The ComAIdian Project – Blue Team Report

Yash Bhartia [*1] **Dillon Carlos** [*1] **Ehsan Fanaian** [*1] **Aakaash Kapoor** [*1] **Chirag Kashyap** [*1]
Isaac Kim [*1] **Naomi Okiddy** [*1] **Kush Patel** [*1] **Christine Tech** [*1] **Nathan Telles** [*1]
Dominic Yang [*1]

Abstract

The Blue Team ComAIdian application applies machine learning to the field of comedy by creating a system that recommends jokes to users based on their demographic information. Our system lets users input information about themselves—such as age, gender, favorite movie genre—and uses those preferences to recommend jokes. As users rate the jokes they have been given, the system learns and improves the quality of its predictions. In this paper, we explore the various attempts made at creating the recommendation system, as well a working recommendation system developed using a modified version of matrix completion. We discuss how this method works, how it has been implemented, as well as its limitations.

1. Introduction

Predicting the quality of a joke is hard – in fact, doing so for a living will earn you an average salary of about $30,000 (art, 2017). In other words, predicting what will make someone laugh is hard and most people who try do so fail miserably in small comedy clubs. The ComAIdian application invokes a novel form of matrix completion, which addresses new user information as it is input into the system.

Our form of matrix completion attempts to address the "cold start" problem in recommendations – it is difficult for any recommender model to make predictions when a previously unobserved agent enters the system (Lika et al., 2014). Other related approaches to recommender systems that have been established over the past decade, while dominating prior recommendation models, also suffer this issue (Koren et al., 2009). By allowing the new user to initially rate a small subsample of jokes, the model can relate the new entrant to the nearest cluster of similar users. Then, the recommended jokes are chosen by two rankings – those most likely to be funny to the user, but also those that are most informative to the model.

The remainder of the paper is organized as follows. Section 2 explains the development of the system and why other methods were forgone. Section 3 introduces the primary prediction model. Section 4 discusses the active recommender system in detail. Section 5 discusses the key results. Section 6 introduces additional tuning of the final model, and lastly section 7 suggests further research on the topic. Additional materials in the appendix specify the design of the web application, our communications platform, and blue team member contributions to the project.

[*]Equal contribution [1]University of California, Davis. Correspondence to: <>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017. Copyright 2017 by the author(s).

2. Initial Approaches to Recommendation System

Before delving into the actual implementation of our recommendation system, let us first discuss the initial approaches to this project.

2.1. Euclidean Distance

The first attempt made at creating a recommendation system was to use the Euclidean distance algorithm to predict jokes to users. In the context of our problem, Euclidean distance is a straight-line distance measure of two users based upon their joke rating vectors. More formally stated, if $x_i = i^{\text{th}}$ joke rating for user x and $y_i = i^{\text{th}}$ joke rating for user y, then the Euclidean distance can be represented with the following formula:

$$\text{distance} = \sqrt{\sum_i (x_i - y_i)^2}$$

We wanted to make a joke recommender system that initially used the joke rater features to recommend the first few jokes, but later gave less weight to the features and more weight to the Euclidean distance between users. Essentially, we wanted to create an active learning model based on the framework outlined by Shimodaira (2014). For the initial jokes outputted, we created a similarity model based off of Hamming distance (Shimodaira, 2014). In Hamming distance, two strings of equal length are compared and the number of places where both strings are identical are outputted. For example, "cat" and "mat" would have a Hamming distance of 2. For our problem, the strings were the joke rater features, so the Hamming distance would calculate how many features two joke raters had in common. For a new user, the algorithm would take the Hamming distance between the new user and all other users and multiply that across all of the other users ratings. Then, the algorithm would sum those modified ratings and rank the jokes. The joke to be outputted would be the one with the highest overall sum.

This passive part of the algorithm worked as planned. Where we got into trouble was implementing and blending both the Euclidean distance and Hamming distance algorithms. We did not want to arbitrarily assign weights to both distance algorithms. Because of this reason and since matrix completion was making a better, purely numerical algorithm, we decided to pursue other types of algorithms and pushed the passive Hamming distance algorithm to Github.

The point of this algorithm was to give Software Engineering and UI/UX a dummy algorithm to build the system around until we could replace it with a better more substantial algorithm. Because of this, we never ended up testing accuracy or MSE, since the intention was to replace it with a much better algorithm in the future. Although, things did not go as planned, we believe the approach was correct, but communication between teams needed to be better.

2.2. Reinforcement Learning

Another method that we wished to integrate into our application was reinforcement learning. Reinforcement learning focuses on the idea that agents choose actions in an environment to attain the maximum reward. A few methods can be used with reinforcement learning such as dynamic programming and Markov Decision Processes (MDP). A few team members were already familiar with Markov Decision Processes, so we decided to attempt MDPs.

Markov Decision Processes have three main components: states, actions, and policy (Kalyanakrishnan, 2014). The policy is the probability of an agent taking an action from any given state, with this action resulting in a reward. Our biggest challenge was to create a proper simulation of how jokes, users, and ratings can be attributed to the Markov Decision Process. At first, we thought states could be the

jokes and actions would be to introduce the user features one at a time, then see how the policy reacts. However, this idea was flawed since the policy would not be truly learning and failed to be universal for jokes and users. We settled on letting the states correspond to users, actions equaling jokes, and the policy being universal.

One reinforcement learning technique called Q-learning would have worked well to find the optimal policy, and it is a natural fit for a finite MDP (Matiisen, 2015). After talking to the professor, however, we were advised to use a different method: active learning. There would be states for each joke and more states per number of jokes learned from. The actions would be based on if the user started at the initial joke, then a probabilistic policy would exist for the reset of the jokes based on their rating of that initial joke. This would be universal for all users. After much contemplation, we realized that this would be too complex to program, therefore we moved on to matrix completion.

3. Attempted Algorithms

Now that we have explored the various attempts at this recommendation system, let us now discuss the algorithms that were actually tested: artificial neural networks and logistic regression.

3.1. Artificial Neural Networks

As Artificial Neural Networks (ANNs) are considered to be the backbone of machine learning and are more widely used than any other method, we tried to predict jokes to users by utilizing ANNs. Contrary to most other algorithms, a "basic" ANN model does not exist, therefore our first issue was to design the structure of the ANN. The ANN model would change depending on how the software engineering team wanted to display and implement the algorithm. Through discussion, our teams agreed upon creating an ANN model that would predict the best joke for a particular user and update the algorithm based on the user's rating for that joke.

However, we still had a decision to make. Did we want an ANN that took in user features as input and predict the jokes best suited for that user? Or did we want to use as input user and joke pairs and predict a rating based off of that. While the latter method seemed rather complicated, it made sense to implement it considering the few features we had for jokes and users. Since there were X users and Y jokes, we could essentially be dealing with only X or Y samples. But the pairwise input gave us an advantage as we were able to create a multitude of samples, XY samples to be precise. With this many input samples and the ratings for each user and joke pair, everything seemed to be on track. But we came across a bigger problem—-we had too few joke features.

