(Foggy) Crystal Ball Through the Lens of Facebook

CS 229 Final Report

Justin Li, Yangluo Jim Wang

December 14, 2012

1 Introduction

Given the abundance of information on Facebook profiles, particularly the favorites section where users "like" various forms of entertainment, such as movies and television shows, there is ample opportunity to employ a machine learning algorithm in order to learn the preferences of users and to make informed recommendations that may be of interest to these users. In particular, we applied a machine learning algorithm to generate television show recommendations. For this work, we partnered with Screaming Velocity [1], a start-up company who provided scripts and resources for mining Facebook profiles, through which 16000+ Facebook profiles were obtained. Screaming Velocity also provided a dataset on the most popular American television shows from which we obtained a set of descriptive genres.

First we built a bag-of-words model to describe both the input feature space generated from Facebook profiles, such as movie interests and group affiliations, as well as a feature space of relevant genres associated with the list of television shows. Then we implemented a Support Vector Machine (SVM) algorithm to predict which genres a user is likely to prefer given his/her Facebook profile.

This work has useful practical applications, directly so in the entertainment industry. Retailers may be able to use similar approaches to cater to personalized customer preferences and interests. This type of work can further enable more comprehensive data-driven predictions and recommendations by coupling with growing prevalence in the mobile media and social media spaces. Similar work can also be generalized to other situations where automated recommendations may be useful.

2 Data Processing

2.1 Data extraction

Screaming Velocity has developed and provided a set of Python scripts that, with user consent, scan through an individual's Facebook profile and the profiles of his/her Facebook friends. By advertising on the CS229 Piazza page, they collected more than 16,000 individual profiles. We downloaded this set of profile data, anonymized to protect privacy, in a large `.xml` formatted file. Based on the `.xml` file structure, we wrote a MATLAB script to extract the file into a struct, with one entry corresponding to each individual. More specificially, each entry contains 12 fields from the individual's public profile, listed in order: gender, locale, atheletes, teams, about, TV, movies, music, books, activities, interests, and sports. For example, a user might have these fields:

```
Gender: Male
Locale: US
Atheletes: Andrew Luck
Teams: Stanford Cardinals, Indianapolis Colts
About: N/A
TV: Battlestar Galactica, Eureka, The Beginning
Movies: Monty Python and the Holy Grail, Iron Man 2, The Avengers
```

```
Music: N/A
Books: The Lord of the Rings, Alexander the Great, The Illiad
Activities: N/A
Interests: N/A
Sports: N/A
```

Then, for every field i, we built a cell array d_i where each element d_{ij} of d_i was itself a cell array of the entries for user j regarding field i. For example, the first field, `Gender`, may be represented as d_1 ={{'Male'}, {'Female'}, {'Female'}, {'Male'}...}, indicating the first user is male, the second is female, and so on. Likewise, the 6^{th} field, `TV`, may be represented as d_6 ={{'Battlestar Galactica, Eureka, The Beginning'}, {'The Office, Star Trek'},...}.

Now, we defined a *feature* to be an element in the inner cell array. So 'Female' would be a feature in `Gender`, and 'The Office' a feature in `TV`. We then defined the *dictionary* to be the set of unique features in a field. Given n total users and m_i unique features in field i, then for each field i we build a sparse matrix X_i of size $m_i \times n$,

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(i)} & \cdots & x^{(n)} \end{bmatrix}$$

where each $x^{(j)} \in \{0,1\}^{m_i}$ is the feature space for user j. For example, `Gender` has two features, `Male` and `Female`, in its dictionary, so the example above may be represented by

$$X_1 = \begin{bmatrix} 1 & 0 & 0 & 1 \cdots \\ 0 & 1 & 1 & 0 \cdots \end{bmatrix}$$

where the first row represents `Male` and the second `Female`. $X_{11} = 1$ indicates the first user is a male, and $X_{12} = 0$ indicates the second user is not a male, and so on.

In the `TV` example, each user could have multiple TV preferences, so d_{ij} could have multiple elements. Furthermore, some of these elements could be repeated by other users. Thus we specify each *feature* to be a unique element. We parsed each d_{ij} to extract features, delimited by a comma. Continuing with the d_6 example above, we would construct a vector $x^{(1)}$ for the first user as follows:

$$x^{(1)} = [1, 1, 1, 0, 0, 0, \cdots]^T,$$

where the first five features correspond to the first five elements of the `TV` dictionary, which are Battlestar Galactica, Eureka, The Beginning, The Office, Star Trek. `1` indicates the user likes this TV show, and `0` means the user did not specify he/she liked it. Now, we repeat this process for all 12 fields to get a large sparse matrix X

$$X = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_{12} \end{bmatrix}.$$

Notice X is particularly row-sparse, since features are not commonly shared. To decrease the sparsity, we ignored features shared by only two or fewer users. This means that if at most two users liked a particular sports team, for example, then we will not use this sports team in our training data. It is worth noting that since we implemented exact string matching to find common features, we run the risk of losing data that is actually shared by multiple users if they spelled it differently (e.g. `Houston Texans` versus `Texans`). Given that we have 16000+ users, losing this amount of data is probably fine.

