Final Project Report:
Crouching Tiger Hidden Markov Model

Alex Baucom, Steven Chen, and Brad Thompson

Introduction

This report aims to detail the accomplishments of the team Crouching Tiger Hidden Markov Model in the 2015 CIS 520 Machine Learning Competition at Penn. In this report we will cover many of the simple models we tried and explain what was good and bad about each of them. Also, we will discuss many of the ideas that we were able to incorporate into our final model, which ended up winning 1st place by a substantial margin. In addition, we will look at some of the other ideas that we tried, but did not help improve our final model. Lastly, we will summarize our overall results and show some interesting visualizations of our model as well as provide some insights into what we think might have been happening behind the scenes.

Individual Models

At the outset of the project, our goal was to find some of the best models individually and then combine them at a later stage. This section details some of our various models that we experimented with in order to find the ones that worked the best. A summary of all of our methods and their accuracy can be found in Table 1 under the Results section.

Nearest Neighbors

An easy model to implement in MATLAB was a KNN model on the words. The reasoning behind this was that maybe a KNN model could capture similarities between genders across all the words and not just look at a few words here or there. Instead of trying to find a handful of really important words, KNN would be able to find the closest matches in terms of word choices and frequency, which, in theory, would indicate the same genders. However, our cross validation resulted in a 32% loss, which wasn't a very promising start and so this method was quickly abandoned. We think KNN performed so poorly because there was not a large enough number of samples to compare each person to, so there were never very many neighbors that were actually close by, resulting in a lot of misclassification.

Naïve Bayes

Another easy model to quickly throw together was a Naïve Bayes model on the words. In class, we talked about these being useful for the 'bag-of-words' model, which is exactly what we had in this case. A quick cross validation test showed that Naïve Bayes had a 21% error, which was quite a step up from the KNN model, but still not great, considering at this point we were still trying to beat the 86% accuracy baseline.

Undeterred, we thought maybe adding more features and doing some sort of combined model, like we talked about in class, would help. So we added in the image features since those would be easier to deal with than the images. We decided a simple regression might work well for the features so we ran LASSO on the image features (which had a 31% error by itself) and then regressed the outputs from the Naïve Bayes model on the words and the LASSO on the features together using the LASSO method again.

Needless to say, this haphazard method did not fare so well, and managed 21% error in cross validation and 0.7881 on the leaderboard. This suggests that the feature model did not add any new information to the overall model since the overall model had almost exactly the same error as the Naïve Bayes model alone.

Looking back now, it is clear that there were some major shortcomings with this model. Firstly, the Naïve Bayes assumption of conditional independence is clearly not valid for the words we have; however, it often can work well even when the assumption is not valid, so maybe more tweaking with that model could have produced better results. Second, we didn't perform any sort of normalization on the features and, since they were definitely not all of the same scale and unit type, this likely caused problems with the feature model. Third, we ran the final regression on the binary prediction outputs from the first two models, which amounted to the Naïve Bayes model getting weighted higher and outvoting the feature model every time there was a disagreement. Clearly, we were going to need to get more clever with our models and not just throw any random method at the problem and expect it to work.

Boosting

This time we decided to do a bit more research on various methods and what they were best for before trying them out. We figured that the words probably had the most information that would be easily accessible, so we started with them. A bit of Googling later and we found a Microsoft Machine Learning Algorithm Cheat Sheet that suggested boosted decision trees might work well for the words. MATLAB has a very nice, built-in method for fitting boosted trees so we tried using AdaBoost on simple decision stumps. After running cross validation on several different values for the number of weak learners we created the graph shown in Figure 1. This suggested that using AdaBoost with 2000 decision stumps could achieve an error of only 12%. So we tried a submission to the leaderboard and sure enough: 0.8823.

This worked surprisingly well for just throwing one method at the problem. This is likely because using so many weak learners allowed a very nonlinear decision surface to be learned that captured a lot more information about the words and their indications as to the gender of the person. Interestingly, Figure 1 seems to indicate that even using only 100-200 weak learners is enough to capture a vast majority of the information. This makes sense, as there are likely only a hundred or so words that are *very* distinctly male or female and, once we go much past that, so many words will be common to both genders that they will not help very much, which is why so many more iterations are required to tease out only a little bit more information here. For more in-depth results and analysis from this model, please see the Results section.

