Face to a Tweet

Yuki Zaninovich

Abstract

Emojis are small digital images that help express emotion and show a recent trend in digital, written communication. Because of the vast number and diversity in meanings of such emojis, they have the potential to provide sentiment analysis of higher degrees, as standard methods are either binary or trinary. This project explores the extent to which Natural Language Processing (NLP) can aid in predicting the emoji that would most likely accompany a given sentence. Twitter posts that contain at least one emoji were scraped and pre-processed before being analyzed. A Maxent model trained with word identity, emoji correlation, and sentiment values was used to classify the "best representing" emoji. The accuracy of this classifier was then compared against a baseline information retrieval algorithm.

Introduction

Whether it be helping make decisions on stock purchase or gauge the public opinion on a current event, analysis of Twitter posts has been a recent trend in the NLP world. The most common form of analysis is sentiment analysis, in which programmers build NLP algorithms to understand the emotions of social media users. However, such forms of analysis are typically binary classification into positive or negative sentiment, and we can only measure the extent to which a person is happy or sad if the values aren't discretized.

On the other hand, analyzing emojis can be very productive. Since there are over 800 of them and can accompany posts in multiples, it significantly raises the ceiling on the dimensionality of sentiment analysis. Unfortunately, just like languages, each emoji doesn't have a one-to-one mapping between usage and meaning: the fire emoji can mean that something is cool or trendy (due to the slang word "lit") or is literally on fire. This leads to ambiguities, and the context must be processed and understood in order to make accurate conclusions of its meaning. These specific conditions make NLP the perfect toolbox for puzzling out the problem, as it is equipped with a plethora of methods adept to disambiguating complex problems involving the human language.

Dataset(s)

Twitter posts were the dataset of choice for this project, as they are the main platform for most standard sentiment analysis and is potent of emojis just like any other social media platform. The Twitter API was used to retrieve 5,500 and 10,000 Tweets that contained 2 and 20 different types of emojis respectively and were saved into text files.

When reading in these Tweets for analysis, a host of pre-processing transpired, including lowercasing and removing user tags and white spaces to allow for better generalization and

prevent overfitting, which occurs when a classifier trains itself too specifically to a particular training set. Spell-check was not implemented in order to stop feature obfuscation from happening (e.g. vowel repetition indicating excitement). The emojis that were originally in the post were also removed and stored as class labels for that instance. It is important to note that every Tweet can contain multiple different emojis, giving each instance more than one truth value. Roughly 90% of the dataset was for training the classifier, a proportion chosen to adhere to the 10% inference rule.

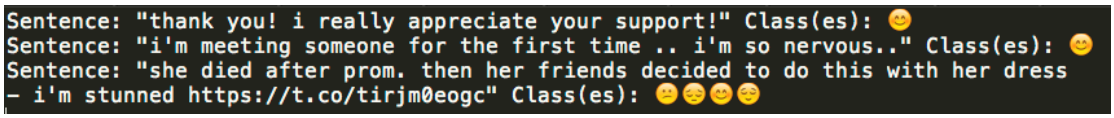

Figure 1: Even smiley emojis are ambiguous in usage and can appear in multiples

Baseline method

An information retrieval algorithm with word count vectors was used to baseline my experimental method. As shown in class, the word count vector model is quite simplistic because its sole feature is precisely the frequency of words and gives no thought to other spatially correlated characteristics.

Every emoji was treated as its own document, and posts were “bucketed” into these documents according to the emoji they contain before being converted into vectors. The Cosine Similarity algorithm was implemented as shown in class, with the exception of not having a denominator since it does not affect the distance between vectors and we are not concerned with the magnitude of the vectors.

Experimental method

For this problem, Maxent was chosen because it is most reliable when trying to classify entire instances into discrete classes. All weight optimization and joint probability generation was handled by NLTK’s MaxEntClassifier class, and the max number of iterations was set to 10 in an effort to reduce chances of overfitting and bring down time cost. Features thought to be most relevant to the problem were implemented and utilized, including word identity, emoji correlation, and sentiment analysis with sarcasm detection.