2.2 Inputs and Labels

Although we use X as the input to make predictions about TV preferences, we first need to make some modifications. Since X_6 is the data for TV preferences, we leave X_6 out of X and instead set the test matrix $Y = X_6$. If there are m total non-TV features and n users, then X is binary matrix of size $m \times n$. If there are r TV tokens, then Y is a binary matrix of size $r \times n$.

Initially we used individual TV shows as labels. Later, we found that this Y matrix was much too sparse and decided to classify by genres instead (see Section 3.1 for details). Screaming Velocity provided

a document which grouped 347 TV shows into 84 genres (note a TV show could fall into several different genres). For the new genres-based Y matrix, $Y_{ij} = 1$ means user j liked a TV show which is part of genre i, and 0 otherwise. Notice Y_{ij} could be greater than 1 as each user may have liked several TV shows in this genre. For our algorithm, we ignored this information as we only considered a binary classification. But in the future, it is possible to extend our work to multiclass classification. Furthermore, a TV show can be more strongly associated with one genre over another. Such association would normally require many unbiased people to watch each TV show and rate them. Since we did not have such information or resource available, we ignored this fact and acknowledge that our algorithm could be improved given such knowledge.

3 Model Selection

With properly formatted data, we can begin applying machine learning techniques in order to build a useful predictive model. Here we describe the development steps we went through in building a model and selecting parameters.

3.1 SVM Modeling

We planned on using a set of support vector machines, one SVM for each television show in the given list. In particular, with X representing the input feature space and Y modified to be a vector of 1's and -1's for each television show, the SVM fit some model for each television show. With this sort of naive first implementation, we find that the SVM claims to have excellent results, with prediction errors ranging from 0.01% to 2%. However, prediction error presents an incomplete picture of the trained model: since the outputs (i.e. labels) are generally sparse, say 10 out of 16,000 individuals having a ' 'like" for a television show, predicting a -1 as the output regardless of the input gives a very low prediction error that is not meaningful. Indeed, quantifying performance using false positives and false negatives showed that these naive models in fact perform quite poorly (i.e. 100% false positive).

In order to handle the issue of sparsity, we switched from modeling specific television shows to predicting television show genres. This approach greatly reduced sparsity and produced more realistic predictions. However, there were still a number of genres which were particularly sparse. As such, our SVM only learned models for genres with enough entries over some threshold. In our final model, we set the threshold to 500 people and thereby trained on 32 genres. Initially, we chose a model based on the linear SVM package `liblinear` [2] as used in our Homework 2.

3.2 Model Results and Parameter Optimization

Predicting by genres gave more realistic results. However, the training and testing error, as shown in the learning curve in Figure 1(a), did not converge (there was a 20% difference between them), which indicated overfitting. It turned out that our data could not be sufficiently modeled by a linear SVM like `liblinear`, so we switched to using `libsvm` developed by the same authors [2]. This gave significantly improved results, with an average training and testing error around 10%, as shown in Figure 1(b). In both Figures 1(a) and 1(b), we also included a green dashed line illustrating the baseline prediction accuracy, which was computed by assuming a uniform distribution based purely on the fraction of individuals who liked a genre. In order to further break down the distribution of the errors, Figure 2 illustrates the learning curves for each genre. From this, we see that while most genres had prediciton error between 3-15%, a few of the genres had considerably higher error rates ($> 20\%$). These errors were the average of 5-fold cross validations.

There are two parameters in `libsvm` that we optimized to get better performance. In particular, `libsvm` used the radial basis function $f(u, v) = \exp(-\gamma|u - v|^2)$ as the kernal type and implemented a cost function for regularization, which is controlled by a c parameter. We ran a grid search algorithm to find the best γ and c parameters for the SVM for each genre using both brute force double `for` loops in MATLAB and the `grid.py` script supplied by `libsvm`. It took a significant amount of time to find the optimal parameters for even a single genre, on the order of six hours, for `grid.py` and even longer for our own brute force method. Consequently, we focused only on optimizing genres with error rates of over 20%, which are the top six lines in Figure 2, using `grid.py`. With these new optimal cost and gamma parameters, we retrained our SVM models and found that the error rates for all six genres were reduced by 5%-15%.

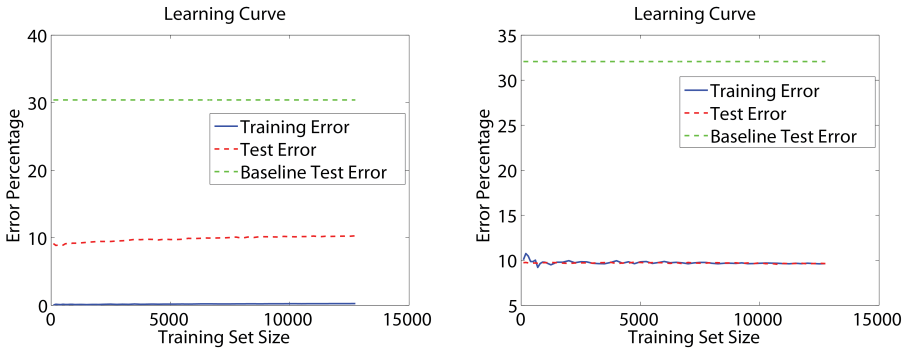

Figure 1: (a) Learning Curve plot using the liblinear SVM package, with a clear gap between training and testing error. (b) Learning Curve plot using the libsvm SVM package, eliminating the overfitting introduced by the liblinear package. All errors are estimated by 5-fold cross validation.