Since we believed that we did not have enough joke features to accurately predict joke ratings, we decided to add the joke text length as an extra joke feature. To obtain the database information, we first binarize the genders using LabelEncoder() provided by the scikit learn library (Pedregosa et al., 2011). Similarly we encoded the joke categories and the lengths of jokes to integers using the same LabelEncoder() function. Next, we iterated through all the samples (corresponding to users) to map the remaining user features including birth country, majors, and preferred joke types, to their corresponding integers values. We then used one-hot encoding, transforming the joke rating to binary bits for numbers in the range $[1, 5]$. To model the system, we used stochastic gradient descent (SGD) for backpropagation to determine the optimal values for the network weights and developed 8 hidden layers. A visual of this model can be seen in Figure 1.

By testing the effects of the number of hidden layers and the number of nodes per hidden layer on accuracy and mean squared error (MSE), we analyze trends to visualize which type of ANN structure to utilize and measure overall ANN performance. To find the MSE, we made the max-

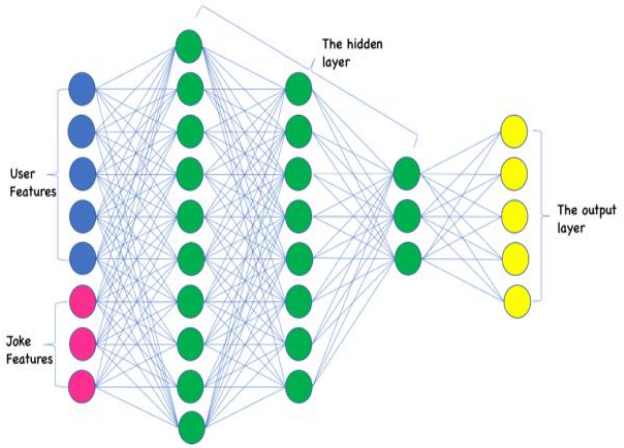

Figure 1: ANN

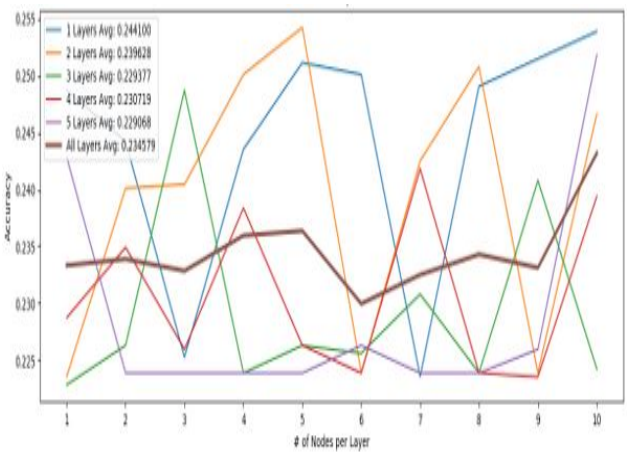

Figure 2: Accuracy vs Number of Nodes and Layers. Activation Function: Categorical Cross Entropy

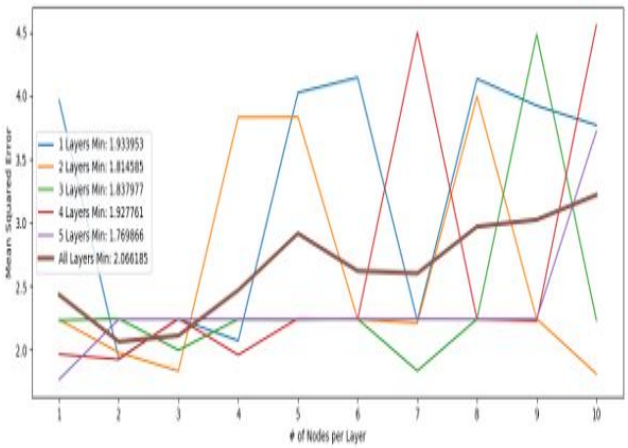

Figure 3: MSE vs Number of Nodes and Layers. Activation Function: Categorical Cross Entropy

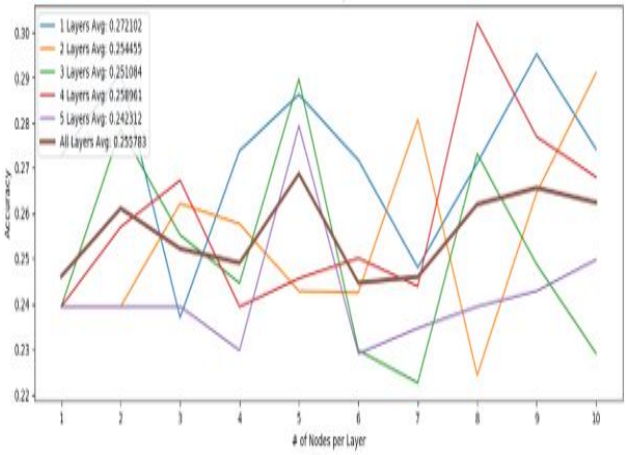

Figure 4: Accuracy vs Number of Nodes and Layers. Activation Function: Tanh

imum value of the output layer the significant value and converted from one-hot encoding to a scalar rating score. Models were fit over 1000 epochs, each with a batch size of 100 and each using MSE as a loss function. Two activation functions were tested: hyperbolic tangent function (tanh function) and categorical cross entropy. The dataset was split using 80% of the data as the training set and 20% for the testing sets, and a dropout ratio of 0.1 was used per hidden layer to prevent overfitting.

As seen in both Figure 2 and Figure 4, we found that the average accuracy of the model is slightly higher on average if there is a greater number of nodes and smaller number of layers. This is shown with the 1 and 2 layer models (blue and orange lines respectively) in Figure 4 and 1, 2, and 3 layer models (blue, orange, and green lines respectively) in Figure 2 which shows higher accuracy peaks and overall AUC than their higher layered counterparts. There does appear to be a few spikes in accuracy for certain configurations such as 5 nodes per layer across all numbers of layers in Figure 4. The MSE is more difficult to analyze as it starts similarly for all layers, most likely due to overfitting over the relatively small

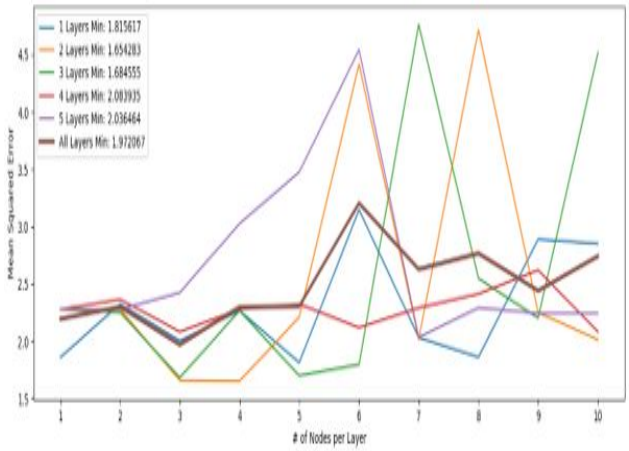

Figure 5: MSE vs Number of Nodes and Layers. Activation Function: Tanh

dataset, but starts to vary greatly with increasing model complexity in both Figures 4 and 2.