Word Identity

This is the most straightforward of the features chosen. The role of emojis is to evoke emotion in text, so it would make sense for words that do so explicitly would have a high correlation with said emoji, especially for those that depict specific nouns; the only exception is when such emojis are used to substitute the actual word itself, e.g. "Lmao who remember the \"I <love emoji> Boobies\" bracelets” but its occurrences are so low (15 in 10,000 of this dataset) that they can be counted as outliers.

Features were assigned binary values indicating whether the word existed anywhere in the Tweet. Because of this, the one immediate downside that surfaces is its neglect of word orientation; one can randomize the order of all words within an instance and the feature will return the exact same result (bigrams were not implemented due to their significant pre-processing time overhead and inability to increase scope of project). This can be catastrophic especially when concerning negation words, which can turn sentences containing even the most positively or negatively correlated word to be misleading. However, this problem is handled by the sentiment feature.

Sentiment

Due to the high relevance of sentiment in emoji detection described earlier, it would also make sense to include it as a feature.

Sentiments of entire sentences were captured using the SentimentIntensityAnalyzer class from nltk.sentiment.vader. This package was chosen amongst other numerous others because of its proven ability to handle negation words and slang, which are characteristics not handled by the unigram feature. Features were assigned discrete values depending on whether the analyzer detected positive, neutral, or negative sentiment from the post the most.

The one base this analyzer does not cover is how sentiment can differ continuously within posts, especially in posts involving sarcasm. For example, the sentence “I absolutely love being cheated on.” is classified as positive sentiment because the strength of “love” as a positive sentiment word eclipses the negative sentiment emitted by “cheated” because there are no negation words involved. Had the sentence been processed in parts, the analyzer would have noticed there were counteracting forces within the sentence.

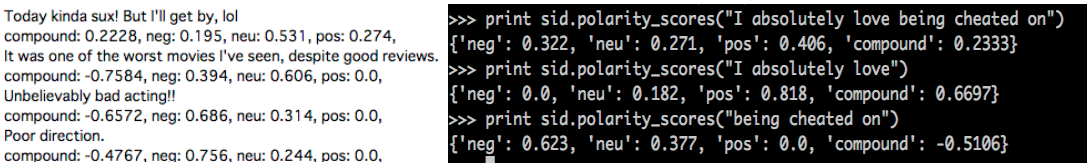

Figure 2: SentimentIntensityAnalyzer handles negation and slang but not sarcasm

Because these inflection points can occur in more than one location within a sentence (e.g. “a person who is nice to you but is rude to the waiter is not a nice person.”), there are 2^(# words in post) permutations of inflection points, which can be difficult to evaluate brute force. This alongside knowledge of dynamic programming algorithms learned in class (which were bottom up) was the inspiration for a sarcasm detection algorithm that is top down and brings the time complexity to polynomial time O(n^3). Below is the pseudocode:

```
TRAINING

For all strictly positive or negative Tweets:
    sentence_sentiment = sentiment of sentence
    for all adjacent pairs of words x, y in Tweet:
        compute distance between sentence_sentiment and sentiment of two sub-sentences,
        where first sentence ends with x and second starts with y

mean = average distance between non-sarcastic tweet sentiment and sentiment of two subsentences
std_dev = standard deviation of this mean

TESTING

sentence_sentiment = sentiment of query
for all adjacent pairs of words x, y in query:
    -compute distance between sentence_sentiment and sentiment of two sub-sentences,
    where first sentence ends with x and second starts with y
    -normalize this value using mean and std_dev

max_outlier = max normalized value that is at least 2 standard deviations away from mean (0 if doesn't exist).
split at index that max_outlier was found, and recurse on both sub-sentences until no max_outlier is found.
```

Figure 3: Pseucode for Sarcasm Detection

The intuition behind this new algorithm is that the average deviation of sentiment of subsentences from that of an entire, unsarcastic sentence is normally distributed. This was conjectured due to the observation that strictly positive sentences would not have negation words or negative sentiment words, so no matter where you "cut", you will still end up with a positive sentiment subsentence. If this is the case, then one could detect sarcastic sentences by normalizing its subsentences' distance from the overall and determining if it is likely that sentence came from the same distribution, otherwise known as a Z-test. Here we use 2 as a threshold, meaning any sentence with a normalized value less than 2 or greater than -2 is deemed to be part of the distribution, which was arbitrarily chosen as it is the default threshold in most statistical analyses.