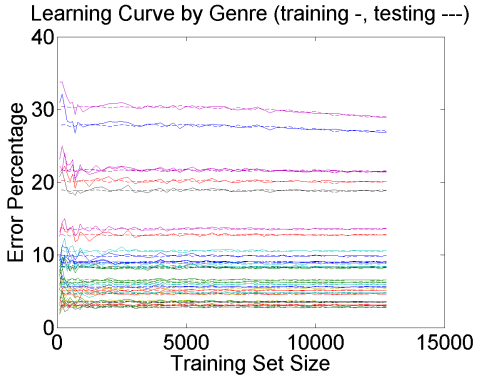

Figure 2: Learning Curve plot using the libsvm SVM package, for each genre. A solid and dashed line of the same color comprise one training and testing pair for a genre.

The end result is that in terms of prediction error for television show genre, our model is nominally accurate. However, in Table 1, we see that for the six genres with the highest error rates and for which we optimized with grid search, precision is high, but recall is low. We further note that recall is more important than precision for our application in terms of providing a meaningful metric, as false negatives are more detrimental than false positives for evaluating predictions. Overall then, with the exception of `Drama`, the model failed to make many useful predictions. In part, this is because the grid search on the `libsvm` implementation using `grid.py` optimized c and γ to improve accuracy rather than precision or recall. Furthermore, due to the sparse nature of the genre data, many of the other genres most of the predictions are still predicting false, hence accuracy being high but not recall. It is somewhat unclear as to why the `Drama` genre shows significant improvement. The six most populated genres all had roughly the same population, but only `Drama` had a reasonable recall percentage, having increased from 0.2% to 55.3% after grid optimization. After considering the shows Excel file that we used to determine genres from television shows, it may be that there is some underlying linkage between the other genres that interferes with the assumption of independence. Many of the television shows that fall into any of the other six genres often contain two or three more of the six genres as well. Alternately, it may be that the grid search in these other genres were based on an incorrect value function. So, for future work, we suggest writing a grid optimizer which optimizes for recall for each of the genres, noting that this will be computationally expensive given the size of our feature space.

	Before grid search		After grid search	
Genres	Precision	Recall	Precision	Recall
Comedy	97.6%	4.6%	97.4%	5.3%
Debauchery	100%	0.2%	100%	0.5%
Drama	100%	0.2%	66.3%	55.3%
Goofy	96.2%	3.2%	97.7%	3.9%
Satire	100%	0.2%	100%	1.2%
Sitcom	0%	0%	100%	0.2%

Table 1: Precision and recall for the six genres with the highest error rates.

4 Conclusions and Suggested Future Work

In conclusion, we implemented a machine learning algorithm to predict individuals' TV show genre preferences given information about their Facebook public profiles. We first pre-processed 16000+ Facebook public profile data: extracting out relevant features, constructing feature and label matrices, and re-processing to make the matrices denser. Then we built a model by training a radial basis function-type kernal SVM on the data and achieved an average test and training error of 10%. However, low recall and high precision indicated that the data was still too sparse, and further optimization and more data are needed. As a result, in general it was not successful in making genre predictions.

This work provides insight into some challenges and solutions encountered in predicting information given Facebook data. We hope this work may be of use to Screaming Velocity and other parties who are interested in predicting user preferences. Broadly speaking, this work can also be extended to general situations desiring automated recommendations. For future work, we think the following are worthy items to address:

1. Use PCA to condense data and make it less sparse. Right now, we simply throw away rows that are too sparse. A better way to do this may be to condense rows.
2. Run grid search to optimize c and γ for recall.
3. Consider weighting different features. For example, books and interests may be better predictors than location, so they would get a higher weighting.
4. A neural network can be used to introduce tags for dependencies, maybe using UFLDL [3].

5 Acknowledgements

This work could not have been done without the generous help from Screaming Velocity and our head TA, Andrew Maas. Graham Darcey and Wayne Yurtin from Screaming Velocity helped us write the script to collect Facebook data and distributed the data to us. Andrew was always available to discuss our project and provided guidance when we were stuck. We would also like to thank the 16000+ people who contributed their data. Last but not least, we would like to thank Andrew Ng and all the TAs for an awesome class and for their time reading our final report. Hope you enjoyed it!

References

[1] Screaming Velocity. `http://www.screamingvelocity.com/`. Last accessed Dec 12, 2012.

[2] Chih-Chung Chang and Chih-Jen Lin. LIBSVM : a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/ cjlin/libsvm`.

[3] Unsupervised Feature Learning and Deep Learning (UFLPL). `http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial`.