We also considered whether it would be more beneficial to use other activation functions and test if they could produce better results. We found better average accuracy results after training with tanh activation function, which had an average accuracy average of 0.255. Despite these considerations, the computed accuracy did not exceed our baseline expectations of majority positive predictions (over 50 percent accuracy) and the MSE values appeared to vary greatly. With this, we had doubts that ANN would be the most practical route.

In terms of project API implementation, we settled on a very straightforward approach. The main constraint was the long time it took to train the ANN model. In order to tackle any processing delay, we decided to pre-train an ANN model for each user (according to their features) and save the weights in a file. This way, when a certain user logs in, we can simply fetch their specific model from the backend and update the weights accordingly if they enter more data. Jokes will be predicted by simply inserting the profiles weights into the backend model and predicting.

Although ANN was giving substantial results, these results were not satisfactory compared to Matrix Completion. The major cause for this discrepancy was the lack of joke features for the ANN and the fact they weren't descriptive enough. The success and the failure of an ANN depends on how closely related the input and the output features are, but for this particular problem, determining the rating of a joke based on what type of joke it is and how long it is did not seem possible. Features like sarcasm, context, and real world truths cannot be quantitatively converted to numbers to predict a joke's rating.

3.2. Logistic Regression

Building upon the categorical nature of the joke rater features, we believed that using logistic regression or other classification techniques could lead to a decent recommender system. Logistic regression is a commonly used classification algorithm that quickly predicts the probability that a sample is a part of a certain class. For this particular problem, we are using the joke rater features to predict what the user would have rated a particular joke. The independent variables are gender, birth country, major, joke genre, joke type, music genre, movie genre and age. We used this set of features to predict if a user would rate a particular joke either 1, 2, 3, 4, or 5.

The first attempt was to make one huge logistic regression predictor, which included the joke rater features and two joke features: category and joke type. This predictor overfitted the data and made the predictions lean heavily towards jokes that were categorized as "Animal" and "Questions." Likewise removing those two joke features gave predictions below 3. It was obvious that this predictor was overfitting the data heavily, so we decided to approach the problem differently.

Instead, we fit a logistic regression predictor for every joke, meaning our algorithm trains 152 (number of jokes) times. Using this approach,

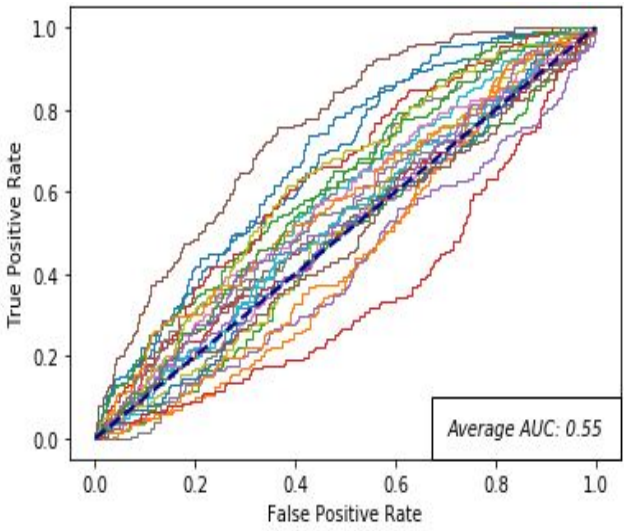

Figure 6: Micro-Averaged ROC Curves Using Passive Logistic Regression

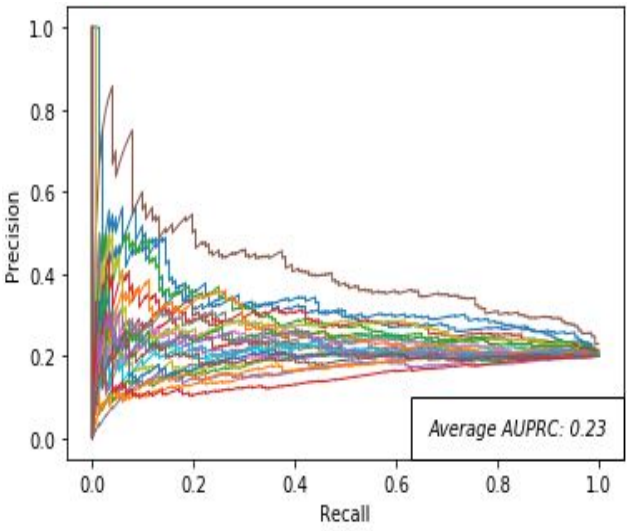

Figure 7: Micro-Averaged PR Curves Using Passive Logistic Regression

we created active and passive versions of this content-based logistic regression algorithm. The passive version of this algorithm trains on each joke once and outputs the jokes by highest predicted rating. The active learning version of this algorithm trains on each joke once per new user rating. This means that once the new user has rated a joke, it adds that rating to the feature space of all users and retrains the logistic regression predictors again. This ensures that the user always gets the joke that is the highest rated by the predictors.

As seen in Figures 6 and 7, the average AUC (0.55) and average AUPRC (0.23) from the predicted ratings of 25 test users are both lower than expected. These values indicate that the algorithm presents only a slight improvement compared to picking a rating at random. In order to get a better representation of the MSE of the passive logistic regression, we also fit a linear SVM and an ensemble voting classifier model (Pedregosa et al., 2011) that used both the logistic regression and SVM. The inspiration for the ensemble model came from the BellKor solution to the Netflix Prize competition, since their final model combined 107 individual models (Bell et al., 2008).

We tried to blend all of our models together, but we could not because of the differing frameworks between the models. However since this framework was already in place, replacing logistic regression with either SVM or the Voting Classifier was done with relative ease. For the passive versions of these models, the Voting Classifier had an average MSE of 2.83, which was the lowest, while the average MSEs of the linear SVM and logistic regression lagged behind at 3.04 and 2.98, respectively (see Figure 8. Based off all the prior information, the passive logistic regression model does not seem as viable as a recommender system because the predictions are not accurate enough.