Emoji Correlation

Lastly, we investigate the relationship between differing emojis appearing simultaneously in posts. This feature was ideated with the assumption that emojis are just like words in that there is semantic dependence amongst them (though not syntactic like "San Francisco" because there is syntactic independence). The only difference here is that there is no real spatial relationship, as we are treating emojis as a characteristic of the instance rather than a part of the language of it. This is why I took a unigram-like approach again over various Markov Models taught in class because Markov Models classify parts of instances and individual states rather than entire sentences.

Evaluation

Below are the accuracy results for each method, alongside differing combinations of features.

P.S. Please feel free to try out my classifier! Just type python emojifier.py (you'll need NLTK installed) and when it prompts you, type a Tweet enclosed in "".

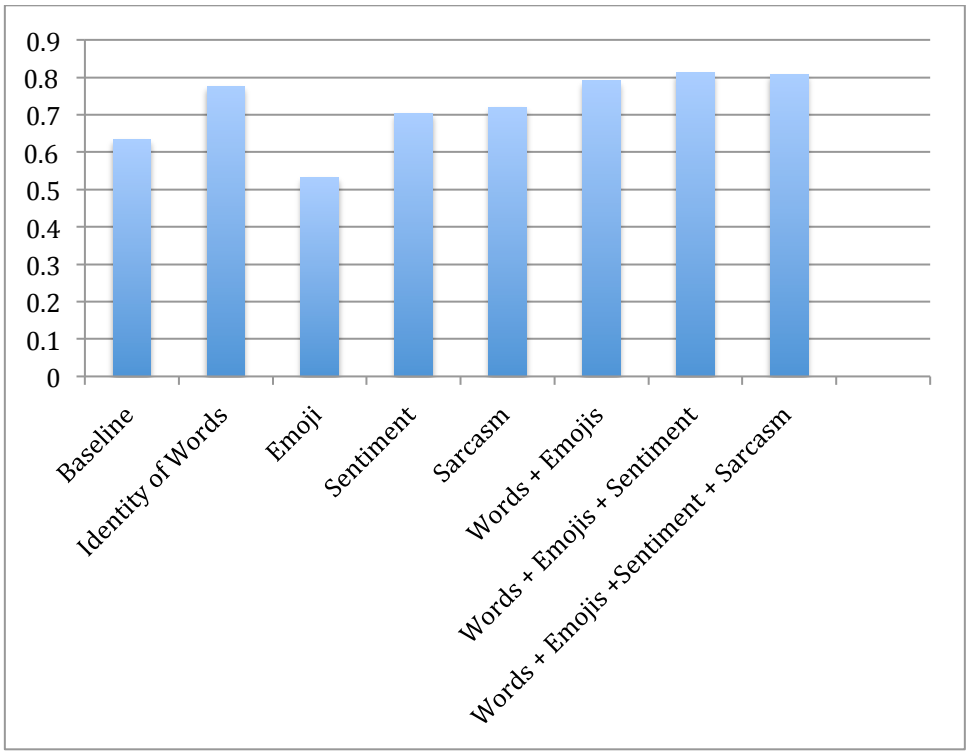

Figure 4: 2 Classes

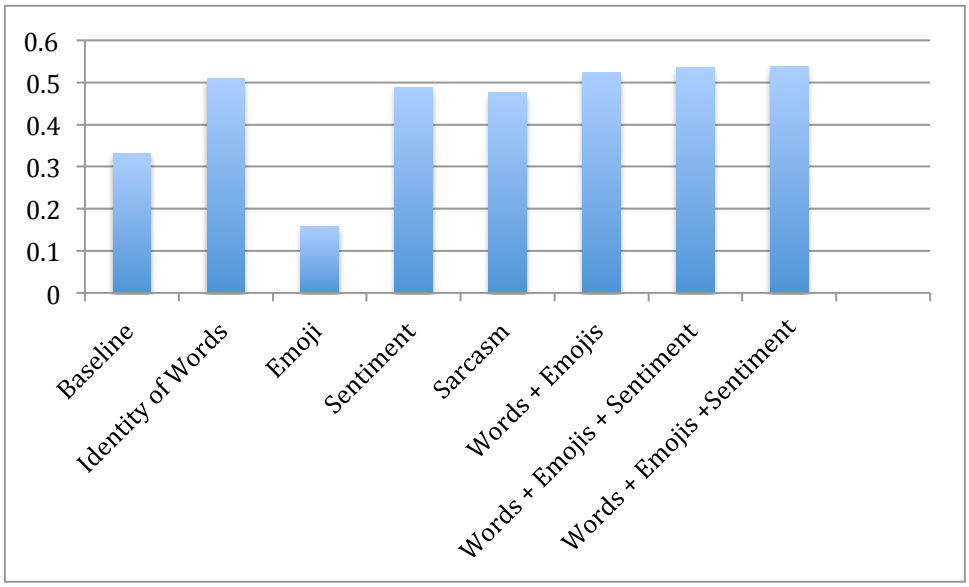

Figure 5: 20 Classes

Luckily, the experimental method outperformed the baseline across most feature usage (other than emoji correlation), accuracy increased with more features, and accuracy unanimously decreased between 2 and 20 classes. Here we investigate the successes of individual features.

Identity of Words is the clear winner for most essential feature.

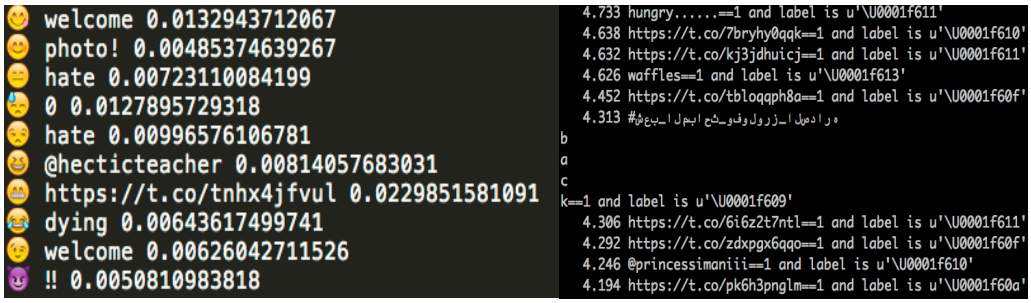

Figure 6: Most correlated feature and its weight of emojis with 2 classes for baseline (left) and maxent with all features included (right)

As we observe the weights of the most correlated features for the emojis, we see why it yields such a high accuracy: 70% of the top 10 informative features for 2 classes and an astounding 100% for 20 classes are word identity features. The identities of these "best features" also gives us insight on why overfitting is so extreme: most of the top 100 informative features for both the baseline and MaxEnt are links to websites and tags of users. This must have resulted because I duplicates were not removed, so spammers who posted website links on numerous posts caused the classifier to think the links were highly correlated with the emoji they used. My prediction on how emojis that depict nouns having the most weighted feature seems to have been incorrect, as all of the emojis with highest correlated features are faces (indicated by how their Unicode end in …1f6XX). Perhaps they were not represented enough in the data, as the most used emojis are faces, to be visible enough to the classifier.

On the other hand, emotion correlations were quite the letdown. On average, every Tweet contained 1.8085454545 emojis, so we have a scarcity problem this time (unlike for word identities when we had too many spammers). However, this was not due to incorrect pre-processing, but helped identify a characteristic of our problem domain, which is that emojis do not frequently accompany one another.