Later on in the project, we did some testing with various methods of boosting and concluded that using LogitBoost did slightly better than AdaBoost (see Table 1 in Results section), so that method was what got incorporated into our final model, even though all our initial results were with AdaBoost. Additionally, due to file size limitations, in the final model our boosting method was reduced to 1000 stumps to conserve space.

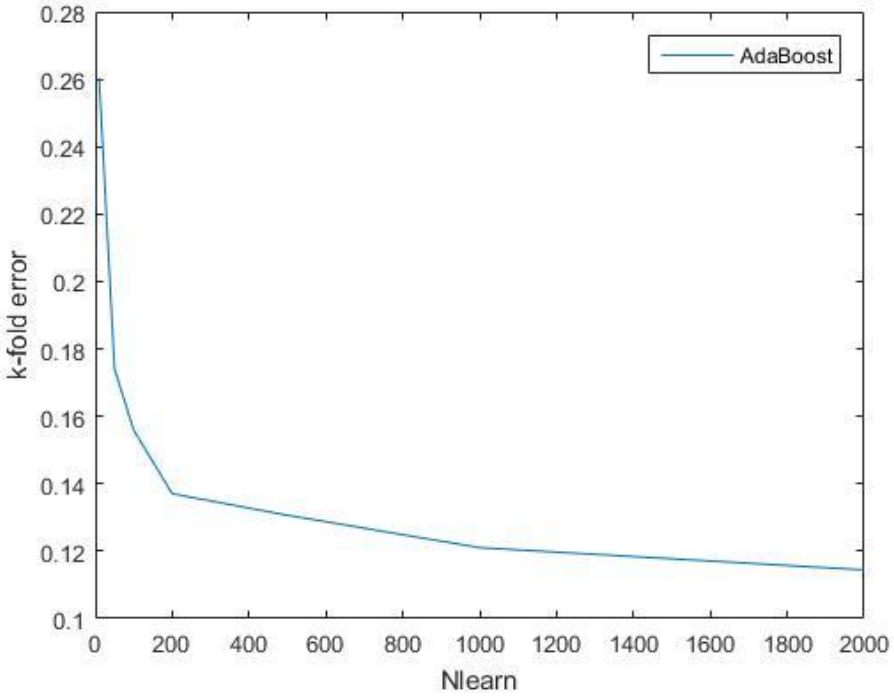

FIGURE 1. COMPLEXITY VS. ERROR FOR ADABOOST METHOD ON WORDS

Image Analysis

Once we had a fairly good model of the words, we decided to turn our attention to the images and see what sort of model we could build to make use of them. After more research and experimentation, we settled on using an adaptation of the open source software VLFeat. It had a lot of good image processing and feature extraction that was very useful for machine learning. Additionally, it works very well with MATLAB, which made it easy to integrate into our project.

The heavy processing for the model is in the feature extraction of the images. This was done by using VLFeat to train an unsupervised encoder on the training and testing images. This encoder used dense SIFT to extract feature descriptions, ran PCA on these feature descriptions, augmented the features with their coordinate locations, and then clustered everything using k-means. Once this encoder was trained, any new images would be encoded using this same process, resulting in a huge dimensionality reduction. After this encoding, a simple linear SVM was used to make the final prediction for each observation. On its own, this model achieved 84% accuracy.

The key to the success of this model was the VLFeat library, which we can take no credit for. It was able to easily extract the most useful information from all the images which was what allowed an SVM to make any sort of accurate prediction. Without using any kind of feature extraction and dimensionality reduction, an SVM by itself would be trying to separate which pixels were the most informative and, at that level, there just isn't enough of a correlation between certain pixels and gender to be of any real use.

Going Back to the Words

We thought that the words had the most signal, so we went back and tried to see how other methods would work. We remembered that from previous homework assignments, a histogram intersection kernel SVM worked well with words. We originally tried to adapt the code we submitted for that homework, but it was clear that it would take too long to run. After digging around VLFeat, we found that they had a function to create a histogram intersection kernel and run a SVM on it. VLFeat is written in C++ and integrated into MATLAB using MEX, so it is much faster than code we would write ourselves. After testing, this model achieved 86% accuracy. We then decided to modify the word features and change the absolute counts to percentages. After this modification, this model achieved 89% accuracy. Most importantly, the predictions from this model were 6% different than the predictions from the boosting model, so we realized that both models were getting different examples wrong and combining them into the same model would improve accuracy.