On the other hand, there is some hope that the active logistic regression algorithm could be a decent recommender system. According to figures 10 and 11, the average AUC and average AUPRC from 25 test users were 0.66 and 0.33, respectively. Changing from a passive model to an active model increased the AUC by 0.11 and the AUPRC by 0.10. Likewise, according to figure 9 the average MSEs of the linear SVM, logistic regression, and voting classifier models also

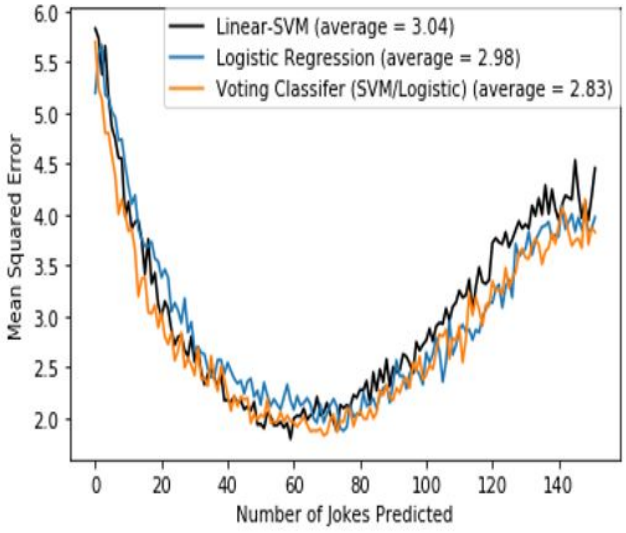

Figure 8: MSE of Different Passive Learning Algorithms Averaged over 25 Iterations

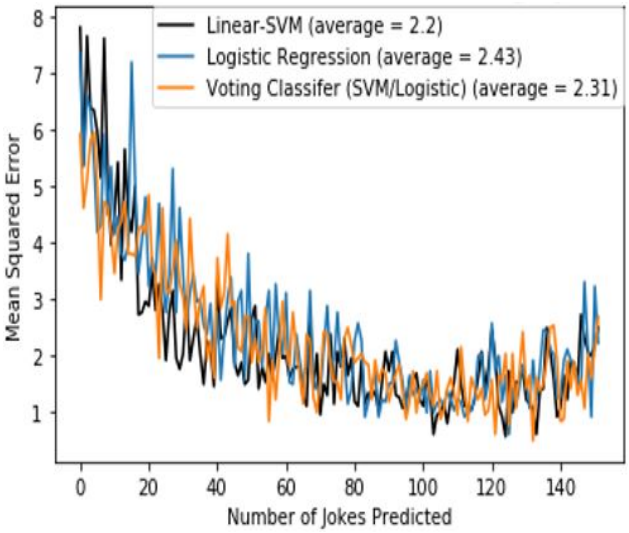

Figure 9: MSE of Different Content-Based Active Learning Algorithms

decreased by a non-negligible margin. Changing from a passive to an active model decreased the MSE of the linear SVM by 0.84, the logistic regression by 0.55, and the voting classifier by 0.52. This means that out of the six models using this framework, the active linear SVM ended up being the best one. Still more testing should be done to replicate those results. We were not able to use more advanced testing methods because the active learning framework took very long to test. For example, for one test user it would take up to four minutes to get predictions and errors for one algorithm. Because of this, we were only able to use the same 25 test users in order to compare the MSEs of all three algorithms.

In terms of implementing our algorithm on a website, the active learning models require sessions that maintain until the user wants to exit the recommender application. This is because they continuously update the training database to convert the ratings of previously rated jokes into the feature space and then delete the instances of the joke from the database. This is the active learning part of the algorithm since it takes the feedback from the user and retrains in order to get better predictions. Regardless, the algorithm is lightweight and fast from a user experience point of view. Training and outputting another joke in a ipython kernel took roughly five seconds and should theoretically decrease as a user progresses further into the joke database. On faster computers or an actual server, this time should decrease even further. Based off of the above information, we can safely state that an active learning model from above can be viable as a joke recommender system.

4. Recommender System

Recommender systems are generally divided into two paradigms: collaborative-based filtering and content-based filtering. Collaborative-based filtering methods focus more on the similarities between users ratings on particular interests. This technique predicts ratings for users through a collaborative approach; the intuition behind this method being users with similar interests on items are more likely to have similar ratings for a different item. By contrast, content-based filtering methods focus more on the fit between a user profile and an items profile to predict that users rating for that item. Content-based filtering methods have been used successfully for recom-

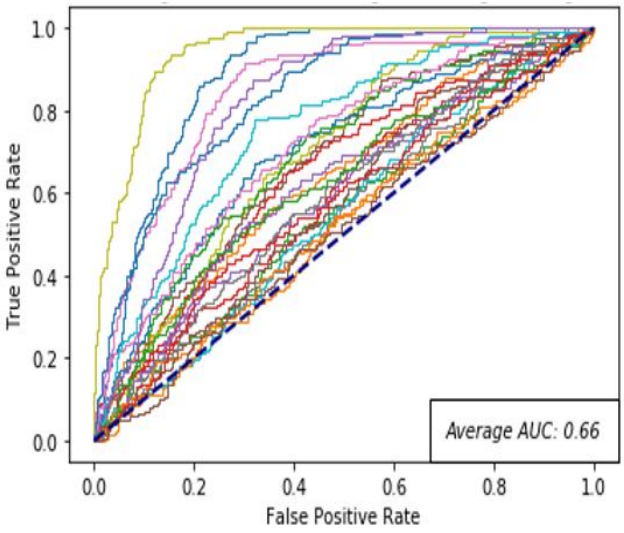

Figure 10: Micro-Averaged ROC Curves Using Active Logistic Regression

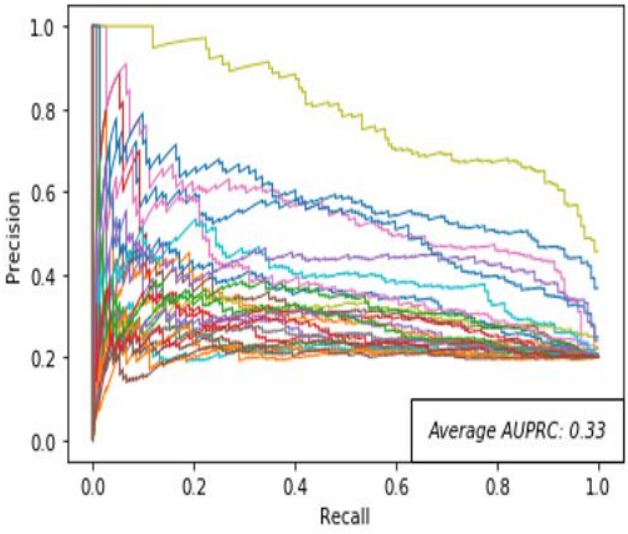

Figure 11: Micro-Averaged PR Curves Using Active Logistic Regression

mending items to users, such as web-pages and books. (cite(basilico2004unifying)) However, we decided a content-based filtering approach would not be appropriate for the problem of recommending jokes given the limited feature set for jokes, which could lead to difficulties in distinguishing appropriate jokes for a certain user. To this end, we developed a collaborative-based filtering method centered around the idea of matrix completion.