It is quite impressive how my sarcasm detector matched the nltk sentiment analyzer. As we can see below, it was successful in detecting not only obvious sarcastic tweets, but also ones that had multiple layers of negation.

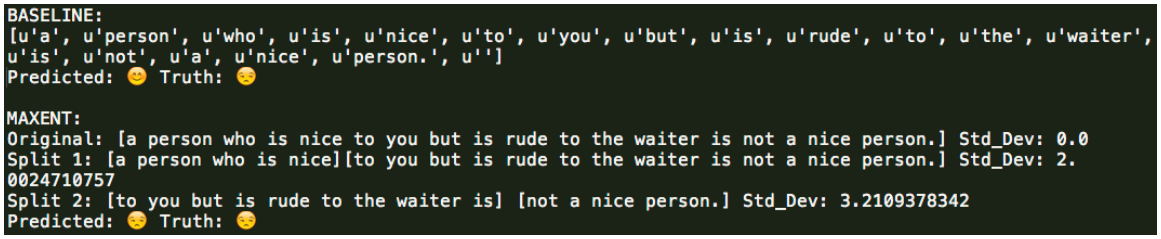

Figure 7: Sarcasm Detector correctly splitting example from before

```
Original: [i absolutely love being cheated on.] Std_Dev: 0.0
Split 1: [i absolutely love] [being cheated on.] Std_Dev: 2.98
```

```
Original: [somehow i got a 97 on my spanish exam but failed my pre calc exam ??? okay] Std_Dev: 0.0
Split 1: [somehow i got a 97 on my spanish exam but] [failed my pre calc exam ??? okay] Std_Dev: 2.82021866669
```

As one can tell, the algorithm seems to also just split at negation words that were either preceded or followed by positive phrases, which explains why accuracy of running MaxEnt exclusively with Sarcasm Detector (perhaps not the most apt name anymore) almost matches the accuracy of running exclusively with generic sentiment analysis. But alas, everything doesn't go as planned.

```
Original: [Dont let this sick world hold you back, speak your language be proud of your
culture https://t.co/gwdfpirkft] Std_Dev: 0.0
Split 1: Dont let this sick] [world hold you back, speak your language be proud of your
culture] Std_Dev: 2.33399036934
```

In this example, although one can observe that there is indeed a disparity in sentiment, the division did not occur in the correct location. This is because the classifier does not detect punctuation since it was not tokenized; had it noticed the comma individually, perhaps it would have learned what independent clauses are and divided accordingly.

Conclusions

Within the domain of this problem, 50% accuracy for 20 emojis is viable. A coin flip is much more comforting to be taken on emoji recommendation than more sensitive applications like cancer detection, and it is likely the user does not necessarily need the "best" emoji for texting if this was implemented as text recommendation.

However, pre-processing was definitely something that I could have done better. Not detecting duplicate Tweets was a big oversight on my part, and was the main reason overfitting occurred, at least on the most informative features. Implementing some sort of L1 regularization could have also helped tremendously (though there could have been losses on predictive power). Tokenizing punctuation would have also helped my Sarcasm Detector as previously discussed.

I am slightly disappointed in the contribution of the Sarcasm Detector to the overall accuracy, so there could either be more tweaking done on its parameters (optimize Z-test threshold, use better sentence analyzer, pre-process more, etc.) or be revisited as a whole with machine learning algorithms that are not NLP specific such as Support Vector Machines. It would also be interesting to actually implement this in a mobile phone and test its performance on real social media users. It could do online learning by observing the incremental usage of emojis of that certain user and freely overfit as it pleases. The most similar implementation that already exists is on iMessage, in which the emojis are suggested as substitutes as words (which I assume is implemented using n-grams because

it is quite identical to text recommendation). Overall, I feel the results certainly exceeded expectations and am optimistic of the value in pursuing real-world implementation ☺