We believe that the 6% difference between the two models is due to the different philosophies of the LogitBoost algorithm and the SVM, as well as the difference between using absolute features and percentage features. The LogitBoost algorithm is trying to minimize a logistic cost function where every training point contributes to the cost, whereas the SVM only considers the support vectors. These are two fundamentally different approaches which we believe would give different predictions to close calls. In addition, by using different types of word features, we hoped to capture different types of signals. For example, percentage word features focuses on the differences in word distributions, whereas the absolute word counts also captures how often a user tweets versus other users. We hoped that these differences would give the models enough independence of each other so that ensembling would boost the overall accuracy.

Building an Ensemble

Once we had a couple good models, we experimented with different methods to combine them. Our thinking was that a good ensemble model would be able to find signals that indicate when each model will predict incorrectly. We first started by taking the boosted word model and the image SVM model and trained a simple linear SVM on the scores. This brought our score up to 92.7% and put us in first place! However, we were afraid of overfitting, and sure enough, when we added in the third model (SVM on the words), our training set became linearly separable and our test accuracy became much worse.

To address the overfitting, we tried using a simple voting model between the three models, but it also gave us a 92.7% accuracy. While a simple voting model is better than using each model individually, it is more of a baseline case for an ensemble model. It is naïve in that it equally weights each model and doesn't try to separate out which situations a specific model will get predictions right or wrong, which is really what we were after.

After searching around a little more, we saw an interesting Kaggle post about stacking. The idea of stacking is that you train one model with half of your data to get scores for the other half and vice versa. This allows the combined model to be trained using only out of sample predictions. To predict on the test set you train each of the individual models on the entire training set and get scores on your test set, and then you use these scores as features in your final model (the stacked model) to make your final predictions.

We thought that this was an interesting concept because it reminded us of a neural network. The higher level is able to become a little more abstract and pick out higher level concepts and get a sense of the bigger picture. In our case, the bigger picture was identifying when a specific model gets predictions wrong. We decided to expand our score features from three to thirteen by including interaction terms between the scores because we thought that the interactions also contained some signal. Once we implemented this stack ensemble method, and reverted back to an SVM on the scores rather than a voting scheme, our score shot up to 94.12%. We tried building different models using the image features, but we were not able to improve our final score, so we decided to not include them (see the Trial and Error section for more details).

One interesting follow up would be to combine all of our words, images, and image feature data and train a joint model on the combined data. We only trained our models independently of each other and tried to capture the interactions through this stacked ensemble method. However, by using the independent score features, we are abstracting away some of the finer detail and probably throwing away some of the interaction signal. For example, we noticed that some of the images were pictures of women, but the words indicate that the user is male, and the actual gender is male. It is probably the case that the user has a picture of his girlfriend or an actress that he likes as his profile picture. When we train our models separately, we get a strong female image score and a strong male word score. It is obvious that an equal voting model would not work well in this case because it would not be able to differentiate that in this situation, the image prediction is wrong. We hoped that the stacked SVM was able to better determine when the word score is more reliable than the image score, and it does perform better, indicating that it is somewhat able to capture this type of interaction. However, there are also probably many situations where there is a strong female image score and a strong word male score, but the user is female. At the actual word, image, and image feature level, this female user may look completely different than the previous male user even though both give the same scores. By training separately and abstracting up a level, we may be throwing away a lot of signal that could help us differentiate these two scenarios and correctly identify the gender in both. By aggregating all of the features together and training a joint model, we might be able to further improve our accuracy.

Final Semi-Supervised Touch

Finally, after getting predictions for all of our test data, we decided to augment our training data with our test data and their predictions and train all of our models again on this augmented set with 10,000 examples. We believe that by including this test data, the benefits of seeing more types of features outweighed the cons of having potentially mislabeled genders and would thus improve our final accuracy. We tested this hypothesis by splitting up our training data into two halves. We used only the first half of the training data to predict the test data and got a 92.8% accuracy. We then used the first half to predict the second half of the training data, and used that composite set to predict our test data getting a 93.5% accuracy. We believe that this semi-supervised addition boosted our score about 0.5-1%.