4.1. Introduction to Active Learning

One goal of the project is to implement a system that learns from users as they rate jokes. This is better than a system that does not learn as it is difficult to make a prediction solely based on a users characteristics. Instead, a better profile of the user is compiled based on how they actually rate jokes. After a joke is selected (the prediction), the user provides their own rating (the actual value). Any discrepancy in the values (the error) is used to recalibrate the model to make a better prediction in the next iteration. This is called active learning. As will be seen later, active learning produces a dramatic improvement in the quality of predictions over time.

4.2. Matrix Completion

This method draws inspiration from the Netflix Prize competition (Bell et al., 2008). The goal of the competition was to create a system that would be able to provide movie recommendations for a given user based on their ratings of movies that they have already seen. Competitors were provided with a database of movie ratings; m users, each having rated a subset of n movies. This database could be encoded in an $m \times n$ matrix A, wherein A_{ij} is the rating of given by user i for item j. The issue with this model is that it will be in all likelihood a sparse matrix since a user may have only watched a small fraction of all n movies, leaving a vast majority of the entries empty. A matrix completion approach to solving this problem would in turn attempt to fill in the remaining ratings for given users by drawing information from similar users whilst avoiding introducing new information haphazardly.

Our main reason for adopting a matrix-completion approach to the recommendation system problem was for its flexibility in incorporating new ratings into the model. Compared to an artificial neural network which requires predetermined input features, thereby limiting the feature

space to the user features for a new user, matrix completion algorithms allow for the matrix to be extended to include new users, new ratings, and possibly new jokes. In essence, this extends our feature space for a given user to include all the jokes they have already rated in addition to their user profile. Also, given the small size of the dataset worked with, the matrix completion algorithms are immediate and can therefore be used for active learning to train the model as ratings are received by new users.

A wide range of researchers have approached the task of solving the matrix completion problem from a variety of angles. A small collection of these algorithms have been implemented in the **fancyimpute** library (Alex Rubinsteyn), from which we draw examples for testing. A naive approach to filling in the empty entries would be to simply take some statistic, say the mean or median, of a given column—in this case a joke—and fill in any missing entry within that column with that central statistic. This method could be improved greatly considering it does not take into account a given users other ratings, but it will serve as a benchmark by which we can compare other matrix completion algorithms.

A slightly more informed matrix completion algorithm is the k-nearest neighbors (kNN) algorithm. This algorithm takes a weighted combination of the k closest users ratings, which is measured by a variety of metrics such as mean squared difference, Pearson correlation coefficient, and cosine distance. More sophisticated matrix completion algorithms have been developed that focus more on fitting a matrix X, which is of as low rank as possible while still agreeing with the observed ratings for every user. More formally, they try to solve the optimization problem

$$\begin{aligned} &\text{minimize } \text{rank}(X) \\ &\text{subject to} X_{ij} = A_{ij} \end{aligned}$$

Minimizing the rank prevents introducing any new information into the matrix and uses the simplest explanation as possible from the information given. Naturally, the above optimization problem is rather difficult to solve, therefore quite a few heuristic methods which involve minimizing the nuclear norm have been proposed to solve this problem. One of these methods include the SOFT-IMPUTE algorithm, proposed by Hastie et al. (2010) (Mazumder et al., 2010), and a method involving convex optimization (referred to as NuclearNormMinimization) developed by Candes and Recht (Candès & Recht, 2009). A separate method (referred to as iterativeSVD in this report) involves iteratively computing the k largest eigenvalues and the corresponding eigenvectors and using a regression based on a specific rows representation in these eigenvectors to predict missing values. This method has been used to estimate missing values in gene expression in DNA microarrays (Troyanskaya et al., 2001). In developing our own method for our recommender system, we tested these various methods to find an appropriate matrix completion algorithm.

4.3. Algorithm

Our approach to constructing a recommender system can be divided into three distinct stages: selecting a class of similar users, running a matrix completion algorithm, and then deciding how to sample a joke to recommend. In this manner, we determined three particular areas in our method which we could vary to possibly improve the performance of our recommender system. Stated more concretely, the algorithm is stated in Algorithm 1.

We expand on each of the steps for making new recommendations. In order to improve predicting ratings for a given user, we wished to first find a submatrix B of the closest users. Measuring closeness is then a matter of creating a one-hot encoding of each of the categorical features for each user and then computing the squared difference of each of the user features and ratings for

Algorithm 1 Recommend Joke

for each new user **do**
  Add a new empty row to matrix A
end for
Select a small random sample (5) of jokes for this user to rate
for each new recommendation **do**
  Select most similar users based on user profile and ratings
  Construct submatrix B with these users and the current user's rows
  Apply matrix completion to this submatrix B
  From the predicted ratings, either decide to recommend the joke with predicted rating not previously recommended, or a joke that informs the recommender system the most
end for

every given user. Due to the small size of the data set, this is feasibly done in a short amount of time. We then construct the submatrix B using the 50 closest users.

After constructing the submatrix, we then used kNN to complete the missing entries of our matrix. This was done by calculating the mean-squared difference between all non-empty ratings for any two given users and taking the inverse of this value, and using that as the measure of similarity between users. This could potentially lead to some users having infinite similarity, but we simply replaced this with a number larger than any other similarity recorded. More formally, for users u_1, u_2 with ratings vectors r_1, r_2, their similarity is given by

$$s(u_1, u_2) = \min \left\{ \frac{1}{\sum_{i=1}^{m} (r_{1i} - r_{2i})^2}, \ L \right\}$$

where L is larger than any other finite similarity observed. Then for any given missing joke j for user u, we can fill in the rating by taking the set of

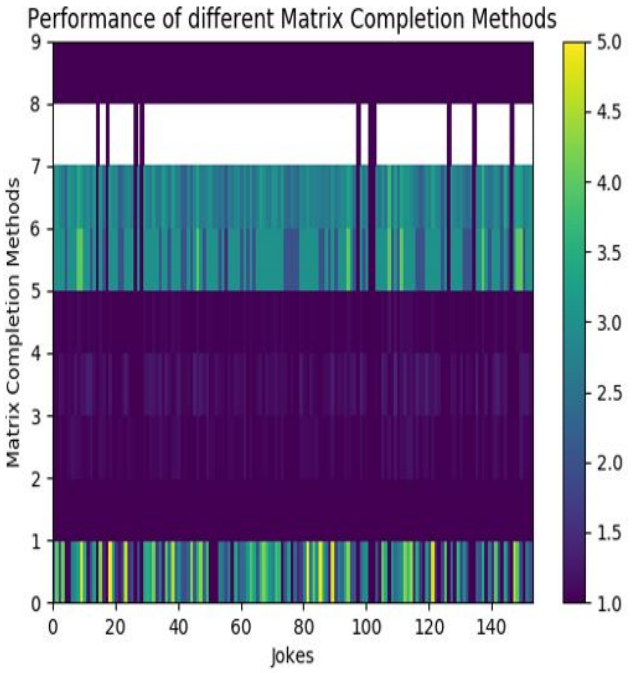

Figure 12: This plot illustrates different predicted ratings for a specific user. The first row is the user's actual ratings and the second row is the collection of known ratings. The remaining rows are the predicted ratings by the following methods in order: mean, median, soft impute, iterativeSVD, nuclear norm minimization, random.