Trial and Error

The final model we arrived at was the result a long process of trial and error. In this section we will talk about some of the attempts we made that didn't make it into the final model.

Initially we planned to incorporate a model of the features into our final ensemble as well. However, from the beginning it was clear that the features didn't have nearly as much information as the words or the images. In order to combat this limitation we performed a feature expansion on the features first, and then constructed an ensemble of Naïve Bayes, K-means, SVM, and Logistic Regression. We experimented with 2 ensembles, one that used a simple voting scheme, and another that ran a logistic regression on the scores produced by the individual models.

The individual results of these classifiers before feature expansion and ensemble produced accuracies in the range of 66% to 70%, and after the feature expansion and implementing the ensemble, our accuracy climbed as high as 71%. Nonetheless we were disappointed that the feature expansion and ensemble methods weren't increasing our accuracy more. We realized that the ensemble was not particularly effective in this case because the different models were mainly getting the same predictions right and wrong, so averaging across several models wasn't helping.

We concluded that the features by themselves just contained too little information to produce high accuracy even with feature expansion and an ensemble, so we tried to extract additional features of our own from the images. First we took a simple histogram of the colors used in each image. A logistic regression on this feature alone produced accuracies of over 60% so we added this feature to our set.

Next we went after the faces themselves. Using MATLAB's vision.CascadeObjectDetector we attempted to detect the eyes, nose and mouth in each image and take the ratios of the distance between them. However, while MATLAB's eyes, nose, and mouth detector worked fabulously on larger images, it could find all of these feature in only 8% or the twitter images because of their size. Surprisingly, just expanding the images before running the detection increased detection dramatically, but increased the runtime to half an hour.

Finally, we tried an eigenface approach to image analysis, but found out quickly that this method is only effective if the faces are all the same size and facing the same direction. To solve the first problem, we returned to the vision.CascadeObjectDetector and used it to just place a bounding box around the entire face, which it did with high accuracy. Then we were able to run an eigenface method on just the pixels in the bounded box. However, this didn't solve the issue of the rotation. It occurred to us that training over the output of the eigenface method and the angles together might be able to isolate eigenface comparisons between similarly rotated faces. However, this approach was beginning to get unwieldy, and furthermore we weren't convinced that the training set was large enough for the approach to be successful.

At this point we set the features aside. We tried incorporating the results we had from the provided features and the color analysis into our final ensemble, but found this wasn't actually increasing our accuracy, so we left the features out in the end.

In another independent approach, we attempted to do a feature selection on the words before training on them by using stemming. We found a Matlab implementation of the Porter Stemmer, and we used this to condense the words matrix down to around thrity-seven hundred words. We hoped that this would both boost our accuracy on the words and reduce the size of our word models. We tested our LogitBoost method on the stemmed words but surprisingly got no improvement on either front. We realized that because Logit Boost takes samples of a set size to train over, reducing the total size of the training set wouldn't actually impact our model size. With regards to accuracy, we concluded that the

only reason we wouldn't be getting a boost in accuracy from this approach is if our model was already implicitly weeding out redundancies, so we were satisfied with how our original models were working on the original set of words and discarded the stemming approach as well.

Results and Pretty Pictures

The results of all of our major models are summarize in Table 1 and, as the table shows, there was a lot of incremental improvement over time. The biggest 'jump' however, came from combining the images and the words together which gave us a model with 92% accuracy. At this time in the competition, our team was sitting right in the middle of the leaderboard with ~89% and the 3% increase was what launched us up to the top of the leaderboard, where we remained for almost the entire rest of the competition.

TABLE 1. SUMMARY OF RESULTS

Method	Accuracy
KNN on words	68%
NB on words	79%
AdaBoost on words	88%
SVM on images	84%
Ensemble on features	71%
LogitBoost on words	89%
SVM on words	90%
LogitBoost on words + SVM on images	92%
LogitBoost on words + SVM on images + SVM on words	94%

As we worked on our model and sought to find new ways to improve it, we came across some interesting things. When we first combined the words and the images and used an SVM on them, we decided to plot what was going on since we were dealing with a 2D SVM. This produced the image shown in Figure 2. A closer look at the hyperplane can be seen in Figure 3.

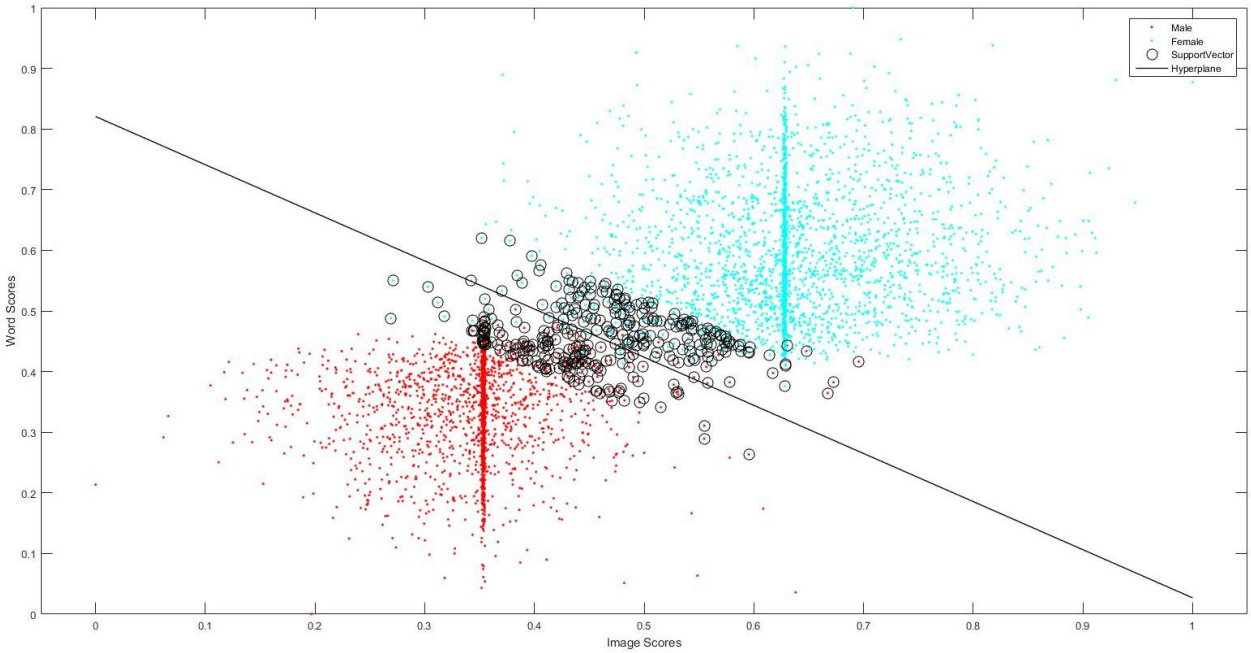

FIGURE 2. SVM ON WORDS AND IMAGES

These plots show a number of interesting things going on here.

Firstly, the data is very close to being perfectly linearly seperable. It's hard to see, but looking closely at (or zooming into) Figure 3, one can see that there are a few points very close to the hyperplane that are misclassified but, for the most part, everything is classified correctly in the central cluster. However, looking towards the edges of the central mass, we can see there are a lot more points on the wrong side of the hyperlane. This is one of the big reasons the SVM worked so well: even when it had a decent number of points that it misclassified at either end, the hinge loss didn't skew the hyperlane so much as to mess up all the points it was already classifying correctly in the center.

Secondly, the image anlsysis seemed to be producing a lot of images that were scored at ~0.35 or ~0.64 as the dense vertical clusters in the plot indicate. We never ended up figuring out exactly why this was, but one possibility is that all of the plain headshots that people use for their profile pictures were all getting scored very similarly (which is a very good thing!).

Once we added in the SVM on the words and used interation terms to capture more information, it was no longer possible to make a nice, easy plot of the SVM, but this 2D case gave us a pretty good indication that using the scores from our individual models was doing a very good job of seperating the data.