k closest users, say $\{u_1, \ldots, u_k\}$, and computing the value as

$$r_{uj} = \frac{\sum_{i=1}^{k} A_{u_i j} s(u, \ u_i)}{\sum_{i=1}^{k} s(u, \ u_i)}$$

We note that it would be very easy to substitute an alternative matrix completion algorithm to fill in the missing values. In fact, we do this for the purposes of comparison with the other methods of matrix completion, and to observe precisely when k nearest neighbors performs best. An example of the predicted ratings for this method of matrix completion (as well as other matrix completion algorithms) for a specific user's ratings vectors can be seen in Figure 12.

After, we perform this computation for each joke,

we can then simply read off the jokes with the best predicted ratings. In the language of active learning, this would be exploiting our predictive abilities. If we wanted to improve our model, we might decide to instead pick the joke with the most uncertainty (measured possibly by variance or the size of confidence interval fitted to a jokes rating). We do not use this method, but an in-depth discussion of how it may be used is included in (Settles, 2009).

5. Results

In order to measure the performance of separate algorithms, we collected our data set of X jokes and Y users and split our ratings matrix into a training set of $Y - 1$ users and a testing set of the remaining user. We adopted a leave-one-out scheme wherein we then had Y different testing sets wherein each user acted as the single left one out. Next, to simulate a new user, we erased all of the ratings of the test user and then selected a random sample of 5 jokes which would serve as their first ratings (taken from his original ratings, naturally). We would then apply each of our matrix completion algorithms and then compute the mean squared error between our predicted ratings and the original ratings as a score for the performance of a given matrix completion algorithm. These scores are contained in Table 1.

Note that despite its simplicity, the mean performs quite well compared to most of the other methods. There is perhaps a slight improvement with the kNN algorithm, but not a significant one. This comparison is measured using only 5 known jokes, so there should be a greater improvement as the number of known jokes increases.

To measure the performance of the different algorithms as more ratings were received for a specific user, we designed another test. This test consisted of selecting a specific user, clearing their ratings, receiving a single rating, running the matrix completion algorithm, computing the error, and then repeating this process until we have received 20 ratings. This way we will have tracked the errors of our system and we should ideally see an improvement as more ratings are learned. However we note that due to dependence on the exact sample chosen, that there is high variability in the performance (see Figure 13).

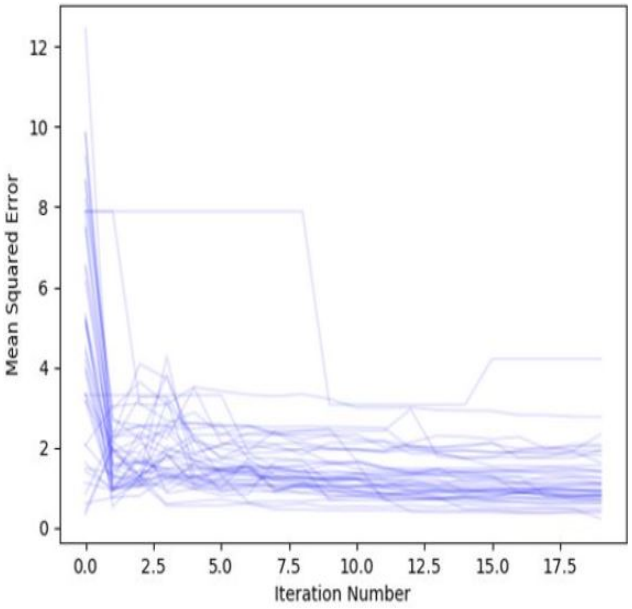

Figure 13: Performance as Number of Ratings Increases

To more accurately capture average performance, we ran the above process for each algorithm 40 times, and calculated the average errors over all the trials. Specific iterations are recorded in Table 2 and a plot of the average errors can be found in Figure 14.

One of the flaws that our matrix completion algorithms suffer from is the cold-start problem wherein we must predict ratings for a user who has not rated any items yet. This is evidenced by algorithms averaging more than 4 error indicating that the predicted ratings were more than 2 off on average. This is terrible performance considering that predicting 3 for every rating will be 2 off at worst. Note also that all the algorithms improve greatly as N increases.

From the ensemble of matrix completion algorithms, it is clear that the k-nearest-neighbors

Scores	
Method	Mean Square Error
Mean	1.633
Iterative SVD	2.118
Soft Impute	3.099
KNN	1.620
Random	2.871

Table 1: Mean Squared Error Scores

Measurements as Iterations Increase					
Method	N=0	N=5	N=10	N=15	N=19
Mean	1.635	1.583	1.532	1.481	1.440
Iterative SVD	4.360	2.043	1.779	1.645	1.524
Soft Impute	4.424	2.976	1.900	1.445	1.247
KNN	5.257	1.417	1.208	1.104	1.070

Table 2: Mean Squared Error Scores

matrix completion algorithm outperforms the remaining. We see this through the use of cross validation with a leave-one-out scheme; we find that the mean square error for the k-nearest-neighbors is the lowest by a significant margin compared to the Soft Impute and Iterative SVD algorithms. It is roughly even with simple imputation using the mean of ratings for a specific joke. With regard to this metric, we find that the KNN is among the top performers achieving lower mean square errors than that obtained by other algorithms (artificial neural networks, logistic regression) in this report.

In addition, it is in the realm of active learning where the kNN algorithm truly shines. After only having learned ratings for five jokes, the kNN approach on average outperforms all other matrix completion approaches. Compared to using neural networks whose parameters and features have to be predefined (meaning we can only work with a small collection of user features), a matrix completion algorithm can be easily updated to incorporate a given users ratings as possible features for the model. Then, by actively updating our recommendation model online with the ratings continuously being read in by new users rating jokes, we can observe the increased precision of our model as it learns users' preferences.

6. Fine-tuning the Algorithm

After having determined the specific matrix completion algorithm, we still had to determine some parameters for the algorithm, in particular the number of neighbors k used for kNN. In order to arrive at a specific value of k, we performed the same cross-validation techniques as before, but instead of applying different matrix completion algorithms, we varied the parameter k. Plots of the mean square errors of the algorithm after having learned 5 ratings for a specific user are depicted in Figure 15.