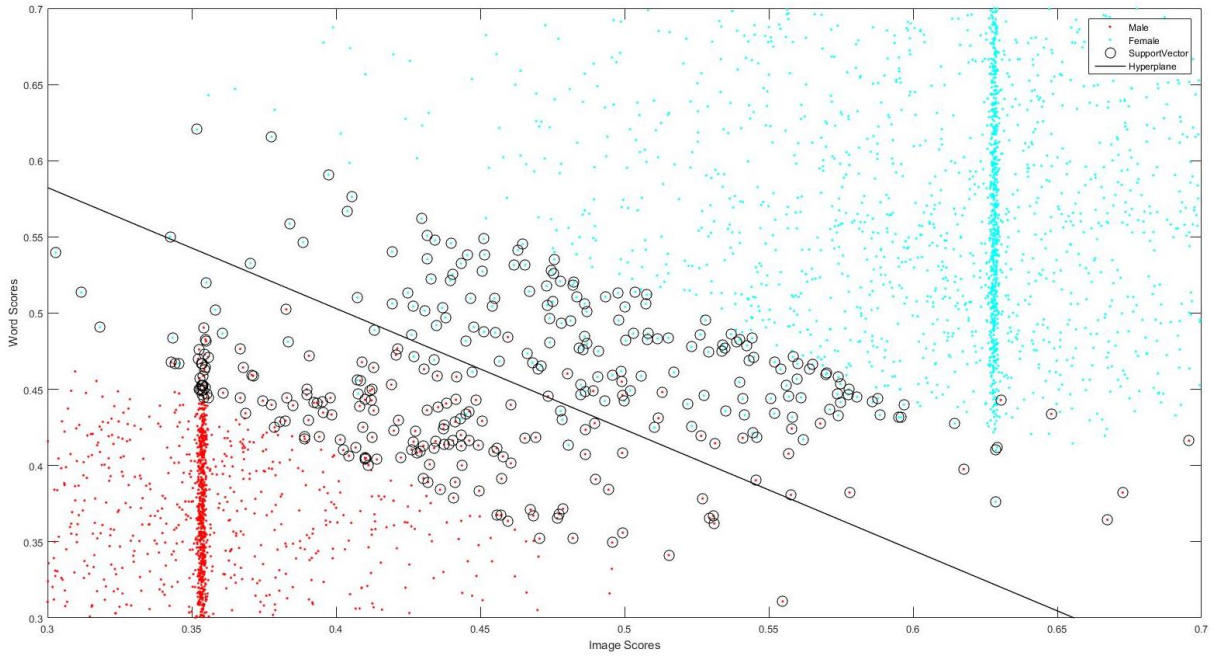

FIGURE 3. ZOOMED IN VIEW OF HYPERPLANE

While exploring various ways we could pull other interesting visualizations from our model, we experimented with trying to identify the image features that were being used for classification. Unfortunately, this proved to be rather difficult and we had to abandon that effort. Instead, we looked at the LogitBoost model that we used on the words to see what words it classified as the most effective at predicting gender. The resulting visualization can be seen in Figure 4 (you might need to zoom in to see it better). This figure shows 50 of the most predictive words in the model and their relative correlations to male and female genders.

There are a few interesting things about this model that we discovered when creating this visualization. First, the LogitBoost method chose only 494 unique words when using 1000 decision stumps. Almost all words were repeated at least once and many of the most telling words such as ‘wife’, ’gay’, ’cute’, and ’yay’ were repeated several times, with each decision stump splitting on a different number of usages. This makes sense, as when a person used the word ‘cute’ more than 3 times it meant the person was very likely to be a female, but even if they only used it more than 2 times, there was still a strong correlation with being female, so another split was chosen there. Another thing to note is that after these 50 words that are strongly correlated with gender, the correlation of individual words almost completely disappears. This is consistent with the cross validation results shown in Figure 1 where only 100-200 weak learners (possibly the 50-100 most correlated words each repeated a couple times) is sufficient to classify most people.