From the plot, we can see that the error decreases as k increases until k is approximately 10 where it flattens out. As k gets very large, the error seems

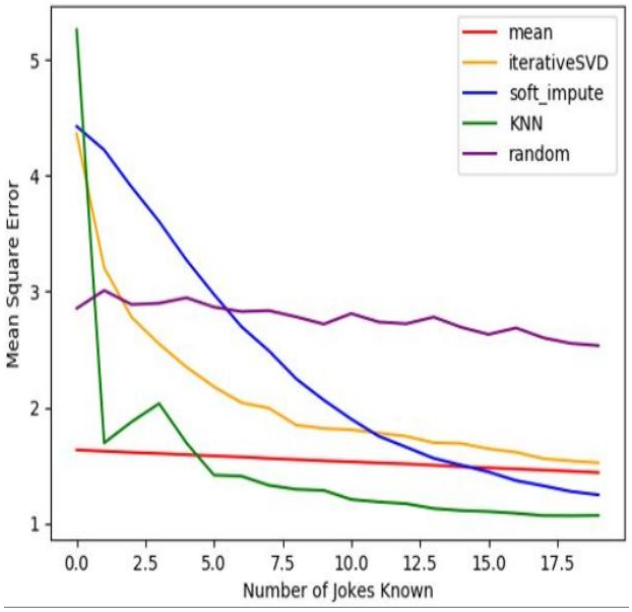

Figure 14: Mean Squared Errors

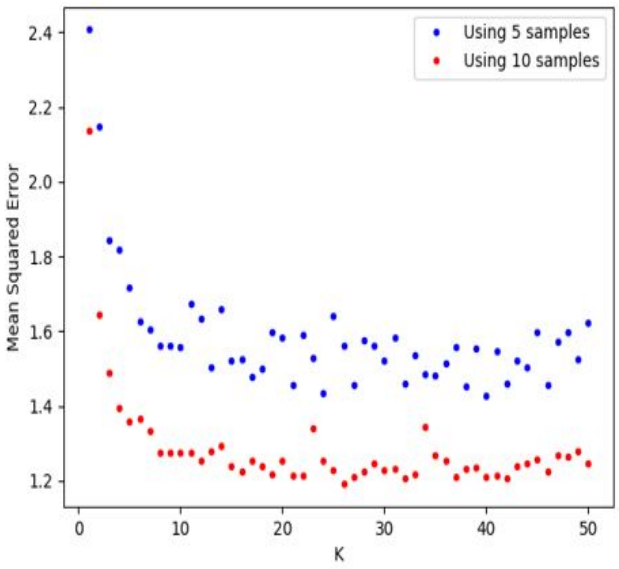

Figure 15: Mean Squared Error for kNN at Various Levels of K

to begin increasing. This coincides with our expectations given that as we increase k, we include more users of farther distance from the predicted user, meaning that the corresponding weight for their contribution is relatively small. Increasing k past a certain point therefore has limited effect on the observed error as seen in the figure.

We would like to choose k to minimize the error, but we would also like constrain our model to a small k so as to minimize the computation time. To this end, we adopt a heuristic wherein our choice of k is determined with the intention to minimize the average of a moving window of errors. Put more concretely, if e_k is the mean squared error observed for a choice of k, we want to find k^* such that

$$k^* = \arg\min_k \frac{1}{n} \sum_{i=1}^{n} e_{k+i-1}$$

where n is the choice of window size. We take this $n = 20$. When we applied this to the above errors, we found a value of $k = 24$ when our model knew 5 ratings of the user, and a value of $k = 26$ when our model knew 10 ratings of the user. With this knowledge, we take the value of k used in our algorithm to be $k = 25$, simply the average of the two observed values.

7. Further Research

While we have observed moderate success in applying collaborative filtering in the form of matrix completion, there are still many areas on which the algorithm can be improved. For example, it is unclear whether our method could be easily extended to a large dataset where we have possibly millions of users and jokes. Naive kNN would have a time complexity of $O(nmk)$ where n is the number of users, m is the number of jokes, and k is the number of nearest neighbors. This quickly becomes infeasible for large n and m values. The other matrix completion algorithms discussed in the text suffer from similar computational complexities, taking seconds to complete matrices of size $10^4 \times 10^4$ (Mazumder et al., 2010). With such a large amount of data, performing matrix completion after every rating would be quickly out of the question, especially when taking into account that there could be multiple users using the recommender at once. Ma-

trix completion perhaps makes more sense as an algorithm which is not updated online, but rather once every hour or so.

If we were to still work with the small dataset, we would wish to improve on our model by taking into account user features possibly in the matrix to be completed. We could then possibly use matrix completion to predict missing user features (of course with a sensible scheme to convert real-valued predictions into categorical data). We would also wish to take into account information about the jokes themselves. In this manner, we could adopt a kind of hybrid collaborative-context filtering scheme wherein we use in conjunction other users ratings and the joke features to predict joke ratings. This type of scheme was used to success in (Basilico & Hofmann, 2004).

References

Comedy — careers — salary, Oct 2017. URL https://www.theartcareerproject.com/careers/comedy/.

Alex Rubinsteyn, Sergey Feldman. fancyimpute. URL https://pypi.python.org/pypi/fancyimpute.

Basilico, Justin and Hofmann, Thomas. Unifying collaborative and content-based filtering. In *Proceedings of the twenty-first international conference on Machine learning*, pp. 9. ACM, 2004.

Bell, Robert M, Koren, Yehuda, and Volinsky, Chris. The bellkor 2008 solution to the netflix prize. *Statistics Research Department at AT&T Research*, 2008.

Candès, Emmanuel J and Recht, Benjamin. Exact matrix completion via convex optimization. *Foundations of Computational mathematics*, 9 (6):717, 2009.

Kalyanakrishnan, Shivaram. An introduction to reinforcement learning. 2014.

Koren, Y., Bell, R., and Volinsky, C. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, Aug 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.263.

Lika, Blerina, Kolomvatsos, Kostas, and Hadjiefthymiades, Stathes. Facing the cold start problem in recommender systems. *Expert Systems with Applications*, 41(4, Part 2): 2065 – 2073, 2014. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2013.09.005. URL http://www.sciencedirect.com/science/article/pii/S0957417413007240.

Matiisen, Tambet. Guest post (part i): Demystifying deep reinforcement learning-nervana. *Nervana.[Online]. Available:*

http://www. nervanasys. com/demystifying-deep-reinforcement-learning/.[Accessed: 08 May 2016], 2015.

Mazumder, Rahul, Hastie, Trevor, and Tibshirani, Robert. Spectral regularization algorithms for learning large incomplete matrices. *Journal of machine learning research*, 11 (Aug):2287–2322, 2010.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830, 2011.

Settles, Burr. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

Shimodaira, Hiroshi. Similarity and recommender systems. *School of Informatics, The University of Eidenburgh*, 21, 2014.