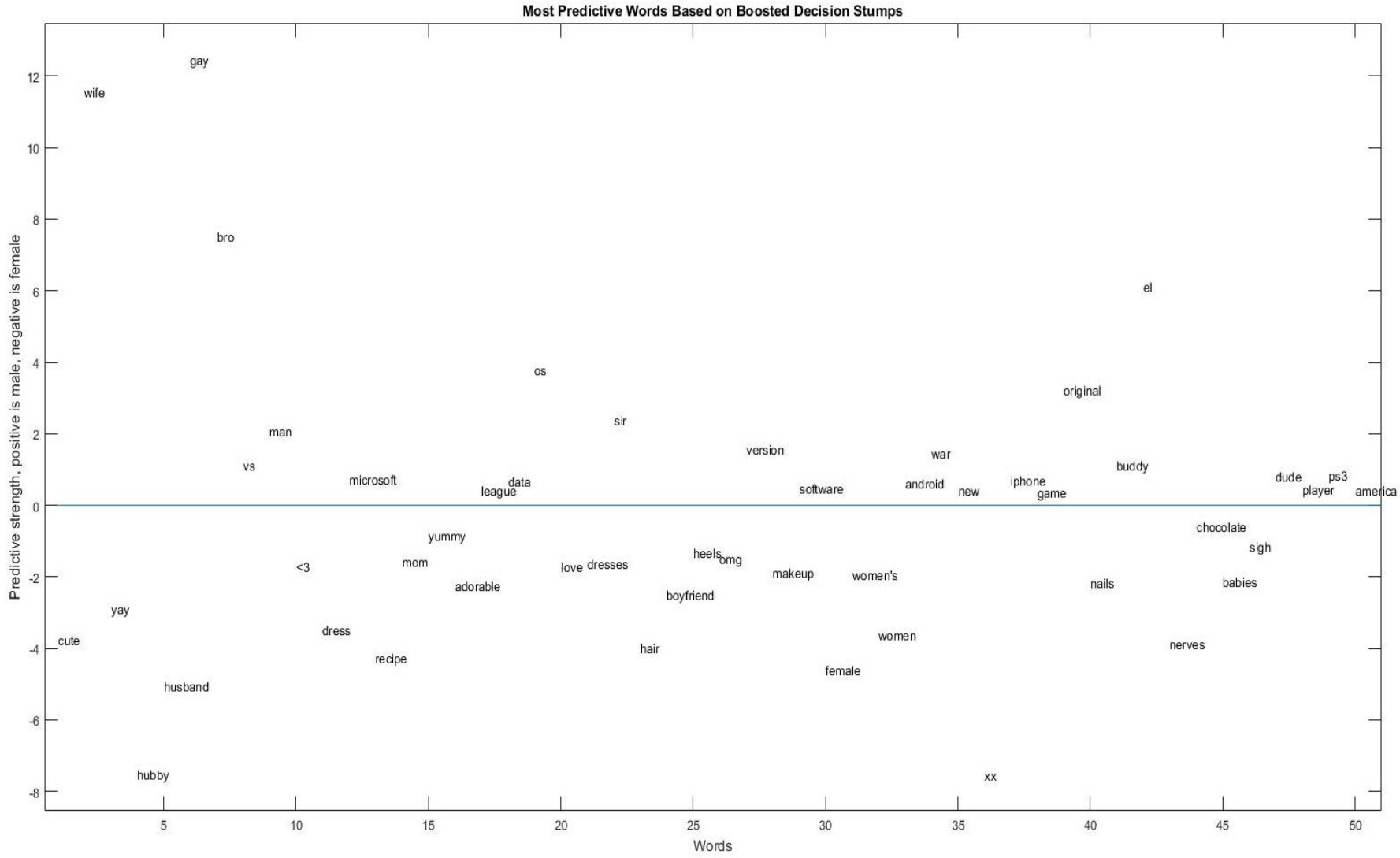

FIGURE 4. PREDICTIVE STRENGTH OF WORDS IN LOGITBOOST MODEL

Conclusion

As demonstrated by our results in the competition, this pragmatic, educated-trial-and-error approach worked very well, as it tends to for much of the machine learning field. Not only were we able to make a model that worked well in a competition, we were able to learn about the sort of process it takes to arrive at this kind of model.

When we started out on this project, none of us had any dreams or ambitions of taking home first place, we just figured we would try our best to make a model that could at least get past the baselines. But once we unexpectedly leapt into first place, we began to get a lot more excited and searched for a better understanding the kinds of tricks and optimizations that were needed in order to eek out a few more tenths of a percent. We learned that cross validation was our best friend. We learned that overfitting is far too easy to do and we have to get really sneaky to avoid it. And we learned that it takes a lot of effort and patience in order to get a really good model; it doesn't just happen because you threw one model at it and it perfectly captured everything (well, unless that model was a deep net…).