Troyanskaya, Olga, Cantor, Michael, Sherlock, Gavin, Brown, Pat, Hastie, Trevor, Tibshirani, Robert, Botstein, David, and Altman, Russ B. Missing value estimation methods for dna microarrays. *Bioinformatics*, 17(6):520–525, 2001.

Contributions

This section documents the contributions of individual team members. There are three tables of note. The first table lists grades assigned by project management via a combination of liaison oversight of the sub-teams and internal, anonymous surveys. Naomi facilitated the surveys, but was graded independently by the rest of the project management team. The original survey results will be sent along with the report to the Professor.

The second table provides anonymous feedback about what could have been done differently or better. The third table provides anonymous examples of skills and techniques that team members gained by working on the project.

Implementation of the ComAIdian WebApp

The web application provides a way for users to see jokes tailored to their tastes. It takes a users information, and predicts jokes that the user would find funny. The application also provides visualization of the user profiles used in the data. The user interface has a menu bar at the top containing all the tabs that the user can explore. These tabs are the Homepage, Recommended Jokes, Visual Data, About Us, and the user's Profile.

The Homepage can be accessed at any time by clicking the ComAIdian link in the upper left corner of the screen. This page provides a link to create a profile or log in if the user has not already done so, and a link to the recommended jokes tab if the user has logged in.

Before proceeding to the jokes, the user must create a profile. The user must submit an email address and create a password. There is a password confirmation field, and the user cannot create the account if the password and confirmation password do not match. The profile page also requires the user's age, major, birth country, three preferred joke genres, preferred music and movie

genres.

After creating a profile, the user must read five selected jokes, and give a rating out of 5 stars for each of the jokes. Once these ratings are completed, the page displays three jokes that the user will find funny, as predicted by the machine learning algorithm. These jokes are pulled from the jokes database, and have been selected as the jokes that best match the profile that the user has submitted. Each of these jokes has a 5-star rating that can be filled in by a user. There is a button at the bottom of the page to generate new jokes once the user is done reading the current three jokes.

The Visual Data contains charts that show all the profiles used to create the joke predictor. It contains three pie charts displaying majors of the joke raters for each gender group. This shows the percentage of each major that contributed to the dataset among the males, females, and users who preferred not to say their gender. There is also a histogram that shows the frequency of each age group from the joke raters.

The next tab, About Us, displays each group within the Blue Team, and the names of the members in each. There is also a tab where the user can update their profile information. In this tab, the user may change their password (after first inputting their current password), and change any data or preferences.

This web application was created using Ruby on Rails. Ruby on Rails is a framework for handling the model, controller, and view aspects of the website. This framework uses a RESTful API and CRUD operations to maintain the relation between interface and back end. Rail's seamless integration with the database model allows for the joke ratings model and users model to be easily integrated with each other. Additionally, the site uses Ajax to update the ratings of the user without requiring them to submit any form. This feature allows users to only rate some of the jokes on the page and still their ratings considered.

Communications

For our communication, we used a combination of Discord and e-mail. Discord is a VoIP application that allows for voice and text chat between large groups of people. By allowing for sub channels to be created, it ensured that subteam members did not have to read through messages from other sub-teams to find anything relevant to them, and also that they could view other sub team channels as well, in order to keep themselves up to date without having to pester others for information. They worked well together, with Discord allowing for more direct communication between members and e-mail being used for general announcements, as well as making sure that members had a back-up/follow-up to any important links that needed to be sent out en-masse.

For our organization, we used Trello, which uses boards to organize a project into a more collaborative undertaking. The Trello boards enabled team members to see where the project was at any given state. Along with the boards, were a few info cards that detailed what the board was and how to work with the different resources that were being used throughout the project. We also used GitHub and Google Docs, to ease sharing files and code, as well as make creating presentations a truly collaborative effort.

Grades		
Name	Grade	Comments
Okiddy, N	A	Trello, Presented, Surveys
Subramanian, R	A	consistent, good communication
Kim, H	C	Sub-team disapproved, dropped contact
Carlos, D	A	Report, communications, presentations
Daniels-Soles, A	A	2 sub-teams, good ratings
Perano, D	A	Essential SE work
van Tonningen, A	A	75% Work logs
Hegarty, D	A	Work logs
Tang, M	C	Low work logs
Ai, Eric	C	Low work logs
Rajakarunanayake, H	D	No work, wrote SE supp. report
Chan, S	B	50% Work logs
Xu, E	B	50% Work logs
Rahman, F	F	Did not work
Wang, D	B	50% Work logs
Kashyap, C	A	–
Patel, K	A	Rishi vouches
Dang, S	D	Little/No work
Tran, A	B-	–
Tech, C	A	Report
Bhartia, Y	A	–
Kapoor, A	B	–
Kim, I	C	low ratings
Chen, H	B	–
Fanaian, E	C	low ratings
Leung, T	D	Never comm.
Nguyen, D	A	Took charge
Lin, H	D	Never comm.
Schuler, G	D	Low / no comm.
Sun, Y	A	showed initiative
Yin, G	A	made efforts to help
Telles, N	A+	Stellar, above and beyond requirements, XC earned
Leung, A	C	Started helping at end

Comments: Improving the Project
What could be done differently? Why?
Assign myself to a different role with more application time.
Assigned the project earlier, or make it less work.
A lot of people dropped out of the class and that made it less fun. Have the professor be more involved with team members especially by making sure we can get updated lists of who drops the class.
Been more involved in algorithm development
Create guidelines at the beginning of the project to be followed.
Discuss with UI/Visualization more
Done more programming to better understand the algorithms
Go to SE meetings earlier, as there was no UI PM or organization
Have a better PM. I feel as if he did not directly assign work with deadlines...
I honesty felt this whole project was disorganized. I implemented regular regression but it wasnt used
I wish the teams were smaller. Also, I would have chosen Slack for communication.
I would have emphasized to other teammates how much we needed critical infrastructure like a database server sooner.
More small groups / less overall group members
Take the lead in certain situations.
Use ReactJS to develop the Front end page

Comments: What We Learned
What skills did you gain from this project?
Bash scripting, docker, testing in python, and working other teams.
Communication skills and deadline planning.
Extended knowledge about ML
front end programming
How to identify people who are not working and keep getting work done without them.
I have a much greater familiarity with the machine learning libraries in python.
I learned how to design ANN for predicting a joke for the new user that inserts his/her information.
I learned more about machine learning theory because there was a lot of research involved to pick algorithms to implement.
I also learned how to clean up messy data.
I learned the main concepts behind ANNs and how to use them in a real world scenario.
I realised I'm good at administrative work
working with SQLAlchemy was new for me.
learning about web application tools like flask, react and deployment tools like docker, aws, apache
Python Visualization. Matrix Completion. How team members are all different skilled, but each person can bring something to the table.
using tensorflow for graphing
Working in large groups